МЕЖДУНАРОДНЫЙ СОЮЗ ЭЛЕКТРОСВЯЗИ

МСЭ-Т

СЕКТОР СТАНДАРТИЗАЦИИ
ЭЛЕКТРОСВЯЗИ МСЭ

Z.140

(07/2001)

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

Методы формального описания (FDT)

Древовидно-табличная комбинированная нотация версии 3 (TTCN-3): базовый язык

Рекомендация МСЭ-Т Z.140

(Ранее "Рекомендация МККТТ")

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 последовательностей сообщений	Z.120–Z.129
ЯЗЫКИ ПРОГРАММИРОВАНИЯ	
CHILL: Язык высокого уровня МСЭ-Т	Z.200–Z.209
ЯЗЫК "ЧЕЛОВЕК–МАШИНА"	
Общие принципы	Z.300–Z.309
Базовый синтаксис и диалоговые процедуры	Z.310–Z.319
Расширенный язык MML для видеотерминалов	Z.320–Z.329
Спецификация интерфейса "человек–машина"	Z.330–Z.399
КАЧЕСТВО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ЭЛЕКТРОСВЯЗИ	Z.400–Z.499
МЕТОДЫ ПРОВЕРКИ ДОСТОВЕРНОСТИ И ТЕСТИРОВАНИЯ	Z.500–Z.599

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

Рекомендация МСЭ-Т Z.140

Древовидно-табличная комбинированная нотация версии 3 (TTCN-3): базовый язык

Резюме

В настоящей Рекомендации определяется базовый язык древовидно-табличной комбинированной нотации версии 3 (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 обобщенной архитектуры посредника объектных запросов (COBRA), тестирование прикладных программных интерфейсов (API) и др. Применение TTCN-3 не ограничено аттестационным тестированием; этот язык может использоваться для многих других видов тестирования, таких, например, как тестирование взаимодействия, устойчивости, ухудшений, систем и интеграции. Спецификация тестовых примеров для протоколов физического уровня выходит за рамки данной Рекомендации.

Источник

Рекомендация МСЭ-Т Z.140 была подготовлена 10-й Исследовательской комиссией МСЭ-Т (2001–2004 гг.) и утверждена 22 июля 2001 года в соответствии с процедурой, изложенной в Резолюции 1 ВАСЭ.

ПРЕДИСЛОВИЕ

Международный союз электросвязи (МСЭ) является специализированным учреждением Организации Объединенных Наций в области электросвязи. Сектор стандартизации электросвязи (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.

Всемирная ассамблея по стандартизации электросвязи (ВАСЭ), которая проводится каждые четыре года, определяет темы для изучения Исследовательскими комиссиями МСЭ-Т, которые, в свою очередь, разрабатывают 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 данной Рекомендации термин "администрация" используется для краткости и обозначает как администрацию электросвязи, так и признанную эксплуатационную организацию.

ПРАВА ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

МСЭ обращает внимание на то, что практическое применение или реализация данной Рекомендации может включать в себя использование заявленного права интеллектуальной собственности. МСЭ не занимает какую бы то ни было позицию относительно подтверждения, обоснованности или применимости заявленных прав интеллектуальной собственности, независимо от того, отстаиваются ли они членами МСЭ или другими сторонами вне процесса подготовки Рекомендации.

На момент утверждения настоящей Рекомендации МСЭ не получил извещения об интеллектуальной собственности, защищенной патентами, которые могут потребоваться для реализации данной Рекомендации. Однако те, кто будет применять Рекомендацию, должны иметь в виду, что это может не отражать самую последнюю информацию, и поэтому им настоятельно рекомендуется обращаться к патентной базе данных БСЭ.

СОДЕРЖАНИЕ

Стр.

Стр.

Стр.

Рекомендация МСЭ-Т Z.140

Древовидно-табличная комбинированная нотация версии 3 (TTCN-3): базовый язык

1 Предмет рассмотрения

В настоящей Рекомендации определяется базовый язык TTCN версии 3 (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 COBRA, 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. Спецификация тестовых примеров для протоколов физического уровня выходят за рамки данной Рекомендации.

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 [1] и графический формат представления [2]. Спецификация этих форматов выходит за рамки данной Рекомендации.

В данной Рекомендации определяется нормативный способ использования совместно с TTCN-3 языка ASN.1, определенного в Рекомендациях МСЭ-Т серии X.680, а именно в Рекомендациях [7], [8], [9] и [10]. Гармонизация других языков с TTCN-3 выходит за рамки настоящей 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 Ссылки

Нижеуказанные документы содержат положения, которые путем ссылки на них в данном тексте образуют положения настоящей Рекомендации.

Ссылки являются либо конкретными (определяемыми датой публикации и/или номером издания или номером версии), либо неконкретными.

– В отношении конкретных ссылок последующие пересмотренные версии неприменимы.

– В отношении неконкретных ссылок применимы самые последние версии.

[1] Рекомендация МСЭ-Т Z.141 (2001), *Древовидно-табличная комбинированная нотация версии 3 (TTCN-3): Табличный формат представления.*

[2] Рекомендация МСЭ-Т Z.142 (проект), *Древовидно-табличная комбинированная нотация версии 3 (TTCN-3): Графический формат.*

[3] Рекомендация МСЭ-Т X.290 (1995), *Методология аттестационного тестирования ВОС и структура Рекомендаций о протоколах для применений МСЭ-Т – Общие понятия.*

ИСО/МЭК 9646-1: 1994, *Информационные технологии – Взаимосвязь открытых систем – Методология и структура аттестационного тестирования – Часть 1: Общие понятия.*

[4] Рекомендация МСЭ-Т X.292 (1998), *Методология аттестационного тестирования ВОС и структура Рекомендаций о протоколах для применений МСЭ-Т – Древовидно-табличная комбинированная нотация (TTCN).*

ИСО/МЭК 9646-3: 1998, *Информационные технологии – Взаимосвязь открытых систем – Методология и структура аттестационного тестирования – Часть 3: Древовидно-табличная комбинированная нотация (TTCN).*

[5] Рекомендация МСЭ-Т Т.50 (1992), *Международный эталонный алфавит (МЭА) (Ранее Международный алфавит № 5, или МА5) – Информационные технологии – Набор 7-битовых кодированных знаков для обмена информацией.*

ИСО/МЭК 646: 1991, *Информационные технологии – Набор 7-битовых кодированных знаков ИСО для обмена информацией.*

[6] ИСО/МЭК 10646-1: 1993, *Информационные технологии – Универсальный набор многооктетных кодированных знаков (UCS) – Часть 1: Архитектура и базовая многоязычная плоскость*.

[7] Рекомендация МСЭ-Т X.680 (1997)|ИСО/МЭК 8824-1: 1998, *Информационные технологии – Абстрактно-синтаксическая нотация 1 (ASN.1): Спецификация базовой нотации.*

[8] Рекомендация МСЭ-Т X.681 (1997)|ИСО/МЭК 8824-2: 1998, *Информационные технологии – Абстрактно-синтаксическая нотация 1 (ASN.1): Спецификация информационных объектов.*

[9] Рекомендация МСЭ-Т X.682 (1997)|ИСО/МЭК 8824-3: 1998, *Информационные технологии – Абстрактно-синтаксическая нотация 1 (ASN.1): Спецификация ограничений.*

[10] Рекомендация МСЭ-Т X.683 (1997)|ИСО/МЭК 8824-4: 1998, *Информационные технологии – Абстрактно-синтаксическая нотация 1 (ASN.1): Параметризация спецификаций ASN.1.*

[11] Рекомендация МСЭ-Т X.690 (1997)|ИСО/МЭК 8825-1: 1998, *Информационные технологии – Правила кодирования ASN.1: Спецификация базовых правил кодирования (BER), канонических правил кодирования (CER) и различительных правил кодирования (DER).*

[12] Рекомендация МСЭ-Т X.691 (1997)|ИСО/МЭК 8825-2: 1998, *Информационные технологии – Правила кодирования ASN.1: Спецификация уплотненных правил кодирования (PER).*

3 Определения терминов и сокращения

3.1 Определения терминов

В настоящей Рекомендации определяются следующие термины:

3.1.1 совместимый тип (compatible type): TTCN-3 не имеет строгой классификации по типам, но этот язык требует совместимости типов. Переменные, константы, шаблоны и т. п. имеют совместимые типы, если они сводятся к одному и тому же базовому типу и в случае присвоения, сопоставления и т. д. не нарушают свойства подтипов (например, диапазоны, ограничения длины).

3.1.2 порт связи (communication port): 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.

3.1.3 особое состояние (exception): При синхронной связи особое состояние (если оно определено) порождается отвечающим объектом, когда на удаленный вызов процедуры он не может дать обычный ожидаемый ответ.

3.1.4 тестовая последовательность (test suite): Модуль TTCN-3, который посредством команд импорта явно или неявно обеспечивает полную спецификацию всех определений и описаний поведения, необходимых для определения полного набора тестовых примеров.

3.1.5 интерфейс тестовой системы (test system interface): Тестовый компонент, который обеспечивает отображение портов, доступных в (абстрактной) тестовой системе TTCN-3, в порты, предоставляемые реальной тестовой системой.

3.1.6 параметризация типа (type parametrization): Способность переслать тип в виде реального параметра в некоторый параметризованный объект. Этот реальный параметр типа затем завершает спецификацию типа данного объекта. Следует отметить, что данный параметр является не значением типа, а соответственно типом.

3.2 Определения терминов из Рекомендации МСЭ-Т X.290 и стандарта ИСО/МЭК 9646-1

В настоящей Рекомендации применяются следующие термины и их определения из Рекомендации МСЭ-Т X.290 и стандарта ИСО/МЭК 9646-1 [3]:

– заявление о соответствии реализации (ЗСР) (Implementation Conformance Statement, ICS);

– дополнительная информация о реализации для тестирования (ДИРТ) (Implementation eXtra Information for Testing, IXIT);

– тестируемая реализация (ТР) (Implementation Under Test, IUT);

– тестируемая система (ТС) (System Under Test, SUT);

– тестовый пример (test case);

– ошибка тестового примера (test case error);

– тестовая система (test system).

3.3 Определения терминов из Рекомендации МСЭ-Т X.292 и стандарта ИСО/МЭК 9646-3

В настоящей Рекомендации применяются следующие термины и их определения из Рекомендации МСЭ-Т X.292 и стандарта ИСО/МЭК 9646-3 [4]:

– главный тестовый компонент (Main Test Component, MTC);

– параллельный тестовый компонент (Parallel Test Component, PTC);

– семантика "снимков" (snapshot semantics).

3.4 Сокращения

В настоящей Рекомендации используются следующие сокращения:

API Прикладной программный интерфейс

ASN.1 Абстрактно-синтаксическая нотация 1

ASP Примитив абстрактной службы

ATS Абстрактная тестовая последовательность

BNF Форма Бэкуса–Наура

COBRA Обобщенная архитектура посредника объектных запросов

ETS Выполнимая тестовая последовательность

FIFO Первым пришел – первым обслужен

IDL Язык определения интерфейса

IUT Тестируемая реализация (ТР)

MTC Главный тестовый компонент

PDU Протокольный блок данных (ПБД)

PICS Заявление о соответствии реализации протокола (ЗСРП)

PIXIT Дополнительная информация о реализации протокола для тестирования

PTC Параллельный тестовый компонент

SUT Тестируемая система (ТС)

TTCN Древовидно-табличная комбинированная нотация

4 Введение

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 COBRA, 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.

С синтаксической точки зрения TTCN-3 сильно отличается от ранних версий этого языка, определенных в Рекомендации МСЭ-Т X.292 и стандарте ИСО/МЭК 9646-3 [4]. Однако многие из хорошо зарекомендовавших себя базовых функциональных возможностей TTCN были сохранены, а в некоторых случаях развиты. TTCN-3 имеет следующие основные характеристики:

- способность определять конфигурации динамичного одновременного тестирования;
- работа с синхронными и асинхронными системами связи;
- способность определять информацию о кодировании и другие атрибуты (включая пользовательские расширения);
- способность определять шаблоны данных и сигнатур с эффективными механизмами сопоставления;
- параметризация типов и значений;
- присвоение и обработка тестовых вердиктов;
- механизмы параметризации тестовой последовательности и выбора тестового примера;
- комбинированное использование TTCN-3 с ASN.1 [и потенциальное использование с другими языками, такими как язык определения интерфейса (IDL)];
- 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

Исторически TTCN всегда был связан с аттестационным тестированием. Чтобы сделать этот язык доступным в более широком диапазоне применений тестирования как в сфере стандартов, так и в сфере промышленности, в данной Рекомендации 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, который по внешнему виду и функциональным возможностям аналогичен ранним версиям TTCN. Третьей частью, определяемой в проекте Рекомендации МСЭ-Т Z.142 [2], является графический формат представления.

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 между инструментами TTCN;

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 полностью совместим с языком ASN.1, который может факультативно использоваться с модулями TTCN-3 в качестве альтернативного синтаксиса типов и значений данных. Использование ASN.1 в модулях TTCN-3 определяется в Приложении E. Подход, примененный для комбинирования ASN.1 и TTCN-3, мог бы использоваться для поддержки комбинирования других систем типов и значений с TTCN-3. Однако детали этого не определяются в настоящей Рекомендации.

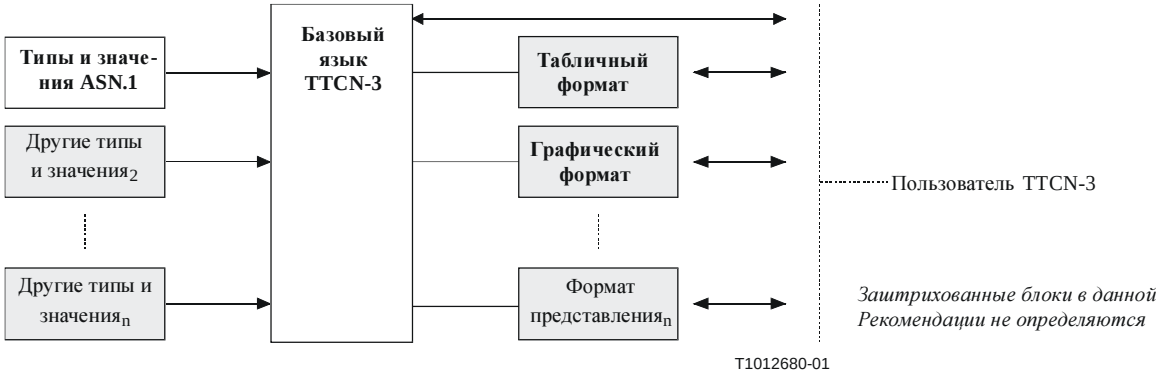


Рисунок 1/Z.140 – Точка зрения пользователя на базовый язык и различные форматы представления

Базовый язык определяется полным синтаксисом (см. Приложение A) и операционной семантикой (см. Приложение B). 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. Функциональные возможности предыдущих версий TTCN, такие как индексы тестовых последовательностей, которые могут быть обеспечены с помощью патентованных инструментов, не входят в состав TTCN-3.

5 Элементы базового языка

Единицей высшего уровня TTCN-3 является модуль. Модуль не может разделяться на субмодули. Он может импортировать определения из других модулей. Модули могут иметь списки параметров для получения некоторой формы параметризации тестовой последовательности, аналогичной механизмам параметризации PICS и PIXIT из TTCN-2.

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. Факультативно с TTCN-3 могут использоваться типы и значения ASN.1 путем их импортирования.

Особый вид значений данных, называемый шаблоном, обеспечивает механизмы параметризации и сопоставления для определения тестовых данных, передаваемых или принимаемых через тестовые порты. Работа этих портов обеспечивает возможности как асинхронной, так и синхронной связи. 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, большинству элементов 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 языка	Соответствующее ключевое слово	Указывается в определениях модулей	Указывается в управлении модулей	Указывается в функциях/ тестовых примерах
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
Определения шаблонов данных/ сигнатур	**template**	Да		
Определения функций	**function**	Да		
Определения именованных альтернатив	**named alt**	Да		
Определения тестовых примеров	**testcase**	Да		
Объявления переменных	**var**		Да	Да
Объявления таймеров	**timer**		Да	Да

5.1 Определения, экземпляры и объявления

В настоящей Рекомендации термин "объявление" используется в общем смысле и охватывает формулирование статического определения или создание некоторого вида динамического экземпляра, в котором дается имя для объекта TTCN-3. Например, определяются типы или константы, при этом команда, такая как вызов функции или объявление переменной, является экземпляром. В обоих случаях эти действия могут быть названы "объявлением".

5.2 Порядок следования элементов языка

В общем случае порядок, в котором могут делаться объявления, и смешивание объявлений с программными командами могут быть произвольными. Однако внутри блока команды, например внутри ветви команды **if-else**, все объявления (если таковые имеются) делаются только в начале блока команды.

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;
   :
}

   :
```

5.2.1 Ссылки вперед

Определения в определяющей части модуля могут даваться в любом порядке. Хотя следует избегать ссылок вперед (для удобства чтения), это не является обязательным. Например, рекурсивные элементы, такие как функции, которые вызывают другие функции и параметризацию модуля, могут привести к неизбежным ссылкам вперед.

Ссылки вперед разрешаются только для объявлений в определяющей части модуля. Ссылки вперед никогда не должны делаться внутри управляющей части модуля, определений тестовых примеров, функций и именованных альтернатив. Это означает, что никогда не должны появляться ссылки вперед на местные переменные, местные таймеры и местные константы.

5.3 Параметризация

TTCN-3 поддерживает как параметризацию *типа*, так и 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* типа и *статического* значения, то есть эта параметризация должна разрешаться во время трансляции;

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

e) именованные альтернативы поддерживают параметризацию *динамических* значений (то есть эта параметризация должна быть разрешена во время выполнения). Поскольку именованные альтернативы не являются единицей контекста, определенные формальные параметры просто заменяются заданными реальными параметрами, когда выполняется (макро) расширение элемента **`named alt`**.

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 типа	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`	Статический во время трансляции	Статическое во время трансляции	*Значения для:* всех базовых типов, всех определяемых пользователем типов и типа **`address`**. ПРИМЕЧАНИЕ. – Определения **`record of, set of, enumerated, port, component`** и **`subtype`** не разрешают параметризацию.
`template`		Динамическое во время выполнения	*Значения для:* всех базовых типов, всех определяемых пользователем типов, типа **`address`**, типа **`component`** и `template`.
`function`		Динамическое во время выполнения	*Значения для:* всех базовых типов, всех определяемых пользователем типов, типа `address`, типа **`component`**, типа **`port`**, **`template`** и **`timer`**.

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

Ключевое слово	Параметризация типа	Параметризация значения	Типы значений, которые могут появляться в списках формальных/реальных параметров
`testcase`		Динамическое во время выполнения	*Значения для:* всех базовых типов и всех определяемых пользователем типов, типа **`address`** и **`template`**.
`signature`		Динамическое во время выполнения	*Значения для:* всех базовых типов, всех определяемых пользователем типов, типа **`address`** и типа **`component`**.
`named alt`		Статическое макрорасширение	*Значения для:* всех базовых типов, всех определяемых пользователем типов, типа **`address`**, типа **`component`**, типа **`port`**, **`template`** и **`timer`**.
ПРИМЕЧАНИЕ. – Примеры синтаксиса и конкретного использования параметризации с различными элементами языка даются в соответствующих разделах этой Рекомендации.			

5.3.1 Передача параметров по ссылке и по значению

Все 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 и обозначаются ключевыми словами **`timer`** и **`port`**. Ключевое слово **`inout`** может использоваться факультативно для указания на передачу по ссылке.

5.3.1.1 Параметры, передаваемые по ссылке

Передача параметров по ссылке имеет следующие ограничения:

a) Только список формальных параметров для элементов **`function`**, **`signature`** и **`testcase`** может содержать параметры, передаваемые по ссылке.

ПРИМЕЧАНИЕ. – Имеются дальнейшие ограничения на способ использования параметров, передаваемых по ссылке, для **`signature`** (см. раздел 22).

b) Реальные параметры должны быть только переменными (например, не константами или шаблонами).

c) Передаваться по ссылке должны только параметры значений (то есть не параметры типов).

Пример:

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

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

5.3.1.2 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 не обязательно
```

5.3.2 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);
```

5.3.3 Пустой список формальных параметров

Если пустым является список формальных параметров для параметризуемого элемента языка TTCN-3 типа функции (то есть **function**, **testcase**, **signature**, **named alt** или функция **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 { ... }
```

5.3.4 Вложенные списки параметров

Обычно собственные параметры параметризованных объектов, указанных в виде реальных параметров, должны быть разрешены в этом списке реальных параметров.

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
{  :
   TC001(MyTemplate(7));
   :
}
```

5.4 Контекстные правила

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

a) модули;

ПРИМЕЧАНИЕ. – Имеются дополнительные ограничивающие правила для групп (см. п. 7.3.1).

b) управляющая часть модуля;

c) функции;

d) тестовые примеры;

e) блоки команд внутри управляющей части, функций и тестовых примеров.

Каждая единица контекста состоит из (факультативных) объявлений и некоторой формы (факультативного) функционального описания. Все единицы контекста, кроме модулей, построены по иерархическому принципу, причем каждый уровень иерархии определяет собственный местный контекст. Объявления из более высокого уровня контекста видны нижним уровням (в пределах одной и той же иерархии контекста). Объявления из нижних уровней контекста невидимы для объявлений из более высокого уровня контекста.

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
      :

   }
}
```

5.4.1 Контекст и совпадение идентификаторов

TTCN-3 не поддерживает совпадение идентификаторов, то есть все идентификаторы в одной иерархии контекста должны быть уникальными. 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 иерархии контекста).

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 (в предположении, что в заголовке модуля нет объявления A)
function MyBehaviourA()
{  :
   const integer A := 1;
   :
}

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

5.4.2 Контекст формальных параметров

Контекст формальных параметров в параметризованном элементе языка (например, в вызове функции) должен быть ограничен определением, в котором появляются эти параметры, и нижележащими уровнями контекста в той же иерархии контекста. То есть они следуют обычным контекстным правилам (см. п. 5.4). Правила для совпадения идентификаторов (см. п. 5.4.1) также применимы к формальным параметрам.

5.5 Идентификаторы и ключевые слова

Идентификаторы TTCN-3 чувствительны к регистру клавиатуры, а ключевые слова TTCN-3 должны записываться только строчными буквами (см. Приложение A).

6 Типы и значения

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 **objid** и **verdicttype**. Структурированные типы, такие как **record**, **set** и **enumerated**, могут конструироваться из этих базовых типов.

Специальные типы, связанные с конфигурациями, такие как **address, port** и **component**, могут использоваться для определения архитектуры тестовой системы (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
	char	range, list
	universal char	range, list
	float	list
	boolean	list
	objid	list
	verdicttype	list
Базовые цепочечные типы	**bitstring**	list, length
	hexstring	list, length
	octetstring	list, length
	charstring	list, length
	universal charstring	list, length

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

Класс типов	Ключевое слово	Подтипы
Определяемые пользователем структурированные типы	**record**	list
	record of	list
	set	list
	set of	list
	enumerated	list
	union	list
Специальные типы для конфигураций	**address**	
	port	
	component	

6.1 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) **char** (знак): тип, который различает значения, являющиеся знаками из Рекомендации МСЭ-Т Т.50 и ИСО/МЭК 646 [5].

Значения типа **char** могут быть заключены в двойные кавычки (") или перечисляться с использованием заранее определенной преобразующей функции с положительным целочисленным значением ее кодирования в качестве аргумента.

Порядок следования значений типа **char** определяется целочисленным значением их кодирования, то есть операторы отношений = =, <, >, ! =, > = и < = могут использоваться для сравнения значений типа **char**.

c) **universal char** (универсальный знак): тип, различаемые значения которого являются одиночными знаками из стандарта ИСО/МЭК 10646-1 [6].

Значения типа **universal char** могут быть заключены в двойные кавычки (") или перечисляться с использованием заранее определенной преобразующей функции с положительным целочисленным значением ее кодирования в качестве аргумента.

Порядок следования значений типа **universal char** определяется целочисленным значением их кодирования, то есть операторы отношений = =, <, >, ! =, > = и < = могут использоваться для сравнения значений типа **universal char**.

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

Числа с плавающей запятой выражаются как: <*мантисса*>*<*основание*> $^{<\text{показатель степени}>}$

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. (Символ * означает знак умножения.)

Представление чисел с плавающей запятой ограничено основанием со значением 10. Значения с плавающей запятой (точкой) могут выражаться разными способами:

- нормальным 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 $-123456789*10^{-6}$); либо

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

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

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

f) **objid** (идентификатор объекта): тип, различаемые значения которого представляют собою набор всех идентификаторов объектов, прикрепленных согласно правилам из [7], [8], [9] и [10]. Например:

{itu-t(0) identified-organization(4) etsi(0)},

или альтернативно: {itu-t identified-organization etsi},

или альтернативно: { 0 4 0 }.

g) **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 "цепочка" и их значения

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

ПРИМЕЧАНИЕ. – 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) нулей и единиц, которым предшествует одинарная кавычка (') и за которыми следует пара знаков 'B. Например:

```
'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:

```
0 1 2 3 4 5 6 7 8 9 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. Например:

```
'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:

```
0 1 2 3 4 5 6 7 8 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. Например:

```
'FF96'O
```

d) **charstring** (цепочка знаков): это типы, различаемыми значениями которых являются нуль (знаков), один или более знаков из Рекомендации МСЭ-Т T.50 и стандарта ИСО/МЭК 646 [5]. Тип "цепочка знаков", которому предшествует ключевое слово **universal**, означает типы, различаемыми значениями которых являются нуль, один или более знаков из стандарта ИСО/МЭК 10646-1 [6].

Значения типа **charstring** и типа **universal charstring** выражаются произвольным числом (возможно, нулем) знаков из соответствующих наборов знаков; перед этими знаками и за ними располагаются двойные кавычки (").

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. Например, "abcd" представляет буквенную цепочку "abcd".

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.

Таблица 4/Z.140 – Единицы длины, используемые в спецификациях длин полей

Тип	Единицы длины
bitstring	биты
hexstring	шестнадцатеричные цифры
octetstring	октеты
цепочки знаков	знаки

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

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

6.2 Определяемые пользователем подтипы и их значения

Определяемые пользователем типы обозначаются ключевым словом **type**. С помощью определяемых пользователем типов можно определять подтипы (такие как списки, диапазоны и ограничения длины) для **integer** и различных цепочечных типов.

6.2.1 Списки значений

TTCN-3 позволяет определять список различаемых значений любого заданного типа из таблицы 3. Значения в таком списке должны быть базового типа и должны представлять собой правильное подмножество значений, определенных для рассматриваемого базового типа. Подтип, определенный этим списком, ограничивает разрешенные значения данного подтипа значениями из этого списка.

Например:

```
type bitstring MyListOfBitStrings ('01'B, '10'B, '11'B);
```

6.2.2 Диапазоны

TTCN-3 позволяет определять диапазон значений для типов **integer, char** и **universal char** (или производных от этих типов). Подтип, определенный таким диапазоном, ограничивает разрешенные значения данного подтипа значениями этого диапазона, включая нижнюю и верхнюю границы. Например:

```
type integer MyIntegerRange (0 .. 255);
```

6.2.2.1 Неограниченные диапазоны

Чтобы определить неограниченный диапазон целых чисел, можно использовать ключевое слово infinity вместо значения, указывающего на отсутствие нижней или верхней границы. Верхняя граница должна быть больше нижней границы или равна ей. Например:

```
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

Для значений типов **integer, char** и **universal char** (или производных от этих типов) возможны смешанные списки и диапазоны. Например:

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

6.2.3 Ограничения длины цепочки

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

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

В таблице 4 указаны единицы длины для разных цепочечных типов.

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.

6.3 Структурированные типы и значения

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 некоторой краткой функции инициализации. Например:

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

// Или
const MyRecordType MyRecordValue:= {'11001'B, true, "A string"}
```

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

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

6.3.1 Тип "запись" и его значения

TTCN-3 поддерживает упорядоченные структурированные типы, известные как **record** (запись). Элементами типа "запись" могут быть любые базовые типы или определяемые пользователем типы, 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. Константа, являющаяся типом записи, не должна содержать переменных (включая параметры модуля) в качестве значений поля ни прямо, ни косвенно.

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

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

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

```
type record MyEmptyRecord {  }
```

Значение записи присваивается отдельно для каждого элемента. Например:

```
var integer MyIntegerValue:= 1;

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

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

Либо значение записи присваивается с помощью функции инициализации. Например:

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

Для факультативных полей разрешается опускать значение с помощью символа пропуска параметра. Например:

```
// Отметим, что это тождественно следующей записи, то есть значение field2 не определено
MyRecordValue.field1 := MyIntegerValue;
MyRecordValue.field3 := "A string"
```

6.3.1.1 Ссылки на вложенные поля записи

Ссылки на элементы вложенных записей делаются с помощью пары *RecordId.ElementId*. Например:

```
MyVar1 := MyRecord1.MyElement1;
// Если запись является вложенной, то ссылка может быть наподобие такой
MyVar2 := MyRecord1.MyElement1.MyRecord2.MyElement2;
```

6.3.1.2 Факультативные элементы в записи

Факультативные элементы в record указываются с помощью ключевого слова optional. Например:

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

6.3.2 Тип "множество" и его значения

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. Например:

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

Нотация функции инициализации для устанавливаемых значений не должна использоваться для значений типов **set**.

6.3.2.1 Факультативные элементы в "множестве"

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

6.3.3 Записи и множества одних и тех же типов

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**. Например:

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

Нотация значений для **record of** и **set of** такая же, как для массивов (см. п. 6.4).

6.3.4 Перечислительный тип и его значения

TTCN-3 поддерживает перечислительные типы. Типы **enumerated** используются для построения типов, которые требуют только отдельного именованного множества значений. В операциях с перечислительными типами должны использоваться только именованные идентификаторы и только операторы присвоения, эквивалентности и упорядочения.

Каждое именованное значение может факультативно иметь связанное целочисленное значение, которое определяется после имени в круглых скобках. Эти значения используются системой только для того, чтобы разрешить применение родственных операторов. Если явные целые числа не даны, то предполагается, что порядок следования начинается с нуля. Например:

```
type enumerated MyEnumType
{
   Monday, Tuesday, Wednesday, Thursday, Friday
}

// Действительным текущим значением MyEnumType может быть
var MyEnumType Today      := Monday;
var MyEnumType Tomorrow   := Tuesday;
// при этом истинно утверждение Today < Tomorrow
```

6.3.5 Объединения

TTCN-3 поддерживает типы **union**. Они аналогичны типам **records**, за исключением того, что 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. Например:

```
type union MyUnionType
{
   integer       number,
   charstring    string
}
// Действительным текущим значением MyUnionType может быть
var MyUnionType age;
age.number := 34;
```

Нотация функции инициализации для установки значений не должна использоваться для значений типа **union**.

Ключевое слово **optional** не должно использоваться с типами **union**.

6.4 Массивы

Как во многих языках программирования, массивы не считаются типами в TTCN-3. Вместо этого они могут определяться в момент объявления какой-либо переменной. Например:

```
var integer MyArray[3];   // Текущие значения целочисленного массива
                          // из 3 элементов с индексами от 0 до 2
```

Значения элементов массива должны быть совместимы с соответствующим объявлением переменных. Значения могут присваиваться индивидуально или все сразу. Например:

```
MyArray[0]:= 10;
MyArray[1]:= 20;
MyArray[2]:= 30;

// либо с помощью функции инициализации
MyArray:= {10, 20, 30};
```

Индексами массива являются выражения, которые должны иметь положительные значения **integer**, включая значение "нуль". По умолчанию индексация массивов TCNN-3 начинается с цифры 0 (нуль).

Размеры массива определяются с помощью выражений с постоянными, которые должны иметь положительное значение integer. Размеры массива могут определяться также с помощью диапазонов. В таких случаях нижнее и верхнее значения диапазона определяют нижнее и верхнее значения индекса. Например:

```
var integer MyArray[1 .. 5];    // Текущие значения целочисленного массива из
                                // 5 элементов с индексами от 1 до 5
MyArray[1] := 10;   // Самый нижний индекс
MyArray[5] := 50;   // Самый верхний индекс
```

Массивы записей типов дают возможность определять многоразмерные массивы. Например:

```
// 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;
```

6.5 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.

6.6 Параметризация типов

Параметризация типов позволяет применять фиктивные идентификаторы типов, которые играют роль держателей места для типа. Это значит, что любой тип может оставаться открытым в определителе TTCN-3, пока он не будет привлечен во время трансляции.

ПРИМЕЧАНИЕ. – Это является обобщением понятия метатипа PDU из TTCN-2.

Реальный тип известен только в случае реального использования параметра типа. Например:

```
type record MyRecordType(MyMetaType)
{
   boolean  field1,
   MyMetaType  field2   // MyMetaType не относится к конкретному типу
}

var MyRecordType(integer) MyRecordValue :=
{
   field1 := true,
   field2 := 123       // MyMetaType теперь имеет тип integer
}
```

6.7 Совместимость типов

В TTCN-3 отсутствует строгая классификация по типам, но этот язык требует совместимости типов. Переменные, константы, шаблоны и т. п. имеют совместимые типы, если они сводятся к одному и тому же базовому типу, а в случаях присвоения, сопоставления и т. д. – не нарушается образование подтипов (например, диапазонов, ограничений длины). Например:

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

// Тогда
x := 20; // Действительное присвоение
y := 20; // НЕ действительное присвоение, так как 20 не находится в диапазоне y

y := 5;  // Действительное присвоение

x := y;  // Действительное присвоение, так как значение y находится в диапазоне x
y := x;  // НЕ действительное присвоение, так как значение x не находится в
         // диапазоне y

x := 5;  // Действительное присвоение
y := x;  // Действительное присвоение, так как значение x теперь находится
         // в диапазоне y
```

6.7.1 Преобразование типов

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

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

7 Модули

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, за которым следует факультативный идентификатор объекта.

ПРИМЕЧАНИЕ. – Идентификатор модуля является неформальным текстовым именем модуля.

7.2 Параметризация модулей

Список параметров **module** определяет набор значений, которые выдаются тестовой средой во время выполнения. В процессе выполнения теста эти значения рассматриваются как константы. Например:

```
module MyParameterizedModule(integer TS_Par1, boolean TS_Par2, hexstring TS_Par3)
   { ... }
```

ПРИМЕЧАНИЕ. – Это обеспечивает функциональную возможность, аналогичную параметрам тестовой последовательности TTCN-2, которые обеспечивают значения PICS и PIXIT для этой тестовой последовательности.

7.2.1 Безусловные значения параметров модуля

Для случаев, когда реальные значения параметров модуля не выдаются тестовой средой во время выполнения тестов, для параметров модуля разрешается определять безусловные (по умолчанию) значения. Это делается путем присвоения в списке параметров модуля. Например:

```
module MyModuleDefaultParameter (integer Par1 : = 1234, boolean Par2 : = false)
{ . . . }
```

7.3 Определяющая часть модуля

Определяющая часть модуля дает определения верхнего уровня для модуля. Эти определения могут использоваться в любом другом месте в этом модуле, включая управляющую часть. 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 или функциях.

ПРИМЕЧАНИЕ. – TTCN-3 не поддерживает объявление переменных в определяющей части модуля, а поддерживает их объявление только в управляющей части. Это означает, что глобальные переменные не могут быть определены в TTCN-3.

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 тестовую последовательность, если это требуется. Это означает, что все идентификаторы объявлений в наборе групп (включая вложенные группы) на любом заданном уровне группирования должны быть уникальными. Другими словами, группы и вложенные группы не имеют ограничений, *за исключением* ограничения на контекст любых атрибутов, приданных этой группе с помощью соответствующей команды **with**. В таких случаях команда **with** из внешней группы отменяется командой **with** из внутренней группы.

Пример:

```
// Совокупность определений
group MyGroup
{
   const integer MyConst:= 1;
   :
   type record MyMessageType { ... }
   }
   // Группа шагов теста
   group MyTestStepLibrary
   {
      group MyGroup1
      {
         function MyTestStep11() { ... }
         function MyTestStep12() { ... }
         :
         function MyTestStep1n() { ... }
      }
      group MyGroup2
      {
         function MyTestStep21() { ... }
         function MyTestStep22() { ... }
         :
         function MyTestStep2n() { ... }
      }
   }
```

7.4 Управляющая часть модуля

Управляющая часть модуля 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; // Местная управляющая переменная
      :
      MyTestCase1(); // Последовательное выполнение тестовых примеров
      MyTestCase2();
      :
   }
}
```

7.5 Импортирование из модулей

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.

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 all from MyModuleB;  // Контекст импортированных определений
                               // является глобальным для MyModuleA
   type record MyMessageType { ... }
   :
   function MyBehaviourC()
   {
      const integer MyConstant := 2;
      // здесь не может использоваться импорт
      :
   }
   :
   control
   {  // Здесь не может использоваться импорт
      :
   }
}
```

7.5.1 Правила использования импорта

При использовании импорта должны применяться следующие правила:

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

b) по умолчанию все определения, зависимые от других определений (например, типов **record**), импортируются вместе со всеми определениями, от которых они зависят. Если желательно не импортировать такие зависимые определения, то может использоваться директива **nonrecursive**;

c) кроме того, могут импортироваться группы определений. Однако группы используются только для целей структурирования и не имеют единиц контекста. Следовательно, разрешается импортировать подгруппы, то есть группы, которые определены внутри другой группы.

7.5.2 Импортирование одиночных определений

Могут быть импортированы одиночные определения. Например:

```
import type MyType from MyModuleC;
```

7.5.3 Импортирование всех определений модуля

Может быть импортировано полное содержимое определяющей части модуля (но не сам реальный модуль), например:

```
import all from MyModule;
```

7.5.4 Импортирование групп

Могут быть импортированы группы. Например:

```
import group MyGroup from MyModule;
```

Подгруппы, то есть группы, определенные внутри другой группы, также импортируются с помощью этой команды.

7.5.5 Импортирование определений одного и того же вида

Могут быть импортированы блоки из определений одного и того же вида. Например:

```
import all template from MyModule;
```

7.5.6 Рекурсивный импорт сложных определений

По умолчанию с помощью оператора **import** неявно импортируются рекурсивные определения, то есть определения, которые ссылаются на другие определения. Примерами рекурсивных определений являются типы **record** вместе с их компонентными типами или функции, вызывающие другие функции. Например:

```
import type MyType from MyModuleC;
```

Все неявно импортированные определения видны на самом верхнем уровне контекста и могут использоваться после команды **import**.

ПРИМЕЧАНИЕ. – Местные определения среди окружающих определений, например, объявления местных констант внутри функции, никогда не будут видны.

Пример:

```
// Задано
module MyModuleA
{  :
   function MyBehaviourB() { ... }
   function MyBehaviourA()
   {  :
      MyBehaviourB();
      :
      const integer LocalConst:= 1000;
      :
   }
}

// Тогда
module MyModuleB
{  :
   import function MyBehaviourA from MyModuleA;
   :
}
// Будет также импортирован и будет виден MyBehaviourB. Константа LocalConst будет
// все еще встроена в MyBehaviourA и не будет видна (за пределами
// MyBehaviourA).
```

Если определения, импортированные из одного модуля, зависят от определений из другого модуля, то определения из этого другого модуля также импортируются; то есть импорт неявно импортирует зависимые определения из модуля третьей стороны. Это следует из правила, что с импортированным определением обращаются так же, как и с определением, которое определено в самом модуле.

Если желательно запретить рекурсивные импорты, то используется директива **nonrecursive**. Например:

```
import type MyType from MyModuleC nonrecursive;
```

7.5.7 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, импорта групп или рекурсивных определений. 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 присутствовать префикс.

Пример:

```
module MyModuleA
{  :
   type bitstring MyTypeA;
   import type MyTypeA from SomeModuleC; //     // Когда MyTypeA относится к типу
                                               // "цепочка знаков"
   import type MyTypeB from SomeModuleC; //     // Когда MyTypeA относится к типу
                                               // "цепочка знаков"
   :
   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.

7.5.8 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. В таких случаях это определение импортируется 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.

7.5.9 Импорт и параметры модуля

Если импортируемое определение использует какой-либо параметр модуля, то этот параметр также включается в список параметров модуля в *импортирующем* модуле.

7.5.10 Импорт определений из модулей не-TTCN

Ключевое слово **language** используется для обозначения случаев, когда определения типов импортируются из модулей не-TTCN. Например:

```
Import type MyASN1Type from MyASN1Module language "ASN.1 : 1997";
```

По умолчанию языком является TTCN-3. Например:

```
import type MyType from MyModule;
// это то же, что и
import type MyType from MyModule language "TTCN-3";
```

8 Тестовые конфигурации

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 систем. (См. рис. 2.)

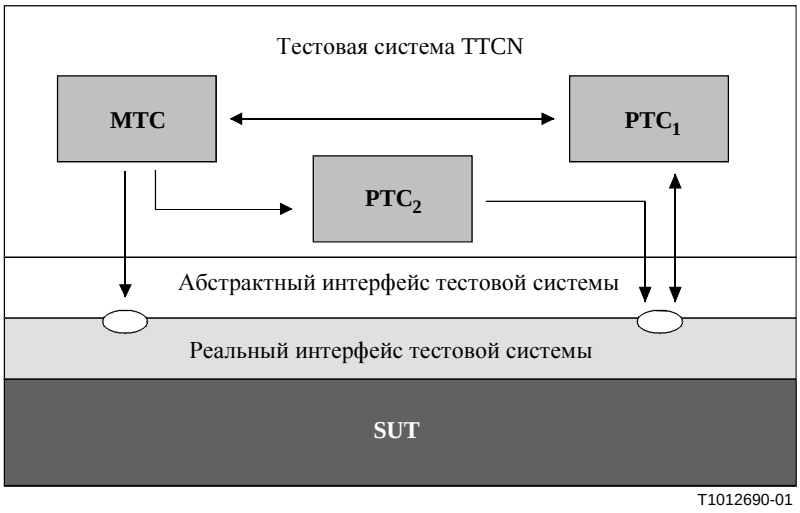


Рисунок 2/Z.140 – Схематическое представление типичной тестовой конфигурации TTCN-3

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 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.

Связь между компонентами внутри тестовой системы, а также между этими компонентами и интерфейсом тестовой системы осуществляется через порты связи.

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. 21.2).

8.1 Модель связи портов

Тестовые компоненты могут быть соединены с другими компонентами и с интерфейсом тестовой системы. Ограничения на число соединений, которое может иметь компонент, отсутствуют, но компонент не должен соединяться с самим собой. Разрешены соединения "один со многими".

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. 24.2.1).

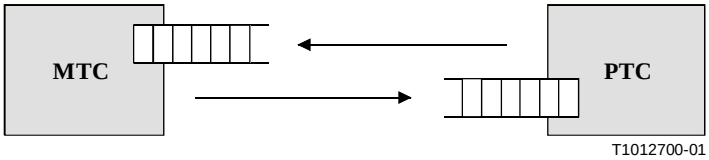


Рисунок 3/Z.140 – Модель портов связи TTCN-3

8.2 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.

8.3 Определение типов портов связи

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 смешанный. Это должно указываться ключевым словом **`message`** или **`procedure`** внутри определения типа связанного порта.

Порты имеют направления. 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. Например:

```
// 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.

При использовании ключевого слова **all** в одном из списков, связанных с типом порта, всем типам и/или всем процедурным сигнатурам, определенным в модуле, будет разрешено прохождение через этот порт связи.

Например:

```
// Порт на базе сообщений, который разрешает переносить через него любое значение
// всех предопределенных типов и определяемых пользователем типов в обоих направлениях
type port MyAllMesssagesPortType message
{
   inout all
}
```

8.3.1 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 этом порту могут использоваться
// все типы и все сигнатуры, определенные в модуле, для
// связи либо с SUT, либо с другими тестовыми компонентами
type port MyAllMixedPortType mixed
{
  inout  all
}
```

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 21), то есть **start, stop** и **clear**, должны выполнять операции в обеих очередях (в произвольном порядке), если они вызваны идентификатором смешанного порта.

8.4 Определение типов компонентов

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. Однако это не долджно пониматься так, что между компонентами имеется какое-либо соединение через эти порты.

Пример:

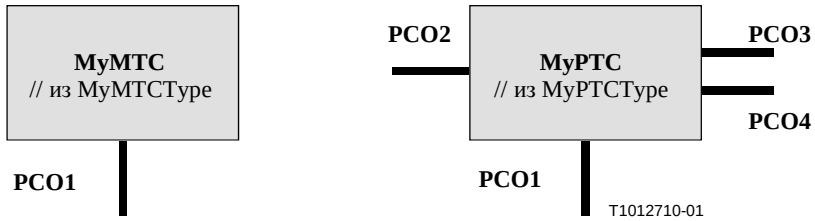


Рисунок 4/Z.140 – Типичные компоненты

```
type component MyMTCType
{
   port MyMessagePortType       PCO1
}

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

8.4.1 Объявление местных переменных и таймеров в компоненте

Возможно объявлять переменные и таймеры, местные для конкретного компонента. Например:

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

Эти объявления видны для всех функций, которые выполняются в данном компоненте. Это 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. 5.4. Поэтому каждый новый экземпляр компонента будет иметь свой собственный набор переменных и таймеров, указанный в определении компонента (включая любые начальные значения, если они устанавливаются).

8.4.2 Определение компонентов с множеством портов

Возможно определять множества портов в определениях типов компонентов (см. также п. 21.9). Например:

```
type component My3pcoCompType
{
   port MyMessageInterfaceType PCO[3]
   // Определяет тип компонента, который имеет множетсво из 3 портов.
}
```

8.5 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. 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;
```

8.6 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. 21.1). Дополнительно компонентные ссылки выдаются стандартными функциями **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 21) для установления тестовых конфигураций, а также в частях **from, to** и **sender** операций связи для целей адресации (см. раздел 22).

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 двух переменных 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 передатчик должен быть
// выбран. 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 MyCompType1 -> value MyMessage sender MyInst1 {}
   [] PCO1.receive(M1:*) from MyCompType2 -> value MyMessage sender MyInst2 {}
   [] PCO1.receive(M1:*) from MyCompType3 -> 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};
 :
```

8.7 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).

```
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. Порты интерфейса тестовой системы автоматически реализуются вместе с МТС при запуске выполнения тестового примера, то есть когда тестовый пример вызывается из управляющей части модуля.

Операцией, выдающей адрес интерфейса тестовой системы, является **system**. Она может использоваться для адресации портов тестовой системы. Например:

```
map (MyNewComponent : 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 заголовках модулей, в управлении модулем, в тестовых примерах и в функциях. Определения констант обозначаются ключевым словом **const**. Значение константы присваивается в порту объявления. Например:

```
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**. Внешние константы должны сводиться к некоторому значению во время трансляции. Например:

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

Внешняя константа может иметь произвольный тип, но этот тип должен быть известен в модуле, то есть он должен быть базовым типом, определенным в модуле или импортированным из какого-либо другого модуля. Отображение этого типа во внешнее представление какой-либо внешней константы не входит в область рассмотрения настоящей Рекомендации. Механизм введения внешней константы в модуль также выходит за рамки данной Рекомендации.

10 Объявление переменных

Переменные указываются ключевым словом var. Переменные можно объявлять и использовать в управлении модулем, в тестовых примерах и в функциях. Они не объявляются и не используются в заголовке модуля (то есть глобальные переменные не поддерживаются в TTCN-3). Объявление переменной может иметь присвоенное ей факультативное начальное значение. Например:

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

Использование неинициализованных переменных во время выполнения приводит к ошибке тестового примера.

11 Объявление таймеров

Таймеры можно объявлять и использовать в управлении модулем, в тестовых примерах и в функциях. Таймеры не объявляются и не используются в определяющей части модуля. 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 относится к типу **float**, причем основной единицей является секунда. Например:

```
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
```

Для управления таймерами могут использоваться таймерные операции **start**, **stop**, **read** и **timeout** (см. раздел 23). Например:

```
// При использовании MyTimer2 может иметь
MyTimer2.start(10);    // 10 секунд
MyTimer2.start(180);   // 3 минуты
```

11.1 Таймеры в качестве параметров

Таймеры можно переслать с помощью ссылок только в функции и именованные альтернативы. Таймеры, пересланные в функцию или именованную альтернативу, известны внутри определения поведения функции или именованной альтернативы.

Таймер, пересланный в качестве параметра ссылки, может использоваться как любой другой таймер, то есть он не нуждается в объявлении. Запущенный таймер также можно переслать в функцию или именованную альтернативу. Таймер продолжает свою работу, то есть его не останавливают неявно. Таким образом, возможные события, связанные с тайм-аутом, могут быть обработаны внутри функции или именованной альтернативы, 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.

Сложные сообщения могут определяться типами "запись" (см. п. 6.3.1). Например:

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

Сообщения, безусловно, могут подразделяться на отдельные структуры, например:

```
// Информационный элемент типа 1 (IEType1). Аналогичные
// объявления для IEType2 … IETypeN
type record IEType1
{
   IEFieldType1   iefield1,
   IEFieldType2   iefield2,
   :
   IEFieldTypeN   iefieldN
}

// Сообщение, содержащее информационные элементы
type record MyMessageType
{
   IEType1  field1,
   IEType2  field2,
   :
   IETypeN field3
}
```

12.1 Факультативные поля сообщения

По умолчанию все поля в сообщении являются обязательными. Факультативные поля сообщения указываются с помощью ключевого слова **optional**. Например:

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

13 Объявление процедурных сигнатур

Процедурные сигнатуры (или сигнатуры, для краткости) нужны для синхронной связи. Процедура может быть вызвана либо в SUT (то есть этот запрос выполняет тестовая система), либо в тестовой системе (то есть запрос выполняет SUT).

Как для процедур, вызываемых из SUT, так и для процедур, вызываемых из тестовой системы, полная процедура **signature** определяется в модуле TTCN-3.

Внутри определения **signature** список параметров может содержать идентификаторы параметров, типы параметров и их направление (то есть **in**, **out** или **inout**). Отметим, что направлением параметров считается то, которое видно *вызываемой* стороне, а не *вызывающей* стороне. Например:

```
signature MyRemoteProc (in integer Par1, out float Par2, inout integer Par3)
   return integer;
// Это определяет удаленную процедуру MyRemoteProc. MyRemoteProc выдает
// целочисленное значение и имеет три параметра: один параметр in типа
// integer, один параметр out типа float и один параметр inout типа integer
```

Процедура **call** приведет на вызываемой стороне либо к выполнению операции **reply** (нормальный случай), либо к порождению особого состояния. Действия в результате принятого вызова процедуры определяются принимающей стороной (см. раздел 22).

13.1 Исключение реальных параметров

Разрешается исключать реальные параметры из списка реальных параметров сигнатуры. Это указывается путем замены исключаемого реального параметра в занимаемой им надлежащей позиции на ключевое слово **omit** (опустить). Например:

```
ParameterList (Par1, omit, Par3)    // Par2 исключен
```

ПРИМЕЧАНИЕ. – Это часто необходимо, когда процедурные сигнатуры используются в синхронной связи.

13.2 Определение особых состояний

Особые состояния представляются в TTCN-3 в виде значений специфического типа, причем могут использоваться даже шаблоны и механизмы сопоставления.

ПРИМЕЧАНИЕ. – Преобразование особых состояний, генерируемых в SUT, в соответствующий тип зависит от применяемых инструментов и от системы, поэтому в данной Рекомендации оно не рассматривается.

Особые состояния определяются в виде списка особых состояний, 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 путем представления с помощью специфических значений этих типов).

Пример:

```
signature MyRemoteProc (in integer Par1, out float Par2, inout integer Par3)
   return integer exception(ExceptionType1, ExceptionType2);

// Вызов MyRemoteProc может порождать особые состояния типа ExceptionType1
// или особые состояния ExceptionType2
```

14 Объявление шаблонов

Шаблоны используются либо для передачи набора отдельных значений, либо для проверки, согласуются ли принятые значения со спецификацией шаблона.

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.

14.1 Объявление шаблонов для сообщений

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, кроме специальных типов (**port, component, address**).

```
// 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);
```

Однако, забегая вперед, отметим, что наиболее общим будет использование с записями, как показано на примерах в последующих разделах.

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.

Пример:

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

```
// Шаблоном сообщения может быть
template MyMessageType MyTemplate:=
{
   field1 := 1,
   field2 := "My string",
   field3 := true
}
// а соответствующей операцией передачи может быть
MyPCO.send(MyTemplate);
```

ПРИМЕЧАНИЕ. – Шаблоны могут использоваться также для особых состояний, если определен соответствующий тип.

14.1.2 Шаблоны для принимаемых сообщений

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

Пример:

```
// Задано определение сообщения
type record MyMessageType
{
   integer    field1,
   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

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.

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 процедур вызова

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 С).

Пример:

```
// 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 параметр Par3 имеет атрибут сопоставления,
// а не какое-либо значение
MyPCO.call(RemoteProc:Template3);

// Шаблоны никогда не выдают значений. В случае Par2 и Par3 значения, выдаваемые
// вызовом, должны быть найдены с использованием раздела присвоений в конце
// команды вызова
```

14.2.2 Шаблоны для приема запросов 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 С. Никакого связывания входящих значений с шаблоном не происходит. Любые параметры **in** в процессе сопоставления игнорируются.

Пример:

```
// Действительный getcall, он будет соответствовать, если Par2 == 2 и Par 3 == 3
MyPCO.getcall(RemoteProc:Template1);

// Действительный getcall, он будет соответствовать, если Par3 == 3, а Par 2 имеет
// любое значение
MyPCO.getcall(RemoteProc:Template2);

// Действительный getcall, он будет соответствовать при любых значениях Par3 и Par2
MyPCO.getcall(RemoteProc:Template3);
```

14.3 Механизмы сопоставления шаблона

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

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

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

Идентификатор типа является факультативным. Например:

```
MyPCO.receive ("abcxyz");
```

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

Механизмы сопоставления разделяются на четыре группы:

a) специфичные значения (то есть выражение, представляющее конкретное значение);

b) специальные символы, которые могут быть использованы *вместо* значений:

- (…) список значений;
- **complement** (…): дополнение к списку значений;
- omit: значение пропущено;
- ? : универсальный символ для любого значения;
- * : универсальный символ для любого значения или для отсутствия значения вообще (то есть пропущенного значения);
- (lower **to** upper): диапазон целочисленных значений от нижней границы до верхней границы включительно;

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);

d) специальные символы, которые описывают *атрибуты* значений:

- **length**: ограничения длины для цепочек 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 5. В левом столбце данной таблицы перечислены все эквивалентные типы TTCN-3 и ASN.1, определенные в Рекомендациях МСЭ-Т серии X.680 [7], [8], [9] и [10], к которым применяются эти механизмы сопоставления. Полное описание каждого механизма сопоставления можно найти в Приложении C.

Таблица 5/Z.140 – Механизмы сопоставления TTCN-3

Используется для значений типа	**Зна-чение**	**Вместо 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					Да
integer	Да	Да	Да	Да	Да	Да	Да				Да
char	Да	Да	Да	Да	Да	Да	Да				Да
universal char	Да	Да	Да	Да	Да	Да	Да				Да

Таблица 5/Z.140 – Механизмы сопоставления TTCN-3

Используется для значений типа	Зна-чение	Вместо 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
float	Да	Да	Да	Да	Да	Да					Да
bitstring	Да	Да	Да	Да	Да	Да		Да	Да	Да	Да
octetstring	Да	Да	Да	Да	Да	Да		Да	Да	Да	Да
hexstring	Да	Да	Да	Да	Да	Да		Да	Да	Да	Да
character strings	Да	Да	Да	Да	Да	Да		Да	Да	Да	Да
record	Да	Да	Да	Да	Да	Да					Да
record of	Да	Да	Да	Да	Да	Да		Да	Да	Да	Да
array	Да	Да	Да	Да	Да	Да		Да	Да	Да	Да
set	Да	Да	Да	Да	Да	Да					Да
set of	Да	Да	Да	Да	Да	Да		Да	Да	Да	Да
enumerated	Да	Да	Да	Да	Да	Да					Да
union	Да	Да	Да	Да	Да	Да					Да

14.4 Параметризация шаблонов

Шаблоны для операций как передачи, так и приема могут быть параметризованы. Формальные параметры шаблона могут содержать шаблоны, функции и специальные символы сопоставления. Должны соблюдаться правила для списков формальных и реальных параметров, определенные в п. 5.3.

Пример:

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

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

14.4.1 Параметризация с атрибутами сопоставления

Чтобы дать возможность пересылать атрибуты сопоставления в виде параметров, перед полем типа добавляется специальное ключевое слово template. Это образует параметр шаблона и в результате расширяет разрешенные параметры для соответствующего типа с целью включения подходящего набора атрибутов сопоставления (см. Приложение C) вместе с нормальным набором значений. Поля параметров шаблона не должны вызываться с помощью ссылок.

Пример:

```
// Шаблон
template MyMessageType MyTemplate (template integer MyFormalParam):=
{  field1 := MyFormalParam,
   field2 := pattern "abc*xyz",
   field3 := true
}
// может использоваться следующим образом
pco1.receive(MyTemplate(?));
// либо, если field1 определено в виде факультативного
pco1.receive(MyTemplate(omit));
```

14.5 Пересылка шаблонов в виде параметров

Только определения **function, testcase, named alt** и **template** могут иметь шаблоны в качестве формальных параметров.

Пример:

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

14.6 Модифицированные шаблоны

Обычно шаблон определяет набор базовых, или безусловных (по умолчанию), значений либо символов сопоставления для каждого поля, указанного в соответствующем определении. 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.

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

Пример:

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

// затем записывается
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
   field2 := "A modified string",
   field3 := omit   // field3 должно быть указано как факультативное
                    // в соответствующем типе record
}
```

```
// это то же, что и запись
template MyRecordType MyTemplate2 :=
{
   field1 := 123,
   field2 := "A modified string",
   field3 := omit
}
```

14.6.1 Параметризация модифицированных шаблонов

Если базовый шаблон является списком 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 параметры не должны пропускаться; однако такой шаблон может иметь дополнительные (добавленные) параметры, если они желательны;

b) список формальных параметров в каждом модифицированном шаблоне должен следовать за именем шаблона;

c) параметризованные шаблоны в полях шаблонов не должны быть модифицированы или явно пропущены в модифицированном шаблоне.

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
   field2 := "A modified string",
   field3 := omit   // field3 должно быть указано как факультативное
                    // в соответствующем типе record
}
```

14.6.2 Шаблоны, модифицированные "инлайн"

Помимо создания явно именованных модифицированных ограничений TTCN-3 позволяет определять "инлайновые" модифицированные ограничения.

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

Все изменения полей шаблона должны производиться только с помощью параметризации или с помощью "инлайновых" выведенных шаблонов во время выполнения операций связи (например, **send, receive, call, getcall** и т. п.). Действие этих изменений в значении поля шаблона не распространяется на последовательность шаблонов для соответствующего сеанса связи.

Нотация вида *MyTemplateId.Fieldid* не должна использоваться для установки или запроса значений в шаблонах при событиях связи. Для этой цели используется символ "–>" (см. раздел 22).

14.8 Операция сопоставления

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

```
template integer LessThan10 := (1..10);

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

   if(match(RxValue, LessThan10)) { ... }
   ...
}
```

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.

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

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

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

15 Операторы

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 6.

Таблица 6/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 7. 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 7/Z.140 – Старшинство операторов

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

15.1 Арифметические операторы

Арифметические операторы представляют операции сложения, вычитания, умножения, деления и определения модуля. Операнды этих операторов должны быть типа **integer** (включая производные **от integer**) или **float** (включая производные от **float**), за исключением **mod**, 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 плюс (+) или минус (–) используется в качестве унарного оператора, 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 целое значение **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|          если  x >= 0
         = 0                  если  x < 0 и  x rem |y| = 0
         = y + x rem |y|      если  x < 0 и  x rem |y| < 0
```

Таблица 8 иллюстрирует различие между операторами **mod** и **rem**:

Таблица 8/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 Операторы цепочки

Предопределенные операторы цепочки выполняют сцепление (конкатенацию) типов "цепочка" (строка). Операндами могут быть любые совместимые значения типа "цепочка". Операцией является простое сцепление слева направо. Никакие формы арифметического сложения не подразумеваются. Типом результата является совместимый тип "цепочка". Например:

```
'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 быть произвольного типа. Все остальные операторы отношения должны иметь операнды только типа **integer** (включая производные от **integer**) или **float** (включая производные от **float**). Во всех случаях два операнда должны быть совместимого типа. Типом результата этих операций является **boolean**.

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 операторов 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**.

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**, **octetstring**. Тип результата побитовых операторов должен быть таким же, как у 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:

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

Побитовый оператор **and4b** принимает два операнда. 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:

```
'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. 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:

```
'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. 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 0. 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:

```
'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**, **octetstring** или **integer**. 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) **bitstream** или **integer**, то применяемой единицей сдвига является 1 бит;

b) **hexstrin**, 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.

ПРИМЕЧАНИЕ 1. – Если левый операнд имеет тип **integer**, то для каждого бита, сдвинутого влево, это эквивалентно умножению левого операнда на два.

ПРИМЕЧАНИЕ 2. – Если применение операции сдвига влево к левому операнду привело к перегрузке, зависящей от системы, то присваивается вердикт "ошибка".

Пример:

```
'111001'B  << 2 дает  '100100'B
'12345'H   << 2 дает  '34500'H
'1122334455'O  << (1+1)  дает  '3344550000'O
32  << 2  дает  128
–32 << 2  дает  –128
```

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.

ПРИМЕЧАНИЕ 3. – Если левый операнд имеет тип **integer**, то для каждого бита, сдвинутого вправо, это эквивалентно целочисленному делению левого операнда на два (2).

ПРИМЕЧАНИЕ 4. – Когда левый операнд имеет тип **integer**, а его значение отрицательно, при выполнении сдвига вправо должен сохраняться бит знака.

Пример:

```
'111001'B  >> 2 дает  '001110'B
'12345'H   >> 2 дает  '00123'H
'1122334455'O  >> (1+1)  дает  '0000112233'O
32  >> 2 дает  8
-32 >> 2 дает  –8
```

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) **bitstream**, 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.

Пример:

```
'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:

```
'100001'B  @> 2 дает  '0110001'B
'12345'H   @> 2 дает  '45123'H
'1122334455'O  @> (1+2)  дает  '3344551122'O
"abcdefg"  @> 3 дает  "efgabcd"
```

16 Функции

Функции используются в TTCN-3 для выражения поведения теста 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. Это обозначается ключевым словом **return**, за которым следует идентификатор типа. Если не указано ключевое слово **return**, то функция является пустой. В TTCN-3 отсутствует явное ключевое слово для обозначения такой пустоты. Ключевое слово **return**, когда оно использовано в теле функции, побуждает эту функцию определить и выдать значение, совместимое с типом выдаваемого результата. Например:

```
// Определение функции MyFunction, не имеющей параметров
function MyFunction() return integer
{

   return 7;// Когда функция заканчивается, выдается целочисленное значение 7
}
```

ПРИМЕЧАНИЕ. – Функции TTCN-3 заменяют операции тестовых последовательностей и процедурные определения тестовых последовательностей из TTCN-2. Могут объявляться неформальные функции в виде внешних функций с поясняющими комментариями либо с помощью пустой формальной функции с комментариями.

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.

```
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 содержит портовые операции, то тип связанного компонента указывается с помощью ключевого слова **runs on** в заголовке функции, чтобы определить число, тип и идентификаторы доступных портов. Единственным исключением из этого правила будет случай, когда все порты, используемые внутри функции, пересылаются в виде параметров.

Если функция содержит портовые операции, то либо все порты, используемые внутри функции, должны пересылаться в виде параметров, либо тип связанного компонента должен указываться с помощью **runs on** в заголовке функции, чтобы определить число, тип и идентификаторы доступных портов. Например:

```
function MyFunction() runs on MyComponent return integer
{
   :
}
```

Экземпляры разных типов компонентов могут использовать одну и ту же функцию, если они удовлетворяют следующему правилу:

"Пусть C1 и C2 будут двумя типами компонентов, а FUNC – функцией, которая указывает C1 в ее разделе runs on. *Экземпляр компонента типа C2 может использовать FUNC, если определение типа C2 содержит полное описание типа C1. Это означает, что C2 содержит те же имена для адресации портов того же типа, что и C1."*

16.1 Параметризация функций

Функции могут быть параметризованы. Должны выполняться правила для списков формальных параметров, определенные в п. 5.3. Например:

```
function MyFunction2(inout integer MyPar1)
{
                            // MyFunction 2 не выдает никакого значения,
   MyPar1 := 10 * MyPar1; // но изменяет значение MyPar1, которое
                            // передается по ссылке
}

function MyFunction3() runs on MyPTCType
{
                            // MyFunction3 не выдает никакого значения,
var integer MyVar := 5;    // но использует портовую операцию send и,
   PCO1.send(MyVar);       // следовательно, нуждается в разделе runs on,
                            // чтобы привести идентификаторы портов
                            // с помощью указания некоторого типа компонента
)
```

16.2 Вызов функций

Функция вызывается путем указания ее имени, а также с помощью списка реальных параметров. Функции, которые не выдают значений, могут вызываться прямо. Функции, которые выдают значения, могут вызываться внутри выражений. Должны выполняться правила для списков реальных параметров, определенные в п. 5.3.

```
MyVar := MyFunction4();      // Значение, выданное функцией MyFunction4, присвоено
                             // MyVar. Типы выданного значения и MyVar
                             // должны быть одинаковыми

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 **start**. Эти ограничения описываются в п. 21.5.

16.3 Предопределенные функции

TTCN-3 содержит ряд предопределенных (встроенных) функций (см. таблицу 9), которые не требуется объявлять перед использованием.

Таблица 9/Z.140 – Список предопределенных функций TTCN-3

Категория	Функция	Ключевое слово
Функции преобразования	Преобразовать значение **integer** в значение **char**	**int2char**
	Преобразовать значение **char** в значение **integer**	**char2int**
	Преобразовать значение **integer** в значение **universal char**	**int2unichar**
	Преобразовать значение **universal char** в значение **integer**	**unichar2int**
	Преобразовать значение **bitstring** в значение **integer**	**bit2int**
	Преобразовать значение **hexstring** в значение **integer**	**hex2int**
	Преобразовать значение **octetstring** в значение **integer**	**oct2int**
	Преобразовать значение **charstring** в значение **integer**	**str2int**
	Преобразовать значение **integer** в значение **bitstring**	**int2bit**
	Преобразовать значение **integer** в значение **hexstring**	**int2hex**
	Преобразовать значение **integer** в значение **octet**	**int2oct**
	Преобразовать значение **integer** в значение **charstring**	**int2str**
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**
Функции присутствия/ 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**

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 параметров, должны быть ограниченными.

Полное описание предопределенных функций дается в Приложении D.

17 Тестовые примеры

Тестовые примеры являются специальным видом функций. Их выполнение в управляющей части модуля связано с командой **execute** (см. п. 26.1). 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, конкретизируются порты в интерфейсе тестовой системы, создается MTC 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 определении 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, запросившей
      // этот тестовый пример
}

// либо тестовый пример, в котором конкретизируется только МТС
testcase MyTestCaseTwo() runs on MyMtcType2
{
   :  // Поведение, определенное здесь, выполняется в МТС, запросившей
      // этот тестовый пример
}
```

18 Программные команды и операции

Основными программными элементами управляющей части модулей и функций TTCN-3 являются базовые программные команды, такие как выражения, присвоения, циклические конструкции и др., команды поведения, такие как параллельное 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 команд являются одним из механизмов для групповых команд. Блоки команд могут использоваться в разных единицах контекста, то есть в управлении модулем, в функциях и поведении тестов. Вид команд, которые могут использоваться в блоке, будет зависеть от единицы контекста, в которой этот блок используется. Например, в блоке команд, появившемся в какой-либо функции, должны использоваться только те программные команды, которые могут использоваться в функциях.

Общие контекстные правила описаны в п. 5.4.

Блок команд синтаксически эквивалентен одиночной команде; следовательно, везде в функции, где разрешена команда, может появиться блок. Это означает, что блоки могут быть вложенными. Объявления, если они имеются, делаются в начале блока. Эти объявления видны только внутри этого блока и для вложенных субблоков.

Команды в блоке должны выполняться в порядке их появления. Разрешается определение пустого блока команд, то есть { }. Пустой блок команд означает, что никакие действия не выполняются.

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
Базовые программные команды			
Выражения	`( . . . )`	Да	Да
Присвоения	`: =`	Да	Да
Регистрация	`log`	Да	Да
Label и Goto (Метка и перейти к)	`label/goto`	Да	Да
Если - еще If-else (-)	`if ( . . . ) { . . . }` `else { . . . }`	Да	Да
Для цикла	`for ( . . . ) { . . . }`	Да	Да
Во время цикла	`while ( . . . ) { . . . }`	Да	Да
Делать во время цикла	`do { . . . }` `while ( . . . )`	Да	Да
Остановить выполнение	`stop`	Да	Да
Программные команды поведения			
Альтернативное поведение	`alt { . . . }`	Да (примечание 1)	Да
Именованная альтернатива	`named alt { . . . }`	Да (примечание 1)	Да
Перемежающееся поведение	`interleave { . . . }`	Да (примечание 1)	Да
Активировать элемент по умолчанию	`activate`	Да (примечание 1)	Да
Деактивировать элемент по умолчанию	`deactivate`	Да (примечание 1)	Да
Управление выдаваемыми результатами	`return`		Да
Операции конфигурации			
Создать параллельный тестовый компонент	`create`		Да
Соединить компонент с другим компонентом	`connect`		Да
Разъединить два компонента	`disconnect`		Да
Отобразить порт в тестовый интерфейс	`map`		Да
Убрать отображение порта из интерфейса тестовой системы	`unmap`		Да
Взять адрес MTC	`mtc`		Да
Взять адрес интерфейса тестовой системы	`system`		Да
Взять собственный адрес	`self`		Да
Запустить выполнение тестового компонента	`start`		Да
Остановить выполнение тестового компонента	`stop`		Да
Проверить окончание PTC	`running`		Да
Ожидать окончания PTC	`done`		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
Разъединить порт	`clear`		Да
Разъединить и дать доступ к порту	`start`		Да
Прекратить доступ к порту (принимающему и передающему)	`stop`		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	`verdict.set`		Да
Взять местный вердикт	`verdict.get`		Да
Операции SUT			
Удаленные операции, выполняемые SUT	`sut.action`		Да
Выполнение тестовых примеров			
Выполнить тестовый пример	`execute`	Да	Да (примечание 2)
ПРИМЕЧАНИЕ 1. – Может использоваться только при управлении с таймерными операциями. ПРИМЕЧАНИЕ 2. – Может использоваться только в функциях и именованных альтернативах, которые используются в управлении модулем.			

19 Базовые программные команды

Базовыми программными командами являются выражения, присвоения, операции, циклические конструкции и др. Все базовые программные команды могут использоваться в управляющей части модуля и в функциях TTCN-3.

Таблица 11/Z.140 – Обзор базовых программных команд TTCN-3

Базовые программные команды	
Команда	**Связанное ключевое слово или символ**
Выражения	`( . . . )`
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`**

19.1 Выражения

TTCN-3 позволяет определять выражения с помощью операторов, указанных в разделе 15. Выражения создаются из других (простых) выражений. Выражения могут содержать функции. Результатом выражения должно быть значение определенного типа, причем используемые операторы должны быть совместимы с типом операндов. Например:

```
(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). Например:

```
((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 элемент того же типа, что и указанный в левой стороне. Действием присвоения будет сцепление переменной (которая может быть также элементом в **`record`**, **`set`** и т. п.) со значением выражения. Присвоение не должно содержать неограниченных переменных. Все присвоения происходят в порядке, в котором они появляются, то есть они обрабатываются слева направо. Например:

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

19.3 Команда Log

Команда **`log`** обеспечивает средства для записи цепочки знаков в каком-либо регистрирующем устройстве, связанном с управлением тестом или тестовым компонентом, в котором эта команда используется. Например:

```
log ("Line 248 in PTC_A");
// Цепочка "Line 248 in PTC_A" записывается в некоторое
// регистрирующее устройство тестовой системы
```

ПРИМЕЧАНИЕ. – В настоящей Рекомендации не рассматриваются комплексные возможности регистрации и трассировки, которые могут зависеть от инструментальных средств.

19.4 Команда Label

Команда **label** позволяет определять метки в тестовых примерах, функциях, именованных альтернативах и в управляющей части модуля. Команда **label** может использоваться свободно, подобно другим программным командам поведения TTCN-3, в соответствии с синтаксическими правилами, определенными в Приложении A. Она может использоваться до или после любой команды TTCN-3, но, например, не в качестве первой команды в альтернативе команды **alt** или **interleave** (см. п. 20.2.7).

19.5 Команда Goto

Команда **goto** может использоваться в функциях, тестовых примерах, именованных альтернативах и в управляющей части модуля TTCN. Команда **goto** выполняет переход к **label** или к началу команды **alt** для того, чтобы перейти к повторному поведению (см. п. 20.2.8).

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 $\texttt{statementblock}_x$ указывает на блок команд.

Пример:

```
if (date == "1.1.2000") 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. Например:

```
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 командой, которая тотчас повторяет цикл **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. Например:

```
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**. Например:

```
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. Например:

```
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, она останавливает функционирование всего модуля. Когда она используется в функции, которая выполняет режим, она останавливает соответствующий тестовый компонент.

20 Программные команды поведения

Программные команды поведения могут использоваться в тестовых примерах, функциях и управлении модулем, за исключением команды **return**, которая используется только в тестовых примерах и функциях. 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 12/Z.140 – Обзор программных команд поведения TTCN-3

Программные команды поведения	
Команда	**Связанное ключевое слово или символ**
Альтернативное поведение	**alt** { . . . }
Именованная альтернатива	**named alt** { . . . }
Перемежающееся поведение	**interleave** { . . . }
Активировать элемент по умолчанию	**activate**
Деактивировать элемент по умолчанию	**deactivate**
Управление выдаваемыми результатами	**return**

20.1 Последовательное поведение

Простейшей формой поведения является набор команд, которые выполняются последовательно, как показано на рис. 5:

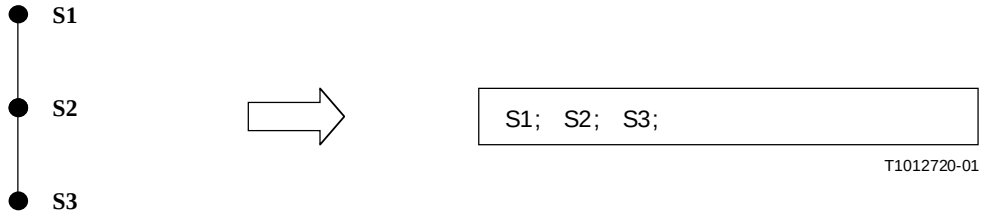


Рисунок 5/Z.140 – Иллюстрация последовательного поведения

Отдельные команды в последовательности отделяются разделителем "**;**". Например:

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

20.2 Альтернативное поведение

При более сложной форме поведения последовательности команд выражаются в виде наборов возможных альтернатив, образующих дерево путей выполнения, как показано на рис. 6:

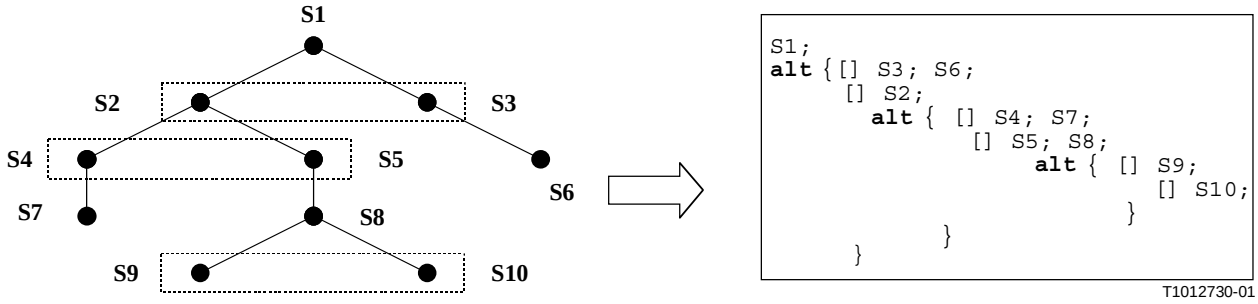


Рисунок 6/Z.140 – Иллюстрация альтернативного поведения

Команда **alt** обозначает разветвление 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** и **done**. Команда **alt** обозначает набор возможных событий, которые должны быть сопоставлены с конкретным "снимком" (стоп-кадром) (см. п. 20.2.1).

ПРИМЕЧАНИЕ. – Команда **alt** соответствует альтернативам на том же уровне структурирования в TTCN-2. Однако имеются три значительных различия:

a) выражения **boolean** для запрещения альтернатив могут применяться только в альтернативной команде;

b) невозможно проверить очередь в порту с помощью булева выражения и затем запретить альтернативу;

c) невозможно вызвать функцию в качестве альтернативы в команде **alt**, за исключением случая, когда последним выбором в этой альтернативе является защита **else** (то есть [**else**]) (см. п. 20.2.3).

Пример:

```
// Использование вложенных альтернативных команд
:
alt
{
[] L1.receive(DL_REL_CO:*)                  // Принят кадр UA или DM; уровень 2 освобожден
   {  verdict.set(pass);
      TAC.stop;
      TNOAC.start;
      alt {
      [] L1.receive(DL_EST_IN)              // Принят SABME
         {  TNOAC.stop;
            verdict.set(pass);
         }
      [] TNOAC.timeout
         {  L1.send(DEL_EST_RQ:*);
            TAC.start;
            alt {
            [] L1.receive(DL_EST_CO:*)      // Принят UA; установлено звено данных
               {  TAC.stop;
                  verdict.set(pass)
               }
            [] TAC.timeout                  // Нет ответа
               {verdict.set(inconc)}
            [] L1.receive                   // Как OTHERWISE в TTCN-2
               {verdict.set(inconc)}
            }
         }
      [] L1.receive                         // Как OTHERWISE в TTCN-2
         {verdict.set(inconc)}
      }
   }
[] TAC.timeout                              // Нет ответа
   {verdict.set(inconc)}
[] L1.receive                               // Как OTHERWISE в TTCN-2
   {verdict.set(inconc)}
}
:

// Использование альтернативы с булевыми выражениями (или с защитой)
:
alt {
[]     L1.receive(MyMessage1)
       {verdict.set(fail)}
[x>1]  L2.receive(MyMessage2)               // Булево выражение/защита
       {verdict.set(pass)}
[x<=1] L2.receive(MyMessage3)               // Булево выражение/защита
       {verdict.set(inconc)}
}
:
```

```
// Использование done в альтернативах
:
alt {
   [] MyPTC.done {
         verdict.set(pass)
      }
   [] any port.receive {
         goto alt
      }
}
:
```

20.2.1 Выполнение альтернативного поведения

Альтернативные команды в команде **alt** обрабатываются в порядке их появления. В операционной семантике TTCN-3 (см. Приложение B) предполагается, что состояние любого события не может измениться за время обработки попытки сопоставления одной альтернативы из набора альтернатив. Это означает, что для принимаемых событий и тайм-аутов используется семантика "снимков", то есть каждый раз в пределах набора альтернатив берется "снимок" событий, которые приняты, и таймеров, которые запущены. Только те события, которые указаны в "снимке", могут сопоставляться в следующем цикле с помощью альтернатив.

ПРИМЕЧАНИЕ 1. – Эта семантика точно такая же, как в TTCN-2.

ПРИМЕЧАНИЕ 2. – Синхронные события (например, **call**) блокируют такой цикл до завершения вызова.

20.2.2 Выбор/отмена выбора альтернативы

Когда необходимо, можно разрешать/запрещать какую-либо альтернативу с помощью булева выражения, размещаемого между квадратными скобками '[ ]' в этой альтернативе. Например:

```
[MyVar = = 3] PCO.receive (MyMessage) { }
```

Открывающая и закрывающая квадратные скобки '[' ']' должны располагаться в начале каждой альтернативы, даже если они пустые. Это не только повышает удобочитаемость, но и необходимо для того, чтобы синтаксически различать альтернативы.

20.2.3 Ветвь Else в альтернативах

Когда необходимо, в альтернативной команде можно определять одну ветвь, которая всегда берется, если не мóжет быть взята другая предварительно определенная альтернатива. Если определена ветвь **else**, то все определенные потом альтернативы излишни; то есть они никогда не могут быть достигнуты. Например:

```
:

alt {
[]       L1.receive(MyMessage1)
         {  verdict.set(fail);
            MyComponent.stop
         }
[x>1]    L2.receive(MyMessage2)    // Булево выражение/защита
         {  verdict.set(pass);
            :
         }
[x<=1]   L2.receive(MyMessage3)    // Булево выражение/защита
         {  verdict.set(inconc);
            :
         }
[else]   {  MyErrorHandling();     // Ветвь else
         verdict.set(fail);
         MyComponent.stop;
         }
   }
:
```

Следует отметить, что значения по умолчанию всегда добавляются к концу всех альтернатив. Если ветвь **else** определена, то активированное **default** никогда не будет вводиться.

ПРИМЕЧАНИЕ. – Использовать **else** можно также в именованных альтернативах.

20.2.4 Объявление именованных альтернатив

Альтернативы, которые используются во многих местах, могут быть определены в именованной альтернативе, обозначаемой двумя ключевыми словами **named alt**. Именованные альтернативы определяются глобально в определениях модулей. Когда она запрошена, **named alt** является идентичной конструкции поведения **alt**, за исключением того, что она имеет идентификатор и допускает параметризацию.

Когда она указана, **named alt** имеет такой же эффект, как и подстановка макроса. Она может быть указана в любом месте в определении поведения (режима), когда оно действительно для включения нормальной конструкции **alt**.

Пример:

```
// Определение именованных макроальтернатив
named alt HandlePCO2()
{
   [] PCO2.receive(DL_EST_IN)
      {PCO2.send(DL_EST_CO)}

   [] PCO2.receive(DL_EST_CO) {}
      // ничего не делать
}

// "Инлайновое" использование именованной alt
testcase TC001() runs on MyPTCtype
{
   :
     HandlePCO2();        // Вызвать именованную alt
   :
}

// Которая расширяется до
testcase TC001() runs on MyPTCtype
{
   :
   alt   {
         [] PCO2.receive(DL_EST_IN)
            {PCO2.send(DL_EST_CO)}
         [] PCO2.receive(DL_EST_CO) {}
            // ничего не делать
   }
   :
}
```

20.2.5 Расширение альтернатив именованными альтернативами

В дополнение к прямому инлайновому указанию возможно также явное расширение альтернатив, определенных в конструкции **named alt**, с помощью команды **expand**. Команда **expand** может помещаться в любой позиции внутри команды **alt**; она будет вводить связанные защитные ограничения от **named alt** в этой позиции.

Пример:

```
// Использование именованной alt путем расширения
testcase TC002()runs on MyPTCtype
{
  :
```

```
   alt   {
      [] PCO1.receive(DL_EST_IN)
         {PCO1.send(DL_EST_CO)}
      [] PCO1.receive(DL_EST_CO) {}
         // ничего не делать
      [expand] HandlePCO2()  // Расширить именованные альтернативы alt до
                             // указанной команды alt
      }
}

// Которая расширяется до
testcase TC002()runs on MyPTCtype
{
   :
   alt   {
         [] PCO1.receive(DL_EST_IN)
            {PCO1.send(DL_EST_CO)}
         [] PCO1.receive(DL_EST_CO) {}
            // ничего не делать
         [] PCO2.receive(DL_EST_IN)
            {PCO2.send(DL_EST_CO)}
         [] PCO2.receive(DL_EST_CO) {}
            // ничего не делать
   }
}
```

20.2.6 Параметризация именованных альтернатив

Именованные альтернативы могут быть параметризованы типами, значениями, функциями и шаблонами. Так как именованные альтернативы не являются единицей контекста, определенные формальные параметры просто заменяются заданными реальными параметрами, когда выполняется макрорасширение.

Пример:

```
named alt HandleAnyPCO(MyPortT PCO)
{
   [] PCO.receive(DL_EST_IN)
      {PCO.send(DL_EST_CO)}
   [] PCO.receive(DL_EST_CO) {}
      // ничего не делать
}

testcase TC001() runs on MyPTCtype
{
   HandleAnyPCO(PCO2);
   :
   alt   {
         [expand] HandleAnyPCO(PCO1);
         [expand] HandleAnyPCO(PCO2);
   }
}
```

20.2.7 Команда Label в поведении

Команда **label** позволяет определять метки в тестовых примерах, функциях, именованных альтернативах и управляющей части модуля. Она может использоваться до или после любой команды TTCN-3, но не должна быть первой командой в альтернативе команды **alt** или **interleave**.

Пример:

```
label MyLabel;
// Определяет метку MyLabel

// Метки L1, L2 и L3 определяются в следующем кодовом фрагменте 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);
      goto L1;                   // Перейти к метке L1
   }
}
:
```

20.2.8 Команда Goto в поведении

Команда **goto** может использоваться в функциях, тестовых примерах, именованных альтернативах и управляющей части модуля TTCN. Команда **goto** выполняет переход к **label** или к началу команды **alt**, чтобы вызвать повторяющееся поведение.

Повторное определение команды **alt** может быть обеспечено:

a) либо путем использования **goto** <*LabelId*>, когда соответствующая команда метки (**label**) должна быть помещена непосредственно перед ключевым словом **alt** в реальной альтернативе, к которой должен быть осуществлен переход;

b) либо путем использования **goto alt** внутри команды **alt**, которая должна быть повторно определена. В этом случае ключевое слово **alt** может рассматриваться как неявная метка для команды **alt**, в которой используется **goto**.

20.2.8.1 Ограничение на использование Goto

Команда **goto** обеспечивает возможность свободного перехода, то есть вперед и назад, внутри последовательности команд, перехода из отдельной составной команды (например, цикла **while**) и перехода через несколько уровней из вложенных составных команд (например, вложенных альтернатив). Однако использование команды **goto** ограничивается следующими правилами:

a) Не разрешается выходить из функций, тестовых примеров, именованных альтернатив и управляющей части модуля TTCN, а также входить в них.

b) Не разрешается переходить в последовательность команд, определенную в составной команде (то есть в команде **alt**, цикле **while**, цикле **for**, команде **if-else**, цикле **do-while** и команде **interleave**).

c) В качестве исключения из правила a) для именованных альтернатив разрешается использовать **goto alt** внутри именованной альтернативы, чтобы вынудить повторное определение команды **alt**, внутри которой эта именованная альтернатива может быть расширена.

   ПРИМЕЧАНИЕ. – Это правило обеспечивает возможность ограниченного выхода именованной альтернативы для обеспечения функциональной возможности описания вариантов по умолчанию.

d) Не разрешается использовать команду **goto** внутри команды **interleave**.

Пример:

```
// Следующий кодовый фрагмент TTCN-3 содержит
 :
label L1;
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }       // ... переход назад к L1 и
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; }     // ... переход вперед к L2,
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;
PCO2.send(integer: 21);
alt {
   [] PCO1.receive
      { goto alt; }             // ... переход, который вынуждает повторное
                                // определение предыдущей команды alt
   [] PCO2.receive(integer: 67)
      {  label L3;
         PCO2.send(MyVar);
         alt {
            [] PCO1.receive
               { goto alt; } // ... снова переход, который вынуждает
                             // повторное определение предыдущей команды alt
                             // (не той, которая была для предыдущего goto),
            [] PCO2.receive(integer: 90)
               {  PCO2.send(integer: 33);
                  PCO2.receive(integer: 13);
                  goto L4; // ... переход вперед из двух вложенных
                           // команд alt,
               }
            [] PCO2.receive(MyError)
               { goto L3; }  // ... переход назад из текущей
                             // команды alt,
            [] any port.receive
               { goto L2; }  // ... переход назад из двух вложенных
                             // команд alt,
         }
      }
   [] any port.receive
      { goto L2; }  // ... и длинный переход назад из некоторой команды alt
}
label L4;
:
```

20.3 Перемежающееся поведение

В командах **interleave** не должны использоваться команды передачи управления **for, do-while, goto, activate, deactivate, stop, return** 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 расширять команды **interleave** с именованными альтернативами или определять ветви **else** в перемежающемся поведении.

Перемежающееся поведение может быть всегда заменено эквивалентным набором вложенных альтернатив. Процедуры такой замены описываются в Приложении B.

Правила определения перемежающейся команды таковы:

a) сразу после выполнения какой-либо команды приема последовательно выполняются команды неприема, пока не поступит следующая команда приема или не закончится перемежающаяся последовательность;

ПРИМЕЧАНИЕ. – Команды приема являются командами TTCN-3, которые могут появляться в наборах альтернатив, то есть команды **receive, check, trigger, getcall, getreply, catch** и **timeout**. Командами неприема обозначаются все другие команды без переноса управления, которые могут использоваться в команде перемежения.

b) затем определение продолжается с помощью следующего "стоп-кадра".

Операционная семантика перемежения полностью определена в Приложении B.

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);
   }
}
:

// может рассматриваться как 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 Поведение по умолчанию

Поведение по умолчанию может рассматриваться как дополнение (расширение) к команде **alt** или как одиночная операция приема, которая определена особым образом. Поведение по умолчанию определяется путем указания **named alt** и активизируется, после чего оно может быть запрошено и выполнено.

Активизация поведения по умолчанию означает, что альтернативы, определенные в соответствующей **named alt**, добавляются к высшему уровню всех последующих альтернатив.

Поведение по умолчанию добавляется также к любым одиночным (то есть не в команде **alt**) операциям приема, тайм-аутам или командам done. Это объясняется тем, что указанные операции принципиально тождественны одиночной альтернативе. Например:

```
:
MyPort.receive(MyMsg);
:

// Это то же, что и
:
alt {
   [] MyPort.receive(MyMsg) {}
}
:
```

20.4.1 Операции Activate и Deactivate

Поведение по умолчанию активизируется с помощью операции **activate** и деактивизируется с помощью операции **deactivate**. Пустая операция **deactivate** деактивизирует все активные поведения по умолчанию.

В случае нескольких активизаций нескольких именованных альтернатив элементы **alt** должны быть расширены в порядке активизации.

В случае, когда аргументом к активной операции является список именованных альтернатив, элементы **alt** должны быть расширены в порядке, указанном в списке.

Пример:

```
named alt Default1() // определение именованной alt
{
   [] MyPort.check
      {MyBehaviour1()}
}

:
```

```
// внутри определения поведения
activate( Default1() );

CL2.receive(MySetup);

alt{
   [] CL2.receive(MySig1)
      {CL2.send(MySig2)}

   [] CL2.receive(MySig2)
      {CL2.send(mySig1)}
}

// Эта команда деактивизирует поведение по умолчанию Default1
deactivate(Default1);
// Эта команда деактивизирует все предыдущие активизированные поведения по умолчанию
deactivate;

// После определения и активизации alt по умолчанию принципиально расширяется
// до конца любой следующей alt или команд приема

activate ( Default1() );
:
CL2.receive(MySetup);

alt {
      [] CL2.receive(MySig1)
         {CL2.send(MySig2)}
      [] CL2.receive(MySig2)
         {CL2.send(mySig1)}
}

// эквивалентно
:
alt {
      [] CL2.receive(MySetup);  // Одиночный receive теперь становится alt
                                // в своем праве

      [] MyPort.check
         {MyBehaviour1()}
}

alt {
      [] CL2.receive(MySig1)
         {CL2.send(MySig2)}
      [] CL2.receive(MySig2)
         {CL2.send(mySig1)}

      [] MyPort.check
         {MyBehaviour1()}
}
```

20.5 Команда Return

Команда **return** заканчивает выполнение функции и возвращает управление в точку, с которой эта функция была вызвана. Команда **return** факультативно может быть связана с некоторым выдаваемым (возвращаемым) значением. Использование **return** в тестовом примере или при управлении эквивалентно команде **stop**.

Пример:

```
function MyFunction() return boolean
{
   :
   if (date == "1.1.2000") { return false; }
// выполнение остановилось 1 января 2000 г. и выдало false (ЛОЖЬ) в качестве
// указания на неисправность
   :
   return true;  // выдана true (ИСТИНА)
}

function MyBehaviour() return verdicttype
{
   :
   if (MyFunction()) { verdict.set(pass); }   // использование MyFunction в
                                              // команде if
   else { verdict.set(inconc); }
   :
   return verdict.get; // Явная выдача вердикта
}
```

21 Операции конфигурации

Операции конфигурации (см. таблицу 13) используются для установления тестовых компонентов и управления ими. Эти операции используются только в тестовых примерах и функциях TTCN-3 (то есть не используются в управляющей части модуля).

Таблица 13/Z.140 – Обзор операций конфигурации TTCN-3

Операции конфигурации	
Команда	**Имя операции**
Создать параллельный тестовый компонент	**create**
Соединить компонент с другим компонентом	**connect**
Разъединить два компонента	**disconnect**
Отобразить порт интерфейса в порт тестового интерфейса	**map**
Устранить отображение порта из интерфейса тестовой системы	**unmap**
Взять адрес MTC	**mtc**
Взять адрес интерфейса тестовой системы	**system**
Взять собственный адрес	**self**
Запустить выполнение тестового компонента	**start**
Остановить выполнение тестового компонента	**stop**
Проверить окончание PTC	**running**
Ожидать окончание PTC	**done**

21.1 Операция Create

MTC является единственным тестовым компонентом, который автоматически создается при запуске тестового примера. Все остальные тестовые компоненты явно создаются во время выполнения теста с помощью операций **create**. Компонент создается с полным набором его портов, у которых входящие очереди пусты. Кроме того, если указанный порт должен иметь тип **in** или **inout**, то он в состоянии ожидания должен быть готов к приему трафика по соединению.

Так как все компоненты и порты при окончании каждого тестового примера неявно уничтожаются, при вызове каждого тестового примера должна полностью создаваться его требуемая конфигурация компонентов и соединений.

```
//  Этот пример объявляет переменную типа address, которая используется для запоминания
//  ссылки на заново создаваемый компонент типа MyComponentType, являющийся результатом
//  функции create.
:
var MyComponenttype MyNewComponent;
:
MyNewComponent  : = MyComponentType.create;
:
```

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. 8.6) и может использоваться для соединения экземпляров и для целей связи, таких, как передача и прием.

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 компонент). 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.

21.2 Операции Connect и Map

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. (См. рис. 7.)

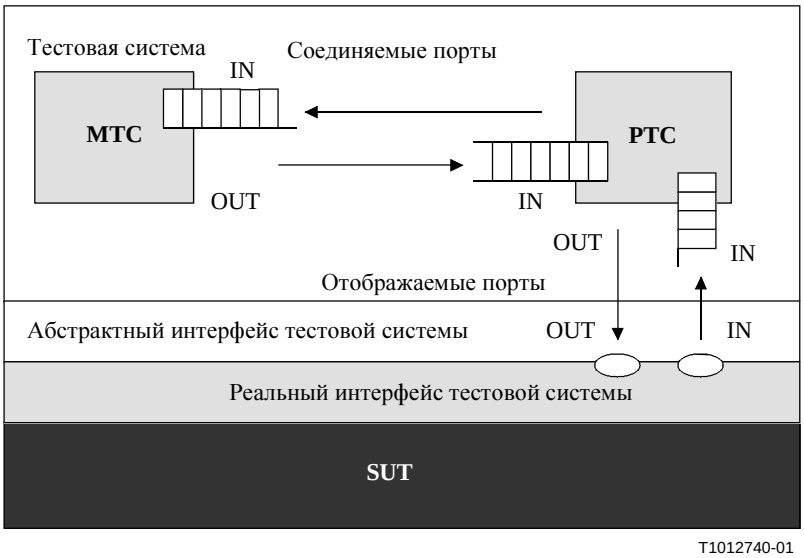


Рисунок 7/Z.140 – Иллюстрация операций connect и map

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.

Имеются две операции – для определения MTC, то есть **mtc**, и для определения портов интерфейса тестовой системы, то есть **system** (см. п. 8.6). Обе операции могут использоваться для определения и соединения портов.

Операции **connect** и **map** могут быть вызваны из любого определения поведения (функции). 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     MyNewComponent;
:
MyNewComponent : = MyComponentType.create;
:
:
connect (MyNewComponent : Port 1, mtc : Port 3);
map (MyNewComponent : Port2, system : PCO1);
:
:
// В этом примере создается новый компонент типа MyComponentType,
// а ссылка на него записывается в переменной MyNewComponent. 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
```

21.2.1 Совместимые соединения

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 $\subseteq$ список in в PORT2 *и* список out в PORT2 $\subseteq$ 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 $\subseteq$ список out в PORT2 *и* список in в PORT2 $\subseteq$ 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.

21.3 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:

```
 :
 :
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
```

21.4 Операции MTC, System и Self

Компонентные ссылки (см. п. 8.6) имеют две операции, **mtc** и **system**, которые выдают ссылки на главный тестовый компонент и на интерфейс тестовой системы соответственно. Кроме того, может использоваться операция **self** для выдачи ссылки на компонент, в котором она вызвана. Например:

```
var MyComponentType MyAddress;
MyAddress := self; // Запомнить текущую компонентную ссылку
```

С компонентными ссылками разрешаются только операции присвоения и эквивалентности.

21.5 Операция Start тестового компонента

Как только компонент создан и соединен, к нему должно быть привязано поведение и должно начаться выполнение этого поведения. Это делается с помощью операции **start** (создание компонента 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. Например:

```
// Предполагается, что порты Port1, Port2, Port3 и PCO1
// надлежащим образом определены и объявлены в соответствующих
// определениях типа порта и типа компонента
 :
var MyComponentType MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
connect(MyNewComponent:Port1, mtc:Port3);
connect(MyNewComponent:Port2, system:PCO1);
 :
 :
MyNewComponent.start(MyComponentBehaviour());
 :
```

```
// В этом примере новый компонент сначала создается, затем соединяется
// с внешней средой и наконец запускается с помощью операции start. Для
// идентификации компонента, который будет выполняться, используется его
// справочный номер.
```

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 либо должна иметь определение **runs on**, которое ссылается на тот же тип компонента, что и заново созданный компонент, либо должна переносить всю необходимую информацию из определения типа компонента в качестве параметров.

– Порты и таймеры могут переноситься в этой функции только в случаях, когда они ссылаются на порты и таймеры в определении типа компонента для заново созданного компонента, то есть порты и таймеры являются локальными для экземпляров компонента и не должны переноситься к другим компонентам.

ПРИМЕЧАНИЕ. – Способность переносить порты в качестве параметров позволяет определять общие функции, которые не привязаны к одному конкретному типу компонентов.

21.6 Операция Stop тестового компонента

Команда **stop** тестового компонента явно останавливает выполнение тестового компонента, в котором вызван **stop**. Эта операция не имеет аргументов. Например:

```
if (date = = "1.1.2000") { stop }/ / Выполнение остановлено 1 января 2000 г.
```

Если останавливаемым тестовым элементом является МТС, то все оставшиеся РТС, которые еще работают, также должны быть остановлены, а тестовый пример закончен.

ПРИМЕЧАНИЕ. – Конкретный механизм остановки всех оставшихся работающих РТС не рассматривается в данной Рекомендации.

По окончании тестового компонента все ресурсы освобождаются либо явно с помощью этой операции **stop** или путем достижения команды **return** в функции, которая первоначально запустила тестовый компонент, либо неявно, когда компонент достигнет конца своего дерева поведения. На любые переменные, хранящие ссылки на остановленные компоненты, не делаются ссылки.

Правила окончания тестовых примеров и вычисления окончательного вердикта для теста описываются в разделе 24.

21.7 Операция Running

Операция **running** позволяет при выполнении поведения в тестовом компоненте установить, завершено ли выполнение поведения в другом тестовом компоненте. 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. Например:

```
if (PTC1.running)                          // Использование running в команде if
{
   // Делайте что-нибудь!
}


while (all component.running != true) { // Использование running в состоянии loop
      MySpecialFunction()
}
```

21.8 Операция Done

Операция **done** позволяет при выполнении поведения в тестовом компоненте установить, завершено ли выполнение поведения в другом тестовом компоненте.

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.

ПРИМЕЧАНИЕ. – Операция **done** TTCN-3 и операция DONE TTCN-2 имеют одинаковую семантику.

Пример:

```
// Использование done в альтернативах
:
alt {
   [] MyPTC.done {
        verdict.set(pass)
      }

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


// 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
```

21.9 Использование массивов компонентов

Операции **create, connect, start** и **stop** не работают прямо в массивах компонентов. Вместо этого в качестве параметра обеспечивается специальный элемент – массив. Для компонентов достигается эффект массива путем использования массива ссылок на компоненты и присвоения соответствующих элементов массива результату операции **create**.

```
// 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 MyPTCType1MyPtcType[11];
    :
   for (i:= 1; i<=10; i:=i+1)
   {
      MyPtcAddresses[i] := MyPtcType1.create;
      connect(self:PtcCoordination, MyPtcAddresses[i]:MtcCordination);
      MyPtcAddresses[i].start(MyPtcBehaviour());
   }
    :
}
```

21.10 Использование Any и All с компонентами

Ключевые слова **any** и **all** могут использоваться с операциями конфигурации, как показано в таблице 14.

Таблица 14/Z.140 – Any и All с компонентами

Операция	Разрешается		Пример
	any	**all**	
create			
start			
running	Да, но только от MTC	Да, но только от MTC	**any component.running** **all component.running**
done	Да, но только от MTC	Да, но только от MTC	**any component.done** **all component.done**
stop			

22 Операции связи

TTCN-3 поддерживает связь на базе сообщений (асинхронную) и на базе процедур (синхронную) (см. п. 8.1). Асинхронная связь не блокируется операцией **send**, как показано на рис. 8, где обработка в MTC продолжается немедленно после появления операции **send**. SUT блокируется операцией **receive** до тех пор, пока она не получит передаваемого сообщения.

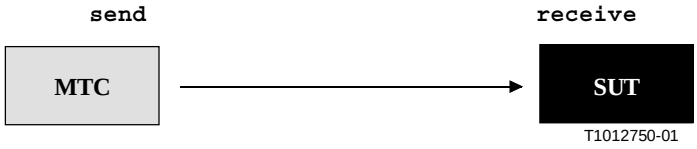


Рисунок 8/Z.140 – Иллюстрация асинхронных send и receive

Синхронная связь является блокирующей при операции **call**, как показано на рис. 9, где операция **call** блокирует обработку в MTC до приема **reply** или особого состояния от SUT. Аналогично **receive, getcall** блокирует SUT до получения вызова.

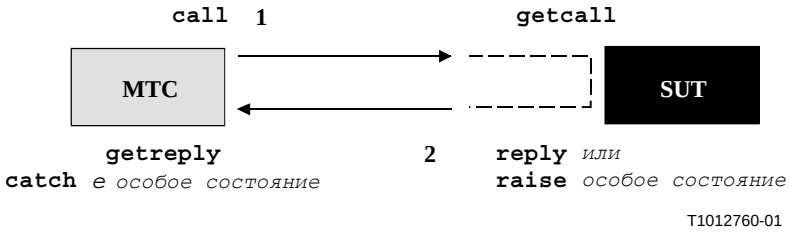


Рисунок 9/Z.140 – Иллюстрация полного синхронного соединения

Операции **send** и **call** вместе называются операциями связи. Эти операции используются только в тестовых примерах и функциях TTCN-3 (то есть не прямо в управляющей части модуля). Операции связи делятся на три группы:

a) компонент передает сообщение, вызывает какую-либо процедуру или отвечает на принятый вызов или порождает особое состояние. Эти действия вместе называются *операциями передачи*;

b) компонент принимает сообщение, вызов процедуры, ответ на ранее вызванную процедуру или вводит особое состояние. Эти действия вместе называются *операциями приема*;

c) управляет доступом к порту, выполняя **clear, start** или **stop**. Эти действия вместе называются *операциями управления*.

Эти операции могут использоваться в портах связи тестового компонента, как обобщается в таблице 15. В случаях комбинированных портов применимы все эти операции.

Таблица 15/Z.140 – Обзор операций связи TTCN-3

Операции связи			
Операция связи	**Ключевое слово**	**Может использоваться в портах на базе сообщений**	**Может использоваться в портах на базе процедур**
Операции передачи			
Передать сообщение	**send**	Да	
Обратиться к вызову процедуры	**call**		Да
Ответить на вызов процедуры от удаленного объекта	**reply**		Да
Породить особое состояние (на принятый вызов)	**raise**		Да
Операции приема			
Принять сообщение	**receive**	Да	
Запустить сообщение	**trigger**	Да	
Принять вызов процедуры от удаленного объекта	**getcall**		Да
Обработать ответ на предыдущий вызов	**getreply**		Да
Ввести особое состояние (от вызываемого объекта)	**catch**		Да
Проверить принятые сообщение/вызов/ особое состояние/ответ	**check**	Да	Да
Операции управления			
Освободить порт	**clear**	Да	Да
Освободить порт и дать доступ к нему	**start**	Да	Да
Остановить доступ к порту (для приема и передачи)	**stop**	Да	Да

22.1 Операции передачи

Операциями передачи являются:

a) **send:** передать сообщение асинхронно;

b) **call:** вызвать процедуру;

c) **reply:** ответить на принятый вызов процедуры от SUT; и

d) **raise:** породить особое состояние в случаях, когда принят вызов процедуры.

22.1.1 Общий формат операций передачи

Операции передачи состоят из части *передача* и, в случае операции **call** на базе процедур, части *ответ и обработка особого состояния*.

Часть "передача":

- описывает порт, в котором должна выполняться определяемая операция;
- определяет значение информации, предназначенной для передачи;
- выдает факультативное адресное выражение, которое уникально (однозначно) определяет партнера по связи в случае соединения типа "один ко многим".

Имя порта, имя операции и значение должны присутствовать во всех операциях передачи. Указание партнера по связи (обозначаемое ключевым словом **to**) факультативно и требуется только в случаях соединений типа "один ко многим", когда принимающий объект должен быть явно определен.

22.1.1.1 Ответ и обработка особого состояния

Ответ и обработка особого состояния нужны только в случаях синхронной связи. 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.

22.1.2 Операция Send

Операция **send** используется для установки значения в очередь порта исходящего сообщения. Это значение может указываться путем ссылки на шаблон, переменную или константу либо может быть определено "инлайн" из некоторого выражения (которое, безусловно, может быть явным значением). При определении "инлайнового" значения должно использоваться факультативное поле типа, если имеется неопределенность в типе передаваемого значения.

Операция **send** должна использоваться только в портах на базе сообщений (или смешанных), а тип передаваемого значения должен содержаться в списке исходящих типов в определении типов порта. Например:

```
MyPort.send (MyTemplate (5, MyVar) );
// Передает шаблон MyTemplate с реальными параметрами 5 и MyVar через
// MyPort.

MyPort.send (integer : 5);
// Передает целочисленное значение 5
```

В случае соединений типа "один ко многим" партнер по связи должен указываться однозначно. Это обозначается ключевым словом to. Например:

```
MyPort.send ("My string") to MyPartner;
// Передает цепочку "My string" компоненту со справочным номером компонента,
// хранящимся в переменной MyPartner.
```

```
MyPCO.send (MyVariable + YourVariable - 2) to MyPartner;
// Передает результат арифметического выражения к MyPartner.
```

22.1.3 Операция Call

Операция **call** используется для указания, что тестовый компонент вызывает процедуру из SUT или из другого тестового компонента. Операция **call** является блокирующейся в том смысле, что она должна ждать получения ответа (то есть **reply**) или особого состояния от вызываемого объекта. Другими словами, операция **call** работает синхронно.

ПРИМЕЧАНИЕ. – Это сравнимо с тестированием функции сервера, то есть SUT является сервером, а компонент играет роль клиента.

Операция **call** применяется только в портах на базе процедур (или смешанных). Определение типа порта, в котором выполняется операция **call**, должно содержать имя процедуры из его списка **out** или **inout**, то есть оно должно разрешать вызывать эту процедуру на этот порт.

Значением операции **call** является сигнатура, которая может определяться либо в виде сигнатуры шаблона, либо "инлайн". Например:

```
signature MyProc (out integer MyPar1, inout boolean MyPar2);
:
MyPort.call (MyProc : {MyVar1, MyVar2});
// Вызывает удаленную процедуру MyProc из MyCL с параметрами in и inout 5
// и MyVar. От этого вызова не ожидается ни вызываемого (возвращаемого) значения,
   ни особого состояния.
// Если один (или оба) из этих параметров определен в качестве параметра inout,
// то его значение не будет рассматриваться, то есть оно не присваивается
// переменной.

// Следующий пример поясняет возможности присвоения значений параметрам in и
// inout в аргументе вызова. Следующая сигнатура предназначена для процедуры,
// подлежащей вызову. Отметим, что MyProc2 не имеет возвращаемого значения и
// особых состояний
signature MyProc2 (in integer A, out integer B, inout integer C);
:
MyPort.call (MyProc2 : {1, -, 3});
//  Указаны только значения параметров in и inout. Возвращаемые значения
параметров
// out и inout не используются после этого вызова и поэтому не присваиваются
// переменным.
```

Все параметры **in** и **inout** сигнатуры должны иметь специальное значение, то есть использовать механизмы сопоставления, такие как *AnyValue*, не разрешается.

Аргументы сигнатуры в операции **call** не используются при запросе имен переменных для параметров **out** и **inout**. Реальное присвоение возвращаемого значения процедуры и значений параметров **out** и **inout** переменным должно явно осуществляться в части "ответ" (**getreply**) и "обработка особого состояния" (**catch**) операции **call**. Это обозначается ключевыми словами **value** и **param** соответственно. Это позволяет использовать шаблоны сигнатуры в операции **call** таким же образом, каким шаблоны могут использоваться для типов.

Обычно предполагается, что операция **call** имеет блокирующую семантику. Однако TTCN-3 поддерживает также неблокирующие вызовы. Считается, что вызов, не имеющий возвращаемых значений, будет неблокирующим вызовом. Особые состояния (если они определены), порожденные вызовом без возвращаемых значений, должны вводиться внутрь последующей команды **alt**. Кроме того, возможно также установить неблокирующую семантику с помощью ключевого слова **nowait** (см. п. 22.1.3.3).

В случае соединений типа "один ко многим" партнер по связи должен быть определен однозначно. Это обозначается ключевым словом **to**. Например:

```
MyPort.call (MyProc : {MyVar1, MyVar2}) to MyPartner;
// В этом примере вызываемая сторона явно определена с помощью справочного номера
// компонента, хранящегося в переменной MyPartner.
```

22.1.3.1 Обработка ответов на Call

Обработка ответа на **call** производится с помощью операции **getreply** (см. п. 22.2.5). Эта операция определяет альтернативное поведение, зависящее от ответа, который был генерирован как результат операции **call**. Например:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner
                              // Где { ... } является "инлайновым" шаблоном
{
   [] MyCl.getreply(MyProc:{MyVar1, MyVar2}) {}
}
```

Если требуется, возвращаемое значение вызываемой процедуры будет явно считано в операции **getreply**. Это выражается с помощью '**- >**' и (факультативного) ключевого слова **value**. Например:

```
MyPort.call (MyProc : {MyVar1, MyVar2}) to MyPartner
{
   [] MyC1.getreply (MyProc : {MyVar1, MyVar2}) - > value MyResult { }
}
// Значение должно быть возвращено с помощью MyProc, которая будет записана в переменной
// MyResult.
```

Аргументы сигнатуры в операции **call** не используются для запроса имен переменных для параметров **out** и **inout**. Реальное присвоение возвращаемого значения процедуры и значений параметров **out** и **inout** переменным должно осуществляться явно в части "ответ" (**getreply**) и "обработка особого состояния" (**catch**) операции **call**. Это обозначается ключевыми словами **value** и **param** соответственно. Это позволяет использовать шаблоны сигнатур в операциях **call** таким же образом, каким шаблоны могут использоваться для типов. Например:

```
MyPort.call(MyProc:{5,MyVar}) to MyPartner
{
   []MyCl.getreply(MyProc:{MyVar1, MyVar2}) -> value MyResult param
      (MyPar1Var,MyPar2Var) {}
}
// В этом примере оба параметра из MyProc определены как параметры inout,
// а их значения после окончания MyProc присваиваются MyPar1Var
// и MyPar2Var.
```

22.1.3.2 Обработка особых состояний, выданных на Call

Обработка особых состояний, выданных на **call**, производится с использованием операции **catch** (см. п. 22.2.6). Эта операция определяет альтернативное поведение, зависящее от особого состояния (если таковое имеется), которое было генерировано как результат операции **call**. Например:

```
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
   exception (ExceptionTypeOne, ExceptionTypeTwo, ExceptionTypeThree);
:
// Следующая операция call показывает механизм обработки getreply и особого состояния
// в операции call
MyPort.call(MyProc3:{5,MyVar}, 30E-3) to MyPartner
{
   [] MyCl.getreply(MyProc3:{MyVar1, MyVar2}) -> value MyResult param
                                                  (MyPar1Var,MyPar2Var) {}
   [] MyPort.catch(MyProc3, MyExceptionOne)
      {                              // Ввести особое состояние
```

```
         verdict.set(fail);          // Установить вердикт и
         stop                        // stop как результат особого состояния
      }
   [] MyPort.catch(MyProc3, MyExceptionTwo)   // Ввести второе особое состояние
      {verdict.set(inconc)}                   // Установить вердикт и продолжить
                                              // после вызова как результат
                                              // второго особого состояния

   [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) {}
                                 // Ввести третье особое состояние, которое может
                                 // появиться, если оценкой MyCondition будет ИСТИНА

   [] MyPort.catch(timeout) {}   // Особое состояние тайм-аута, то есть вызываемая
                                 // сторона не ответила вовремя, ничего не делается
}
```

22.1.3.3 Обработка особых состояний по тайм-ауту, выданных на Call

Операция **call** факультативно может содержать тайм-аут. Это выражается в виде явного значения или константы типа **float**, определяющих продолжительность времени после запуска операции **call**, в течение которого особое состояние **timeout** должно генерироваться тестовой системой. Если в операции **call** нет части со значением тайм-аута, то не будет генерироваться особое состояние **timeout**. Например:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3)
{
   [] MyPort.catch(timeout)
      {
         verdict.set(fail);
         stop
      }
}
// Этот пример показывает вызов со значением тайм-аута 20 мс. Это означает, что
// если вызываемая сторона не выдаст ответ или особое состояние в пределах этого
// времени, то тестовая система будет автоматически генерировать особое состояние
// timeout. Если процедура завершилась без особого состояния timeout, то выполнение
// будет продолжено с помощью команды, следующей за операцией call.
```

Использование ключевого слова **nowait** в части "значение тайм-аута" в операции **call** позволяет вызвать процедуру без ожидания либо окончания, ответа или особого состояния, порожденного вызываемой процедурой, либо особого состояния по тайм-ауту. Например:

```
MyPort.call (MyProc : {5, MyVar}, nowait);
// В этом примере тестовый компонент будет продолжать выполнение без ожидания
// окончания MyProc.
```

В таких случаях возможный ответ или особое состояние должны быть удалены из очереди с помощью операции **getreply** или **catch** в последующей команде **alt**.

22.1.4 Операция Reply

Операция **reply** используется для ответа на ранее принятый вызов согласно процедурной сигнатуре. Операция **reply** используется только в порту на базе процедур (или смешанном). Определение типа для порта должно содержать имя процедуры, к которой относится эта операция **reply**.

Часть "значение" в операции **reply** содержит ссылку на сигнатуру с соответствующим списком реальных параметров и (факультативно) возвращаемое значение. Сигнатура может определяться либо в форме шаблона сигнатуры, либо "инлайн". Все параметры **out** и **inout** сигнатуры должны иметь некоторое конкретное значение, то есть использование механизмов сопоставления, таких как *AnyValue*, не разрешается. Например:

```
MyPort.reply (MyProc2 : {-, 5});
```

```
// Отвечает на принятый вызов MyProc2. Эта MyProc2 не имеет возвращаемого значения,
// но имеет два параметра. Первый параметр – это параметр in, то есть его значение не
// будет ответом и поэтому не нуждается в определении. Второй параметр – это параметр
// out или inout. Его значение равно 5.
```

В случаях соединений типа "один ко многим" партнер по связи должен быть четко указан и должен быть уникальным. Это обозначается с помощью ключевого слова **to**. Например:

```
MyPort.reply (MyProc3 : {-, 5}) to MyPartner;
// Этот пример идентичен предыдущему примеру, но ответ направлен компоненту
// со справочным номером компонента, хранящимся в переменной MyPartner.
```

Если какое-либо значение должно быть возвращено вызывающей стороне, то это должно быть явно сформулировано при помощи ключевого слова **value**.

```
MyPort.reply (MyProc : {5, MyVar} value 20);
// Отвечает на принятый вызов MyProc. Возвращаемое значение процедуры MyProc равно
// 20, а процедура имеет два параметра – out и inout. Их
// значениями будут 5 и MyVar.
```

22.1.5 Операция Raise

Операция **raise** используются для порождения особого состояния. Особое состояние порождается только в порту на базе процедур (или смешанном). Особое состояние – это реакция на принятый вызов процедуры, результат которого ведет к особому событию. Тип особого состояния определяется в сигнатуре вызываемой процедуры. Определение типа порта должно содержать в его списке принятых вызовов процедуры имя процедуры, к которой относится особое состояние.

ПРИМЕЧАНИЕ. – Отношение между принятым вызовом и операцией **raise** не всегда может быть проверено статически. Для тестирования разрешается определять операцию **raise** без связанной операции **getcall**.

Часть "значение" операции **raise** содержит ссылку на сигнатуру, за которой следует значение особого состояния. Например:

```
MyPort.raise (MySignature, MyVariable + YourVariable - 2);
// Порождает особое состояние со значением, которое является результатом
// арифметического выражения в MyPort
```

Особые состояния определяются в виде некоторого типа. Поэтому значение особого состояния может либо выводиться из шаблона, либо быть значением, вытекающим из какого-либо выражения (которое, безусловно, может быть явным значением). Факультативное поле типа в определении значения для операции **raise** используется в случаях, когда необходимо избежать любой неясности в типе значения, подлежащем передаче. Например:

```
MyPort.raise (MyProc, MyExceptionType : {5, MyVar});
// Порождение особого состояния от удаленной процедуры, определенной с помощью
// MyProc, со значением, которое определяется шаблоном MyExceptionTemplate, с
// реальными параметрами 5 и MyVar в порту MyPort
```

В случаях соединений типа "один ко многим" партнер по связи должен быть указан однозначно. Это обозначается ключевым словом **to**. Например:

```
MyPort.raise (MySignature, "My string") to MyPartner;
// Порождает цепочечное особое состояние со значением "My String" в MyPort для
// компонента со справочным номером компонента, хранящимся в переменной MyPartner.
```

22.2 Операции приема

Операциями приема являются:

a) **receive:** принять асинхронно переданное сообщение;

b) **trigger:** запустить получение конкретного сообщения;

c) **getcall:** принять вызов процедуры;

d) **getreply:** обработать ответ на ранее вызванную процедуру;

e) **catch**: ввести особое состояние, которое было порождено как реакция на операцию **call**; и

f) **check:** проверить верхний элемент в очереди конкретного порта.

22.2.1 Общий формат операций приема

Операция приема состоит из части *прием* и 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 (факультативное) 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.

22.2.1.1 Выполнение присвоений при операциях приема

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 для запоминания особых состояний.

Кроме того, часть "присвоение" может использоваться также для присвоения адреса **sender** в сообщении, особого состояния, **reply** или **call** переменной. Это полезно для соединений типа "один ко многим", в которых, например, одно и то же сообщение или вызов могут быть получены от разных компонентов, но сообщение, **reply** или особое состояние должны быть переданы обратно к исходному передающему компоненту.

22.2.2 Операция Receive

Операция **receive** используется для приема значения из очереди порта входящих сообщений. Значение может быть указано ссылкой на шаблон, переменную или константу либо может быть определено "инлайн" из выражения (которое, конечно, может быть явным значением). При определении значения "инлайн" используется факультативное поле типа, чтобы избежать каких-либо неопределенностей в типе принимаемого значения. 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 устраняется из очереди порта, то есть если операция **receive** неуспешна, то выполнение тестового примера продолжается со следующей альтернативой.

Критерии сопоставления относятся к типу и значению принимаемого сообщения. Тип и значение сообщения, которое необходимо принять, могут либо выделяться из шаблона, либо быть значением, взятым из выражения (которое, безусловно, может быть явным значением).

```
MyPort.receive (MyTemplate (5, MyVar));
// Определяет прием значения, которое удовлетворяет условиям, определенным с помощью
// шаблона MyTemplate с реальными параметрами 5 и MyVar.
```

```
MyPort.receive (A<B);
// Определяет прием булева значения true (ИСТИНА) или false (ЛОЖЬ), зависящего от результата
// A < B
```

Факультативное поле типа в критерии сопоставления операции **receive** используется для того, чтобы избежать какой-либо неопределенности с типом принимаемого значения. Например:

```
MyPort.receive (integer : MyVar);
// Определяет прием целочисленного значения, которое имеет то же значение, что и
// переменная MyVar в MyPort. Целое число идентификатора типа (факультативного)
// не является строго необходимым, так как тип уже дан в определении MyVar.
// Однако в случаях сложных и длинных тестов такой идентификатор типа может
// использоваться для повышения удобочитаемости.

MyPort.receive (MyVar);
// Является альтернативой предыдущему примеру.
```

Если сопоставление оказалось успешным, то значение, удаляемое из очереди в порту, может быть записано в переменной, а адрес компонента, передавшего сообщение, может быть вызван и записан в переменной. Это обозначается символом '–>' и ключевым словом **value**. Например:

```
MyPort.receive (MyType :*) from MyPartner -> value MyVar;
// Определяет прием произвольного значения для MyType (от компонента
// с адресом, записанным в переменной MyPartner), которое потом присваивается к
// переменной MyVar. MyVar должна быть типа MyType.
```

В случае соединений типа "один ко многим" операция **receive** может быть ограничена определенными партнерами по связи. Это ограничение обозначается ключевым словом **from**.

```
MyPort.receive (charstring : "Hello") from MyPartner;
// Определяет прием цепочки знаков "Hello" от компонента со справочным номером
// компонента или с адресом, записанным в переменной MyPartner.
```

Можно также запросить справочный номер компонента или адрес отправителя сообщения. Это обозначается ключевым словом **sender**. Например:

```
MyPort.receive (MyTemplate : {5, MyVarOne}) -> value MyVarTwo sender MyPartner;
// Определяет прием значения, которое удовлетворяет условиям, определенным шаблоном
// MyTemplate с реальными параметрами 5 и MyVarOne. После приема это значение
// присваивается переменной MyVarTwo. Справочный номер передающего компонента
// вызывается операцией call и присваивается к переменной MyPartner.

MyPort.receive (A<B) -> sender MyPartner;
/ / Определяет прием булева значения true или false, зависящего от результата
/ / A<B. Справочный номер передающего компонента запрашивается операцией call и
/ / присваивается переменной MyPartner.
```

22.2.2.1 Получение 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.

ПРИМЕЧАНИЕ. – Это эквивалентно команде OTHERWISE из TTCN-2.

Сообщение, принятое с помощью *ReceiveAnyMessage*, не присваивается переменной.

Пример:

```
MyPort.receive;
// Удаляет верхнее значение из MyPort.
```

```
MyPort.receive from MyPartner;
// Удаляет верхнее значение из CL1, если оно является сообщением от компонента
// с указанным адресом.

MyPort.receive -> sender MySenderVar;
// Удаляет верхнее значение из CL1, но запоминает передаваемый экземпляр путем записи
// его справочного номера в MySenderVar.
```

22.2.2.2 Получение в любом порту

Для получения сообщения в любом порту используется ключевое слово **any**. Например:

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

22.2.3 Операция Trigger

Операция **trigger** по определенным критериям сопоставления фильтрует сообщения из потока принимаемых сообщений в заданном входящем порту. Операция **trigger** используется только в портах на базе сообщений (или смешанных), а тип принимаемого значения должен быть включен в список входящих типов в определении типов порта. Все сообщения, которые не удовлетворяют критериям сопоставления, удаляются из очереди без последующих действий, то есть операция **trigger** ожидает следующее сообщение в этой очереди. Если сообщение удовлетворяет критериям сопоставления, то операция **trigger** действует так же, как операция **receive**. Например:

```
MyPort.trigger (MyType :*);
// Определяет, что операция будет инициировать прием первого обнаруженного в порту MyPort
// сообщения типа MyType с произвольным значением
```

Операция **trigger** требует имя порта, критерии сопоставления для типа и значения, факультативные ограничения **from** (то есть выбор партнера по связи) и факультативное присвоение сопоставляемого сообщения и компонента передатчика переменным.

Пример:

```
MyPort.trigger (MyType :*) from MyPartner;
// Определяет, что операция будет инициировать прием первого обнаруженного в порту
// MyCL сообщения типа MyType с произвольным значением, поступающего от компонента
// со справочным номером, идентичным тому, который записан в переменной MyPartner.

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;
// Определяет, что операция будет инициировать прием первого обнаруженного в MyPort
// сообщения типа MyType с произвольным значением. Справочный номер компонента
// передатчика из этого сообщения будет записан в переменной MyPartner.

MyPort.trigger (integer :*) -> value MyVar sender MyPartner;
// Определяет, что операция будет инициировать прием произвольного целочисленного
// значения, которое потом запоминается в переменной MyVar, а справочный номер
// компонента передатчика из этого сообщения будет записан в переменной MyPartner.
```

22.2.3.1 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;
```

22.2.3.2 Запуск в любом порту

Для запуска сообщения в любом порту используется ключевое слово **any**. Например:

```
any port.trigger
```

22.2.4 Операция Getcall

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.

```
MyPort.getcall (MyProc (5, MyVar);
// Будет принят вызов процедуры MyProc в MyCL с параметрами in или inout 5 и
// "значение MyVar".
```

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 обозначается ключевым словом **from**.

```
My.Port.getcall (MyProc : {5, MyVar}) from MyPartner;
// Будет принимать вызов процедуры MyProc в MyCL (с параметрами in или inout 5
// и "значение MyVar") от корреспондента с адресом или справочным номером компонента,
// записанным в переменной MyPartner.
```

Часть "присвоение" операции **getcall** содержит факультативное присвоение значений параметров **in** и **inout** переменным, а также запрос и присвоение адреса вызывающего компонента переменной.

Для запроса значений параметров вызова используется ключевое слово **param**. Например:

```
MyPort.getcall (MyProc : {5, MyVar}) from MyPartner -> param (MyPar1Var, MyPar2Var);
// Оба параметра в MyProc являются параметрами inout, причем такими, значения которых
// присвоены MyPar1Var и MyPar2Var. Идентификация параметров, определенных
// в процедурной сигнатуре, и имен в списке имен переменных, которые следуют за ключевым
// словом param, в вышеуказанной операции приема дана в порядке их расположения в списке.
```

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").

```
MyPort.getcall (MyProc : {5, MyVar}) -> sender MySenderVar;
// Будет принимать вызов процедуры MyProc в MyCL с параметрами in или inout 5 и
// MyVar. Вызывающая сторона запрашивается операцией приема и записывается в
// MySenderVar. Это позволяет обработать вызов одной и той же процедуры от нескольких
// компонентов в одном и том же порту одним и тем же способом. MySenderVar может
// использоваться для ответа или порождения особого состояния к вызывающему компоненту.
```

Аргументы сигнатуры операции **getcall** не используются для пересылки параметров **in** и **inout** в именах переменных. Присвоение значений параметров **in** и **inout** переменным производится в части "присвоение" операции **getcall**. Это позволяет использовать шаблоны сигнатур в операциях **getcall** таким же образом, каким шаблоны используются для типов.

Следующие операции **getcall** показывают возможности использования атрибутов сопоставления и опускания факультативных частей, которые могут быть не важны для спецификации теста.

Пример:

```
MyPort.getcall (MyProc : {5, Var}) -> param (MyPar1Var, MyPar2Var) sender
MySenderVar;

MyPort.getcall (MyProc : {5, *}) -> param (MyPar1Var, MyPar2Var);

MyPort.getcall (MyProc : {*, MyVar}) -> param (-, MyPar2Var);
// 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, integer E,
inout integer F);
// MyProc2 не имеет возвращаемого значения и каких-либо особых состояний

MyPort.getcall (MyProc2 : {*, *, 3, -, -, *}) ->
       param (MyVarIn1, MyVarIn2, MyVarIn3, -, -, MyVarInout1);
// Параметры in A, B и C присвоены переменным MyVarIn1, MyVarIn2 и MyVarIn3,
// параметр inout F присвоен переменной MyVarInout1. Параметры out D и E не требуется
// учитывать в части "присвоение" этой операции приема.

MyPort.getcall (MyProc2 : {*, *, *, -, -, *}) ->
       param (MyVarIn1 : = A, MyVarIn2 : = B, MyVarIn3 : = C, MyVarInout1 : = F);
// 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 (MyVarInout1 : = F);
// Для дальнейшего выполнения тестового примера необходимо только значение параметра
// inout.
```

22.2.4.1 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 из верхнего уровня очереди в CL1, если вызывающей стороной является объект,
// адрес или справочный номер которого записан в переменной MyPartner.

MyPort.getcall -> sender MySenderVar;
// Удаляет вызов из верхнего уровня очереди в CL1, но запоминает вызывающую сторону
// путем записи ее адреса или справочного номера компонента в MySenderVar.
```

22.2.4.2 Прием в любом порту

Для обозначения операции **getcall** в любом порту используется ключевое слово **any**. Например:

```
any port.getcall (MyProc)
```

22.2.5 Операция Getreply

Операция **getreply** используется для обработки ответов на ранее вызванные процедуры. Операция **getreply** используется только в порту на базе процедур (или смешанном). Например:

```
MyPort.getreply (MyProc : {5, MyVar} value 20);
// Принимает ответ процедуры MyProc, в котором возвращаемое значение равно 20,
// а значения двух параметров out или inout равны 5 и значению MyVar.

MyPort.getreply (MyProc2 : {-, 5});
// Принимает ответ от MyProc2. Процедура MyProc2 не имеет возвращаемого значения,
// но имеет два параметра. Первый параметр является параметром in, то есть его значение
// не будет выдаваться в виде ответа и поэтому не будет учитываться для сопоставления.
// Второй параметр является параметром out или inout. Его значением должно быть 5.
```

Она может использоваться либо в **getreply** и в части "особое состояние" вызова, например:

```
MyPort.call (MyProc) to MyPeer
{
   [ ] MyPort.getreply(MyProc:*) {}
   [ ] MyPort.catch {}
}
```

либо внутри команды **alt**, например:

```
MyPort.call (MyProc, nowait) to MyPeer;
:
alt
{
   [ ] MyPort.getreply(MyProc:*) {}
   :
}
```

Операция **getreply**, если она используется в команде **alt**, должна охватывать случаи, когда ответ на предыдущую вызванную процедуру поступает слишком поздно, то есть когда порождено особое состояние по тайм-ауту.

Как и в других операциях приема, механизмы сопоставления в операции **getreply** разрешены для того, чтобы различать ответы от предыдущей вызванной процедуры, каждый из которых отличается по возвращаемому значению и/или значению параметров **out** и **inout**.

```
MyPort.getreply (MyProc1 : {*, MyVar});
// В этом примере отсутствуют ограничения на возвращаемое значение и значение первого
// параметра.

MyPort.getreply (MyProc1 : {*, *});
// Операция getreplay будет сопоставлять любой ответ от MyProc1 с любым возвращаемым
// значением. Звездочки являются определениями шаблонов inline для MyProc1 и
// возвращаемого типа MyProc1.
```

В случаях соединений типа "один ко многим" операция **getreply** позволяет различать разных партнеров по связи с помощью раздела **from**.

```
MyPort.getreply (MyProc2 : {-, 5}) from MyPartner;
// Этот ответ принимается только в случае, когда он исходит от компонента со справочным
// номером,указанным в переменной MyPartner
```

Факультативная часть "присвоение" в операции **getreply** позволяет присваивать значения параметров **out** и **inout** и возвращаемых значений переменным.

Пример:

```
MyPort.getreply (MyProc1 : {*, *} value *) -> value MyReturnValue
        param (MyPar1, MyPar2);
// После приема возвращаемое значение присваивается переменной MyReturnValue, а
// значения двух параметров out или inout присваиваются переменным MyPar1 и MyPar2.

MyPort.getreply (MyProc1 : {*, *} value *) -> value MyReturnValue param (-, MyPar2)
   sender MySender;
// Значение первого параметра не учитывается при дальнейшем выполнении теста, а адрес
// или справочный номер компонента объекта, от которого был получен ответ,
// записывается в переменной MySender.

// Следующие примеры описывают некоторые возможности для присвоения значений
// параметров out и inout переменным. Следующая сигнатура предназначена для
// процедуры, которая вызвана
signature MyProc2 (in integer A, integer B, integer C, out integer D, integer E,
   inout integer F);
// Отметим, что MyProc2 не имеет возвращаемого значения и особых состояний.

MyPort.getreply (MyProc2 :*) -> param (-, -, -, MyVarOut1, MyVarOut2, -,
   MyVarInout1);
// Параметры in D и E присвоены переменным MyVarOut1 и MyVarOut2. Параметр inout
// F присвоен переменной MyVarInout1.

MyPort.getreply (MyProc2 :*) -> param (MyVarOut1 : =D, MyVarOut2 : =E, MyVarInout1 : =F);
// Альтернативная нотация для присвоения значений параметров in и inout переменным.
// Отметим, что имена в списке присвоений указывают на имена, использованные в
// сигнатуре из MyProc2

MyPort.getreply (MyProc2 : {-, -, -, 3, *, *}) -> param (MyVarInout1 :=F);
// Для дальнейшего выполнения тестового примера необходимо только значение параметра
// inout
```

22.2.5.1 Чтение ответа на любой вызов

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.

Пример:

```
MyPort.getreply;
// Удаляет ответ из верхнего уровня очереди в MyPort.

MyPort.getreply from MyPartner;
// Удаляет ответ из верхнего уровня очереди в CL1, если отвечающей стороной является
// объект, адрес или справочный номер компонента которого записан в переменной
// MyPartner.

MyPort.getreply -> sender MySenderVar;
// Удаляет ответ из верхнего уровня очереди в CL1, но запоминает отвечающую сторону
// путем записи ее в переменной MySenderVar
```

22.2.5.2 Чтение ответа в любом порту

Для чтения ответа в любом порту используется ключевое слово **any**. Например:

```
any port.getreply (MyProc)
```

22.2.6 Операция Catch

Операция **catch** используется, чтобы уловить (зафиксировать) особые состояния, порожденные корреспондентом в качестве реакции на вызов процедуры. Операция **catch** используется только в портах на базе процедур (или смешанных). Тип введенного особого состояния указывается в сигнатуре процедуры, которая породила особое состояние.

```
MySyncPort.catch (MySignature, integer : MyVar);
// Определяет фиксацию особого состояния, порожденного процедурой с сигнатурой
// MySignature в порту MySyncPort. Это особое состояние является целочисленным
// значением, которое имеет то же значение, что и переменная MyVar. Целое число
// (факультативное) идентификатора типа не строго необходимо, так как этот тип уже указан
// в определении MyVar. Однако в сложных и длинных тестовых примерах такой
// идентификатор типа может использоваться для повышения удобочитаемости.

MySyncPort.catch (MySignature, MyVar);
// Является альтернативой предыдущему примеру.

MySyncPort.catch (MySignature, A<B);
// Вводит булево особое состояние ИСТИНА и ЛОЖЬ, зависящее от результата A<B,
// которое порождено процедурой с сигнатурой MySignature в порту MySyncPort.
```

Операция **catch** может быть частью принимающей части вызова или может использоваться для определения альтернативы в команде **alt**. Если операция **catch** используется в принимающей части операции **call**, то информация об имени порта и справочном номере сигнатуры для обозначения процедуры, которая породила это особое состояние, является избыточной, так как эта информация вытекает из операции **call**. Однако эта информация должна повторяться для удобства чтения (например, в случае сложных команд **call**).

Особые состояния определяются в виде типов и поэтому могут рассматриваться так же, как сообщения, например, чтобы различать значения одного и того же типа особых состояний, могут использоваться шаблоны.

```
MySyncPort.catch (MySignature, MyTemplate : {5, MyVar});
// Фиксирует особое состояние, порожденное процедурой с сигнатурой MySignature в порту
// MySyncPort, которое удовлетворяет условиям, определенным шаблоном MyTemplate с
// реальными параметрами 5 и MyVar.
```

Для операции **catch** требуются имя порта, критерии сопоставления для типа и значения, факультативное ограничение **from** (то есть выбор партнера по связи), а также факультативное присвоение сопоставляемого особого состояния и компонента **sender** переменным. Например:

```
MySyncPort.catch (MySignature, charstring : "Hello") from MyPartner;
// Фиксирует цепочку Международного алфавита № 5 "Hello", порожденную процедурой
// с сигнатурой MySignature в порту MySyncPort из объекта с адресом или справочным
// номером компонента, записанным в MyPartner.

MySyncPort.catch (MySignature, MyType :*) from MyPartner -> value MyVar;
// Фиксирует особое состояние с произвольным значением типа MyType (порожденное
// процедурой с сигнатурой MySignature в порту MySyncPort из компонента со справочным
// номером, записанным в переменной MyPartner), которое затем присваивается
// переменной MyVar. MyVar должна иметь тип MyType.

MySyncPort.catch (MySignature, MyTemplate (5, MyVarOne)) -> value MyVarTwo sender
   MyPartner;
// Фиксирует особое состояние, порожденное процедурой с сигнатурой MySignature со
// значением, которое удовлетворяет условиям, определенным шаблоном MyTemplate с
// реальными параметрами 5 и MyVarOne. Затем это особое состояние присваивается
// MyVarTwo. Адрес или справочный номер передающего объекта запрашивается
// операцией catch и присваивается MyPartner.
```

22.2.6.1 Особое состояние по тайм-ауту

Имеется одно особое состояние **timeout**, которое вводится операцией **catch**. Особое состояние **timeout** является вариантом действия на случай, когда вызванная процедура не отвечает и не порождает особого состояния в пределах заранее определенного времени. Например:

```
MyPort.catch (timeout);  // Вводит особое состояние timeout.
```

Фиксация особых состояний **timeout** ограничивается частью "обработка особого состояния" вызова. Для операции **catch**, которая обрабатывает особое состояние **timeout**, не разрешаются дальнейшие критерии сопоставления (включая часть **from**) и часть "присвоение".

22.2.6.2 Фиксация любого особого состояния

Операция **catch** без списка аргументов позволяет фиксировать любое действительное особое состояние. В наиболее общем случае не используется ключевое слово **from** и отсутствует часть "присвоение". Эта команда также вводит особое состояние **timeout**.

Например:

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

22.2.6.3 Фиксация в любом порту

Для фиксации особого состояния в любом порту используется ключевое слово **any**. Например:

```
any port.catch (timeout)
```

22.2.7 Операция Check

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.

```
MyAsyncPort.check (receive (integer : 5));
// Будет проверять целочисленное значение 5 в качестве верхнего сообщения в
// асинхронном порту MyAsyncPort.

MyPort.check (getcall (MyProc : {5, MyVar}) from MyPartner);
// Будет проверять вызов процедуры MyProc в MyCL (с параметрами in или inout 5 и MyVar)
// от корреспондента, адрес или справочный номер компонента которого записан в
// переменной MyPartner.

MyPort.check (getreply (MyProc : {5, MyVar) value 20));
// Проверяет ответ от процедуры MyProc в MyPort, в котором возвращаемое значение равно
// 20, а значения двух параметров out и inout равны 5 и значению MyVar.

MySyncPort.check (catch (MySignature, MyTemplate (5, MyVar)));
// Проверяет особое состояние, которое порождено процедурой с сигнатурой MySignature
// в порту MySyncPort и которое удовлетворяет условиям, определенным шаблоном MyTemplate
// с реальными параметрами 5 и MyVar.
```

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 делается копия верхнего элемента и над этой копией выполняется операция приема, указанная в операции **check**. Операция **check** будет неуспешной, если принимающая функция 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 следующая альтернатива для операции проверки. Операция **check** будет успешной, если принимающая функция 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 трансляции.

Пример:

```
MyPort.check (getreply (MyProc1 : {*, MyVar} value *) -> value MyReturnValue
   param (MyPar1));
// В этом примере возвращаемое значение присваивается переменной MyReturnValue,
// а значение первого параметра out или inout присваивается переменной MyPar1.

MyPort.check (getcall (MyProc : {5, MyVar}) from MyPartner -> param (MyPar1Var,
   MyPar2Var));
// В этом примере оба параметра процедуры MyProc считаются параметрами inout, а их
// значения присваиваются MyPar1Var и MyPar2Var.

MyPort.check (getcall (MyProc : {5, MyVar}) -> sender MySenderVar);
// Будет принят вызов процедуры MyProc в MyCL с параметрами in или inout 5 и MyVar.
// Вызывающая сторона запрошена и записана в MySenderVar.
```

22.2.7.1 Проверка любой операции

Операция **check** без списка аргументов позволяет проверять, есть ли в очереди входящего порта что-то, ожидающее обработки. Операция *CheckAny* 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**.

Пример:

```
MyPort.check;

MyPort.check (from MyPartner);

MyPort.check (-> sender MySenderVar);
```

22.3 Управление портами связи

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:** запуск опроса порта и разрешение доступа к порту;
- **stop:** остановка опроса порта и запрещение передающих операций в порту.

22.3.1 Портовая операция Clear

Операция **clear** удаляет содержимое *входящей* очереди в поименованном порту. Если очередь в порту уже пуста, то эта операция не будет выполнять каких-либо действий.

```
MyPort.clear; // Освобождает порт MyPort
```

22.3.2 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. Например:

```
MyPort.start; // Запускает MyPort
```

По умолчанию все порты компонента должны запускаться при выполнении запуска компонента.

22.3.3 Портовая операция Stop

Если порт определен в качестве порта, позволяющего операции приема, такие, как **receive, getcall**, то операция **stop** останавливает опрос поименованного порта. Если порт определен в качестве порта, позволяющего операции передачи, то порт, получивший **stop**, запрещает выполнение таких операций, как **send, call, raise** и т. п. Например:

```
MyPort.stop;  // Останавливает MyPort
```

22.4 Использование Any и All с портами

Ключевые слова **any** и **all** могут использоваться с конфигурациями операций, как показано в таблице 16.

Таблица 16/Z.140 – Any и All с портами

Операция	Разрешено		Пример
	any	**all**	
Приемные операции связи (**receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**)	Да		**any port.receive**
connect / **map**			
start		Да	**all port.start**
stop		Да	**all port.stop**
clear		Да	**all port.clear**

23 Таймерные операции

TTCN-3 поддерживает ряд таймерных операций (см. таблицу 17). Эти операции могут использоваться в тестовых примерах, функциях и в управлении модулем.

Таблица 17/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**

23.1 Таймерная операция Start

Таймерная операция **start** используется для указания на то, что необходимо запустить таймер. Значения таймеров должны иметь тип **float**. Например:

```
MyTimer1.start;          // MyTimer1 запущен с выдержкой по умолчанию.
MyTimer2.start (20E-3); // MyTimer2 запущен с выдержкой 20 мс.
```

Факультативный 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. Счетчик таймера считает от значения с плавающей запятой "нуль" (0,0) до максимума, установленного параметром "выдержка".

23.2 Таймерная операция Stop

Операция **stop** используется для остановки считающего таймера и для удаления его из списка считающих таймеров. Остановленный таймер становится пассивным, а его истекшее время устанавливается в значение с плавающей запятой "нуль" (0,0). Если именем таймера в операции **stop** является **all**, то все считающие (то есть активные) таймеры останавливаются. Например:

```
MyTimer1.stop;       // Останавливает MyTimer1
all timer.stop;      // Останавливает все считающие таймеры
```

Остановка пассивного таймера является действительной операцией, хотя она не дает какого-либо эффекта.

23.3 Таймерная операция Read

Операция **read** используется для запроса времени, которое прошло с момента запуска указанного таймера, и для записи его в указанную переменную. Эта переменная должна иметь тип **float**. Например:

```
var float Myvar;
MyVar := MyTimer1.read;   // Присвоить переменной MyVar время, которое
                          // прошло с момента запуска MyTimer1
```

Применение операции **read** к пассивному таймеру вернет значение в "нуль".

23.4 Таймерная операция Running

Операция **running** используется для проверки, считает ли таймер (то есть что он был запущен и что его выдержка не истекла, а он не был отменен). Эта операция выдает значение **true**, если таймер считает, а в остальных случаях – значение **false**. Например:

```
if (MyTimer1.running) { ... }
```

23.5 Событие Timeout

Операция **timeout** указывает тайм-аут для ранее запущенного таймера. Операция **timeout** может использоваться в альтернативах вместе с операциями **receive, getcall, getreply, catch** и другими таймерными операциями.

Пример:

```
MyTimer1.timeout;   // Проверяет тайм-аут ранее запущенного таймера
                    // MyTimer1
```

Ключевое слово **any** используется для указания **timeout** для любого таймера (а не явно поименованного таймера), запущенного в пределах этого тайм-аута. Например:

```
any timer.timeout;  // Проверяет тайм-аут любого ранее запущенного таймера
```

23.6 Использование Any и All с таймерами

Ключевые слова **any** и **all** могут использоваться с таймерными операциями, как показано в таблице 18.

Таблица 18/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**

24 Операции тестового вердикта

Операции вердикта (см. таблицу 19) позволяют устанавливать и запрашивать вердикты с помощью операций **set** и **get** соответственно. Эти операции используются только в тестовых примерах и функциях.

Таблица 19/Z.140 – Обзор операций тестового вердикта TTCN-3

Операции тестового вердикта	
Команда	**Команда**
Установить местный вердикт	**Verdict.set**
Прочитать местный вердикт	**Verdict.get**

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 реализации. Он используется для слежения за индивидуальным вердиктом каждого тестового компонента (то есть в MTC и во всех до единого PTC).

ПРИМЕЧАНИЕ. – В отличие от TTCN-2, присвоение окончательного вердикта не останавливает выполнение тестового компонента, в котором выполняется поведение. Если требуется, это делается явно с помощью команды stop.

24.1 Вердикт тестового примера

Кроме того, имеется глобальный вердикт, который обновляется при завершении выполнения каждого тестового компонента (то есть MTC и всех PTC). Этот вердикт не доступен для операций **get** и **set**. 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.

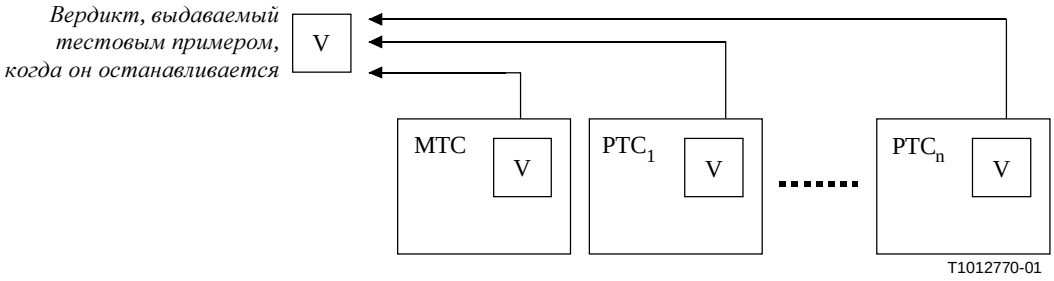


Рисунок 10/Z.140 – Иллюстрация отношений между вердиктами

ПРИМЕЧАНИЕ. – TTCN-3 не определяет реальные механизмы, которые производят обновление местного вердикта и вердикта тестового примера. Эти механизмы зависят от реализации.

24.2 Значения вердиктов и правила перезаписи

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 **set** используется только со значениями **pass**, **fail**, **inconc** и **none**. Например:

```
verdict.set (pass);
verdict.set (inconc);
```

Значение местного вердикта может быть вызвано с помощью операции **get**. Например:

```
MyResult := verdict.get; // где MyResult – это переменная типа verdicttype
```

Во время реализации тестового компонента создается его объект "местный вердикт", который устанавливается в значение **none**.

Когда значение данного вердикта изменяется (то есть используется операция **set**), результат этого изменения должен подчиняться правилам перезаписи, перечисленным в таблице 20. Вердикт тестового примера устанавливается неявно при окончании какого-либо тестового компонента. Результат такой неявной операции также подчиняется правилам перезаписи, перечисленным в таблице 20.

Таблица 20/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:

```
:
verdict.set(pass);  // Местный вердикт устанавливается в pass
:
verdict.set(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.
```

24.2.1 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 **set**. Никакое другое значение вердикта не может заменить вердикт **error**. Это означает, что вердикт **error** может быть только результатом операции тестового примера **execute**.

25 Операции SUT

В некоторых ситуациях тестирования, в которых может отсутствовать явный интерфейс к SUT, может оказаться необходимым заставить SUT инициировать определенные действия (например, передать сообщение тестовой системе).

Это действие может определяться в виде цепочки, например:

```
sut.action("Send MyTemplate on lower PCO");  // Неформальное описание
                                             // действия SUT
```

либо в виде ссылки на шаблон, который определяет структуру сообщения, передаваемого из SUT, например:

```
sut.action(MyTemplate);  // Это эквивалентно команде IMPLICIT SEND
                         // из TTCN-2.
```

В обоих случаях нет спецификации действий, выполняемых в направлении SUT или из SUT, чтобы запустить это действие, а лишь неформально описывается сама необходимая реакция.

Действия SUT могут указываться в тестовых примерах, функциях, именованных альтернативах и в управлении модулем.

26 Управляющая часть модуля

Тестовые примеры описываются в определениях модулей и выполняются при управлении модулем. Все переменные, таймеры и др. (если имеются), определенные в управляющей части модуля, переносятся в тестовый пример путем параметризации, если они должны использоваться в определении режима этого тестового примера, то есть TTCN-3 не поддерживает глобальные переменные любого вида.

Тестовая конфигурация при запуске каждого тестового примера должна сбрасываться в исходное состояние. Это означает, что все операции **create, connect** и т. п., которые могли выполняться в предыдущем тестовом примере, не будут "видны" для нового тестового примера.

26.1 Выполнение тестовых примеров

Тестовый пример вызывается с помощью команды **execute**. В результате выполнения тестового примера должен выдаваться один из тестовых вердиктов **none, pass, inconclusive, fail** или **error**, который может присваиваться переменной для дальнейшей обработки.

Факультативно команда **execute** позволяет контролировать тестовый пример посредством выдержки таймера. Если тестовый пример не закончился в пределах этой выдержки, то результатом выполнения данного тестового примера будет вердикт **error**, а тестовая система прекратит этот тестовый пример.

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".
```

26.2 Окончание тестовых примеров

Тестовый пример заканчивается с окончанием МТС. После окончания МТС все работающие параллельные тестовые компоненты останавливаются с помощью тестирования (то есть 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 24. Реальный местный вердикт тестового компонента становится его окончательным местным вердиктом, когда тестовый компонент остановился сам или когда он был остановлен посредством тестирования (то есть тестовой системой).

ПРИМЕЧАНИЕ 2. – Чтобы избежать кольцевых состояний при вычислении тестовых вердиктов из-за задержанных остановок разных РТС, МТС должен убедиться, что все РТС остановлены (посредством команды done), прежде чем остановиться самому.

26.3 Управление выполнением тестовых примеров

Для указания таких аспектов, как порядок выполнения тестов или возможное число запусков некоторого тестового примера, в управляющей части модуля могут использоваться программные команды, ограниченные теми, которые определены в таблице 11. Например:

```
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.

Тестовые примеры выдают одиночное значение типа **verdicttype**, так что возможно управлять порядком выполнения в зависимости от результата тестового примера. Например:

```
if (MySimpleTestCase() == pass) { log("Success!") }
```

26.4 Выбор тестового примера

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.

ПРИМЕЧАНИЕ. – Это эквивалентно выражениям выбора именованного теста из TTCN-2.

Пример:

```
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. Например:

```
:
function MyTestCaseGroup1()
{  execute(MySimpleTestCase1());
   execute(MySimpleTestCase2());
   execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{  execute(MySimpleTestCase4());
   execute(MySimpleTestCase5());
   execute(MySimpleTestCase6());
}
:
control
{  if (MySelectionExpression1()) { MyTestCaseGroup1(); }
   if (MySelectionExpression1()) { MyTestCaseGroup2(); }
   :
}
:
```

26.5 Использование таймеров при управлении

Таймеры могут использоваться для управления выполнением тестовых примеров. Это может быть сделано путем использования явного тайм-аута в команде execute. Например:

```
MyReturnVal := execute (MyTestCase(), 7E-3); // Переменная типа verdicttype
// Здесь выдаваемый вердикт будет error, если выполнение тестового примера
// не завершилось в пределах 7 мс
```

Можно также использовать таймерные операции. Например:

```
// 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;
:
execute(MyTestCase1());
T1.start;
T1.timeout; // Пауза перед выполнением следующего тестового примера
execute(MyTestCase2());
```

27 Определение атрибутов

Атрибуты связываются с элементами языка TTCN-3 посредством команды **with**. Синтаксис для аргумента команды **with** (то есть реальные атрибуты) определяется просто как нефиксированная текстовая цепочка.

Имеется три вида атрибутов:

a) **display:** позволяет определять атрибуты отображения, относящиеся к конкретному формату представления;

b) **encode:** позволяет ссылаться на конкретные правила кодирования;

c) **extension:** позволяет описывать атрибуты, определяемые пользователем.

27.1 Атрибуты отображения

Все элементы языка TTCN-3 могут иметь атрибуты **display**, чтобы указывать, как конкретные элементы языка должны отображаться, например, в некотором графическом формате.

Специальные цепочки 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 различно интерпретироваться в инструментах, предоставленных разными поставщиками, либо даже могут не поддерживаться.

27.2 Атрибуты кодирования

Правила кодирования определяют, как конкретное значение, шаблон и т. п. кодируется и передается, обычно в виде потока битов, через порт связи. 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.

Атрибут **encode** позволяет связать некоторое указанное правило кодирования или директиву по кодированию с определениями типов TTCN-3 (и только с определениями типов).

Специальные цепочки для атрибутов, относящиеся к атрибутам кодирования ASN.1, можно найти в Приложении E.

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 типов, а самым нижним уровнем – отдельный тип:

a) **module:** кодирование применяется ко всем типам, определенным в модуле, включая базовые типы TTCN-3;
b) **group:** кодирование применяется к группе определений типов, определяемых пользователем;
c) **type:** кодирование применяется к отдельному типу, определяемому пользователем;
d) **field**: кодирование применяется к полю в типе **record** или **set**.

Пример:

```
module MyTTCNmodule
{  :
   import type MyRecord from MySecondModule with {encode "MyRule 1"}
         // Все экземпляры MyRecord будут кодироваться согласно правилу MyRule 1
   :
   type charstring MyType;      // Обычно кодируется согласно определенному
                                // глобальному правилу
   :
   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 "Rule 2"}

}
with {encode "Global encoding rule"}
```

27.2.1 Недействительные правила кодирования

Если желательно указать недействительные правила 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.

27.3 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 могут не поддерживаться.

27.4 Контекст атрибутов

Команда **with** всегда связывает атрибуты с отдельными элементами языка. Можно также связывать атрибуты с несколькими элементами языка путем связывания команды **with** с единицей окружающего контекста или с группой элементов языка.

Команда **with** следует контекстным правилам, определенным в п. 5.4, то есть команда **with**, которая располагается в пределах контекста другой команды **with**, заменяет самую удаленную **with**. Это относится также к использованию команды **with** с группами. Следует проявлять осторожность, когда схема замены используется в сочетании со ссылками на отдельные определения. Общее правило состоит в том, что атрибуты должны присваиваться и перезаписываться в порядке их появления.

Пример:

```
// MyPDU1 будет отображаться как PDU
type record MyPDU1 { ... } with { display "PDU"}

// MyPDU2 будет отображаться как PDU с атрибутом расширения MyRule, зависящим от
// приложения
type record MyPDU2 { ... }
with
{
   display "PDU";
   extension "MyRule"
}

// Следующее определение группы . . .
group MyPDUs {
   type record MyPDU3 { ... }
   type record MyPDU4 { ... }
}
with {display "PDU"}   // Все типы группы MyPDUs будут отображаться как PDU.

// идентично следующему
group MyPDUs {
   type record MyPDU3 { ... } with { display "PDU"}
   type record MyPDU4 { ... } with { display "PDU"}
}

// Пример использования схемы перезаписи из команды 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 MySpecialRule, зависящий от
                                     // приложения.
}
with
{
   display "PDU";      // Все типы группы MyPDUs будут отображаться как PDU и
   extension "MyRule"; // (если не перезаписаны) иметь атрибут расширения MyRule
}

// идентично следующему . . .
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"
}
   }
}
```

27.5 Правила перезаписи для атрибутов

Определение атрибута в единице контекста более низкого уровня будет заменять общее определение атрибута в контексте более высокого уровня. Например:

```
type record MyRecordA
{
   :
} with {encode "RuleA"}

// Ниже показано, что MyRecordA кодируется согласно RuleA, а не RuleB
type record MyRecordB
{  :
   field MyRecordA
} with {encode "RuleB"}
```

Определение атрибута в контексте более низкого уровня может быть перезаписано в контексте более высокого уровня с помощью директивы **override**. Например:

```
type record MyRecordA
{
   :
} with {encode "RuleA"}

// Ниже показано, что MyRecordA кодируется согласно RuleB
type record MyRecordB
{  :
   fieldA   MyRecordA
} with {encode override "RuleB"}
```

Директива замены принуждает все содержащиеся типы во всех низких контекстах принудительно перейти к указанному атрибуту.

27.6 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 type MyType from MyModule with {display "ASP"} // MyType будет
                                                      // отображаться как ASP.

Import group MyGroup from MyModule with
{
   display "ASP";            // Все типы по умолчанию будут отображаться на ASP.
   extension "MyRule"
}
```

Приложение A

Язык BNF и статическая семантика

A.1 Язык BNF для TTCN-3

В данном Приложении определяется синтаксис TTCN-3 с использованием расширенной формы Бэкуса–Наура (далее называемой 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
"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.

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 */. Например:

```
/* Это блочный комментарий,
расположенный на двух строках */
```

Блочные комментарии не должны быть вложенными.

```
/* Это не является /* законным */ комментарием */
```

Строчные комментарии должны открываться парой символов / / и закрываться *<переводом строки>*. Например:

```
// Это - строчный комментарий,
// расположенный на двух строках
```

Строчные комментарии могут следовать за программными командами TTCN-3, но они не должны быть вложены в команду. Например:

```
// Нижеследующее – незаконно
const  // Это MyConst integer MyConst : = 1;

// Нижеследующее – законно
const integer MyConst : = 1; // Это MyConst
```

A.1.5 Терминалы 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 цепочки битов, цепочки шестнадцатеричных чисел, цепочки октетов	B H O
Показатель в числе с плавающей запятой	E

Ниже перечисляются специальные идентификаторы, зарезервированные для предопределенных функций, определенных в Приложении D.

```
int2char, char2int, int2unichar, unichar2int, bit2int, hex2int, int2bit,
int2hex, int2oct, int2str, oct2int, str2int, lengthof, sizeof, ischosen,
ispresent
```

Терминалы TTCN-3, перечисленные в таблице A.3, не должны использоваться в качестве идентификаторов в модуле TTCN-3. Эти терминалы пишутся только строчными буквами.

Таблица A.3/Z.140 – Список терминалов TTCN-3, являющихся зарезервированными словами

action	**fail**	**named**	**self**
activate	**false**	**none**	**send**
address	**float**	**nonrecursive**	**sender**
all	**for**	**not**	**set**
alt	**from**	**not4b**	**signature**
and	**function**	**nowait**	**start**
and4b		**null**	**stop**
any	**get**		**sut**
	getcall	**objid**	**system**
bitstring	**getreply**	**octetstring**	

Таблица A.3/Z.140 – Список терминалов TTCN-3, являющихся зарезервированными словами

boolean	**goto**	**of**	**template**
	group	**omit**	**testcase**
call		**on**	**timeout**
catch	**hexstring**	**optional**	**timer**
char		**or**	**to**
charstring	**if**	**or4b**	**trigger**
check	**ifpresent**	**out**	**true**
clear	**import**	**override**	**type**
complement	**in**		
component	**inconc**	**param**	**union**
connect	**infinity**	**pass**	**universal**
const	**inout**	**pattern**	**unmap**
control	**integer**	**port**	
create	**interleave**	**procedure**	**value**
			valueof
deactivate	**label**	**raise**	**var**
disconnect	**language**	**read**	**verdict**
display	**length**	**receive**	**verdicttype**
do	**log**	**record**	
done		**rem**	**while**
	map	**repeat**	**with**
else	**match**	**reply**	
encode	**message**	**return**	**xor**
enumerated	**mixed**	**running**	**xor4b**
error	**mod**	**runs**	
exception	**modifies**		
execute	**module**		
expand	**mtc**		
extension			
external			

A.1.6 Продукты BNF для синтаксиса TTCN-3

A.1.6.1 Модуль TTCN

```
1.  TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId [ModuleParList]
                BeginChar
                [ModuleDefinitionsPart]
                [ModuleControlPart]
                EndChar
                [WithStatement] [SemiColon]
2.  TTCN3ModuleKeyword ::= "module"
3.  TTCN3ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]
4.  ModuleIdentifier ::= Identifier
5.  DefinitiveIdentifier ::= Dot ObjectIdentifierKeyword "{"
                DefinitiveObjIdComponentList "}"
6.  DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+
7.  DefinitiveObjIdComponent ::= NameForm |
                DefinitiveNumberForm |
                DefinitiveNameAndNumberForm
```

```
8.  DefinitiveNumberForm ::= Number
9.  DefinitiveNameAndNumberForm ::= Identifier  "(" DefinitiveNumberForm ")"
10. ModuleParList ::= "(" ModulePar {"," ModulePar} ")"
11. ModulePar ::= [InParKeyword] ModuleParType ModuleParIdentifier
                  [AssignmentChar ConstantExpression]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Значение ConstantExpression должно иметь тот же тип,
что и установленный тип для Parameter */
12. ModuleParType ::= Type
13. ModuleParIdentifier ::= Identifier
```
12. ModuleParType ::= Type
13. ModuleParIdentifier ::= Identifier

A.1.6.2 Определяющая часть модуля

```
14. ModuleDefinitionsPart ::= ModuleDefinitionsList
15. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
16. ModuleDefinition ::= (TypeDef |
                  ConstDef |
                  TemplateDef |
                  FunctionDef |
                  SignatureDef |
                  TestcaseDef |
                  NamedAltDef |
                  ImportDef |
                  GroupDef |
                  ExtFunctionDef |
                  ExtConstDef) [WithStatement]
```

A.1.6.2.1 Определения Typedef

```
17. TypeDef ::= TypeDefKeyword TypeDefBody
18. TypeDefBody ::= StructuredTypeDef | SubTypeDef
19. TypeDefKeyword ::= "type"
20. StructuredTypeDef ::= RecordDef | UnionDef | SetDef | RecordOfDef |
                  SetOfDef | EnumDef | PortDef | ComponentDef
21. RecordDef ::= RecordKeyword StructDefBody
22. RecordKeyword ::= "record"
23. StructDefBody ::= (StructTypeIdentifier  [StructDefFormalParList] |
                  AddressKeyword)
                  BeginChar
                  [StructFieldDef {"," StructFieldDef}]
                  EndChar
24. StructTypeIdentifier ::= Identifier
25. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar}
                  ")"
26. StructDefFormalPar ::= FormalValuePar | FormalTypePar
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - FormalValuePar должен сводиться к параметру in */
27. StructFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
                   [OptionalKeyword]
28. StructFieldIdentifier ::= Identifier
29. OptionalKeyword ::= "optional"
30. UnionDef ::= UnionKeyword UnionDefBody
31. UnionKeyword ::= "union"
32. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] |
                  AddressKeyword)
                  BeginChar
                  UnionFieldDef {"," UnionFieldDef}
                  EndChar
33. UnionFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
34. SetDef ::= SetKeyword StructDefBody
35. SetKeyword ::= "set"
36. RecordOfDef ::= RecordKeyword OfKeyword [StringLength]  StructOfDefBody
37. OfKeyword ::= "of"
38. StructOfDefBody ::= Type (StructTypeIdentifier | AddressKeyword)
                   [SubTypeSpec]
39. SetOfDef ::= SetKeyword OfKeyword [StringLength]  StructOfDefBody
```

```
40. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
                BeginChar
                NamedValueList
                EndChar
41. EnumKeyword ::= "enumerated"
42. EnumTypeIdentifier ::= Identifier
43. NamedValueList ::= NamedValue {"," NamedValue}
44. NamedValue ::= NamedValueIdentifier ["(" Number ")"]
45. NamedValueIdentifier ::= Identifier
46. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef]
                 [SubTypeSpec]
47. SubTypeIdentifier ::= Identifier
48. SubTypeSpec ::= AllowedValues | StringLength
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - Эти значения должны быть того же типа, что и поле,
разбиваемое на подтипы */
49. AllowedValues ::= "(" ValueOrRange {"," ValueOrRange} ")"
50. ValueOrRange ::= IntegerRangeDef | SingleConstExpression
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - Продукт IntegerRangeDef должен использоваться только
с типами на базе целых чисел * /
51. IntegerRangeDef ::= LowerBound ".." UpperBound
52. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound]
                ")"
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - StringLength должна использоваться только с типами
String либо ограничиваться типами set of и record of * /
53. LengthKeyword ::= "length"
54. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
55. PortDef ::= PortKeyword PortDefBody
56. PortDefBody ::= PortTypeIdentifier PortDefAttribs
57. PortKeyword ::= "port"
58. PortTypeIdentifier ::= Identifier
59. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
60. MessageAttribs ::= MessageKeyword
                BeginChar
                {MessageList  [SemiColon]}+
                EndChar
61. MessageList ::= Direction AllOrTypeList
62. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
63. MessageKeyword ::= "message"
64. AllOrTypeList ::= AllKeyword | TypeList
65. AllKeyword ::= "all"
66. TypeList ::= Type {"," Type}
67. ProcedureAttribs ::= ProcedureKeyword
                BeginChar
                {ProcedureList  [SemiColon]}+
                EndChar
68. ProcedureKeyword ::= "procedure"
69. ProcedureList ::= Direction AllOrSignatureList
70. AllOrSignatureList ::= AllKeyword | SignatureList
71. SignatureList ::= Signature {"," Signature}
72. MixedAttribs ::= MixedKeyword
                BeginChar
                {MixedList  [SemiColon]}+
                EndChar
73. MixedKeyword ::= "mixed"
74. MixedList ::= Direction ProcOrTypeList
75. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
76. ProcOrType ::= Signature | Type
77. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                BeginChar
                [ComponentDefList]
                EndChar
78. ComponentKeyword ::= "component"
79. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
80. ComponentTypeIdentifier ::= Identifier
```

```
81. ComponentDefList ::= {ComponentElementDef [SemiColon]}+
82. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance |
                 ConstDef
83. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
84. PortElement ::= PortIdentifier [ArrayDef]
85. PortIdentifier ::= Identifier
```

A.1.6.2.2 Определения констант

```
86. ConstDef ::= ConstKeyword Type ConstList
87. ConstList ::= SingleConstDef  {"," SingleConstDef}
88. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar
              ConstantExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Значение ConstantExpression должно быть того же типа,
что и установленный тип для константы */
89. ConstKeyword ::= "const"
90. ConstIdentifier ::= Identifier
```

A.1.6.2.3 Определения шаблонов

```
91. TemplateDef ::= TemplateKeyword BaseTemplate  [DerivedDef]
              AssignmentChar TemplateBody
92. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["("
              TemplateFormalParList ")"]
93. TemplateKeyword ::= "template"
94. TemplateIdentifier ::= Identifier
95. DerivedDef ::= ModifiesKeyword TemplateRef
96. ModifiesKeyword ::= "modifies"
97. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
98. TemplateFormalPar ::= FormalValuePar |
              FormalTemplatePar
/* СТАТИЧЕСКАЯ СЕМАНТИКА - FormalValuePar должен сводиться к параметру in */
99. TemplateBody ::= SimpleSpec | FieldSpecList |
              ArrayValueOrAttrib
100. SimpleSpec ::= SingleValueOrAttrib
/* СТАТИЧЕСКАЯ СЕМАНТИКА - SimpleSpec не должно использоваться для сложных типов */
101. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
102. FieldSpec ::= FieldReference AssignmentChar TemplateBody
103. FieldReference ::= RecordRef | ArrayOrBitRef | ParRef
104. RecordRef ::= StructFieldIdentifier
105. ParRef ::= SignatureParIdentifier
/* ОПЕРАЦИОННАЯ СЕМАНТИКА - SignatureParIdentifier должен быть формальным
параметром Identifier из связанного определения сигнатуры */
106. SignatureParIdentifier ::= ValueParIdentifier
107. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - ArrayRef должен быть идентификатором, который
факультативно используется для типов array, для SET OF и SEQUENCE OF из ASN.1, а
также для record, record of, set и set of из TTCN. Эта же нотация может
использоваться для битовой ссылки внутри типа "цепочка битов" ASN.1 или TTCN */
108. FieldOrBitNumber ::= SingleExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - SingleExpression будет сводиться к значению типа
integer */
109. SingleValueOrAttrib ::= MatchingSymbol [ExtraMatchingAttributes] |
              SingleExpression [ExtraMatchingAttributes] |
              TemplateRefWithParList
/* СТАТИЧЕСКАЯ СЕМАНТИКА - VariableIdentifier (доступный с помощью
singleExpression) может использоваться только в "инлайновых" определениях шаблона
для переменных ссылочного типа в текущем контексте */
110. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
111. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
112. ArrayElementSpec ::= NotUsedSymbol | TemplateBody
113. NotUsedSymbol ::= Dash
```

```
114. MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList |
               IntegerRange | BitStringMatch | HexStringMatch |
               OctetStringMatch | CharStringMatch
115. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch
116. BitStringMatch ::= "'" {BinOrMatch} "'" B
117. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
118. HexStringMatch ::= "'" {HexOrMatch} "'" H
119. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
120. OctetStringMatch ::= "'" {OctOrMatch} "'" O
121. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
122. CharStringMatch ::= PatternKeyword CharStringPattern {StringOp
               CharStringPattern}
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Все CharStringPattern должны сводиться к одному и тому
же знаку или типу "цепочка знаков" */
123. CharStringPattern ::= CharStringValue | TemplateRefWithParList
124. PatternKeyword ::= "pattern"
125. Complement ::= ComplementKeyword (SingleConstExpression | ValueList)
126. ComplementKeyword ::= "complement"
127. Omit ::= OmitKeyword
128. OmitKeyword ::= "omit"
129. AnyValue ::= "?"
130. AnyOrOmit ::= "*"
131. ValueList ::= "(" SingleConstExpression {"," SingleConstExpression}+ ")"
132. LengthMatch ::= StringLength
133. IfPresentMatch ::= IfPresentKeyword
134. IfPresentKeyword ::= "ifpresent"
135. IntegerRange ::= "(" LowerBound ".." UpperBound ")"
136. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
137. UpperBound ::= SingleConstExpression | InfinityKeyword
138. InfinityKeyword ::= "infinity"
139. TemplateInstance ::= InLineTemplate
140. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier
               [TemplateActualParList] | TemplateParIdentifier
141. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier |
               TemplateParIdentifier
142. InLineTemplate ::= [(Type | Signature) Colon] [DerivedDef AssignmentChar]
               TemplateBody
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - Поле типа может быть опущено только в том случае,
когда тип неявно однозначен * /
143. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
144. TemplateActualPar ::= TemplateInstance
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - Когда соответствующий формальный параметр не является
типом шаблона, продукт TemplateInstance должен сводиться к одному или нескольким
SingleExpression * /
145. TemplateOps ::= MatchOp | ValueofOp
146. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - Тип значения, выдаваемого выражением, должен быть тем
же, что и тип шаблона, а каждое поле шаблона должно сводиться к одиночному
значению * /
147. MatchKeyword ::= "match"
148. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
149. ValueofKeyword ::= "valueof"
```

A.1.6.2.4 Определения функций

```
150. FunctionDef ::= FunctionKeyword FunctionIdentifier
               "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
               BeginChar
               FunctionBody
               EndChar
151. FunctionKeyword ::= "function"
152. FunctionIdentifier ::= Identifier
153. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
```

```
154. FunctionFormalPar ::= FormalValuePar |
               FormalTimerPar |
               FormalTemplatePar |
               FormalPortPar
155. ReturnType ::= ReturnKeyword Type
156. ReturnKeyword ::= "return"
157. RunsOnSpec ::= RunsKeyword OnKeyword (ComponentType | MTCKeyword)
158. RunsKeyword ::= "runs"
159. OnKeyword ::= "on"
160. MTCKeyword ::= "mtc"
161. FunctionBody ::= [FunctionStatementOrDefList]
162. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
163. FunctionStatementOrDef ::= FunctionLocalDef |
               FunctionLocalInst |
               FunctionStatement
164. FunctionLocalInst ::= VarInstance |
               TimerInstance
165. FunctionLocalDef ::= ConstDef
166. FunctionStatement ::= ConfigurationStatements |
               TimerStatements |
               CommunicationStatements |
               BasicStatements |
               BehaviourStatements |
               VerdictStatements |
               SUTStatements
167. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
168. FunctionRef ::= [GlobalModuleId Dot] FunctionIdentifier
169. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
170. FunctionActualPar ::= TimerRef |
               TemplateInstance |
               Port |
               ComponentRef
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Когда соответствующий формальный параметр не является
типом шаблона, продукт TemplateInstance должен сводиться к одному или нескольким
SingleExpression, то есть должен быть эквивалентен продукту Expression * /
```

A.1.6.2.5 Определения сигнатур

```
171. SignatureDef ::= SignatureKeyword SignatureIdentifier
               "("[SignatureFormalParList] ")" [ReturnType]
               [ExceptionSpec]
172. SignatureKeyword ::= "signature"
173. SignatureIdentifier ::= Identifier
174. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
175. SignatureFormalPar ::= FormalValuePar
176. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
177. ExceptionKeyword ::= "exception"
178. ExceptionTypeList ::= Type {"," Type}
179. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.2.6 Определения тестовых примеров

```
180. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
               "("[TestcaseFormalParList] ")" ConfigSpec
               BeginChar
               FunctionBody
               EndChar
181. TestcaseKeyword ::= "testcase"
182. TestcaseIdentifier ::= Identifier
183. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
184. TestcaseFormalPar ::= FormalValuePar |
               FormalTemplatePar
185. ConfigSpec ::= RunsOnSpec [SystemSpec]
186. SystemSpec ::= SystemKeyword ComponentType
```

```
187. SystemKeyword ::= "system"
188. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "("
                [TestcaseActualParList] ")" ["," TimerValue]   ")"
189. ExecuteKeyword ::= "execute"
190. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
191. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
192. TestcaseActualPar ::=
               TemplateInstance
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Когда соответствующий формальный параметр не является
типом шаблона, продукт TemplateInstance должен сводиться к одному или нескольким
SingleExpression, то есть должен быть эквивалентен продукту Expression */
```

A.1.6.2.7 Определения NamedAlt

```
193. NamedAltDef ::= NamedKeyword AltKeyword NamedAltIdentifier
               "("  [NamedAltFormalParList] ")"
               BeginChar
               AltGuardList                    EndChar
194. NamedKeyword ::= "named"
195. NamedAltIdentifier ::= Identifier
196. NamedAltFormalParList ::= NamedAltFormalPar {"," NamedAltFormalPar}
197. NamedAltFormalPar ::= FormalValuePar |
               FormalTimerPar |
               FormalTemplatePar |
               FormalPortPar
198. NamedAltInstance ::= NamedAltRef  "(" [NamedAltActualParList]")"
199. NamedAltRef ::= [GlobalModuleId Dot] NamedAltIdentifier
200. NamedAltActualParList ::= NamedAltActualPar {"," NamedAltActualPar}
201. NamedAltActualPar ::=
               TimerRef |
               TemplateInstance |
               Port |
               ComponentRef
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Когда соответствующий формальный параметр не является
типом шаблона, продукт TemplateInstance должен сводиться к одному или нескольким
SingleExpression, то есть должен быть эквивалентен продукту Expression */
```

A.1.6.2.8 Определения импорта

```
202. ImportDef ::= ImportKeyword ImportSpec
203. ImportKeyword ::= "import"
204. ImportSpec ::= ImportAllSpec |
               ImportGroupSpec |
               ImportTypeDefSpec |
               ImportTemplateSpec |
               ImportConstSpec |
               ImportTestcaseSpec |
               ImportNamedAltSpec |
               ImportFunctionSpec |
               ImportSignatureSpec
205. ImportAllSpec ::= AllKeyword [DefKeyword] ImportFromSpec
206. ImportFromSpec ::= FromKeyword ModuleId [NonRecursiveKeyword]
207. ModuleId ::= GlobalModuleId [LanguageSpec]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - LanguageSpec может быть опущен только в том случае,
когда указанный модуль содержит нотацию TTCN-3 */
208. LanguageKeyword ::= "language"
209. LanguageSpec ::= LanguageKeyword  FreeText
210. GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]
```

```
211. DefKeyword ::= TypeDefKeyword |
                    ConstKeyword |
                    TemplateKeyword |
                    TestcaseKeyword |
                    FunctionKeyword |
                    SignatureKeyword |
                    NamedKeyword AltKeyword
212. NonRecursiveKeyword ::= "nonrecursive"
213. ImportGroupSpec ::= GroupKeyword GroupIdentifier {"," GroupIdentifier}
                    ImportFromSpec
214. ImportTypeDefSpec ::= TypeDefKeyword TypeDefIdentifier {","
                    TypeDefIdentifier} ImportFromSpec
215. TypeDefIdentifier ::= StructTypeIdentifier |
                    EnumTypeIdentifier |
                    PortTypeIdentifier |
                    ComponentTypeIdentifier |
                    SubTypeIdentifier
216. ImportTemplateSpec ::= TemplateKeyword TemplateIdentifier {","
                    TemplateIdentifier} ImportFromSpec
217. ImportConstSpec ::= ConstKeyword ConstIdentifier {"," ConstIdentifier}
                    ImportFromSpec
218. ImportTestcaseSpec ::= TestcaseKeyword TestcaseIdentifier {","
                    TestcaseIdentifier} ImportFromSpec
219. ImportFunctionSpec ::= FunctionKeyword FunctionIdentifier {","
                    FunctionIdentifier} ImportFromSpec
220. ImportSignatureSpec ::= SignatureKeyword SignatureIdentifier {","
                    SignatureIdentifier} ImportFromSpec
221. ImportNamedAltSpec ::= NamedKeyword AltKeyword NamedAltIdentifier {","
                    NamedAltIdentifier} ImportFromSpec
```

A.1.6.2.9 Определения групп

```
222. GroupDef ::= GroupKeyword GroupIdentifier
                    BeginChar
                    [ModuleDefinitionsPart]
                    EndGroupChar
223. GroupKeyword ::= "group"
224. EndGroupChar ::= "}"
225. GroupIdentifier ::= Identifier
```

A.1.6.2.10 Определения внешних функций

```
226. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
                    "("[FunctionFormalParList] ")" [ReturnType]
227. ExtKeyword ::= "external"
228. ExtFunctionIdentifier ::= Identifier
```

A.1.6.2.11 Определения внешних констант

```
229. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
230. ExtConstIdentifier ::= Identifier
```

A.1.6.3 Управляющая часть

```
231. ModuleControlPart ::= ControlKeyword
                    BeginChar
                    ModuleControlBody
                    EndChar
                    [WithStatement] [SemiColon]
232. ControlKeyword ::= "control"
233. ModuleControlBody ::= [ControlStatementOrDefList]
234. ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
```

```
235. ControlStatementOrDef ::= FunctionLocalInst |
               ControlStatement |
               FunctionLocalDef
236. ControlStatement ::= TimerStatements |
               BasicStatements |
               BehaviourStatements |
               SUTStatements
```

A.1.6.3.1 Экземпляры переменных

```
237. VarInstance ::= VarKeyword Type VarList
238. VarList ::= SingleVarInstance {"," SingleVarInstance}
239. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar
               VarInitialValue]
240. VarInitialValue ::= Expression
241. VarKeyword ::= "var"
242. VarIdentifier ::= Identifier
243. VariableRef ::= (VarIdentifier | ValueParIdentifier)
               [ExtendedFieldReference]
```

A.1.6.3.2 Экземпляры таймеров

```
244. TimerInstance ::= TimerKeyword TimerIdentifier [ArrayDef]
               [AssignmentChar TimerValue]
245. TimerKeyword ::= "timer"
246. TimerIdentifier ::= Identifier
247. TimerValue ::= SingleExpression
/* СТАТИСТИЧЕСКАЯ СЕМАНТИКА - SingleExpression должно сводиться к значению типа
float */
248. TimerRef ::= TimerIdentifier [ArrayOrBitRef]|
               TimerParIdentifier [ArrayOrBitRef]
```

A.1.6.3.3 Операции компонентов

```
249. ConfigurationStatements ::= ConnectStatement |
               MapStatement |
               DisconnectStatement |
               UnmapStatement |
               DoneStatement |
               StartTCStatement |
               StopTCStatement
250. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp
251. CreateOp ::= ComponentType Dot CreateKeyword
252. SystemOp ::= "system"
253. SelfOp ::= "self"
254. MTCOp ::= MTCKeyword
255. DoneStatement ::= ComponentId Dot DoneKeyword
256. ComponentId ::= ComponentIdentifier | (AnyKeyword | AllKeyword)
               ComponentKeyword
257. DoneKeyword ::= "done"
258. RunningOp ::= ComponentId Dot RunningKeyword
259. RunningKeyword ::= "running"
260. CreateKeyword ::= "create"
261. ConnectStatement ::= ConnectKeyword PortSpec
262. ConnectKeyword ::= "connect"
263. PortSpec ::= "(" PortRef "," PortRef ")"
264. PortRef ::= ComponentRef Colon Port
265. ComponentRef ::= ComponentIdentifier | SystemOp | SelfOp | MTCOp
266. DisconnectStatement ::= DisconnectKeyword PortSpec
267. DisconnectKeyword ::= "disconnect"
268. MapStatement ::= MapKeyword PortSpec
269. MapKeyword ::= "map"
270. UnmapStatement ::= UnmapKeyword PortSpec
271. UnmapKeyword ::= "unmap"
```

```
272. StartTCStatement ::= ComponentIdentifier Dot StartKeyword "("
                FunctionInstance ")"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - FunctionInstance может иметь только параметры in */
273. StartKeyword ::= "start"
274. StopTCStatement ::= StopKeyword
275. ComponentIdentifier ::= VariableRef | FunctionInstance
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Переменная, связанная с VariableRef, или тип Return,
связанный с FunctionInstance, должны быть типа компонента */
```

A.1.6.3.4 Портовые операции

```
276. Port ::= (PortIdentifier | PortParIdentifier) [ArrayOrBitRef]
277. CommunicationStatements ::= SendStatement | CallStatement | ReplyStatement |
RaiseStatement |
                ReceiveStatement | TriggerStatement | GetCallStatement |
                GetReplyStatement | CatchStatement | CheckStatement |
                ClearStatement | StartStatement | StopStatement
278. SendStatement ::= Port Dot PortSendOp
279. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
280. SendOpKeyword ::= "send"
281. SendParameter ::= TemplateInstance
282. ToClause ::= ToKeyword AddressRef
283. ToKeyword ::= "to"
284. AddressRef ::= VariableRef | FunctionInstance
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - Выдаваемые VariableRef и FunctionInstance должны быть
типа адреса или компонента * /
285. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
286. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
287. CallOpKeyword ::= "call"
288. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Только параметры out могут быть опущены или определены
с атрибутом сопоставления */
289. CallTimerValue ::= TimerValue | NowaitKeyword
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - Значение должно быть типа float * /
290. NowaitKeyword ::= "nowait"
291. PortCallBody ::= BeginChar
                CallBodyStatementList
                EndChar
292. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
293. CallBodyStatement ::= CallBodyGuard  StatementBlock
294. CallBodyGuard ::= AltGuardChar CallBodyOps
295. CallBodyOps ::= GetReplyStatement | CatchStatement
296. ReplyStatement ::= Port Dot PortReplyOp
297. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")"
                [ToClause]
298. ReplyKeyword ::= "reply"
299. ReplyValue ::= ValueKeyword Expression
300. RaiseStatement ::= Port Dot PortRaiseOp
301. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")"
                [ToClause]
302. RaiseKeyword ::= "raise"
303. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
304. PortOrAny ::= Port | AnyKeyword PortKeyword
305. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                [PortRedirect]
/ * СТАТИЧЕСКАЯ СЕМАНТИКА - Вариант PortRedirect может присутствовать только в
том случае, когда присутствует также вариант ReceiveParameter * /
306. ReceiveOpKeyword ::= "receive"
307. ReceiveParameter ::= TemplateInstance
308. FromClause ::= FromKeyword AddressRef
309. FromKeyword ::= "from"
310. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
311. PortRedirectSymbol ::= "->"
312. ValueSpec ::= ValueKeyword VariableRef
```

```
313. ValueKeyword ::= "value"
314. SenderSpec ::= SenderKeyword VariableRef
/* СТАТИЧЕСКАЯ СЕМАНТИКА – Переменная ref должна быть типа адреса или компонента */
315. SenderKeyword ::= "sender"
316. TriggerStatement ::= PortOrAny Dot PortTriggerOp
317. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                [PortRedirect]
/* СТАТИЧЕСКАЯ СЕМАНТИКА – Вариант PortRedirect может присутствовать только в том
случае, когда присутствует также вариант ReceiveParameter */
318. TriggerOpKeyword ::= "trigger"
319. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
320. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                [PortRedirectWithParam]
/* СТАТИЧЕСКАЯ СЕМАНТИКА – Вариант PortRedirectWithParam может присутствовать
только в том случае, когда присутствует также вариант ReceiveParameter */
321. GetCallOpKeyword ::= "getcall"
322. PortRedirectWithParam ::= PortRedirectSymbol RedirectSpec
323. RedirectSpec ::= ValueSpec  [ParaSpec] [SenderSpec] |
                ParaSpec [SenderSpec] |
                SenderSpec
324. ParaSpec ::= ParaKeyword ParaAssignmentList
325. ParaKeyword ::= "param"
326. ParaAssignmentList ::= "(" (AssignmentList | VariableList) ")"
327. AssignmentList ::= VariableAssignment {"," VariableAssignment}
328. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* СТАТИЧЕСКАЯ СЕМАНТИКА – ParameterIdentifier должен быть из соответствующего
определения сигнатуры * /
329. ParameterIdentifier ::= ValueParIdentifier |
                TimerParIdentifier |
                TemplateParIdentifier |
                PortParIdentifier
330. VariableList ::= VariableEntry  {"," VariableEntry}
331. VariableEntry ::= VariableRef | NotUsedSymbol
332. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
333. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec]
                ")"] [FromClause] [PortRedirectWithParam]
/* СТАТИСТИЧЕСКАЯ СЕМАНТИКА - Вариант PortRedirectWithParam может присутствовать
только в том случае, когда присутствует также вариант ReceiveParameter */
334. GetReplyOpKeyword ::= "getreply"
335. ValueMatchSpec ::= ValueKeyword TemplateInstance
336. CheckStatement ::= PortOrAny Dot PortCheckOp
337. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
338. CheckOpKeyword ::= "check"
339. CheckParameter ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp |
                PortCatchOp | [FromClause] [PortRedirectSymbol SenderSpec]
340. CatchStatement ::= PortOrAny Dot PortCatchOp
341. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause]
                [PortRedirect]
/* СТАТИСТИЧЕСКАЯ СЕМАНТИКА – Вариант PortRedirect может присутствовать только в
том случае, когда присутствует также вариант CatchOpParameter */
342. CatchOpKeyword ::= "catch"
343. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
344. ClearStatement ::= PortOrAll Dot PortClearOp
345. PortOrAll ::= Port | AllKeyword PortKeyword
346. PortClearOp ::= ClearOpKeyword
347. ClearOpKeyword ::= "clear"
348. StartStatement ::= PortOrAll Dot PortStartOp
349. PortStartOp ::= StartKeyword
350. StopStatement ::= PortOrAll Dot PortStopOp
351. PortStopOp ::= StopKeyword
352. StopKeyword ::= "stop"
353. AnyKeyword ::= "any"
```

A.1.6.3.5 Таймерные операции

```
354. TimerStatements ::= StartTimerStatement | StopTimerStatement |
                TimeoutStatement
355. TimerOps ::= ReadTimerOp | RunningTimerOp
356. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
357. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
358. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
359. ReadTimerOp ::= TimerRef Dot ReadKeyword
360. ReadKeyword ::= "read"
361. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
362. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
363. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
364. TimeoutKeyword ::= "timeout"
```

A.1.6.4 Типы

```
365. Type ::= PredefinedType | ReferencedType
366. PredefinedType ::= BitStringKeyword |
                BooleanKeyword |
                CharStringKeyword |
                UniversalCharString |
                CharKeyword |
                UniversalChar |
                IntegerKeyword |
                OctetStringKeyword |
                ObjectIdentifierKeyword |
                HexStringKeyword |
                VerdictKeyword |
                FloatKeyword |
                AddressKeyword
367. BitStringKeyword ::= "bitstring"
368. BooleanKeyword ::= "boolean"
369. IntegerKeyword ::= "integer"
370. OctetStringKeyword ::= "octetstring"
371. ObjectIdentifierKeyword ::= "objid"
372. HexStringKeyword ::= "hexstring"
373. VerdictKeyword ::= "verdict"
374. FloatKeyword ::= "float"
375. AddressKeyword ::= "address"
376. CharStringKeyword ::= "charstring"
377. UniversalCharString ::= UniversalKeyword CharStringKeyword
378. UniversalKeyword ::= "universal"
379. CharKeyword ::= "char"
380. UniversalChar ::= UniversalKeyword CharKeyword
381. ReferencedType ::= [GlobalModuleId Dot] TypeReference
                [ExtendedFieldReference]
382. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                EnumTypeIdentifier |
                SubTypeIdentifier |
                TypeParIdentifier |
                ComponentTypeIdentifier
383. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
384. TypeActualPar ::= SingleConstExpression | Type
```

A.1.6.4.1 Типы массива

```
385. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
386. ArrayBounds ::= SingleConstExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА - ArrayBounds будет сводиться к неотрицательному
значению целого числа * /
```

A.1.6.5 Значения

```
387. Value ::= PredefinedValue | ReferencedValue
388. PredefinedValue ::= BitStringValue |
                BooleanValue |
                CharStringValue |
                IntegerValue |
                OctetStringValue |
                ObjectIdentifierValue |
                HexStringValue |
                VerdictValue |
                EnumeratedValue |
                FloatValue |
                AddressValue
389. BitStringValue ::= Bstring
390. BooleanValue ::= "true" | false
391. IntegerValue ::= Number
392. OctetStringValue ::= Ostring
393. ObjectIdentifierValue ::= ObjectIdentifierKeyword  "{" ObjIdComponentList
                "}"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - ReferencedValue должно быть типа идентификатора объекта */
394. ObjIdComponentList ::= {ObjIdComponent}+
395. ObjIdComponent ::= NameForm |
                NumberForm |
                NameAndNumberForm
396. NumberForm ::= Number | ReferencedValue
/* СТАТИЧЕСКАЯ СЕМАНТИКА - ReferencedValue должно быть типа integer и иметь
неотрицательное значение * /
397. NameAndNumberForm ::= Identifier  NumberForm
398. NameForm ::= Identifier
399. HexStringValue ::= Hstring
400. VerdictValue ::= "pass" | fail | inconc | none | error
401. EnumeratedValue ::= NamedValueIdentifier
402. CharStringValue ::= Cstring | Quadruple | ReferencedValue
/* СТАТИЧЕСКАЯ СЕМАНТИКА - ReferencedValue должно сводиться к типу "цепочка" * /
403. Quadruple ::= "(" Group "," Plane "," Row "," Cell ")"
404. Group ::= Number
405. Plane ::= Number
406. Row ::= Number
407. Cell ::= Number
408. FloatValue ::= FloatDotNotation | FloatENotation
409. FloatDotNotation ::= Number Dot DecimalNumber
410. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
411. Exponential ::= E
412. ReferencedValue ::= ValueReference [ExtendedFieldReference]
413. ValueReference ::= [GlobalModuleId Dot] ConstIdentifier |
                ExtConstIdentifier |
                ValueParIdentifier |
                ModuleParIdentifier |
                VarIdentifier
414. Number ::= (NonZeroNum {Num}) | 0
415. NonZeroNum ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
416. DecimalNumber ::= {Num}
417. Num ::= 0 | NonZeroNum
418. Bstring ::= "'" {Bin} "'" B
419. Bin ::= 0 | 1
420. Hstring ::= "'" {Hex} "'" H
421. Hex ::= Num | A | B | C | D | E | F| a | b | c | d | e | f
422. Ostring ::= "'" {Oct} "'" O
423. Oct ::= Hex Hex
424. Cstring ::= """ {Char} """
425. Char ::= /* КОММЕНТАРИЙ - Знак, определенный соответствующим типом
                CharacterString * /
426. Identifier ::= Alpha{AlphaNum | Underscore}
```

```
427. Alpha ::= UpperAlpha | LowerAlpha
428. AlphaNum ::= Alpha | Num
429. 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
430. 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
431. ExtendedAlphaNum ::= /* КОММЕНТАРИЙ – Знак из любого набора знаков,
              определенного в стандарте ИСО/МЭК 10646-1 */
432. FreeText ::= """ {ExtendedAlphaNum} """
433. AddressValue ::= "null"
```

A.1.6.6 Параметризация

```
434. InParKeyword ::= "in"
435. OutParKeyword ::= "out"
436. InOutParKeyword ::= "inout"
437. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type
              ValueParIdentifier
438. ValueParIdentifier ::= Identifier
439. FormalTypePar ::= [InParKeyword] TypeParIdentifier
440. TypeParIdentifier ::= Identifier
441. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
442. PortParIdentifier ::= Identifier
443. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
444. TimerParIdentifier ::= Identifier
445. FormalTemplatePar ::= [InParKeyword] TemplateKeyword Type
              TemplateParIdentifier
446. TemplateParIdentifier ::= Identifier
```

A.1.6.7 Команда With

```
447. WithStatement ::= WithKeyword WithAttribList
448. WithKeyword ::= "with"
449. WithAttribList ::= "{" MultiWithAttrib "}"
450. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}+
451. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier]
              AttribSpec
452. AttribKeyword ::= EncodeKeyword |
              DisplayKeyword |
              ExtensionKeyword
453. EncodeKeyword ::= "encode"
454. DisplayKeyword ::= "display"
455. ExtensionKeyword ::= "extension"
456. OverrideKeyword ::= "override"
457. AttribQualifier ::= "(" DefOrFieldRefList ")"
458. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
459. DefOrFieldRef ::= DefinitionRef | FieldReference
460. DefinitionRef ::= StructTypeIdentifier |
              EnumTypeIdentifier |
              PortTypeIdentifier |
              ComponentTypeIdentifier |
              SubTypeIdentifier |
              ConstIdentifier |
              TemplateIdentifier |
              NamedAltIdentifier |
              TestcaseIdentifier |
              FunctionIdentifier |
              SignatureIdentifier
461. AttribSpec ::= FreeText
```

A.1.6.8 Команды поведения

```
462. BehaviourStatements ::= TestcaseInstance |
                FunctionInstance |
                ReturnStatement |
                AltConstruct |
                InterleavedConstruct |
                LabelStatement |
                GotoStatement |
                ActivateStatement |
                DeactivateStatement |
                NamedAltInstance
/* СТАТИЧЕСКАЯ СЕМАНТИКА - TestcaseInstance не должен вызываться из существующего
выполняемого тестового примера или цепочки функций, вызванной из какого-либо
тестового примера, то есть тестовые примеры могут быть реализованы только из
управляющей части или из функций, вызванных непосредственно из управляющей части * /
463. VerdictStatements ::= SetLocalVerdict
464. VerdictOps ::= GetLocalVerdict
465. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* СТАТИЧЕСКАЯ СЕМАНТИКА - SingleExpression должно сводиться к значению типа
вердикта * /
/* СТАТИЧЕСКАЯ СЕМАНТИКА - SetLocalVerdict не должен использоваться для
присвоения значения "ошибка" * /
466. SetVerdictKeyword ::= VerdictKeyword Dot SetKeyword
467. GetLocalVerdict ::= VerdictKeyword Dot GetKeyword
468. GetKeyword ::= "get"
469. SUTStatements ::= SUTAction "(" (FreeText | TemplateRefWithParList)  ")"
470. SUTAction ::= SUTKeyword Dot ActionKeyword
471. SUTKeyword ::= "sut"
472. ActionKeyword ::= "action"
473. ReturnStatement ::= ReturnKeyword  [Expression]
474. AltConstruct ::= AltKeyword BeginChar AltGuardList EndChar
475. AltKeyword ::= "alt"
476. AltGuardList ::= {AltGuardElement [SemiColon]}+ [ElseStatement [SemiColon]]
477. AltGuardElement ::= GuardStatement | ExpandStatement
478. GuardStatement ::= AltGuardChar GuardOp  StatementBlock
479. ExpandStatement ::= "["ExpandKeyword "]" NamedAltInstance
480. ElseStatement ::= "["ElseKeyword "]" StatementBlock
481. ExpandKeyword ::= "expand"
482. AltGuardChar ::= "[" [BooleanExpression] "]"
483. GuardOp ::= TimeoutStatement | ReceiveStatement | TriggerStatement |
                GetCallStatement | CatchStatement | CheckStatement |
                GetReplyStatement | DoneStatement
/* СТАТИЧЕСКАЯ СЕМАНТИКА - GuardOp, используемый в управляющей части модуля,
должен содержать только TimeoutStatement * /
484. StatementBlock ::= BeginChar [FunctionStatementOrDefList] EndChar
485. InterleavedConstruct ::= InterleavedKeyword BeginChar InterleavedGuardList
                EndChar
486. InterleavedKeyword ::= "interleave"
487. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
488. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
489. InterleavedGuard ::= "[" "]" GuardOp
490. InterleavedAction ::= StatementBlock
/* СТАТИЧЕСКАЯ СЕМАНТИКА - StatementBlock не может содержать циклических команд,
goto, activate, deactivate, stop, return или вызовов функций * /
491. LabelStatement ::= LabelKeyword LabelIdentifier
492. LabelKeyword ::= "label"
493. LabelIdentifier ::= Identifier
494. GotoStatement ::= GotoKeyword (LabelIdentifier | AltKeyword)
/* СТАТИЧЕСКАЯ СЕМАНТИКА - Вариант AltKeyword может использоваться только в
конструкции ALT */
495. GotoKeyword ::= "goto"
496. ActivateStatement ::= ActivateKeyword "(" NamedAltList ")"
497. ActivateKeyword ::= "activate"
```

```
498. NamedAltList ::= NamedAltInstance {"," NamedAltInstance}
499. DeactivateStatement ::= DeactivateKeyword ["(" NamedAltRefList ")"]
500. DeactivateKeyword ::= "deactivate"
501. NamedAltRefList ::= NamedAltRef {"," NamedAltRef}
```

A.1.6.9 Базовые команды

```
502. BasicStatements ::= Assignment | LogStatement | LoopConstruct |
                ConditionalConstruct
503. Expression ::= SingleExpression | CompoundExpression
/ * СТАТИЧЕСКАЯ СЕМАНТИКА – Выражение не должно содержать конфигурации или
операции вердикта в управляющей части модуля * /
504. CompoundExpression ::= FieldExpressionList | ArrayExpression
505. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec}
                "}"
506. FieldExpressionSpec ::= FieldReference AssignmentChar Expression
507. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
508. ArrayElementExpressionList ::= NotUsedOrExpression  {","
                NotUsedOrExpression}
509. NotUsedOrExpression ::= Expression | NotUsedSymbol
510. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
511. SingleConstExpression ::= SingleExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА – SingleConstExpression не должно содержать переменных или
параметров модуля и должно сводиться к постоянному значению во время трансляции */
512. BooleanExpression ::= SingleExpression
/* СТАТИЧЕСКАЯ СЕМАНТИКА – BooleanExpression должно сводиться к значению типа
boolean * /
513. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
514. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {","
                FieldConstExpressionSpec} "}"
515. FieldConstExpressionSpec ::= FieldReference AssignmentChar
                ConstantExpression
516. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
517. ArrayElementConstExpressionList ::= ConstantExpression {","
                ConstantExpression}
518. Assignment ::= VariableRef  ":=" Expression
/* ОПЕРАЦИОННАЯ СЕМАНТИКА –Expressions в правой части Assignment должно
определять явное значение типа для левой части * /
519. SingleExpression ::= SimpleExpression {BitOp SimpleExpression}
/* ОПЕРАЦИОННАЯ СЕМАНТИКА – Если имеются как SimpleExpressions, так и BitOp, то
SimpleExpressions должны определять конкретные значения совместимых типов * /
520. SimpleExpression ::= SubExpression [RelOp SubExpression]
/* ОПЕРАЦИОННАЯ СЕМАНТИКА – Если имеются как SubExpressions, так и RelOp, то
SubExpressions должны определять конкретные значения совместимых типов * /
/* ОПЕРАЦИОННАЯ СЕМАНТИКА – Если RelOp является "<"|">"|">="|"<=", то каждое
SubExpression должно определять конкретное значение integer, Enumerated или float
(эти значения могут быть значениями TTCN или ASN.1) * /
521. SubExpression ::= Product [ShiftOp Product]
/* ОПЕРАЦИОННАЯ СЕМАНТИКА – Каждый Product должен сводиться к конкретному
значению. Если присутствуют более одного Product, то правый операнд должен быть
типа integer, а если ShiftOp является '< <' или '> >', то левый операнд должен
сводиться к типу bitstring, hexstring, octetstring или integer. Если ShiftOp
является '< @' или '@ >', то левый операнд должен быть типа bitstring, hexstring,
charstring или universal charstring * /
522. Product ::= Term {AddOp Term}
/* ОПЕРАЦИОННАЯ СЕМАНТИКА – Каждый Term должен сводиться к конкретному значению.
Если присутствуют более одного Term, то все Term должны сводиться к типу integer
или float.* /
523. Term ::= Factor {MultiplyOp Factor}
/ * ОПЕРАЦИОННАЯ СЕМАНТИКА – Каждый Factor должен сводиться к конкретному
значению. Если присутствуют более одного Factor, то все Factor должны сводиться к
типу integer или float. * /
524. Factor ::= [UnaryOp] Primary
/* ОПЕРАЦИОННАЯ СЕМАНТИКА – Primary должен сводиться к конкретному значению. Если
UnaryOp присутствует и равен "not", то Primary должен сводиться к типу boolean, а
```

```
если UnaryOp равен "+" или "-", то Primary должен сводиться к типу integer или
float. Если UnaryOp сводится к not4b, то Primary должен сводиться к типу
bitstream, hexstring или octetstring. * /
525. Primary ::= OpCall | Value | "(" SingleExpression ")"
526. ExtendedFieldReference ::= {(Dot StructFieldIdentifier | ArrayOrBitRef)}+
527. OpCall ::= ConfigurationOps | VerdictOps | TimerOps | TestcaseInstance |
               FunctionInstance | TemplateOps
528. AddOp ::= "+" | "-"
/* ОПЕРАЦИОННАЯ СЕМАНТИКА - Операнды операторов "+" или "-" должны быть типа
integer или float (то есть предопределенных типов TTCN или ASN.1) либо
производных от integer или float (то есть поддиапазонов) * /
529. MultiplyOp ::= "*" | "/" | mod | rem
/* ОПЕРАЦИОННАЯ СЕМАНТИКА - Операнды операторов "*", "/", rem или mod должны быть
типа integer или float (то есть предопределенных типов TTCN или ASN.1) либо
производных от integer или float (то есть поддиапазонов) * /
530. UnaryOp ::= "+" | "-" | not | not4b
/* ОПЕРАЦИОННАЯ СЕМАНТИКА - Операнды операторов "+" или "-" должны быть типа
integer или float (то есть предопределенных типов TTCN или ASN.1) либо
производных от integer или float (то есть поддиапазонов). Операнды неоператоров
должны быть типа boolean (TTCN или ASN.1) либо производных от типа boolean.
Операнды оператора not4b будут типа bitstring, octetstring или hexstring * /
531. RelOp ::= "==" | "<" | ">" | "!=" | ">=" | "<="
/* ОПЕРАЦИОННАЯ СЕМАНТИКА - Старшинство операторов определено в таблице 7 * /
532. BitOp ::= "and4b" | xor4b |or4b | and | xor | or | StringOp
/* ОПЕРАЦИОННАЯ СЕМАНТИКА - Операнды операторов and, or или xor должны быть типа
boolean (TTCN или ASN.1) либо производных от типа boolean. Операнды операторов
and4b, or4b или xor4b должны быть типа bitstring, hexstring или octetstring (TTCN
или ASN.1) либо производных от этих типов. * /
/ * ОПЕРАЦИОННАЯ СЕМАНТИКА - Старшинство операторов определено в таблице 7 * /
533. StringOp ::= "&"
/* ОПЕРАЦИОННАЯ СЕМАНТИКА - Операндами оператора string должны быть bitstring,
hexstring, octetstring или charstring * /
534. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
535. LogStatement ::= LogKeyword "(" [FreeText] ")"
536. LogKeyword ::= "log"
537. LoopConstruct ::= ForStatement |
               WhileStatement |
               DoWhileStatement
538. ForStatement ::= ForKeyword "(" Initial [SemiColon] Final [SemiColon] Step
               ")"
               StatementBlock
539. ForKeyword ::= "for"
540. Initial ::= VarInstance | Assignment
541. Final ::= BooleanExpression
542. Step ::= Assignment
543. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
               StatementBlock
544. WhileKeyword ::= "while"
545. DoWhileStatement ::= DoKeyword StatementBlock
               WhileKeyword "(" BooleanExpression ")"
546. DoKeyword ::= "do"
547. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
               StatementBlock
               {ElseIfClause}   [ElseClause]
548. IfKeyword ::= "if"
549. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"
               StatementBlock
550. ElseKeyword ::= "else"
551. ElseClause ::= ElseKeyword  StatementBlock
```

A.1.6.10 Разные продукты

```
552. Dot ::= "."
553. Dash ::= "-"
554. Minus ::= Dash (черточка, тире)
555. SemiColon ::= ";"
556. Colon ::= ":"
557. Underscore ::= "_"
558. BeginChar ::= "{"
559. EndChar ::= "}"
560. AssignmentChar ::= ":="
```

Приложение B

Операционная семантика

В данном Приложении интуитивно и однозначно описывается смысл поведения TTCN-3. Операционная семантика не является формальной, поэтому возможность обрабатывать математические проверки на базе этой семантики весьма ограничена.

Эта операционная семантика дает ориентированную на состояния точку зрения на выполнение модуля TTCN-3. Вводятся разные виды состояний и описывается смысл различных конструкций TTCN-3 путем:

1) использования статической информации для определения исходных состояний перед выполнением какой-либо конструкции; и

2) определения, как выполнение конструкции будет изменять состояние.

Эта операционная семантика ограничивается смыслом поведения TTCN-3, то есть функций, тестовых примеров, управления модулями и языковыми конструкциями для определения тестового поведения, например, операций **send** и **receive**, команд **if-else** или **while**. Смысл некоторых конструкций TTCN-3 объясняется путем замены их конструкциями другого языка. Например, именованные альтернативы являются макросами, и их смысл полностью объясняется путем замены всех ссылок на макросы соответствующими определениями макросов. Это охватывает и обработку поведения по умолчанию.

В большинстве случаев определение семантики какого-либо языка базируется на дереве абстрактного синтаксиса того кода, который необходимо описать. Эта семантика действует не на основе дерева абстрактного синтаксиса, но требует графического представления описаний поведения TTCN-3 в форме потоковых графов. Потоковый граф описывает поток управления в тестовом примере, функции или в управлении модулем. Отображение описаний поведения TTCN-3 в потоковые графы является простым.

B.1 Структура данного Приложения

Данное Приложение разделено на две части:

1) В первой части (см. п. B.2) определяется смысл кратких нотаций и макронотаций TTCN-3 путем их замены другими конструкциями языка TTCN-3. Такие замены в модуле TTCN-3 могут рассматриваться как шаг перед обработкой до того, как модуль может интерпретироваться согласно последующему описанию операционной семантики.

2) Во второй части (см. п. B.3) описывается операционная семантика TTCN-3 с помощью интерпретации потокового графа и модификации состояний.

B.2 Замена кратких нотаций и макровызовов

Прежде чем эта операционная семантика может быть использована для пояснения поведения TTCN-3, краткие нотации должны быть расширены, а ссылки на макросы должны быть заменены на соответствующие определения на текстовом уровне.

Краткими нотациями TTCN-3 являются:

- автономные операции приема;
- операции **trigger**;
- случаи использования ключевого слова **any** в таймерных операциях и операциях приема;
- случаи использования ключевого слова **all** в таймерных и портовых операциях;
- отсутствие команд **return** и **stop** в конце определений функции и тестового примера.

Макросами TTCN-3 являются именованные альтернативы, то есть определения **named alt**. Они вызываются:

- явно вместо команды **all**, то есть они появляются как вызов функции;
- явно в командах **all** с помощью ключевого слова **expand**;
- неявно в случае, когда на них делается ссылка как на режим по умолчанию в командах **activate** и **deactivate**.

В дополнение к кратким нотациям и вызовам макросов операционная семантика требует специальной обработки параметров модулей и глобальных констант, то есть констант, определенных в определяющей части модуля. Все ссылки на параметры модуля и глобальные константы должны заменяться конкретными значениями. Это означает, что предполагается, что значения параметров модуля и глобальных констант могут быть определены до того, как операционная семантика станет уместной.

ПРИМЕЧАНИЕ 1. – Эта обработка параметров модуля и глобальных констант в операционной семантике будет отличаться от их обработки в трансляторе (компиляторе) TTCN-3. Операционная семантика описывает смысл поведения TTCN-3, но не является руководством по реализации транслятора TTCN-3.

ПРИМЕЧАНИЕ 2. – Операционная семантика обрабатывает параметры и местные константы в тестовых компонентах, тестовых примерах, функциях и в управлении модулем наподобие переменных. Ошибочное использование местных констант или параметров in, out и inout должно проверяться статически.

B.2.1 Порядок следования шагов замены

Текстуальная замена кратких нотаций, макровызовов, глобальных констант и параметров модуля должна производиться в следующем порядке:

1) добавление команд **stop** и **return** в управлении модулем, функциях и тестовых примерах;
2) замена глобальных констант и параметров модуля конкретными значениями;
3) вложение одиночных операций приема в команды **alt**;
4) макрорасширение чистых *макровызовов*, что означает:
   - явное расширение команд **alt**, которые содержат ключевое слово **expand** (и ссылаются на определение **named alt**);
   - явное расширение вызовов определений **named alt**;
5) расширение команд **interleave**;
6) расширение поведения по умолчанию;
7) замена всех операций **trigger** эквивалентными операциями **receive** и командами **goto**;
8) замена всех использованных ключевых слов **any** и **all** таймерными и портовыми операциями.

ПРИМЕЧАНИЕ. – Если не придерживаться этого порядка шагов замены, результат замен не будет представлять определяемое поведение.

B.2.2 Добавление операций Stop и Return в описания поведения

TTCN-3 допускает прекращение управления модулем, тестовых примеров и функций, которые не выдают какого-либо значения, без определения явной операции **stop** или **return**. Для операционной семантики предполагается, что отсутствующие операции **return** и **stop** добавляются, то есть операции **stop** добавляются в управление модулем и в тестовые примеры, а операции **return** добавляются в функции.

Пример:

```
// Определения функции и тестового примера без явных команд return и stop в конце
// описаний их поведения

   function MyFunction(inout integer MyPar) {
                                 // MyFunction не выдает значения, но изменяет
      MyPar := 10 * MyPar1;      // значение MyPar, которое передается по ссылке
      if (MyPar == 999) stop;    // Остановка выполняется, если MyPar имеет
                                 // значение 999
      // НЕЯВНАЯ return, если MyPar != 999
   }

   testcase MyTestCase() runs on MyMTCtype {
      MyMTCbehaviour();       // Функция, которая определяет поведение MTC

      // НЕЯВНАЯ stop после выдачи MyMTCbehaviour
   }


// MyFunction и MyTestCase после добавления явных операций return и stop

   function MyFunction(inout integer MyPar) {
                                 // MyFunction не выдает значения, но изменяет
      MyPar := 10 * MyPar1;      // значение MyPar, которое передается по ссылке
      if (MyPar == 999) stop;    // Остановка выполняется, если MyPar имеет
                                 // значение 999

         return;                 // ЯВНАЯ return
      }

      testcase MyTestCase() runs on MyMTCtype {

      MyMTCbehaviour();       // Функция, которая определяет поведение MTC

      stop;                   // ЯВНАЯ stop
      }
```

B.2.3 Замена глобальных констант и параметров модуля

Константы, объявленные в определяющей части модуля, являются глобальными для управления модулем и всех тестовых компонентов, которые появляются во время выполнения модуля TTCN-3. Параметры модуля считаются глобальными константами во время выполнения.

Все ссылки на глобальные константы и параметры модуля должны заменяться реальными значениями перед тем, как операционная семантика начнет интерпретацию этого модуля. Если значение константы или параметра модуля дано в форме выражения, то это выражение должно быть вычислено. Затем результат этого вычисления должен заменить все ссылки на константу или параметр модуля.

B.2.4 Вложение одиночной операции приема в команды Alt

Операциями приема TTCN-3 являются: **receive, trigger, getcall, getreply, catch, check, timeout** и **done**.

ПРИМЕЧАНИЕ. – Операции **receive, trigger, getcall, getreply, catch** и **check** выполняются в портах и позволяют разветвление благодаря приему сообщений, вызовов процедур, ответов и особых состояний. Операции **timeout** и **done** не являются реальными операциями приема, но они могут использоваться так же, как операции приема, то есть как альтернативы в командах **alt**. Поэтому операционная семантика обрабатывает **timeout** и **done** наподобие операций приема.

Операции приема могут использоваться в качестве автономной команды в функции, именованной альтернативе или в тестовом примере. В таком случае операция приема рассматривается как краткий вариант команды **alt** только с одной альтернативой, определенной этой приемной операцией. Для операционной семантики команда **alt**, в которую вложена команда приема, должна заменять все автономные появления операций приема.

Например:

```
// Автономное появление нижеследующего
    :
   MyCL.trigger(MyType:*);
    :

// должно быть заменено на
    :
   alt {
      [] MyCL.trigger (MyType:*);
   }
    :

// или
    :
   MyPTC.done;
    :

// должно быть заменено на
    :
   alt {
      [] MyPTC.done;
   }
    :
```

B.2.5 Макрорасширение

Макрорасширение в TTCN-3 относится к использованию именованных альтернатив (определений **named alt**) в командах **alt** или вместо команд **alt**, то есть ссылки на именованное определение **alt** делаются так же, как на вызов функции в последовательности команд.

B.2.5.1 Расширение именованных альтернатив в командах альтернатив

Расширение именованных альтернатив в командах **alt** относится к определенным ветвям альтернатив, обозначенным заключенным в квадратные скобки ключевым словом **expand**, за которым следует ссылка на определение **named alt** (только как команда этой ветви). В таком случае ветви альтернатив для указанной именованной альтернативы заменяют ветвь с ключевым словом **expand**. В операционной семантике предполагается, что эта замена производится на синтаксическом уровне. Пример такого расширения можно найти в основной части данной Рекомендации.

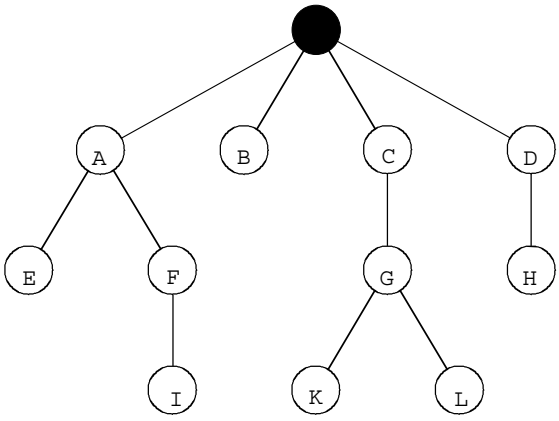
B.2.5.2 Явный вызов именованной альтернативы

Именованные альтернативы могут быть указаны так же, как вызов функции в последовательности команд. В этом случае ссылка расширяется соответствующим определением **named alt**. Пример такого расширения можно найти в основной части данной Рекомендации.

B.2.6 Замена перемежающейся конструкции

Смысл команды **interleave** определяется путем ее замены серией вложенных команд **alt**, которые имеют один и тот же смысл. Алгоритм конструирования замены для команды **interleave** описывается в данном разделе. Эта замена производится на синтаксическом уровне.

Серия вложенных команд **alt** может быть описана с помощью дерева. Вершины (узлы) дерева представляют команды в командах **alt**. Соединение ветвью обозначает команду **alt**, а команды в одной и той же ветви описывают команды в одной и той же альтернативе. Это схематически показано на рис. B.1. На рис. B.1 a) представлено дерево, а на рис. B.1 b) дано соответствующее представление в форме серии вложенных команд **alt**.




```
alt {

     [] A {
          alt {
               [] E ;
               [] F {
                    I
               }
          }
     }
     [] B;
     [] C {
          G;
          alt {
               [] K;
               [] L;
          }
     }
     [] D {
               H
     }
}
```

T1012780-01

a) Дерево **b) Представление a) в стиле TTCN-3**

Рисунок B.1/Z.140 – Вложенные команды alt и соответствующее древовидное представление

Ниже дается конструкция древовидного представления команды **interleave**. Трансформация дерева в серию вложенных команд **alt** является простой и не требует дальнейших пояснений.

Команда **interleave** может рассматриваться как частично упорядоченный набор ***POS*** разрешенных команд TTCN-3. Формально:

- $\boldsymbol{POS} = (\boldsymbol{S}, <)$

где:

$\boldsymbol{S}$ – набор разрешенных команд TTCN-3; и

$< \subseteq (\boldsymbol{S} \times \boldsymbol{S})$ – описывает рефлексивный и транзитивный порядок отношения.

Термин *разрешенные команды TTCN-3* означает, что в командах **interleave** не разрешается использовать управляющие команды переноса **for, while, do-while, goto, activate, deactivate, stop, return** и вызовы определяемых пользователем функций, которые содержат операции связи. Кроме того, не разрешаются также защитные ветви команды **interleave** с булевыми

выражениями для расширения команд **interleave** с именованными альтернативами или для указания ветвей **else**.

Для алгоритма конструирования необходимо определить следующие функции:

- Функция *DISCARD* удаляет элемент ***s*** из частично упорядоченного набора ***POS*** и выдает результирующий частично упорядоченный набор ***POS'***:

  $$\underline{DISCARD}\ (\mathbf{s}, \boldsymbol{POS}) = \boldsymbol{POS}'$$

  где:

  $\boldsymbol{POS}' = (\boldsymbol{S}', <')$; и

  $\mathbf{S}' = \mathbf{S} \setminus \{\mathbf{s}\}$; и

  $<' = < \cap (\boldsymbol{S} \setminus \{\boldsymbol{s}\} \times \boldsymbol{S} \setminus \{\boldsymbol{s}\})$.

- Функция *ENABLED* получает частично упорядоченный набор ***POS*** = (***S***, <) и выдает все элементы, которые не имеют предшественников в ***POS***:

  $$\underline{ENABLED}\ (\boldsymbol{POS}) = \{\boldsymbol{s} \mid \boldsymbol{s} \in \boldsymbol{S} \wedge (< \cap (\boldsymbol{S} \times \{\boldsymbol{s}\}) = \varnothing)\}.$$

- Функция *RECEIVING* получает набор команд ***S*** TTCN-3 и выдает все команды приема из этого набора.

  $$\underline{RECEIVING}\ (\boldsymbol{S}) = \{\boldsymbol{s} \mid \boldsymbol{s} \in \boldsymbol{S} \wedge \underline{kind}\ (\mathbf{s}) \in \{\mathbf{receive, trigger, getcall, getreply, catch, check, done, timeout}\}\}$$

- Функция *SELECT* выбирает случайный элемент ***s*** из заданного набора ***S*** и выдает ***s***.

  $$\underline{SELECT}\ (\boldsymbol{S}) = \boldsymbol{s}, \text{ где } \boldsymbol{s} \in \boldsymbol{S}.$$

ПРИМЕЧАНИЕ. – Функция *kind*, показанная выше в функции *RECEIVING*, не определена формально. Функция *kind* (или тип) выдает вид заданной команды TTCN-3.

Алгоритм конструирования дерева – это рекурсивная процедура, в которой при каждом вызове рекурсии конструируются вершины-преемники для заданной вершины. Эта процедура обеспечивается в нотации псевдокода, похожего на язык Си, которая использует определенные выше функции и некоторую дополнительную математическую нотацию:

```
CONSTRUCT-SUCCESSORS (treeNode *predecessor, partiallyOrderedSet POS) {
   // treeNode ссылается на тип вершины конструируемого дерева
   // partiallyOrderedSet означает тип частично упорядоченного набора
   // команд TTCN-3

   var statement myStmt;                  // для сохранения команды TTCN-3
   var treeNode *newSonNode;              // для обработки новых вершин дерева

// ЗАПРАШИВАЮЩИЕ НАБОРЫ КОМАНД TTCN-3, КОТОРЫЕ НЕ ИМЕЮТ ПРЕДШЕСТВЕННИКОВ В 'POS'
   var statementSet enabStmts := ENABLED(POS);
                        // Все команды без предшественников
   var statementSet enabRecStmts := RECEIVING(enabStmts);
                        // Приемные команды в 'enabStmts'
   var statementSet enabNonRecStmts := enabStmts\enabRecStmts;
                        // Неприемные команды в 'enabStmts'

   if (POS == ∅)
      return;           // КРИТЕРИЙ ОКОНЧАНИЯ ДЛЯ РЕКУРСИИ
   else {
      if (enabNonRecStmts != ∅) { // / Обработка неприемных команд в
                                  // 'enabStmts'
         myStmt := SELECT(enabNonRecStmts);
         newSonNode := create(myStmt, predecessor);
                  // Создание новой вершины дерева, представляющей 'myStmt' в
                  // этом дереве и обновляющей указатели в 'newSonNode' и 'predecessor'.
         CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, POS));
```

```
                  // СЛЕДУЮЩИЙ ШАГ РЕКУРСИИ
      }
      else {   // Обработка событий приема, дерево будет ветвиться
         for each (myStmt in enabRecStmts) {
               newSonNode := create(myStmt, predecessor); // Новая вершина дерева
               CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, POS));
               // СЛЕДУЮЩИЙ ШАГ (ШАГИ) РЕКУРСИИ
         }
      }
   }
}
```

Вначале будет вызвана функция CONSTRUCT-SUCCESSORS с *корневой вершиной* пустого дерева и частично упорядоченный набор команд TTCN-3, описывающий команду **interleave**, которая должна быть заменена. После определения *корневая вершина* может использоваться для доступа к конструируемому дереву.

B.2.7 Расширение безусловных вариантов

Механизм безусловного (по умолчанию) поведения TTCN-3 определяется с помощью механизма макрорасширения. Безусловное поведение должно обеспечиваться в форме определений **named alt**. Определение **named alt**, используемое в качестве безусловного поведения, указывается в команде **activate**. Контекст безусловного варианта определяется командой **activate** и соответствующими командами **deactivate** либо командой **activate** и окончанием этой функции или тестового примера, в котором использована данная команда **activate**. В пределах этого контекста альтернативы всех команд **alt** расширяются с помощью поведения, определенного в активированных определениях **named alt**. В операционной семантике предполагается, что это расширение дано на синтаксическом уровне. Пример механизма расширения можно найти в основной части данной Рекомендации.

B.2.8 Замена триггерных операций

Операция **trigger** фильтрует сообщения по определенному критерию сопоставления из потока сообщений заданного порта. Семантика операции **trigger** может быть описана путем ее замены двумя операциями **receive** и командой **goto**. В операционной семантике предполагается, что эта замена производится на синтаксическом уровне.

Пример:

```
// Следующая триггерная операция ...

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

// должна быть заменена на...

   alt   {
         [] MyCL.receive (MyType:*);
         [] MyCL.receive {
               goto alt
         }
   }
```

Если команда **trigger** используется в более сложной команде **alt**, то замена производится таким же способом.

Пример:

```
// Следующая команда alt содержит команду trigger...
   alt   {
         [] PCO2.receive {
               stop;
            }
         [] MyCL.trigger (MyType:*);
         [] PCO3.catch {
               verdict.set(fail);
               stop;
            }
      }

// которая будет заменена на

   alt   {
         [] PCO2.receive {
               stop;
            }
         [] MyCL.receive (MyType:*);
         [] MyCL.receive {
               goto alt;
            }
         [] PCO3.catch {
               verdict.set(fail);
               stop;
            }
      }
```

B.2.9 Замена ключевых слов 'any' и 'all'

Использование ключевого слова **any** разрешается для:

- таймерных операций **running** и **timeout**;
- операций приема **receive, trigger, getcall, getreply, catch, check.**

Использование ключевого слова **all** разрешается для:

- таймерной операции **stop**;
- портовых операций **start, stop** и **clear**.

Использование обоих ключевых слов разрешается для:

- операций **done** и **running** в компонентах.

B.2.9.1 Замена 'all' в таймерных и портовых операциях

Применение таймерных и портовых операций относится к контексту, в котором они используются. Это означает, что ключевое слово **all** адресует все таймеры и порты, известные в единице контекста, в которой используются **all** (+ операция). Замена использований **all** в таймерных и портовых операциях является простой.

Использование **all port** в операции **start, stop** или **clear** заменяется отдельной операцией **start, stop** или **clear** для каждого известного порта. Использование **all timer** в операции stop заменяется отдельной операцией **stop** для каждого известного таймера.

Пример:

```
// Предположим, что известны порты PCO1, PCO2 и таймеры T1 и T2

    :
   all port.clear;
    :
    :
   all timer.stop;
    :

// будут заменены на

    :
   PCO1.clear;
   PCO2.clear;
    :
    :
   T1.stop;
   T2.stop;
    :
```

B.2.9.2 Замена 'any' в таймерных операциях и операциях приема

Применение таймерных операций и операций приема относится к контексту, в котором они используются. Это означает, что ключевое слово **any** адресует все таймеры и порты (в случае операций приема), известные в единице контекста, в которой используются **any** (+ операция). Замена использований **any** в таймерных операциях и операциях приема простая.

Использование **any port** в операции **receive, trigger, getcall, getreply, catch** или **check** заменяется отдельными альтернативными операциями для каждого известного и возможного порта. "Возможного" означает, что появление **any port.receive** уместно только для портов на базе сообщений.

Использование **any timer** в операции **timeout** заменяется отдельными альтернативными операциями для каждого известного таймера в этой единице контекста.

Пример:

```
// Предположим, что известны порты PCO1, PCO2 и таймеры T1 и T2

   alt   {
         [] PCO2.receive {
               aTestStep();
            }
         [] any port.receive {
               verdict.set(fail);
               stop;
            }
         [] any timer.timeout {
               verdict.set(fail);
               stop;
            }
      }

// будут заменены на

   alt   {
         [] PCO2.receive {
               stop;
            }
```

```
        [] PCO1.receive {
             verdict.set(fail);
             stop;
           }
        [] PCO1.receive {
             verdict.set(fail);
             stop;
           }
        [] T1.receive {
             verdict.set(fail);
             stop;
           }
        [] T2.receive {
             verdict.set(fail);
             stop;
           }
     }
```

Использование **any timer** в операции **running** заменяется отдельными операциями **running** для каждого известного таймера в единице контекста, которая скомбинирована с помощью операций or.

Пример:

```
// Предположим, что в единице контекста известны таймеры T1 и T2

    :
   if (any timer.running) {
      verdict.set(fail);
      stop;
   }
    :

// будут заменены на

    :
   if (T1.running or T2.running) {
      verdict.set(fail);
      stop;
   }
    :
```

B.2.9.3 Ключевые слова 'any' и 'all' в операциях 'done' и 'running'

Операции **any component.done, all component.done, any component.running** и **all component.running** могут выполняться только из МТС. Из-за динамического характера создания тестовых компонентов МТС может не знать всех компонентов, которые были созданы во время выполнения тестового примера. Поэтому выполнение этих операций требует связи со средствами тестирования. Следовательно, **any component.done, all component.done, any component.running** и **all component.running** можно считать системными командами, то есть они не могут быть заменены другими командами.

B.3 Семантика с потоковым графом в TTCN-3

Операционная семантика TTCN-3 базируется на интерпретации потоковых графов. В данном разделе вводятся потоковые графы (см. п. B.3.1), объясняется конструкция потоковых графов, представляющих управление модулем TTCN-3, тестовые примеры, функции и определения типов компонентов (см. п. B.3.2), определяются состояния модуля и компонентов для описания состояний выполнения модуля TTCN-3 (см. п. B.3.3), описываются обработка сообщений, вызовы удаленных процедур, ответы на вызовы удаленных процедур и особые состояния (см. п. B.3.4), поясняется процедура оценки для управления модулем и тестовыми примерами (см. п. B.3.6) и описывается смысл различных команд TTCN-3 (см. п. B.3.7).

B.3.1 Потоковые графы

Потоковый граф – это направленный граф, который содержит помеченные вершины (узлы) и помеченные края. Прохождение через потоковый граф описывает поток управления во время выполнения представленного описания поведения.

B.3.1.1 Рамка потокового графа

Потоковый граф помещается в рамку, определяющую границу этого потокового графа. Имя потокового графа следует за ключевыми словами **flow graph** (это не ключевые слова базового языка TTCN-3) и помещается в верхний левый угол потокового графа. В качестве соглашения предполагается, что имя потокового графа ссылается на описание поведения TTCN, представляемое этим потоковым графом. Простой потоковый граф показан на рис. B.2.

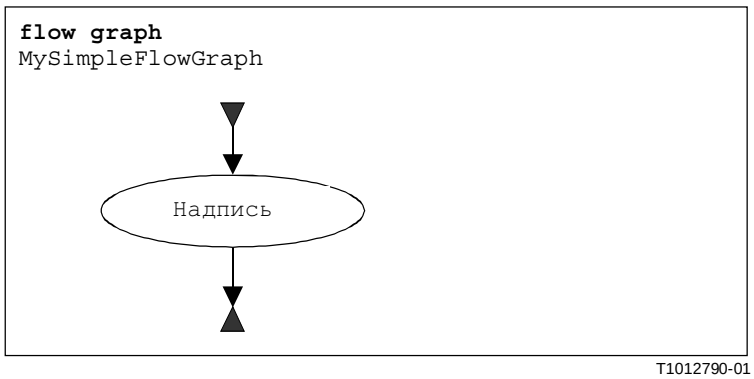


Рисунок B.2/Z.140 – Простой потоковый граф

B.3.1.2 Вершины потокового графа

Потоковые графы состоят из *начальных вершин*, *оконечных вершин*, *базовых вершин* и *ссылочных вершин*.

B.3.1.2.1 Начальные вершины

Начальные вершины описывают начальную точку потокового графа. Потоковый граф должен иметь только одну начальную вершину. Начальная вершина показана на рис. B.3 a).

a) Начальная вершина потокового графа **b) Оконечная вершина потокового графа**

Рисунок B.3/Z.140 – Начальная и оконечная вершины

B.3.1.2.2 Оконечные вершины

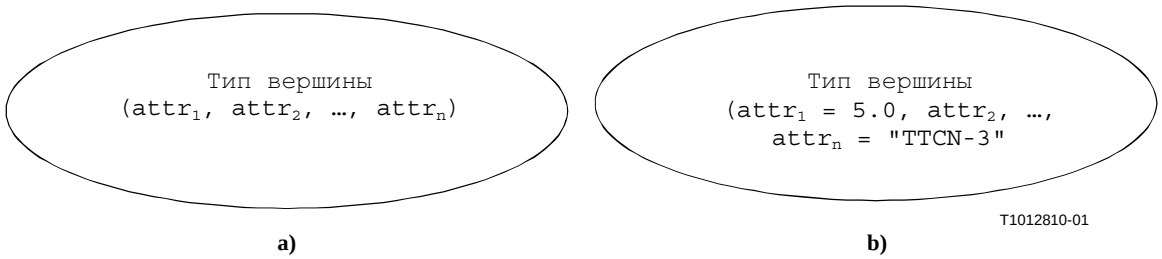
Оконечные вершины описывают оконечные точки потокового графа. Потоковый граф может иметь несколько оконечных вершин или, в случае циклов, ни одной оконечной вершины. Базовые вершины (см. п. B.3.1.2.3) и ссылочные вершины (см. п. B.3.1.2.4), не имеющие вершин-преемников, соединяются с оконечной вершиной, для того чтобы показать, что они описывают последнее действие на пути через потоковый граф. Оконечная вершина показана на рис. B.3 b).

B.3.1.2.3 Базовые вершины

Базовая вершина описывает единицу выполнения, то есть она выполняется за один шаг. Базовая вершина имеет тип и, в зависимости от этого типа, может иметь связанный список атрибутов. Базовая вершина показана на рис. B.4 a).

В надписи базовой вершины атрибуты узла следуют за типом узла и помещаются в круглые скобки. Тип и атрибуты используются для определения действия, которое необходимо реализовать во время выполнения представленной конструкции языка. Атрибуты описывают информацию, которая будет запрашиваться от корреспондирующей конструкции TTCN-3.

Атрибуты имеют значения, и операционная семантика будет запрашивать эти значения путем ссылок на имя атрибута. Если требуется, разрешается присваивать явные значения в базовых вершинах с помощью присвоения '='. Пример показан на рис. B.4 b).

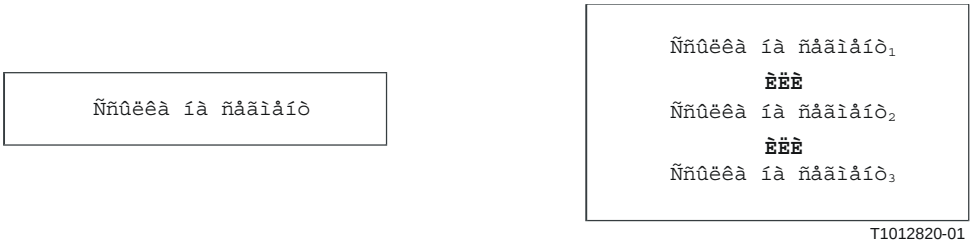


a) **b)**

Рисунок B.4/Z.140 – Базовые вершины с атрибутами

B.3.1.2.4 Ссылочные вершины

Ссылочные вершины отсылают к сегментам потокового графа (см. п. B.3.1.4), которые являются потоковыми подграфами. Смысл ссылочной вершины определяется путем ее замены в потоковом графе на указанный сегмент потокового графа. Надпись в ссылочной вершине дает ссылку на сегмент потокового графа. Ссылочная вершина показана на рис. B.5 a).



a) Одиночная ссылочная вершина **b) Комбинация с OR для трех ссылочных вершин**

Рисунок B.5/Z.140 – Ссылочная вершина

B.3.1.2.4.1 Комбинация с OR для ссылочных вершин

В некоторых случаях одну ссылочную вершину могут заменять несколько сегментов потокового графа. В таких случаях может использоваться оператор `OR` (ИЛИ) для ссылки на несколько сегментов потокового графа (рис. B.5 b). В реальном потоковом графе, представляющем управление модулем, тестовый пример или функцию, представленной конструкцией определяется одна альтернатива.

B.3.1.2.4.2 Многократное появление ссылочных вершин

В некоторых случаях ссылочные вершины одного и того же вида могут появиться в потоковом графе нуль, один или более раз. В обычных выражениях возможное повторение частей обычного выражения описывается с помощью символов операторов '+' (одно или более повторений и '*' (нуль или больше повторений). Как показано на рис. B.6, эти операторы были приспособлены к потоковым графам путем введения ссылочных вершин с двойной рамкой, содержащей соответствующие символы операторов. Одиночная потоковая линия должна заменять ссылочную вершину в случае нулевого числа появлений (используя ссылочную вершину с двойной рамкой и с оператором '*').

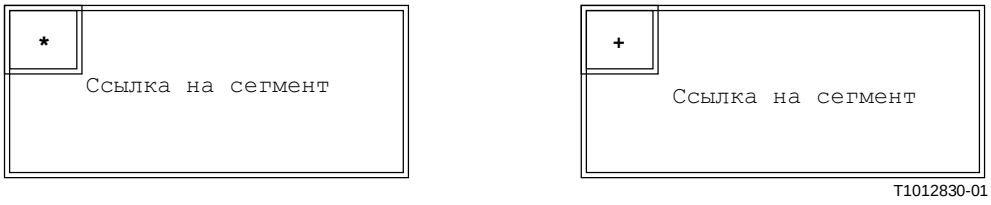


Рис. B.6/Z.140 – Повторение ссылочных вершин

Верхняя граница возможных повторений ссылочной вершины может быть дана в виде целого числа в круглых скобках после символа '*' или '+' в ссылочной вершине с двойной рамкой. Ссылка на сегмент, показанная на рис. B.7, может появиться от нуля до 5 раз.

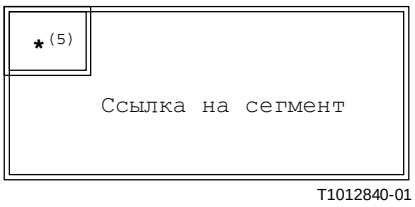
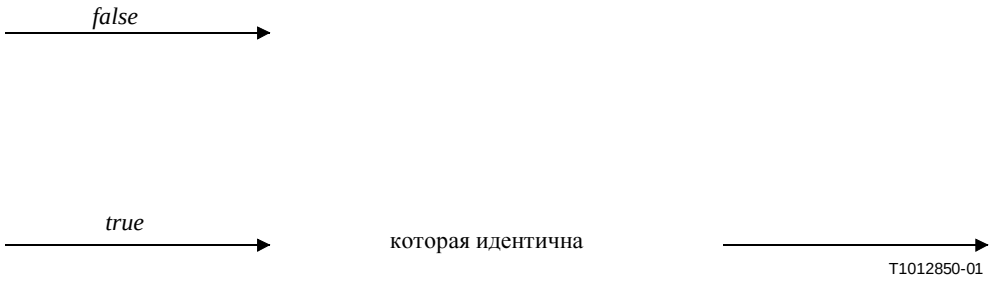


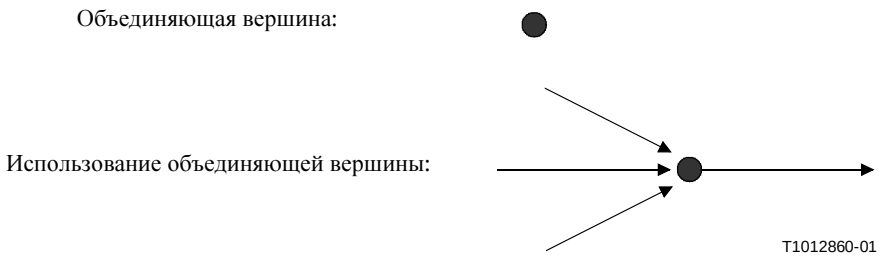
Рисунок B.7/Z.140 – Ограниченное число повторений ссылочной вершины

B.3.1.3 Потоковые линии

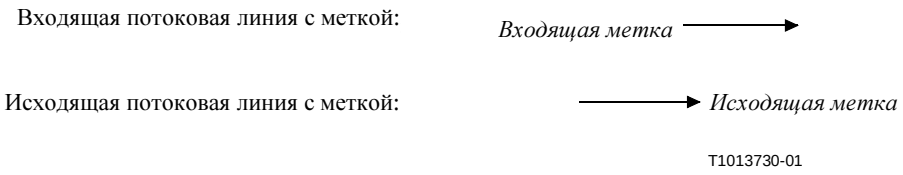
Потоковые линии представляются стрелками. Потоковая линия имеет надпись *true* (ИСТИНА) или *false* (ЛОЖЬ), которая указывает состояние, в котором эта линия выбирается во время интерпретации потокового графа. В качестве краткой нотации разрешается опускать надпись true. Примеры потоковых линий показаны ниже.



Для поддержки объединения нескольких потоковых линий в одну потоковую линию на графическом уровне вводится специальная объединяющая вершина. Объединяющая вершина и пример ее использования показаны ниже.



Вычерчивание длинных потоковых линий, необходимых на больших диаграммах, например для моделирования конструкций TTCN-3 **goto** и **label**, затруднительно. Для этой цели можно использовать исходящие и входящие потоковые линии. Примеры показаны ниже.



Исходящая потоковая линия с меткой соединяется с входящей потоковой линией с меткой, если эти метки идентичны. Метки входящих потоковых линий должны быть уникальными. Если имеются несколько исходящих потоковых линий с одинаковой меткой, то это следует рассматривать как объединение линий во входящую потоковую линию с такой же меткой.

B.3.1.4 Сегменты потокового графа

Сегменты потокового графа – это потоковые подграфы. На них делаются ссылки в ссылочных вершинах, и они определяют смысл такой ссылочной вершины. Сегменты потокового графа могут содержать дальнейшие ссылочные вершины.

Как показано на рис. B.8, сегменты потокового графа имеют определенные интерфейсы, которые содержат входящие и исходящие потоковые линии. Имеется только одна входящая потоковая линия без метки и одна или нуль исходящих потоковых линий без метки. Кроме того, может иметься несколько входящих и исходящих потоковых линий с метками. Входящие и исходящие потоковые линии с метками нужны для описания смысла команд TTCN-3 **goto**.

Сегменты потокового графа заключаются в рамку, а имя сегмента потокового графа помещается после ключевого слова segment в верхнем левом углу рамки. Потоковые линии, описывающие интерфейс сегмента потокового графа, должны пересекать рамку сегмента потокового графа.

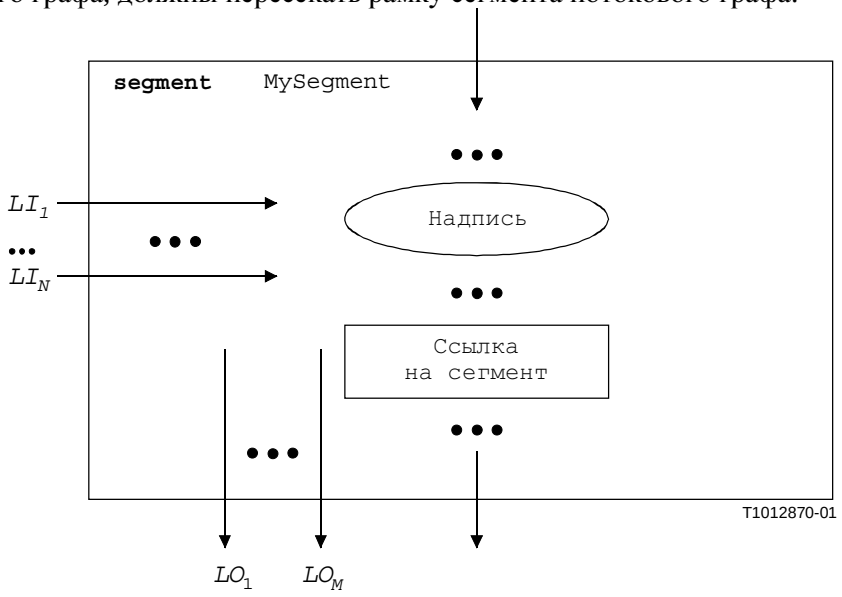
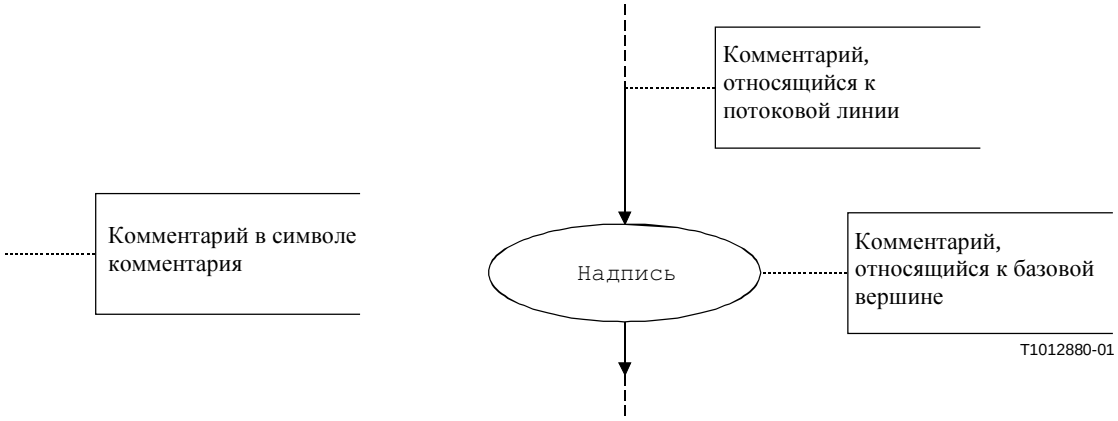


Рисунок B.8/Z.140 – Схематическое описание сегмента потокового графа

B.3.1.5 Комментарии

Для удобства чтения и повышения связности может использоваться специальный символ комментария для привязывания комментариев к вершинам потокового графа и к потоковым линиям. Символ комментария и его использование показаны на рис. B.9.



a) Символ комментария **b) Использование символа комментария**

Рисунок B.9/Z.140 – Представление комментариев в потоковом графе

B.3.1.6 Обращение с описаниями потоковых графов

Процедура анализа операционной семантики представляет собой обход потоковых графов, которые содержат только базовые вершины, то есть все ссылочные вершины вводятся соответствующими определениями сегментов потокового графа. Для поддержки этого обхода требуется функция *NEXT*. Эта функция определяется следующим образом:

<actualNodeRef>.NEXT(<bool>) = <successorNodeRef>,

где:

<*actualNodeRef*> : ссылка из базовой вершины потокового графа;

<*successorNodeRef*> : ссылка на вершины-преемника и той вершины, которая указана ссылкой <*actualNodeRef*>;

<*bool*> : булево выражение, указывающее, какой преемник выдается – с *true* или *false* (см. п. B.3.1.3).

B.3.2 Представление поведения TTCN-3 с помощью потокового графа

В операционной семантике предполагается, что описания поведения TTCN-3 выполняются в виде набора потоковых графов, то есть для каждого описания поведения TTCN-3 должен быть сконструирован отдельный потоковый граф.

Операционная семантика интерпретирует в качестве описаний поведения следующие виды описаний TTCN-3:

a) управление модулем;

b) определения тестовых примеров;

c) определения функций;

d) определения типов компонентов.

Управление модулем описывает тестовую кампанию, то есть порядок выполнения (возможно, повторений) реальных тестовых примеров. Определения тестовых примеров описывают поведение МТС. Определения функций описывают поведение, которое должно выполняться с помощью

управления модулем или тестовых компонентов. Определения типов компонентов, как предполагается, будут описаниями поведения, так как они определяют создание, объявление и инициализацию портов, констант, переменных и таймеров в процессе создания экземпляра типа компонента.

B.3.2.1 Процедура конструирования потокового графа

Потоковые графы, представленные на рис. B.10 и B.11, и сегменты потокового графа, представленные в п. B.3.6, являются только шаблонами. Они содержат *символы подстановки* для информации, которая должна быть предоставлена для образования конкретного потокового графа или сегмента потокового графа. Символы подстановки обозначаются вводными знаками '<' и '>'.

Конструирование представления с потоковым графом для модуля TTCN-3 производится в три шага:

1) Конструируется конкретный сегмент потокового графа для каждой команды TTCN-3 в определениях управления модулем, тестовых примеров, функций и типов компонента.

2) Конструируется конкретный потоковый граф (со ссылочными вершинами) для управления модулем и для каждого определения тестового примера, функции и типа компонента.

3) Все ссылочные вершины в конкретных потоковых графах с помощью пошаговой процедуры заменяются соответствующими определениями сегментов потокового графа, пока все потоковые графы не будут содержать только одну начальную вершину, оконечные вершины и базовые вершины потокового графа.

ПРИМЕЧАНИЕ 1. – Базовые вершины потокового графа описывают базовые неделимые единицы выполнения. Операционная семантика для поведения TTCN-3 базируется на интерпретации базовых вершин потокового графа. В разделе B.3 представлены методы выполнения только для базовых вершин потокового графа.

Замена ссылочной вершины соответствующим определением сегмента потокового графа может привести к появлению несоединенных частей в потоковом графе, то есть частей, которых нельзя достичь из начальной вершины путем обхода потокового графа вдоль потоковых линий. Операционная семантика будет игнорировать несоединенные части потокового графа.

ПРИМЕЧАНИЕ 2. – Несоединенная часть потокового графа является результатом механической процедуры замены. Для конструирования оптимального представления потокового графа необходимо учитывать также различные комбинации команд TTCN-3. Однако задачей данного Приложения является обеспечение правильной и полной семантики, а не оптимальное представление потокового графа.

B.3.2.2 Представление потокового графа для управления модулем

Синтаксическая структура модуля TTCN-3 схематически выглядит так:

```
module <identifier> (<parameter>) <module-definitions-part> control
                                                        <statement-block>
```

Для представления поведения потокового графа подходит только следующая информация:

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

Это сравнимо с определением функции, поэтому представление потокового графа для управления модулем аналогично представлению потокового графа для функции (см. п. B.3.2.4). Семантика будет иметь доступ к потоковому графу, представляющему управление модулем, с помощью имени модуля.

ПРИМЕЧАНИЕ. – Смысл определяющей части модуля выходит за рамких этой операционной семантики. Параметры модуля определяются как глобальные константы во время действия. Ссылки на параметры модуля должны заменяться их конкретными значениями на синтаксическом уровне (см. п. B.2.3).

B.3.2.3 Представление потокового графа для тестовых примеров

Синтаксическая структура определения тестового примера TTCN-3 схематически выглядит так:

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

Здесь `<testcase-interface>` указывает на разделы **runs on** (обязательный) и system (факультативный) в определении тестового примера. Описание потокового графа для тестового

примера определяет поведение MTC. Информация, содержащаяся в **<testcase-interface>**, не подходит для MTC. Она будет использоваться командой **execute**, но не должна быть представлена в представлении потокового графа для тестового примера. Поэтому для такого представления потокового графа подходит только следующая информация:

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

Это сравнимо с определением функции, поэтому представление потокового графа для тестового примера аналогично представлению потокового графа для функции (см. п. B.3.2.4). Семантика будет иметь доступ к потоковым графам, представляющим тестовые примеры, с помощью имен тестовых примеров.

B.3.2.4 Представление потокового графа для функции

Синтаксическая структура функции TTCN-3 схематически выглядит так:

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

Здесь факультативный <function-interface> указывает на разделы runs on и return в определении функции. Информация, содержащаяся в <function-interface>, не подходит для описания поведения. Она будет использоваться для проверок статической семантики, но не должна быть представлена в потоковом графе. Поэтому для такого представления потокового графа подходит только следующая информация:

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

Семантика будет иметь доступ к потоковым графам, представляющим функции, с помощью имен функций.

Схема представления потокового графа для функции показана на рис. B.10. Имя этого потокового графа **<identifier>** указывает имя представляемой функции (или управления модулем, или тестового примера). Вершины потокового графа имеют связанные комментарии, описывающие смысл различных вершин.

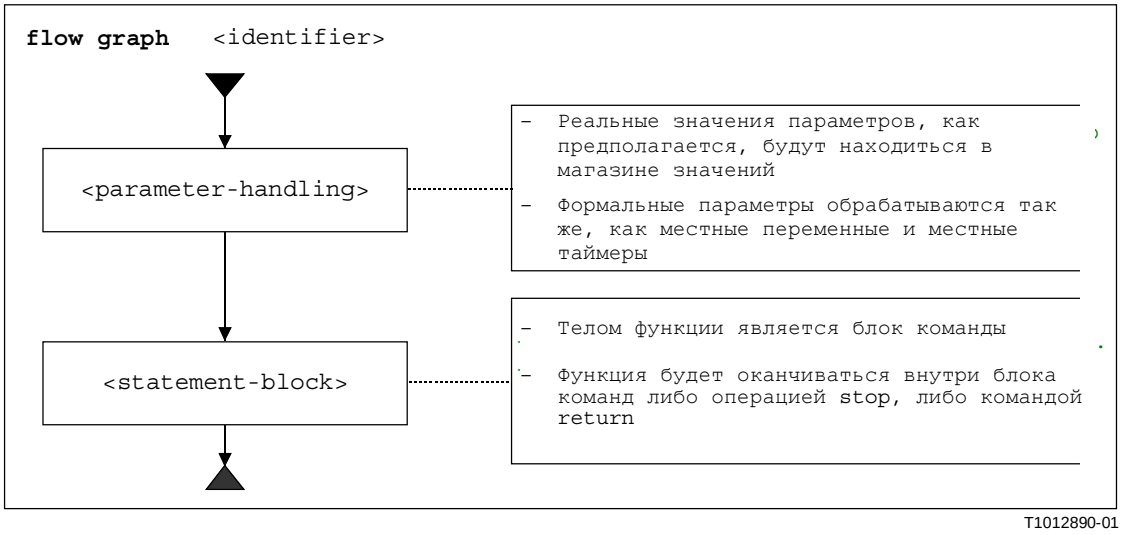


Рисунок B.10/Z.140 – Представление потокового графа для функций

B.3.2.5 Представление потокового графа для определений компонентных типов

Синтаксическая структура определения компонентного типа (типа компонента) схематически выглядит так:

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

Семантика будет иметь доступ к потоковым графам, представляющим типы, с помощью имен компонентных типов.

Схема представления потокового графа для определения компонентного типа показана на рис. B.11. Имя этого потокового графа <identifier> указывает имя представляемого компонентного типа.

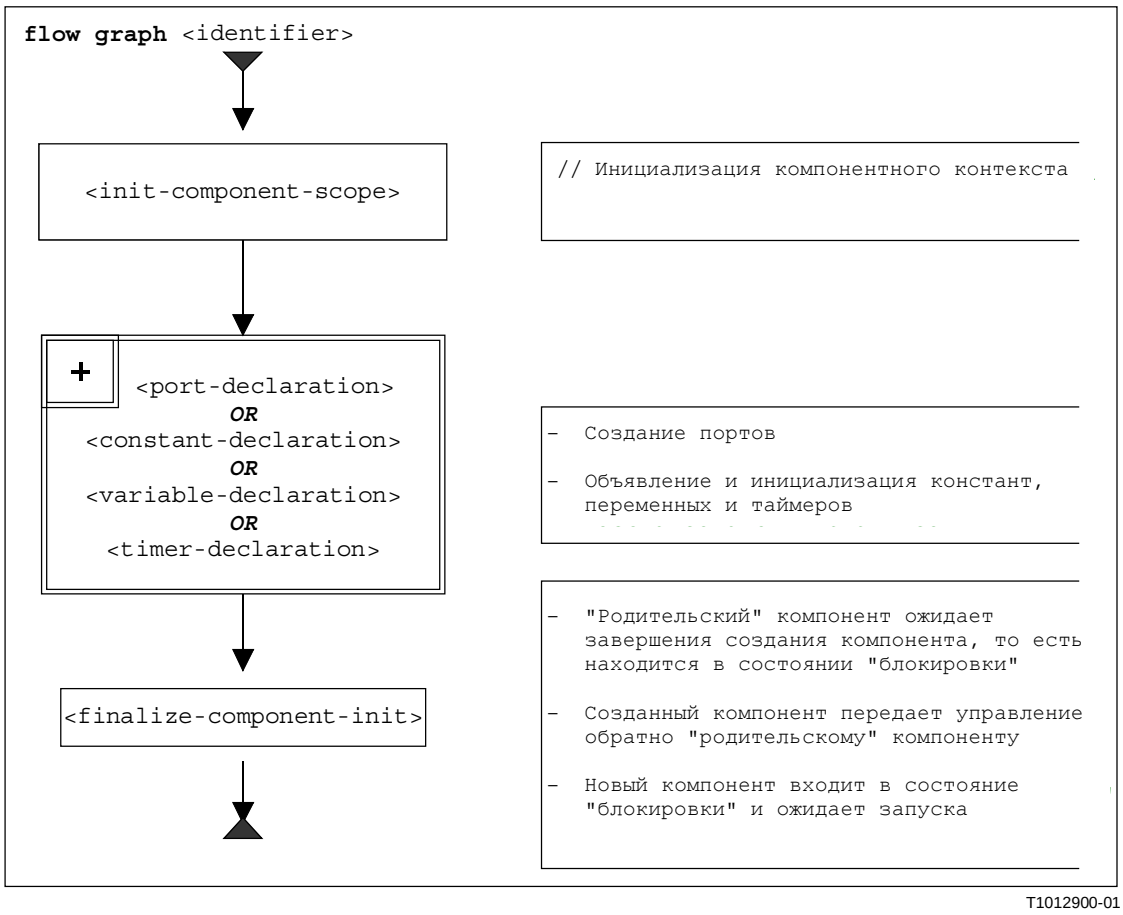


Рисунок B.11/Z.140 – Представление потокового графа для определений компонентных типов

B.3.2.6 Запрос запуска вершин потоковых графов

Для запроса ссылки на начальную вершину потокового графа необходима следующая функция:

Функция GET-FLOW-GRAPH: GET-FLOW-GRAPH (<flow-graph-identifier>)

Эта функция выдает ссылку на начальную вершину потокового графа с именем <*flow-graph-identifier*>. Этот <*flow-graph-identifier*> указывает имя модуля для управления, имена тестовых примеров, имена функций и определения компонентных типов.

B.3.3 Определения состояний для модулей TTCN-3

Во время интерпретации потоковых графов, представляющих поведение TTCN-3, манипулируют с *состояниями модуля*. Состояние модуля – это структурированное состояние, которое состоит из нескольких подсостояний, описывающих состояния тестовых компонентов и портов. В этом подразделе вводятся состояния модуля, состояния компонента и состояния порта. Кроме того, определяются функции для запроса информации от манипулируемых состояний и к ним.

B.3.3.1 Состояние модуля

Как показано на рис. B.12, состояние модуля подразделяется на *список состояний объектов*, *список состояний портов*, ссылку на *MTC* и *TC-VERDICT*. *Список состояний объектов* описывает состояние управления модулем, а также состояния создаваемых тестовых компонентов во время выполнения тестового примера. *Список состояний портов*, ссылка на *MTC* и *TC-VERDICT* действуют только во время выполнения тестового примера.

Список состояний портов описывает состояния разных портов. *MTC* дает ссылку на MTC, *TC-VERDICT* хранит реальный глобальный тестовый вердикт для тестового примера, а DONE является счетчиком, подсчитывающим число обновлений *TC-VERDICT*.

ПРИМЕЧАНИЕ 1. – Число обновлений TC-VERDICT идентично числу тестовых компонентов, которые были завершены.

Поведение управления модулем (M-CONTROL на рис. B.12) обрабатывается так же, как обычный тестовый компонент, а его состояние будет первым элементом в *списке состояний объектов* для состояния модуля.

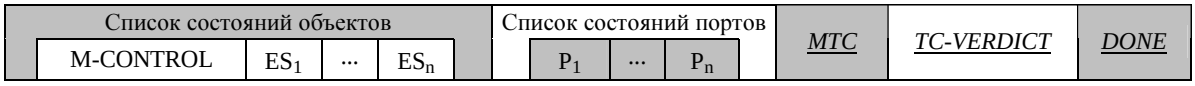


Рисунок B.12/Z.140 – Структура состояния модуля

ПРИМЕЧАНИЕ 2. – Состояния портов могут рассматриваться как часть состояний объектов. Однако благодаря **`connect`** и **`map`** порты становятся видимыми для других компонентов и поэтому они обрабатываются на верхнем уровне состояния модуля.

B.3.3.1.1 Доступ к состоянию модуля

MTC, *SYSTEM*, *TC-VERDICT* и *DONE* являются частями состояния модуля и обрабатываются так же, как глобальные переменные, то есть ключевые слова *MTC* и *TC-VERDICT* могут использоваться для запроса и изменения значений соответствующего состояния модуля.

ПРИМЕЧАНИЕ 1. – Во время интерпретации модуля TTCN-3 существует только одно состояние модуля. Поэтому ключевые слова *MTC* и *TC-VERDICT* могут рассматриваться как уникальные идентификаторы для процедуры оценки.

Для обработки *списка состояний объектов* и *списка состояний портов* могут использоваться списковые операции ***append***, ***delete***, *first* и *length*.

ПРИМЕЧАНИЕ 2. – Списковые операции ***append***, ***delete***, *first* и *length* имеют следующий смысл:

- <*list*>.*append* (<*item*>) добавляет <*item*> в качестве последнего элемента в список <*list*>;
- <*list*>.*delete* (<*item*>) удаляет <*item*> из списка <*list*>;
- <*list*>.*first* ( ) выдает первый элемент из <*list*>;
- <*list*>.*length* ( ) выдает длину списка <*list*>;
- <*list*>.*next* (<*item*>) выдает элемент, который следует за <*item*> в списке, либо NULL, если <*item*> является последним элементом в списке.

B.3.3.2 Состояния объектов

Состояния объектов используются для описания реальных состояний управления модулем и тестовых компонентов. Структура состояния объекта показана на рис. B.13.

<*identifier*>	***STATUS***	*CONTROL-STACK*	**Состояние данных**	**Состояние таймера**	*VALUE-STACK*	*E-VERDICT*

Рисунок B.13/Z.140 – Структура состояния объекта

Здесь <*identifier*> является уникальным идентификатором объекта, то есть управления модулем тестового компонента, в тестовой системе. Такие уникальные идентификаторы создаются неявно для управления модулем, для **`mtc`** и тестовой системы, когда модуль начинает выполнение или тестовый пример выполняется с помощью команды **`execute`**. Этот идентификатор используется для идентификации и адресации объектов в тестовой системе, например, в случае операций **`send`** с разделами `to` или операций **`receive`** с разделами **`from`**.

Часть *STATUS* описывает состояние **`ACTIVATE`** или **`BLOCKED`** управления модулем или тестового компонента. Управление модулем блокировано во время выполнения тестового примера. Тестовый компонент может быть блокирован во время создания других тестовых компонентов, то есть во время выполнения операции **`create`**.

Часть *CONTROL-STACK* является магазином ссылок на вершины потокового графа. Верхним элементом в *CONTROL-STACK* является вершина потокового графа, которая должна интерпретироваться согласно очереди. Этот магазин должен моделировать вызовы функций соответствующим образом.

Состоянием данных считается список списков связываний переменных. Структура списка списков отражает вложенные единицы контекста, обусловленные вложенными вызовами функций. Каждый список в списке списков связываний переменных описывает связывания переменных в определенной единице контекста. Вхождение в единицу контекста или выход из нее соответствует добавлению или исключению списка связываний переменных в *состоянии данных*. Более подробное описание части *состояние данных* в состоянии объекта можно найти в п. B.3.3.2.2.

C*остоянием таймера* считается список списков состояний таймеров. Структура списка списков отражает вложенные единицы контекста, обусловленные вложенными вызовами функций. Каждый список в списке списков состояний таймеров описывает связи таймеров (известных таймеров и их состояний) в определенной единице контекста. Вхождение в единицу контекста или выход из нее соответствует добавлению или исключению списка состояний таймеров в части *состояние таймера*. Более подробное описание части *состояние таймера* в состояние объекта можно найти в п. B.3.3.2.3.

Часть *VALUE-STACK* является магазином значений всех возможных типов, который разрешает промежуточное хранение окончательных или промежуточных результатов операций, функций и команд. Например, в *VALUE-STACK* будет помещен результат выполнения выражения или результат функции mtc. В дополнение к значениям всех типов данных, известных в модуле, определим специальное значение MARK как часть алфавита магазина. Когда происходит выход из единицы контекста, MARK используется для очистки *VALUE-STACK*.

Часть *E-VERDICT* хранит реальный местный вердикт тестового компонента. Если состояние объекта представляет управление модуля, то *E-VERDICT* игнорируется.

B.3.3.2.1 Доступ к состоянию объекта

Части *STATUS* и *E-VERDICT* состояния объекта обрабатываются так же, как глобальные переменные, то есть значения *STATUS* и *E-VERDICT* могут запрашиваться или изменяться с помощью 'dot'-нотации <*identifier*>.STATUS и <*identifier*>.*E-VERDICT*. Член <*identifier*> в этой нотации 'dot' указывает уникальный идентификатор объекта.

Обратиться к *CONTROL-STACK* и *VALUE-STACK* состояния объекта можно с помощью 'dot'-нотации <*identifier*>.*CONTROL-STACK* и <*identifier*>.*VALUE-STACK*.

Доступ к *CONTROL-STACK* и *VALUE-STACK* и манипуляции с ними могут осуществляться с помощью магазинных операций ***push***, ***pop***, ***top***, *clear* и *clear-until*.

ПРИМЕЧАНИЕ. – Магазинные операции ***push***, ***pop***, ***top***, *clear* и *clear-until* имеют следующий смысл:

- <*stack*>.*push* (<*item*>) помещает <*item*> в <*stack*>;
- <*stack*>.*pop* ( ) выбирает (выталкивает) верхний элемент из <*stack*>;
- <*stack*>.*top* ( ) выдает верхний элемент из <*stack*> или NULL, если <*stack*> пуст;
- <*stack*>.*clear* ( ) очищает <*stack*>, то есть выталкивает все элементы из <*stack*>;
- <*stack*>.*clear-until* (<*item*>) выталкивает элементы из <*stack*> до тех пор, пока не будет найдет <*item*> или пока <*stack*> не будет пуст.

Предполагается, что для создания нового состояния объекта доступна функция *NEW-ENTITY*.

NEW-ENTITY (<*entity-identifier*>, <*flow-graph-node-reference*>) создает новое состояние объекта и выдает его порядковый номер. Компоненты нового состояния объекта имеют следующие значения:

- <entity-identifier> является уникальным идентификатором;
- *STATUS* устанавливается в **ACTIVE**;
- <flow-graph-node-reference> является единственным (верхним) элементом в *CONTROL-STACK*;
- *состояние данных* и *состояние таймера* являются пустыми списками;

- *VALUE-STACK* является пустым магазином;
- *E-VERDICT* устанавливается в none.

Во время обхода потокового графа *CONTROL-STACK* часто изменяет свое значение одним и тем же образом: верхний элемент выталкивается из *CONTROL-STACK*, а вершина-преемник вытолкнутой вершины помещается в *CONTROL-STACK*. Эта серия магазинных операций оформляется в функцию *NEXT-CONTROL*:

```
<identifier>.NEXT-CONTROL(boolean <bool>) {
    FlowGraphNodeType successorNode := <identifier>.CONTROL-STACK.NEXT(<bool>).top();
    <identifier>.CONTROL-STACK.pop();
    <identifier>.CONTROL-STACK.push(successorNode).
}
```

B.3.3.2.2 Состояние данных и связывание переменной

Как показано на рис. B.14, *состояние данных* является списком списков связываний переменных. Каждый список связываний переменных описывает связывания переменных в определенной единице контекста. Добавление нового списка связываний переменных соответствует вхождению в новую единицу контекста, например, когда вызвана некоторая функция. Исключение списка связываний переменных соответствует выходу из единицы контекста, например, когда функция выполняет команду return.

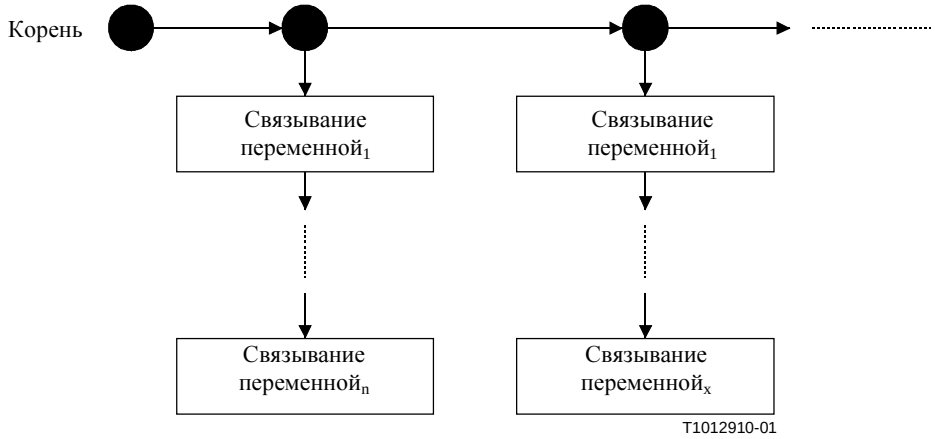


Рисунок B.14/Z/140 – Структура части "состояние данных" в состоянии объекта

Структура связывания переменной показана на рис. B.15. Переменная имеет имя *<var-name>*, *положение* и *значение*. Часть *<var-name>* определяет переменную в единице контекста. *Положение* является уникальным указателем положения значения переменной в памяти. Часть *значение* в связывании переменной описывает реальное значение переменной.

ПРИМЕЧАНИЕ. – Уникальные указатели положения появляются автоматически при объявлении переменной.

<var-name>	**положение**	**значение**

Рисунок B.15/Z.140 – Структура связывания переменной

Различие между именем переменной и положением сделано для формирования вызовов функции и выполнения тестовых примеров с параметризацией значения и сылки соответствующим образом:

a) параметр, пересылаемый с помощью значения, обрабатывается так же, как объявление новой переменной, то есть новое связывание переменной добавляется к списку связываний переменных в контексте вызванной функции или выполняемого тестового примера. Новое связывание переменной использует имя формального параметра в качестве *<var-name>*,

получает новое положение и берет значение, которое переслано в функцию или тестовый пример;

b) параметр, пересылаемый с помощью ссылки, также приводит к новому связыванию переменной в контексте вызванной функции или выполняемого тестового примера. Новое связывание переменной также использует имя формального параметра в качестве <*var-name*>, но не получает нового положения и нового значения. Новое связывание переменной берет копии *положения* и *значения* переменной, пересланные с помощью ссылки.

При обновлении значения переменной, например в случае присвоения переменной, имя переменной используется для идентификации положения, а все связывания переменных с одним и тем же положением обновляются в одно и то же время. Следовательно, при выходе из единицы контекста список переменных, принадлежащий этой единице контекста, может быть исключен без последующего обновления. Благодаря процедуре обновления переменные, пересланные с помощью ссылки, будут автоматически иметь правильное значение.

B.3.3.2.3 Состояние таймера и связывание таймера

Как показано на рис. B.16 и B.17, *состояние таймера* и *состояние данных* в состоянии объекта сравнимы. Оба являются списком списков связываний, а каждый список связываний определяет действительные связывания в определенном контексте. Добавление нового списка соответствует вхождению в новую единицу контекста, а исключение списка связываний – выходу из единицы контекста.

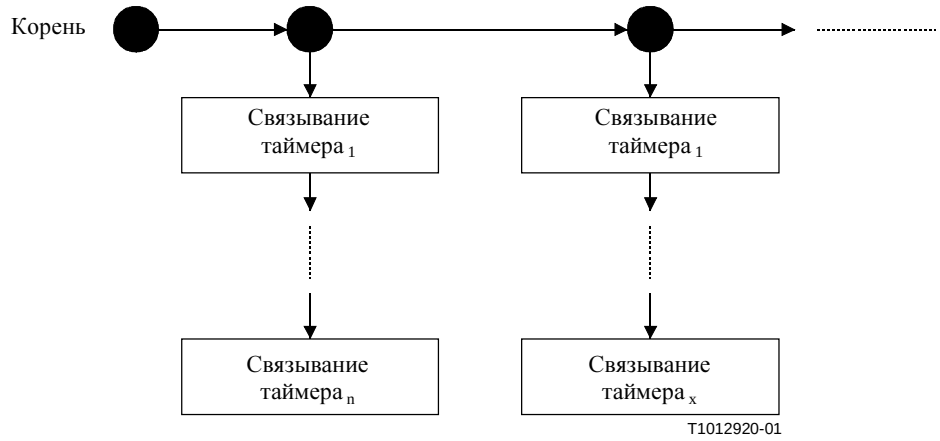


Рисунок B.16/Z.140 – Структура части "состояние таймера" в состоянии объекта

Структура связывания таймера показана на рис. B.17. Значения частей <*timer-name*> и *положение* аналогичны значениям <*var-name*> и *положение* в связывании переменной (рис. B.15).

<*timer-name*>	**Положение**	*STATUS*	*DEF-DURATION*	*ACT-DURATION*	*TIME-LEFT*

Рисунок B.17/Z.140 – Структура связывания таймера

Часть *STATUS* указывает, является ли таймер активным, неактивным или его выдержка истекла. Соответствующими значениями *STATUS* являются **`IDLE`**, **`RUNNING`** и **`TIMEOUT`**. Часть *DEF-DURATION* описывает выдержку таймера по умолчанию. Часть *ACT-DURATION* хранит реальную выдержку, с которой считающий таймер был запущен. Часть *TIME-LEFT* описывает реальную длительность, которую считающему таймеру осталось считать до истечения выдержки.

ПРИМЕЧАНИЕ. – *DEF-DURATION* будет неопределенной, если объявлено, что таймер не имеет выдержки по умолчанию. *ACT-DURATION* и *TIME-LEFT* устанавливаются в 0,0, когда таймер остановлен или когда у него истекла выдержка. Если таймер запущен без установленной выдержки, то значение *DEF-DURATION* копируется в *ACT-DURATION*. Если таймер запущен без какой-либо определенной выдержки, то произойдет динамическая ошибка.

Таймер может пересылаться в функции только с помощью ссылки, то есть этот механизм аналогичен механизму для переменных, описанному в п. B.3.3.2.2. Это означает создание нового связывания

таймера (с именем формального параметра в виде <*timer-name*>), которое берет копии *положения*, *STATUS*, *DEF-DURATION*, *ACT-DURATION* и *TIME-LEFT* от таймера, который переслан с помощью ссылки. При обновлении <*timer-name*> одновременно обновляются все связывания таймеров с одним и тем же *положением*.

B.3.3.2.4 Доступ к таймеру и состояниям данных

Значение переменной может быть затребовано с помощью dot-нотации <*identifier*>.<var-name>, где <identifier> указывает уникальный *идентификатор* объекта. Для изменения значения переменной должна использоваться функция *VAR-SET*:

<*identifier*>.*VAR-SET* (<*var-name*>, <*value*>)

устанавливает значение переменной <*var-name*> в реальный контекст объекта с уникальным идентификатором <*identifier*>. Кроме того, значения всех переменных с тем же положением, что и у переменной <*var-name*>, будут также установлены в <*value*>.

Значения *STATUS*, *DEF-DURATION*, *ACT-DURATION* и *TIME-LEFT* таймера <*timer-name*> могут быть затребованы с помощью dot- нотации:

<*identifier*>.<*timer-name*>.*STATUS*;

<*identifier*>.<*timer-name*>.*DEF-DURATION*;

<*identifier*>.<*timer-name*>.*ACT-DURATION*;

<*identifier*>.<*timer-name*>.*TIME-LEFT*.

Для изменения значений *STATUS*, *DEF-DURATION*, *ACT-DURATION* и *TIME-LEFT* таймера <*time-name*> должна использоваться общая операция TIMER-SET. Например:

<*identifier*>.*TIMER-SET* (<*timer-name*>, *STATUS*, <*value*>

устанавливает значение *STATUS* таймера <*timer-name*> в реальном контексте объекта с уникальным идентификатором <*identifier*> в значение <*value*>. Кроме того, *STATUS* всех таймеров с тем же положением, что и у таймера <*timer-name*>, будут также установлены в <*value*>. Функция TIMER-SET также может использоваться для изменения значений *DEF-DURATION*, *ACT-DURATION* и *TIME-LEFT*.

Для обработки переменных, таймеров и единиц контекста должны быть определены следующие функции:

a) Функция *INIT-VAR*: <*identifier*>.*INIT-VAR* (<*var-name*>, <*value*>)

создает новое связывание переменной для переменной <*var-name*> с начальным значением <*value*> в реальной единице контекста объекта с уникальным идентификатором <*identifier*>. Использование ключевого слова **NONE** в качестве <*value*> означает, что создается переменная с неопределенным начальным значением.

b) Функция *INIT-TIMER*: <identifier>.*INIT-TIMER* (<*timer-name*>, <*duration*>)

создает новое связывание таймера для таймера <*timer-name*> с выдержкой по умолчанию <*duration*> в реальном контексте объекта с уникальным идентификатором <*identifier*>. Использование ключевого слова **NONE** в качестве <*duration*> означает, что создается таймер без выдержки по умолчанию.

c) Функция *GET-VAR-LOC*: <*identifier*>.*GET-VAR-LOCATION* (<*var-name*>)

запрашивает положение переменной <*var-name*>, принадлежащей объекту с уникальным идентификатором <*identifier*>.

d) Функция *GET-TIMER-LOC*: <*identifier*>.*GET-TIMER-LOCATION* (<*timer-name*>)

запрашивает положение таймера <*timer-name*>, принадлежащего объекту с уникальным идентификатором <*identifier*>.

e) Функция *INIT-VAR-LOC*: <*identifier*>.*INIT-VAR-LOC* (<*var-name*>, <*location*>)

создает новое связывание переменной для переменной <*var-name*> с положением <*location*> в реальной единице контекста объекта с уникальным идентификатором <*identifier*>. Эта переменная будет инициализирована со значением другой переменной, имеющей то же положение <*location*>.

ПРИМЕЧАНИЕ 1. – Переменные с одинаковым положением являются результатом параметризации с помощью ссылки. Из-за обработки параметров ссылки согласно п. B.3.3.2.2 все переменные с одним и тем же положением будут иметь идентичные значения в течение всего жизненного цикла.

f) Функция *INIT-TIMER-LOC*:<*identifier*>.*INIT-TIMER-LOC* (<*timer-name*>, <*location*>)

создает новое связывание таймера для таймера <*timer-name*> с положением <*location*> в реальной единице контекста объекта с уникальным идентификатором <*identifier*>. Таймер будет инициализирован со значениями *STATUS*, *DEF-DURATION*, *ACT-DURATION* и *TIME-LEFT* другого таймера, имеющего то же положение <*location*>.

ПРИМЕЧАНИЕ 2. – Таймеры с одинаковым положением являются результатом параметризации с помощью ссылки. Из-за обработки параметров ссылки на таймер согласно п. B.3.3.2.3 все таймеры с одним и тем же положением будут иметь идентичные значения для *STATUS*, *DEF-DURATION*, *ACT-DURATION* и *TIME-LEFT* в течение всего жизненного цикла.

g) Функция *INIT-VAR-SCOPE*: <*identifier*>.*INIT-VAR-SCOPE ( )*

инициализирует новый контекст переменной в "состоянии данных" объекта с уникальным идентификатором <*identifier*>, то есть добавляется пустой список в качестве первого списка в списке списков связываний переменных.

h) Функция *INIT-TIMER-SCOPE*: <*identifier*>.*INIT-TIMER-SCOPE ( )*

инициализирует новый контекст таймера в "состоянии таймера" объекта с уникальным идентификатором <*identifier*>, то есть добавляется пустой список в качестве первого списка в списке списков связываний таймеров.

i) Функция *DEL-VAR-SCOPE*: <*identifier*>.*DEL-VAR-SCOPE ( )*

удаляет контекст переменной в "состоянии данных" объекта с уникальным идентификатором <*identifier*>, то есть первый список в списке списков связываний переменных удаляется.

j) Функция *DEL-TIMER-SCOPE*: <*identifier*>.*DEL-TIMER-SCOPE ( )*

удаляет контекст таймера в "состоянии таймера" объекта с уникальным идентификатором <*identifier*>, то есть первый список в списке списков связываний таймеров удаляется.

B.3.3.3 Состояния портов

Состояния портов используются для описания реальных состояний портов. Структура состояния порта показана на рис. B.18. Часть <*port-name*> указывает имя порта, используемое тестовым компонентом <*owner*>, которому принадлежит этот порт, для обозначения порта. Часть *STATUS* указывает реальные состояния порта. Порт может быть в состоянии **`STARTED`** или **`STOPPED`**.

ПРИМЕЧАНИЕ. – Порт в тестовой системе уникально (однозначно) определяется тестовым компонентом-владельцем <*owner*> и местным для <*owner*> именем порта <*port-name*>.

Часть *список соединений* состояния порта отслеживает соединения между разными портами в тестовой системе. Механизм этого поясняется в п. B.3.3.3.1.

Часть *очередь значений* состояния порта содержит элементы данных, которые получены в этом порту, но еще не использованы.

<*port-name*>	<*owner*>	*STATUS*	**Список соединений**	**Очередь значений**

Рисунок B.18/Z.140 – Структура состояния порта

B.3.3.3.1 Обработка соединений между портами

Соединение между двумя тестовыми компонентами осуществляется путем соединения двух их портов с помощью операции **`connect`**. Поэтому компонент может потом использовать имя своего местного порта при адресации удаленной очереди. Как показано на рис. B.19, *соединение* представляется в состояниях обеих соединенных очередей с помощью пары <*remote-entity*> и <*remote-port-name*>. Часть

<*remote-entity*> является уникальным идентификатором тестового компонента, который владеет удаленным портом. Часть <*remote-port-name*> указывает местное имя, используемое в <*remote-entity*> для адресации очереди. TTCN-3 поддерживает соединения портов типа "один ко многим", поэтому все соединения порта заносятся в список.

ПРИМЕЧАНИЕ 1. – Соединения, осуществленные операциями **map**, также обрабатываются в этом списке соединений. Операция **map**: **map** (*PTC1 : MyPort*, **system**.*PCO1*) приводит к новому соединению (**system**, *PCO1*) в состоянии порта для *MyPort*, принадлежащего *PTC1*. Удаленная сторона, к которой присоединен *PCO1*, находится внутри SUT. Его поведение не рассматривается в этой семантике.

ПРИМЕЧАНИЕ 2. – В операционной семантике ключевое слово **system** рассматривается как символический адрес. Соединение (**system**, <*port-name*>) в списке соединений порта указывает, что этот порт отображается в порт <*port-name*> в интерфейсе тестовой системы.

<*remote-entity*>	<*remote-port-name*>

Рисунок B.19/Z.140 – Структура соединения

B.3.3.3.2 Обработка состояний портов

Обработка состояний портов поддерживается следующими методами:

a) Функция *NEW-PORT*: *NEW-PORT(<owner>,<port-name>)*

создает новый порт и выдает его справочный номер. Новый порт принадлежит владельцу <*owner*> и имеет имя <*port-name*> для этого порта, определяемого тестовым компонентом <*owner*> и именем порта <*port-name*>. Состоянием нового порта будет **STARTED**, а *список соединений* и *очередь значений* будут пустыми.

b) Функция *GET-PORT*: *GET-PORT(<owner>, <port-name>)*

выдает ссылку на порт, определяемый тестовым компонентом <*owner*>, который владеет портом, и именем порта <*port-name*>.

c) Функция *GET-REMOTE-PORT*: *GET-REMOTE-PORT(<owner>, <port-name>, <remote-entity>)*

выдает ссылку на порт, который принадлежит тестовому компоненту <*remote-entity*> и соединен с портом, определенным с помощью <*owner*> и <*port-name*>. Выдается символический адрес **SYSTEM**, если удаленный порт отображается в порт интерфейса тестовой системы.

ПРИМЕЧАНИЕ 1. – Функция *GET-REMOTE-PORT* выдает **NULL**, если удаленный порт отсутствует или не может быть однозначно идентифицирован. Специальное значение **NONE** может использоваться в качестве значения параметра <*remote-entity*>, если удаленный объект неизвестен или не запрошен, то есть для этого порта существует только соединение типа "один к одному".

d) *STATUS* порта обрабатывается как переменная. Он может адресоваться путем указания *STATUS* с запросом *GET-PORT*:

GET-PORT(<owner>,<port-name>).STATUS

e) Функция *ADD-CON*: *ADD-CON(<owner>, <port-name>, <remote-entity>, <remote-port-name>)*

добавляет соединение (<*remote-entity*>,<*remote-port-name*>) к списку соединений порта <*port-name*>, принадлежащего владельцу <*owner*>.

f) Функция *DEL-CON*: *DEL-CON(<owner>, <port-name>, <remote-entity>, <remote-port-name>)*

удаляет соединение (<*remote-entity*>, <*remote-port-name*>) из списка соединений порта <*port-name*>, принадлежащего владельцу <*owner*>.

Обращение к очереди значений в состоянии порта и манипулирование ею могут выполняться с помощью известных операций с очередью *enqueue*, *dequeue*, *first* и *clear*. Функция *GET-PORT* или *GET-REMOTE* указывает очередь, к которой следует обратиться.

ПРИМЕЧАНИЕ 2. – Операции с очередью *enqueue*, *dequeue*, *first* и *clear* имеют следующий смысл:

- <*queue*>.*enqueue* (<*item*>) вводит <*item*> в качестве последнего элемента в <*queue*>;
- <*queue*>.*dequene* ( ) удаляет первый элемент из <*queue*>;
- <*queue*>.*first* ( ) выдает первый элемент из <*queue*> или NULL, если <*queue*> пуста;
- <*queue*>.*clear* ( ) удаляет все элементы из <*queue*>.

B.3.3.4 Общие функции для обработки состояний модуля

В операционной семантике предполагается наличие следующих функций для обработки состояний модуля.

ПРИМЕЧАНИЕ. – Во время интерпретации модуля TTCN-3 существует только одно состояние модуля. Предполагается, что компоненты состояния модуля хранятся в глобальных переменных, а не в каком-либо сложном объекте данных. Поэтому следующие функции предназначаются для работы с глобальными переменными и не адресуются к конкретным объектам состояния модуля.

a) Функция *DEL-ENTITY*: *DEL-ENTITY(<entity-identifier>)*

удаляет объект с уникальным идентификатором <*entity-identifier*>. Это удаление охватывает:

– удаление состояния объекта <*entity-identifier*>;

– удаление всех портов, принадлежащих объекту <*entity-identifier*>;

– удаление всех соединений, в которых участвует <*entity-identifier*>.

b) Функция *EXISTING*: *EXISTING(<entity-identifier>)*

выдает *true*, если существует какое-либо состояние объекта для объекта <*entity-identifier*>. В остальных случаях она выдает *false*.

c) Функция *UPDATE-REMOTE-REFERENCES*:

UPDATE-REMOTE-REFERENCES(<source-entity>, <target-entity>)

обновляет переменные и таймеры с одинаковым положением в обоих объектах. Значениями, которые используются для обновления, являются значения переменных и таймеров, принадлежащие объекту-источнику <*source-entity*>.

B.3.4 Сообщения, вызовы процедуры, ответы и особые состояния

Обмен информацией между тестовыми компонентами, а также между тестовыми компонентами и SUT имеет отношение к *сообщениям*, *вызовам процедуры*, *ответам на вызовы процедуры* и *особым состояниям*. Для целей связи эти элементы должны конструироваться, кодироваться и декодироваться. Конкретное кодирование, то есть преобразование типов данных TTCN-3 в биты и байты, и декодирование, то есть преобразование битов и байтов в типы данных TTCN-3, выходят за рамки операционной семантики. В данной Рекомендации *сообщения*, *вызовы процедуры*, *ответы на вызовы процедуры* и *особые состояния* рассматриваются на концептуальном уровне.

B.3.4.1 Сообщения

Сообщения относятся к асинхронной связи. Между объектами, которые осуществляют связь, возможен обмен значениями любых типов данных (предопределенных и определяемых пользователем). Как показано на рис. B.20, в операционной семантике сообщение рассматривается как структурированный объект, который состоит из частей *передатчик* и *значение*. Часть *передатчик* определяет объект, передающий сообщение, а часть *значение* определяет значение сообщения.

Рисунок B.20/Z.140 – Структура сообщения

ПРИМЕЧАНИЕ. – Операционная семантика дает только модель принципов TTCN-3. Когда и как информация *передатчика* передается или должна передаваться и/или приниматься, зависит от реализации тестовой системы, то есть в некоторых случаях информация *передатчика* может быть частью части *значение* в сообщении и поэтому не будет отдельной частью структуры сообщения.

B.3.4.2 Вызовы процедуры и ответы

Вызовы процедуры и ответы на процедуры относятся к синхронным вызовам. Они определяются аналогично значениям записи с компонентами, представляющими параметры. В операционной семантике вызовы процедуры и ответы на них рассматриваются также аналогично значениям в структурированных типах. Структура вызова сообщения и структура ответа представлены на рис. B.21 и B.22.

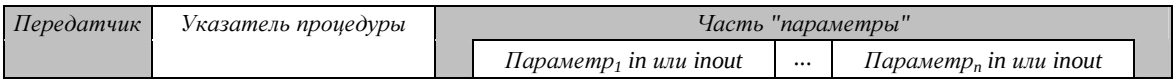


Рисунок B.21/Z.140 – Структура вызова процедуры

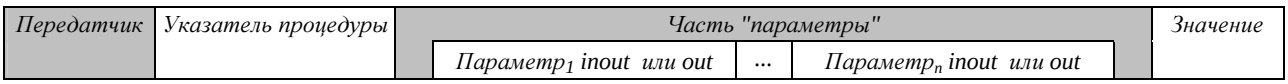


Рисунок B.22/Z.140 – Структура ответа на вызов процедуры

Части *передатчик* и *указатель процедуры* имеют одинаковый смысл на обоих рисунках. Часть *передатчик* указывает объект, передающий вызов или ответ на вызов процедуры. Часть *указатель процедуры* определяет процедуру, к которой относится вызов или ответ. *Часть "параметры"* при вызове процедуры (на рис. B.21) указывает параметры **in** и параметры **inout**, а *часть "параметры"* при ответе (на рис. B.22) указывает параметры **inout** и параметры **out** той процедуры, к которой относятся вызов и ответ. Кроме того, ответ имеет часть *значение* для выдачи значений в ответе на процедуру.

ПРИМЕЧАНИЕ 1. – Как отмечалось в предыдущем примечании, операционная семантика дает только модель принципов TTCN-3. Когда и как информация, описанная на рис. B.21 и B.22, передается или должна передаваться и/или приниматься, зависит от реализации тестовой системы.

ПРИМЕЧАНИЕ 2. – Для вызова процедуры параметры **out** не подходят и опущены на рис. B.21. Для ответа на вызов процедуры параметры **in** не подходят и опущены на рис. B.22.

B.3.4.3 Особые состояния

Особые состояния также относятся к синхронной связи. Структура особого состояния показана на рис. B.23. Она состоит из трех частей. Часть *передатчик* определяет передатчик особого состояния; часть *указатель процедуры* определяет процедуру, к которой относится это особое состояние, а часть *значение* дает значение особого состояния. Тип значения особого состояния определяется в сигнатуре процедуры, приведенной в части *указатель процедуры*. В общем случае им может быть любой из предопределенных или определяемых пользователем типов данных TTCN-3.

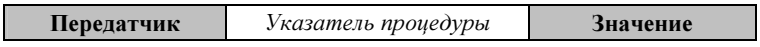


Рисунок B.23/Z.140 – Структура особого состояния

B.3.4.4 Конструирование сообщений, вызовов процедуры, ответов и особых состояний

Для передачи сообщения, вызова процедуры, ответа на вызов процедуры или особого состояния применяются операции **send, call, reply** и **raise**. Все эти операции передачи строятся одинаково:

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

Части <*port-name*> и <*sending-operation*> определяют порт и операцию, используемые для передачи элемента. В случае соединений типа "один ко многим" должен указываться элемент <*receiver*>. Элемент, который будет передаваться, конструируется с помощью <*send-specification*>. Эта "спецификация передачи" может использовать конкретные значения, ссылки на шаблоны, значения переменных, константы, выражения, функции и т. п. для конструирования и кодирования передаваемого элемента.

В операционной семантике предполагается, что имеется общая функция *CONSTRUCT-ITEM*:

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

  выдает сообщение, вызов процедуры, ответ на вызов процедуры или особое состояние в зависимости от <*sending-operation*> и <*send-specification*>. Информация <*sender*> также считается частью передаваемого элемента (рис. B.20–B.23).

B.3.4.5 Сопоставление сообщений, вызовов процедуры, ответов и особых состояний

Для приема сообщения, вызова процедуры, ответа на вызов процедуры или особого состояния применяются операции **`receive, getcall, getreply`** и **`catch`**. Все эти операции строятся одинаково:

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

Части <*port-name*> и <*receiving-operation*> определяют порт и операцию, используемые для приема элемента. В случае соединений типа "один ко многим" для выбора конкретного передающего элемента <*sender*> может использоваться раздел **`from`**. Принимаемый элемент должен соответствовать условиям, определенным в <*matching-part*>, то есть он должен сопоставляться. В части <*matching-part*> могут использоваться конкретные значения, ссылки на шаблоны, значения переменных, константы, выражения, функции и т. п. для описания условий сопоставления.

В операционной семантике предполагается, что имеется общая функция *MATCH-ITEM*:

- *MATCH-ITEM(*<*item-to-check*>, <*matching-part*>, <*sender*>*)*

  выдает true, если <*item-to-check*> удовлетворяет условиям из <*matching-part*>, а <*item-to-check*> был передан из <*sender*>; в остальных случаях она выдает false.

B.3.4.6 Выборка информации из принятых элементов

Информация из принятых сообщений, вызовов процедуры, ответов на вызовы процедуры и особых состояний может быть выбрана в <*assignment-part*> (см. п. B.3.4.5) функций приема **`receive, getcall, getreply`** и **`catch`**. Часть <*assignment-part*> описывает, как переменным присваиваются параметры вызовов процедуры и ответов на них, выдаваемые значения, закодированные в ответах, сообщениях, особых состояниях, а также идентификатор элемента <*sender*>.

В операционной семантике предполагается, что имеется общая функция *RETRIEVE-INFO*:

- *RETRIEVE-INFO(*<*item-received*>, <*assignment-part*>, <*receiver*>*)*

  Все значения, которые необходимо выбрать согласно <*assignment-part*>, выбираются и присваиваются переменным, перечисленным в этой <*assignment-part*>. Присвоения производятся с помощью операции VAR-SET, то есть переменные с одним и тем же положением обновляются одновременно.

B.3.5 Записи вызовов функций и тестовых примеров

Функции и тестовые примеры вызываются (или выполняются) по их именам и списку реальных параметров. Реальные параметры имеют справочные номера для параметра "ссылка" и конкретные значения для параметра "значение", как определено формальными параметрами в определении функции или тестового примера. В операционной семантике вызовы функции и вызовы тестовых примеров обрабатываются с помощью *call records*, как показано на рис. B.24. Значение *BEHAVIOUR-ID* является именем функции или тестового примера, часть "параметры значения" дает конкретные значения <$parId_1$> … <$parId_n$> для формальных параметров <$parId_1$> … <$parId_n$>. Часть "параметры

ссылки" дает ссылки на положения имеющихся переменных и таймеров. Соответствующая запись вызова должна быть сконструирована до момента, когда функция или тестовый пример могут быть выполнены.

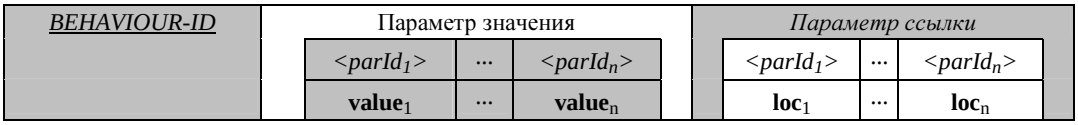


Рисунок B.24/Z.140 – Структура записи вызова

B.3.5.1 Обработка записей вызовов

Имя функции или тестового примера и реальные значения параметров могут быть запрошены с помощью dot-нотации, например, <*myRecord*>.<*parId*$_n$> или <*myRecord*>.*BEHAVIOUR-ID*, где <*myRecord*> является указателем к записи вызова.

Предполагается, что для конструирования вызова будет доступна функция *NEW-CALL-RECORD*:

- *NEW-CALL-RECORD(<behaviour-name>)*

  создает новую запись вызова для функции или тестового примера <*behaviour-name*> и выдает указатель для новой записи. Поля параметров новой записи вызова имеют неопределенные значения.

- *<call-record>.INIT-CALL-RECORD()*

  создает переменные и таймеры для обработки параметров значения и ссылки в реальном контексте функции или тестового примера. Переменные для обработки параметров значения инициализируются с соответствующими значениями, имеющимися в записи вызова. Переменные и таймеры для обработки параметров ссылки получают обеспеченное положение. Кроме того, они получают значение существующей переменной или таймера в другой единице контекста того компонента, в котором эта запись вызова была создана.

B.3.6 Процедура оценки для модуля TTCN-3

B.3.6.1 Фазы оценки

Процедура оценки (определения, вычисления) для модуля TTCN-3 состоит из:

1) *фазы инициализации;*

2) *фазы обновления;*

3) *фазы выбора;* и

4) *фазы выполнения.*

Фазы 2), 3) и 4) повторяются до окончания управления модулем. Процедура оценки описывается с помощью сочетания неформального текста, псевдокода и функций, введенных в предыдущих разделах.

B.3.6.1.1 Фаза 1: Инициализация

Фаза инициализации состоит из следующих действий:

а) **Объявление и инициализация переменных**

INIT-FLOW-GRAPHS(); // Инициализация обработки потокового графа.
// *INIT-FLOW-GRAPHS* поясняется в п. B.3.6.2.

– *Entity* : = **NULL**; // *Entity* будет использоваться для ссылки на
// состояние объекта. Состояние объекта
// указывает либо на управление модулем, либо
// на тестовый компонент.

– *AllEntities* : = **NULL**; // *AllEntities* будет списком состояний объектов

– *AllPorts* : = **NULL**; // *AllPorts* будет списком состояний портов
– *MTC* : = **NULL**; // *MTC* будет отсылать к MTC, когда выполняется
// тестовый пример
– *TC-VERDICT* : = **none**; // *TC-VERDICT* будет хранить реальный вердикт
// тестового примера, когда выполняется
// тестовый пример
– *DONE* : = 0; // Во время выполнения тестового примера *DONE*
// подсчитывает число тестовых компонентов,
// которые окончены.

ПРИМЕЧАНИЕ. – Глобальные переменные *AllEntities*, *AllPorts*, *MTC*, *TC-VERDICT* и *DONE* формируют состояние модуля, которое подвергается манипуляциям во время интерпретации модуля TTCN-3.

b) **Создание и инициализация управления модулем**

– *Entity* : = *NEW-ENTITY* (*GET-UNIQUE-ID*(), *GET-FLOW-GRAPH* (<*moduleId*>));
// Новое состояние объекта создается и
// инициализируется с начальной вершины
// потокового графа, представляющего поведение
// управления для модуля с именем <*moduleId*>.
// *GET-UNIQUE-ID* поясняется в п. B.3.6.2.
– *Entity*.*INIT-VAR-SCOPE*(); // Новый контекст переменной
– *Entity*.*INIT-TIMER-SCOPE*(); // Новый контекст таймера
– *Entity*.*VALUE-STACK*.*push*(**MARK**); // Метка (MARK) опубликована в
// магазине значений
– *AllEntities*.*append*(*Entity*); // Новый объект включается в состояние модуля.

B.3.6.1.2 Фаза II: Обновление

Фаза обновления относится ко всем действиям, которые находятся за рамками операционной семантики, но влияют на интерпретацию модуля TTCN-3. Фаза обновления состоит из следующих действий:

a) **Продвижение времени**: Все считающие таймеры обновляются, то есть значения *TIME-LEFT* считающих таймеров (возможно) уменьшаются, а если из-за обновления у какого-либо таймера истекла выдержка, то обновляются связывания соответствующего таймера, то есть TIME-LEFT устанавливается в 0,0, а *STATUS* – в TIMEOUT;

b) **Поведение SUT**: Сообщения, удаленные вызовы процедуры, ответы на удаленные вызовы процедуры и (возможно) особые состояния, полученные от SUT, помещаются в очереди портов, в которых будут производиться соответствующие операции приема.

ПРИМЕЧАНИЕ. – В этой операционной семантике не делаются какие-либо предположения относительно продвижения времени и поведения SUT.

B.3.6.1.3 Фаза III: Выбор

Фаза выбора состоит из следующих двух действий:

a) **Выбор**: Выбирается незаблокированный объект, то есть объект, который имеет значение *STATUS*, равное **ACTIVE**;

b) **Хранение**: Идентификатор выбранного объекта записывается в глобальную переменную *Entity*.

B.3.6.1.4 Фаза IV: Выполнение

Фаза выполнения состоит из следующих двух действий:

a) **Шаг выполнения выбранного объекта**: Выполняется верхняя вершина потокового графа в *CONTROL-STACK* из *Entity*;

b) **Проверка критерия окончания**: Останавливается выполнение, если закончилось управление модулем, то есть список состояний объекта стал пустым, в остальных случаях следует продолжить с фазы II.

ПРИМЕЧАНИЕ. – Шаг выполнения выбранного объекта может рассматриваться как вызов процедуры. Проверка критерия окончания производится, когда шаг выполнения заканчивается, то есть возвращает управление.

B.3.6.2 Глобальные функции

Процедура оценки использует глобальные функции *INIT-FLOW-GRAPHS* и *GET-UNIQUE-ID*:

a) *INIT-FLOW-GRAPHS* рассматривается как функция, которая инициализирует обработку потокового графа. Эта обработка может охватывать создание потоковых графов и обработку указателей к потоковым графам и вершинам потокового графа.

b) *GET-UNIQUE-ID* рассматривается как функция, которая выдает уникальный идентификатор каждый раз, когда она вызывается. Уникальный идентификатор может быть реализован в виде переменной счетчика, которая увеличивается на 1 и выдается каждый раз, когда вызывается *GET-UNIQUE-ID*.

Псевдокод, применяемый в последующих разделах для описания выполнения вершин потокового графа, использует функции *CONTINUE-COMPONENT*, *RETURN*, **`***DYNAMIC-ERROR***:`**

c) *CONTINUE-COMPONENT*, реальный тестовый компонент, продолжает свое выполнение с вершины, лежащей на верху управляющего магазина, то есть управление не передается обратно к процедуре оценки модуля, описанной в этом подразделе.

d) *RETURN* передает управление обратно к процедуре оценки модуля, описанной в этом подразделе. Функция *RETURN* является последним действием '*шага выполнения выбранного объекта*' в фазе *выполнение*.

e) **`***DYNAMIC-ERROR***:`** указывает на появление динамической ошибки. Сама процедура обработки ошибок выходит за рамки операционной семантики. Если динамическая ошибка появилась, то все последующее поведение модуля считается неопределенным.

ПРИМЕЧАНИЕ. – Появление динамической ошибки относится к поведению теста. Динамическая ошибка, описанная операционной семантикой, указывает на проблему в использовании TTCN-3, например, неправильное использование или состояние конфликта.

B.3.7 Определения сегментов потокового графа для конструкций TTCN-3

Операционная семантика представляет поведение TTCN-3 в форме потоковых графов. Алгоритм конструирования для потоковых графов, представляющих поведение, был описан в п. B.3.2.1. Он базируется на шаблонах для потоковых графов и сегментов потоковых графов, которые должны использоваться при конструировании конкретных потоковых графов для определений управления модулем, тестовых примеров, функций и типов компонентов, описанных в модуле TTCN-3. Определения шаблонов для сегментов потокового графа можно найти в этом подразделе. Они представлены не в логическом, а в алфавитном порядке (английских названий).

Определения сегментов потокового графа даются в виде рисунков. Вершины порогового графа представляются в левой части рисунка, а комментарии, связанные с вершинами и потоковыми линиями, помещаются в правой части. Описательные комментарии даются для ссылочных вершин, а комментарии в форме псевдокода связываются с базовыми вершинами. Псевдокод описывает, как интерпретируется базовая вершина, то есть изменяет состояние модуля. Он использует функции, определенные в предыдущих частях раздела B.3, и глобальные переменные, объявленные и инициированные в процедуре оценки для модулей TTCN-3 (в п. B.3.6). В этом подразделе приводится также общий обзор всех функций и ключевых слов, используемых псевдокодом.

B.3.7.1 Команда Alt

В команде **`alt`**, представленной на рис. B.25 в виде потокового графа, различаются команды **`alt`**, которые имеют ветвь **`else`**, и команды **`alt`**, которые не имеют ветви **`else`**.

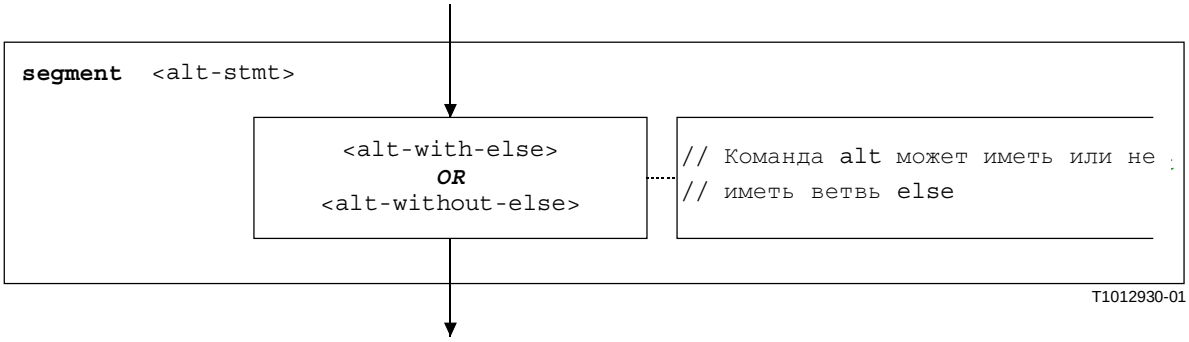


Рисунок B.25/Z.140 – Сегмент <alt-stmt> потокового графа

Сегменты **<**alt-with-else**>** и **<**alt-without-else**>** потокового графа показаны на рис. B.26 и B.27. Ветвь **else** является блоком команды, который не требует дальнейших пояснений. Однако оба сегмента потокового графа весьма похожи; разница лишь в том, что ветвь **else** обеспечивает определенное завершение команды **alt**, а команда **alt** без ветви **else** может зациклиться.

Оба сегмента потокового графа имеют входную вершину и помимо одной входящей потоковой линии имеют дополнительную потоковую линию с меткой <altId>. Это – символическая метка для команды **alt**. Она указывает адресата команды goto **alt** и определяет также цикличность в сегменте <alt-without-else> потокового графа. Оба сегмента потокового графа имеют также определенную выходную точку, показанную с помощью метки <altIdExit> и вершины alt-exit.

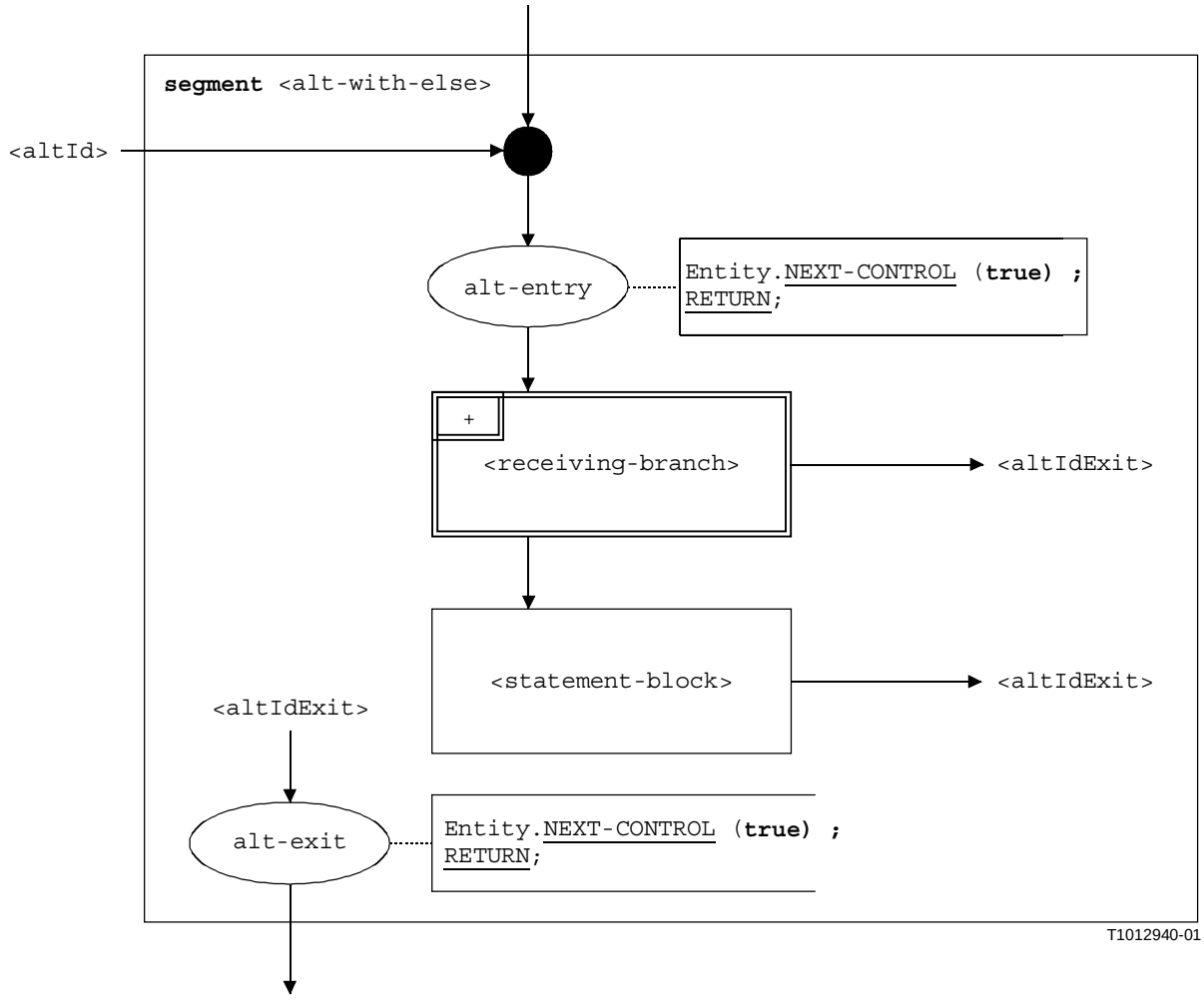


Рисунок B.26/Z.140 – Сегмент <alt-with-else> потокового графа

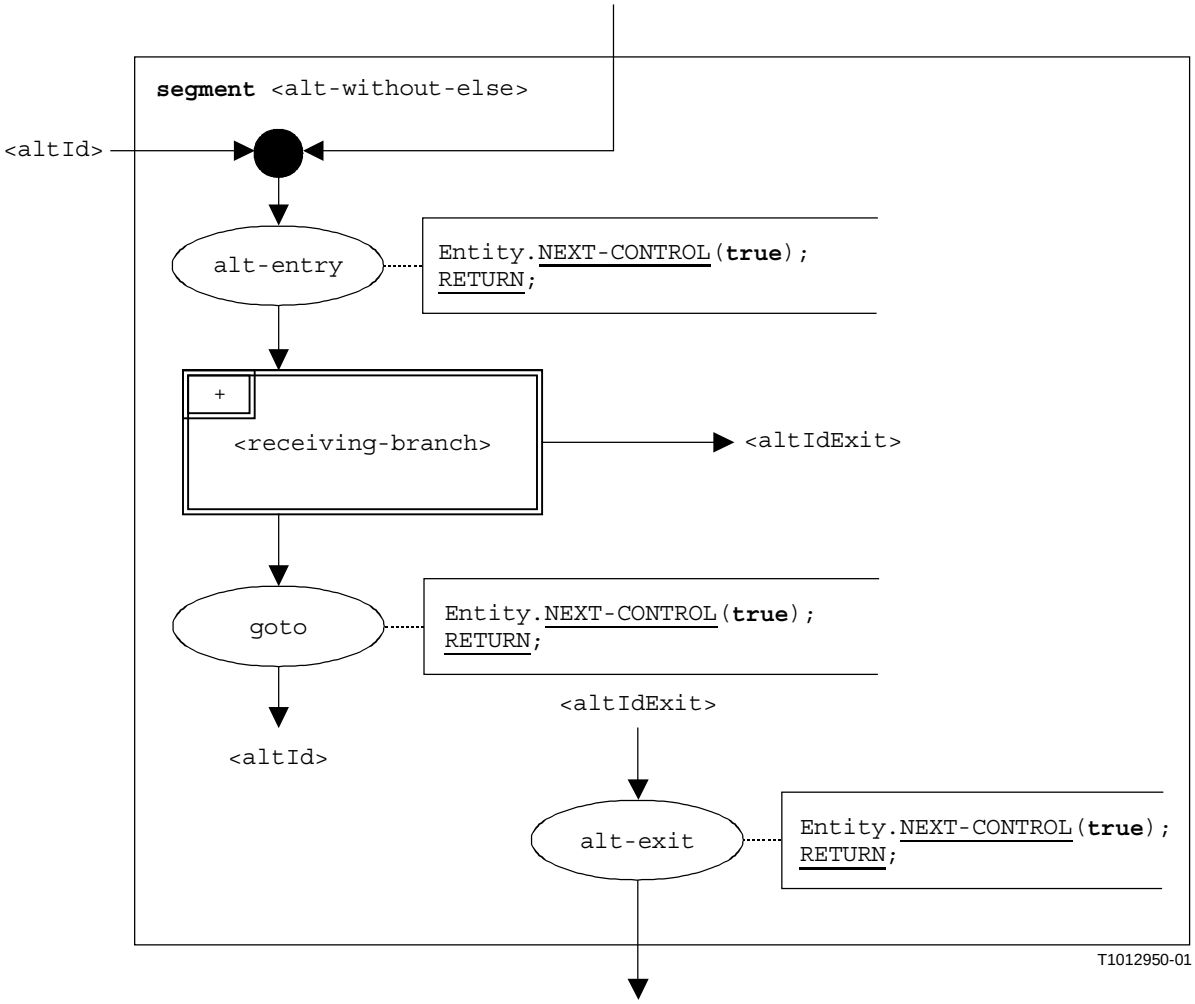


Рисунок B.27/Z.140 – Сегмент <alt-without-else> потокового графа

B.3.7.1.1 Сегмент <receiving-branch> потокового графа

Выполнение сегмента <receiving-branch> потокового графа показано на рис. B.28.

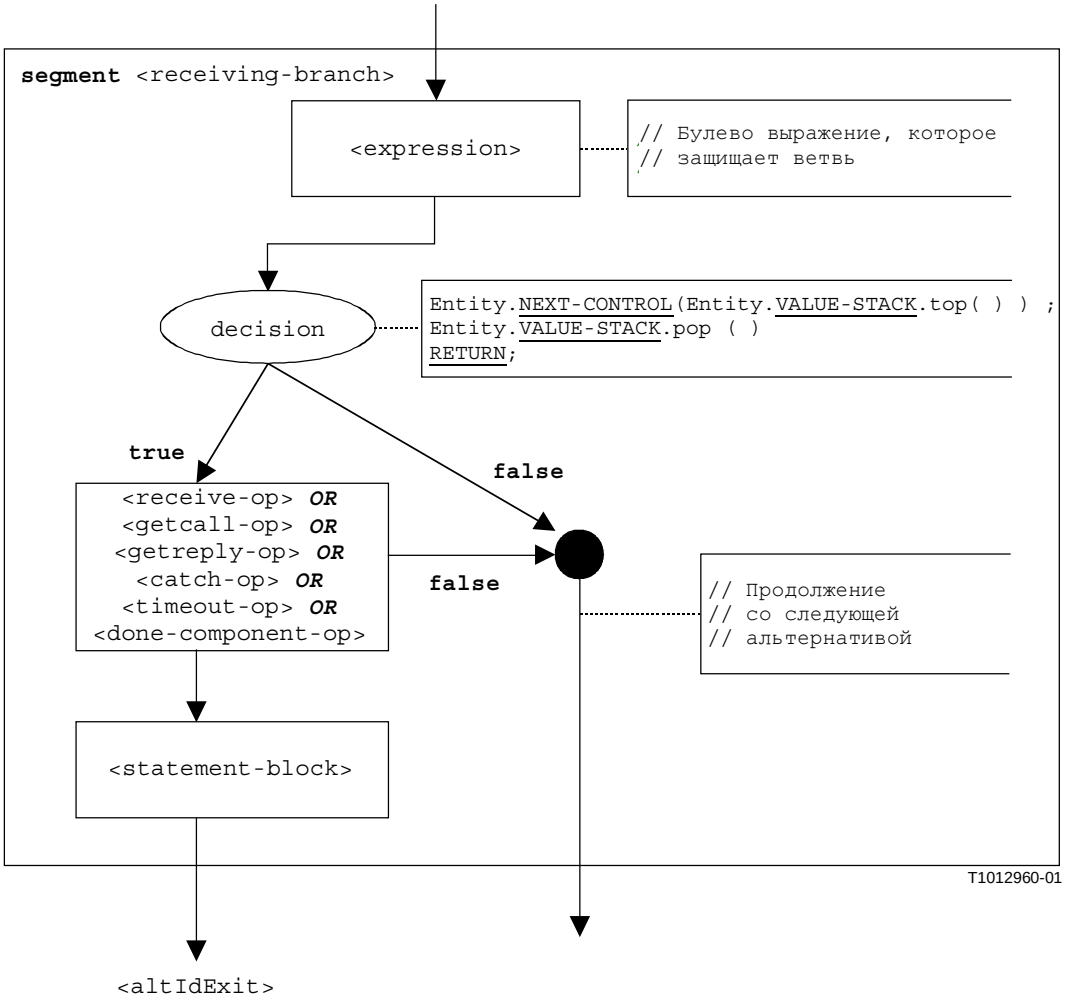


Рисунок B.28/Z.140 – Сегмент <receiving-branch> потокового графа

B.3.7.2 Команда Assignment

Синтаксической структурой команды assignment является:

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

Значение выражения <expression> присваивается переменной <varId>. Выполнение команды присвоения определяется сегментом <assignment-stmt> потокового графа, показанным на рис. B.29.

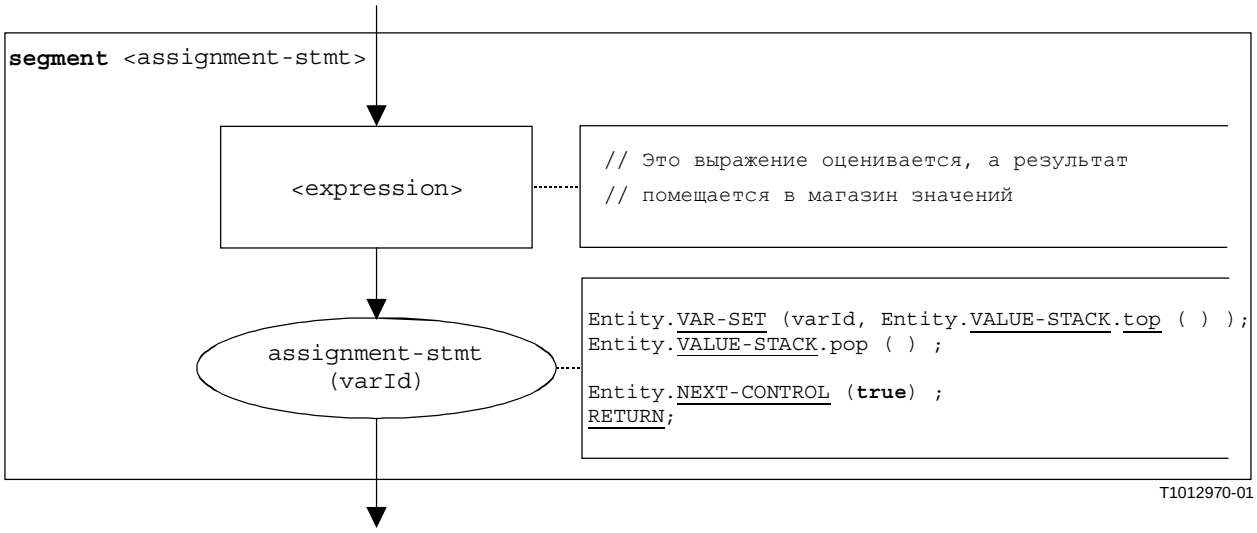


Рисунок B.29/Z.140 – Сегмент <assignment-stmt> потокового графа

B.3.7.3 Операция Call

Синтаксической структурой операции **call** является:

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

Факультативная <blocking-info> содержит либо ключевое слово **nonblocking**, либо выдержку особого состояния с тайм-аутом. Факультативное <component_expression> в разделе **to** указывает принимающий объект. Он может быть представлен в форме значения переменной или значения, выдаваемого функцией. Факультативная <call-reception-part> обозначает альтернативные возможности приема в случае блокирования операции **call**.

Операционная семантика различает *блокирующие* и *неблокирующие* операции **call**. Вызов не блокирует, если он ожидает отсутствие ответов или если используется ключевое слово "nonblocking". Блокирующий вызов имеет <call-reception-part>.

Сегмент <call-op> потокового графа на рис. B.30 определяет выполнение операции **call**. Он отражает разницу между блокирующим и неблокирующим вызовами.

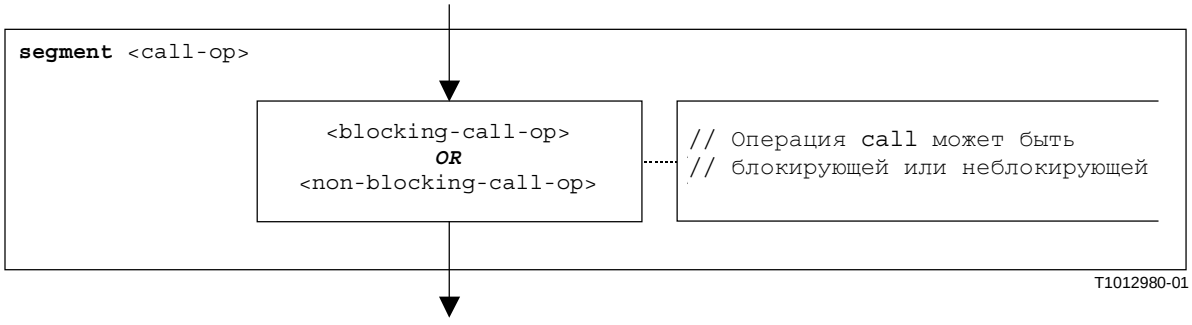


Рисунок B.30/Z/140 – Сегмент <call-op> потокового графа

Принимающий объект для операций блокирующего и неблокирующего вызовов может быть описан в форме выражения. Эти возможности показаны на рис. B.31 и B.32.

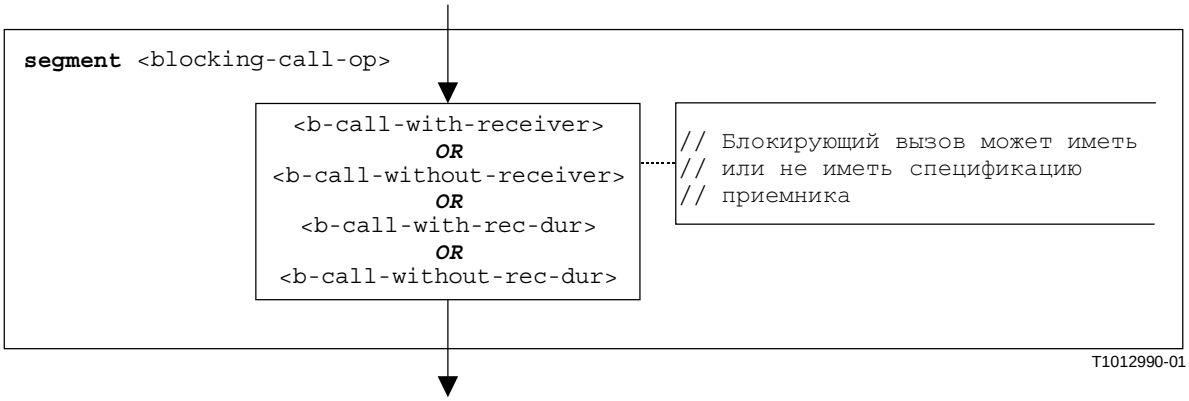


Рисунок B.31/Z.140 – Сегмент <blocking-call-op> потокового графа

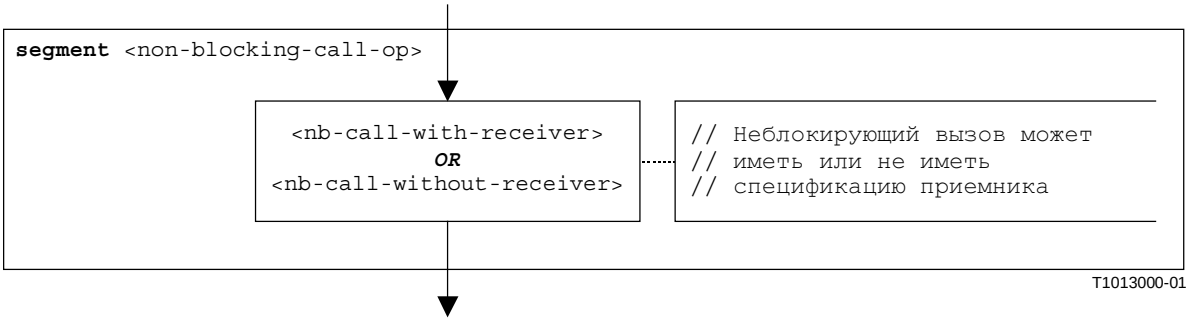


Рисунок B.32/Z.140 – Сегмент <non-blocking-call-op> потокового графа

B.3.7.3.1 Сегмент <nb-call-with-receiver> потокового графа

Сегмент <nb-call-with-receiver> потокового графа на рис. B.33 определяет выполнение неблокирующей операции **call**, в которой приемник описан в форме выражения.

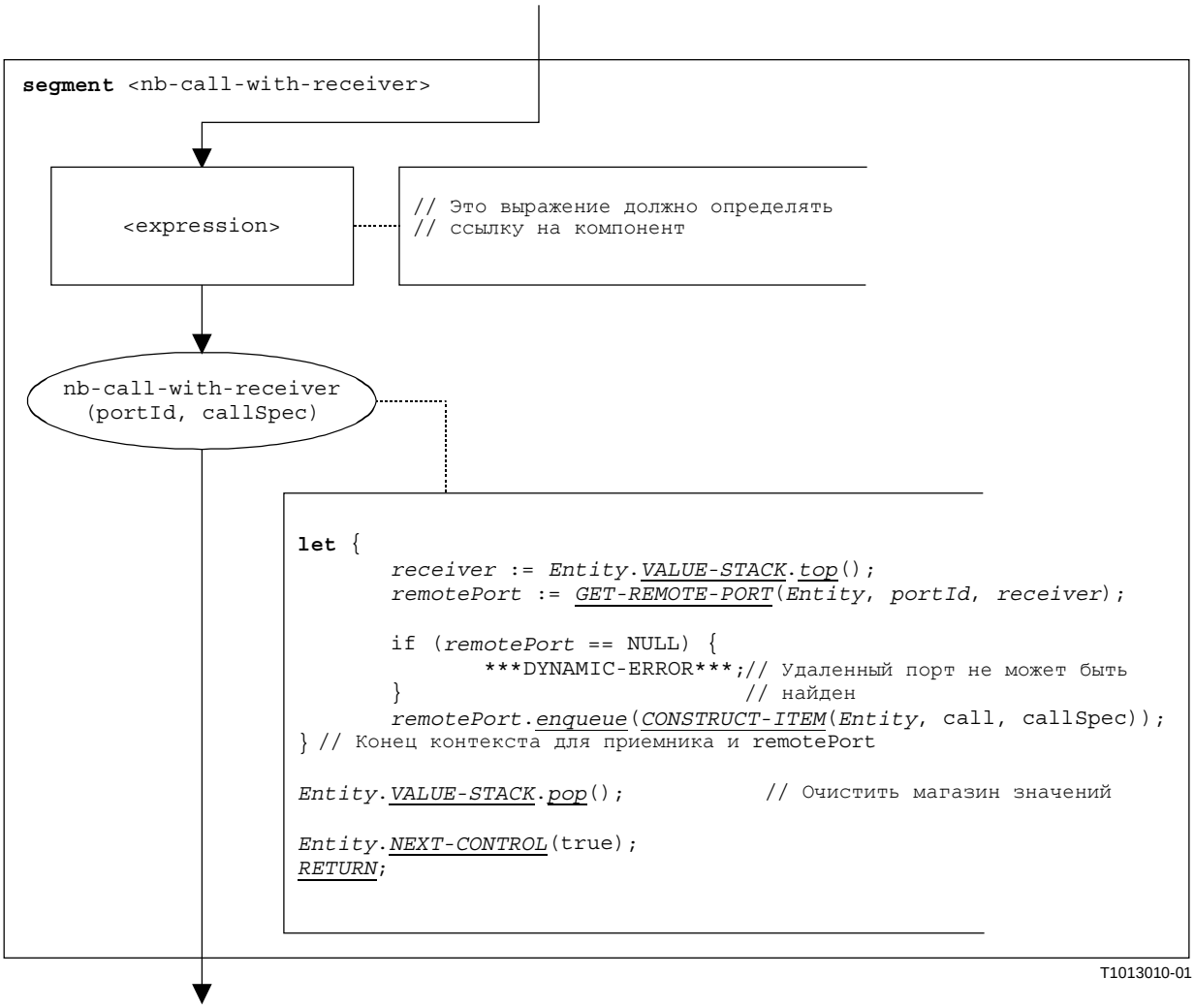


Рисунок B.33/Z.140 – Сегмент <nb-call-with-receiver> потокового графа

B.3.7.3.2 Сегмент <nb-call-without-receiver> потокового графа

Сегмент <nb-call-without-receiver> потокового графа на рис. B.34 определяет выполнение неблокирующей операции **call** без раздела **to**.

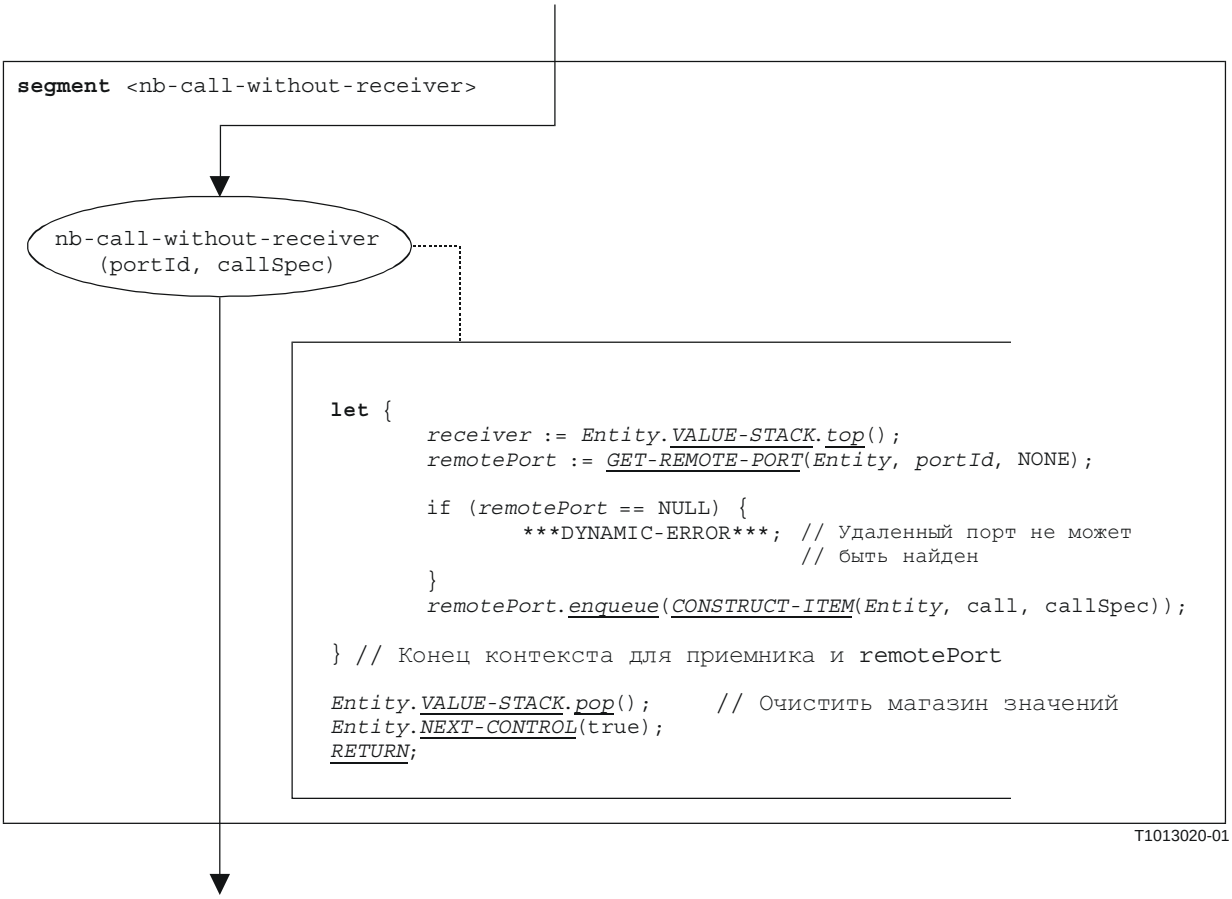


Рисунок B.34/Z.140 – Сегмент <nb-call-without-receiver> потокового графа

B.3.7.3.3 Сегмент <b-call-with-receiver>

Блокирующие вызовы моделируются в виде неблокирующего вызова, за которым следует команда **alt**. Сегмент <b-call-with-receiver> описывает выполнение блокирующего вызова без выдержки в качестве таймерной защиты, но с описанием приемника для этого вызова. Такой сегмент потокового графа показан на рис. B.35.

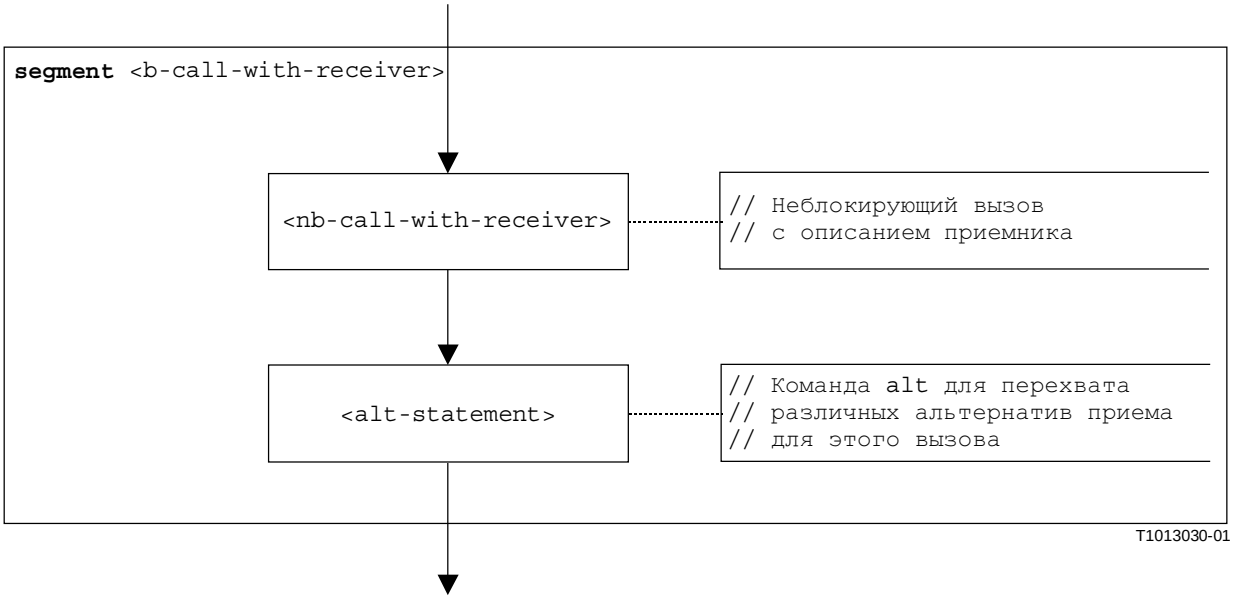


Рисунок B.35/Z.140 – Сегмент <b-call-with-receiver> потокового графа

B.3.7.3.4 Сегмент <b-call-without-receiver> потокового графа

Сегмент <b-call-without-receiver> потокового графа описывает выполнение блокирующего вызова без защиты выдержкой таймера и без описания приемника для этого вызова. Такой сегмент потокового графа показан на рис. B.36.

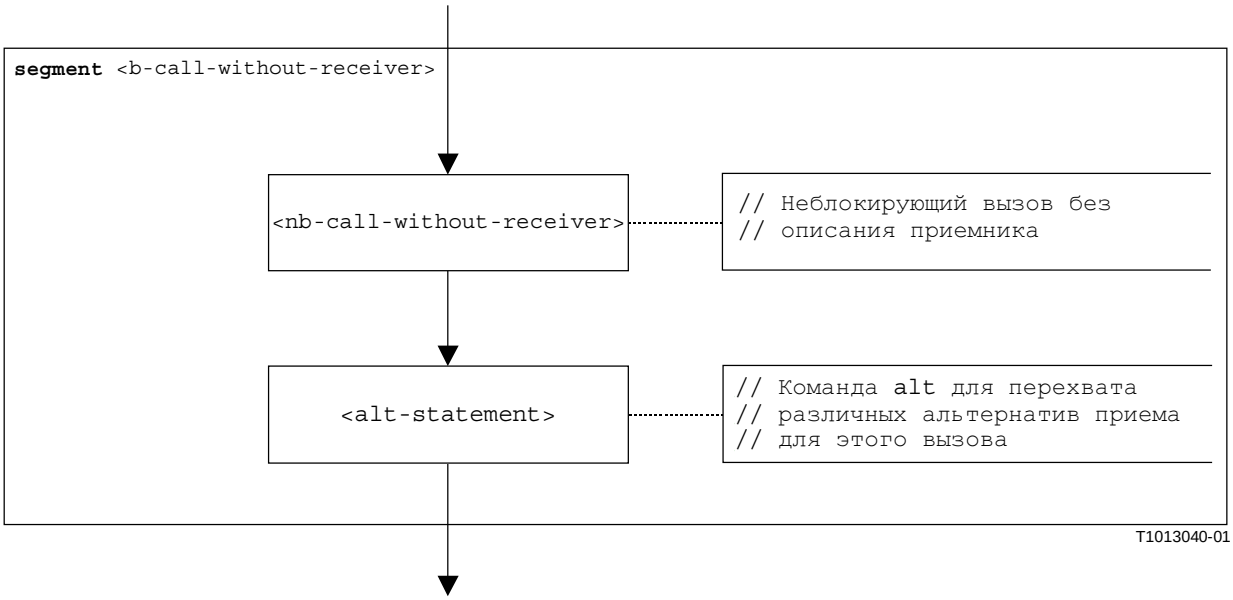


Рисунок B.36/Z.140 – Сегмент <b-call-without-receiver> потокового графа

B.3.7.3.5 Сегмент <b-call-with-rec-dur> потокового графа

Блокирующие вызовы, защищенные с помощью таймеров, моделируются в виде неблокирующего вызова, за которым следует команда **alt**. Для выдержки запускается специальный системный таймер SYS-TI. Улавливающая ветвь таймера в команде **alt** указывает на этот системный таймер. Сегмент <b-call-with-rec-dur> потокового графа описывает выполнение блокирующего вызова с выдержкой в качестве таймерной защиты и с описанием приемника для этого вызова. Такой сегмент потокового графа показан на рис. B.37.

ПРИМЕЧАНИЕ. – Обработка системного таймера производится лишь неформальным образом. Реализация зависит от тестового оборудования.

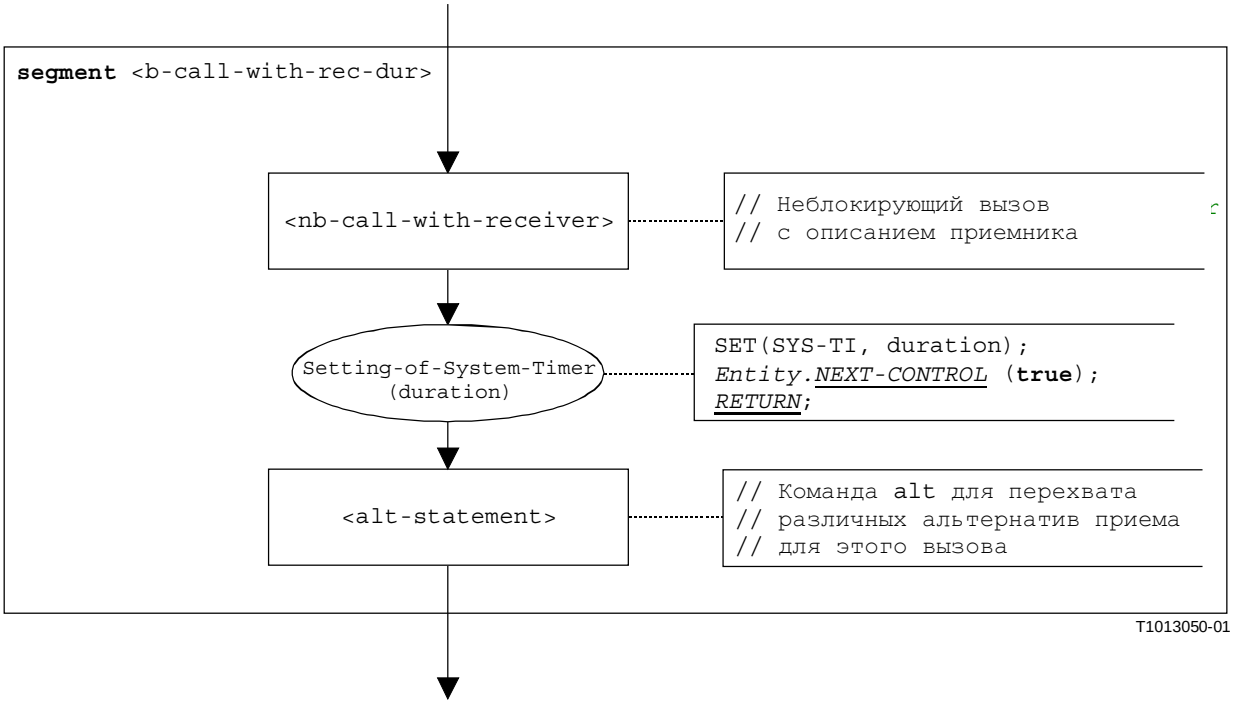


Рисунок B.37/Z.140 – Сегмент <b-call-with-rec-dur> потокового графа

B.3.7.3.6 Сегмент <b-call-without-rec-dur> потокового графа

Сегмент <b-call-without-rec-dur> потокового графа описывает выполнение блокирующего вызова с выдержкой в качестве таймерной защиты и без описания приемника для этого вызова. Такой сегмент потокового графа показан на рис. B.38.

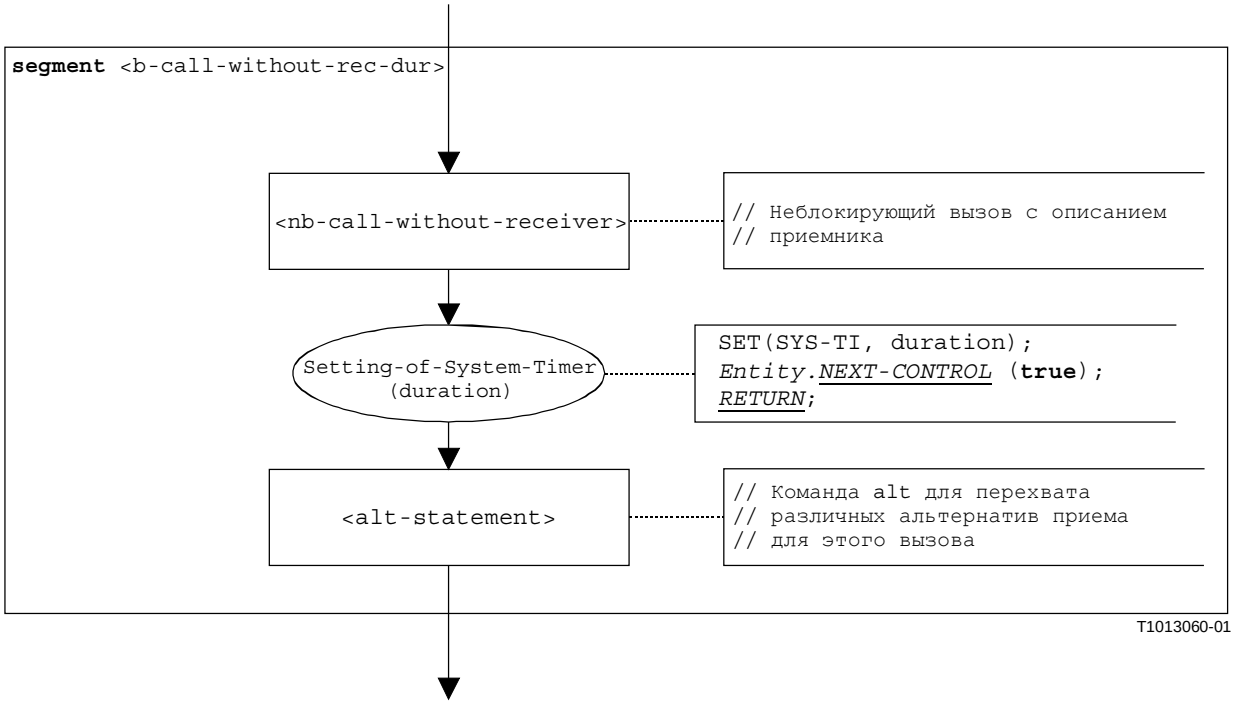


Рисунок B.38/Z.140 – Сегмент <b-call-without-rec-dur> потокового графа

B.3.7.4 Операция Catch

Синтаксической структурой операции catch является:

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

Факультативное <component_expression> в разделе **from** указывает на передатчик особого состояния. Он может быть представлен в форме значения переменной или значения, выдаваемого функцией, то есть он считается выражением. Факультативная <assignmentPart> означает присвоение захваченной информации, если захваченное особое состояние соответствует спецификации сопоставления <matchingSpec> и (факультативному) разделу **from**.

Сегмент <catch-op> потокового графа на рис. B.39 определяет выполнение операции **catch**.

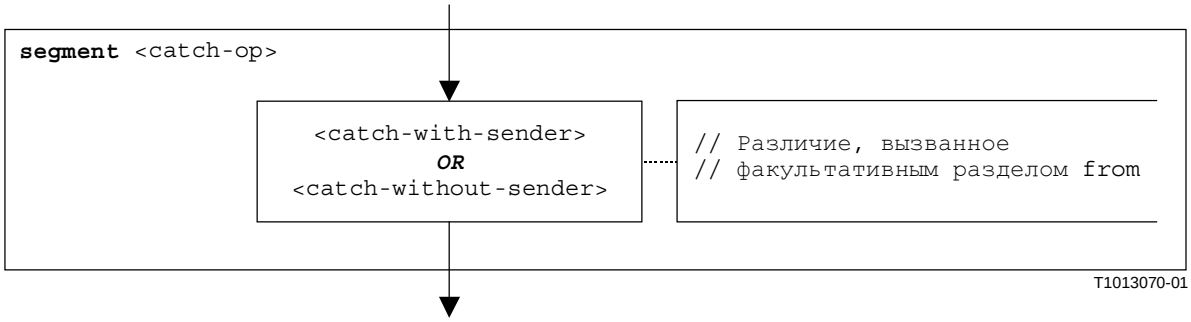


Рисунок B.39/Z.140 – Сегмент <catch-op> потокового графа

B.3.7.4.1 Сегмент <catch-with-sender> потокового графа

Сегмент <catch-with-sender> потокового графа на рис. B.40 определяет выполнение операции **catch**, в которой передатчик описан в форме выражения.

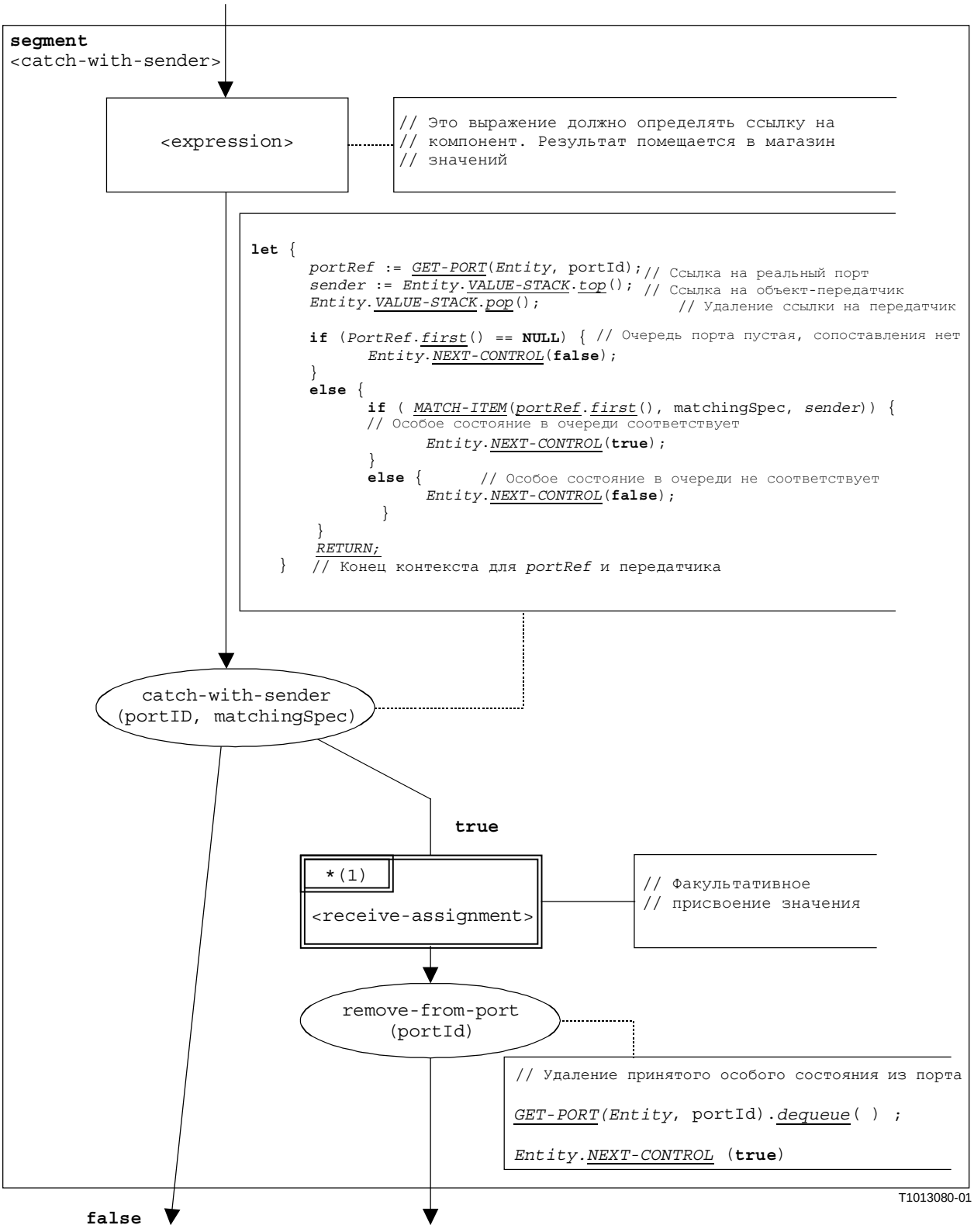


Рисунок B.40/Z.140 – Сегмент <catch-with-sender> потокового графа

B.3.7.4.2 Сегмент <catch-without-sender> потокового графа

Сегмент <catch-without-sender> потокового графа на рис. B.41 определяет выполнение операции **catch** без раздела **from**.

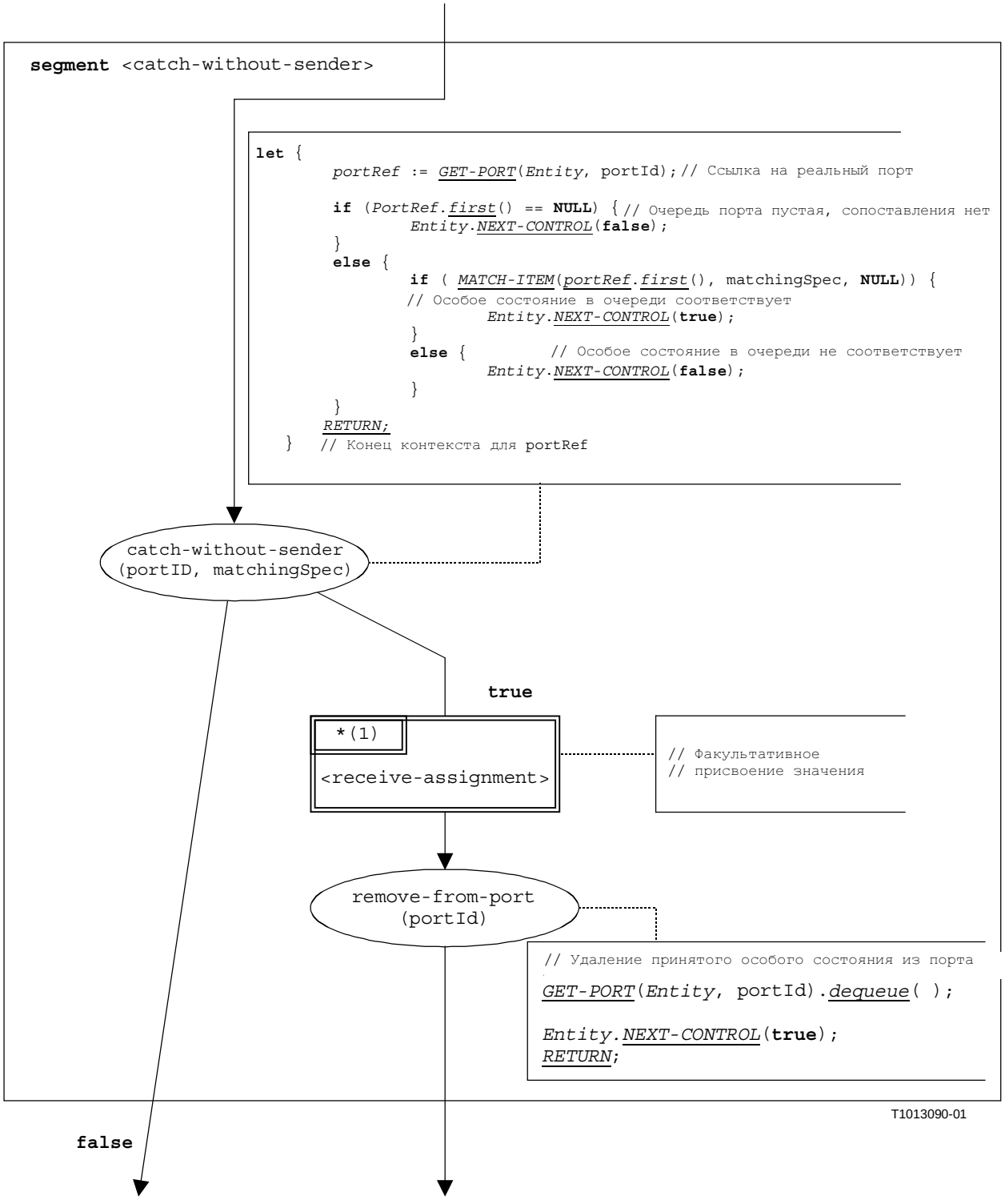


Рисунок B.41/Z.140 – Сегмент <catch-without-sender> потокового графа

B.3.7.5 Портовая операция Clear

Синтаксической структурой портовой операции **clear** является:

<portId>**.clear**

Сегмент <clear-port-op> потокового графа на рис. B.42 определяет выполнение портовой операции **clear**.

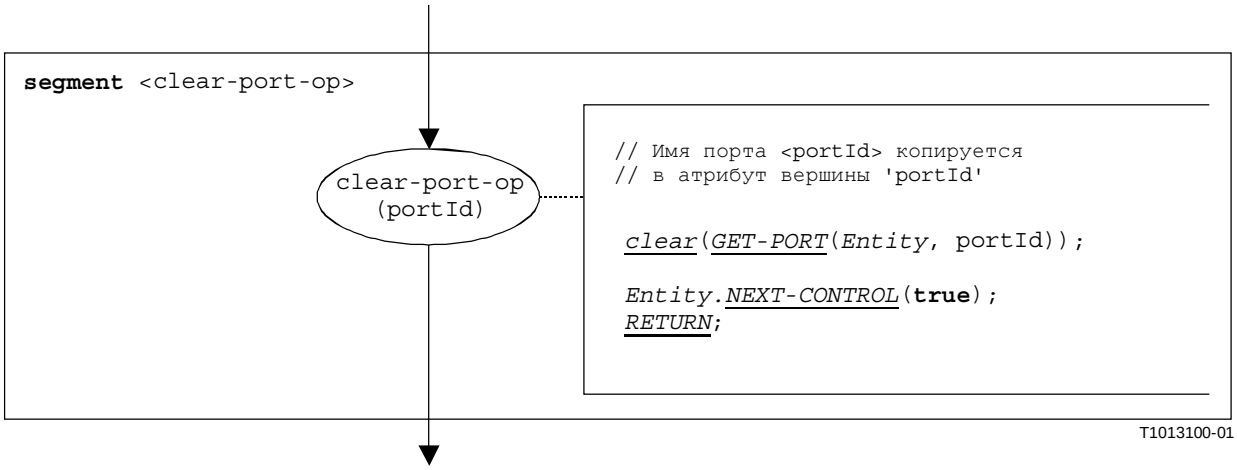


Рисунок B.42/Z.140 – Сегмент <clear-port-op> потокового графа

B.3.7.6 Операция Connect

Синтаксической структурой операции connect является:

connect(<component_expression$_1$>.<portId1>,<component_expression$_2$>.<portId2>)

Идентификаторы <portId1> и <portId2> рассматриваются в качестве идентификаторов портов в соответствующих тестовых компонентах. Компоненты, которым принадлежат эти порты, указываются ссылками на компоненты <component_expression$_1$> и <component_expression$_2$>. Эти ссылки могут быть записаны в переменных или могут выдаваться функцией. Для простоты мы считаем их выражениями, которые определяют ссылку на компонент. Поэтому для хранения ссылок на компоненты используется магазин значений.

Выполнение операции **connection** определяется сегментом <connect-op> потокового графа, показанным на рис. B.43. В этом описании потокового графа первое выражение, подлежащее оценке, ссылается на <component_expression$_1$>, а второе выражение – на <component_expression$_2$>, то есть <component_expression$_2$> является верхним значением в магазине значений, когда выполняется вершина connect-op.

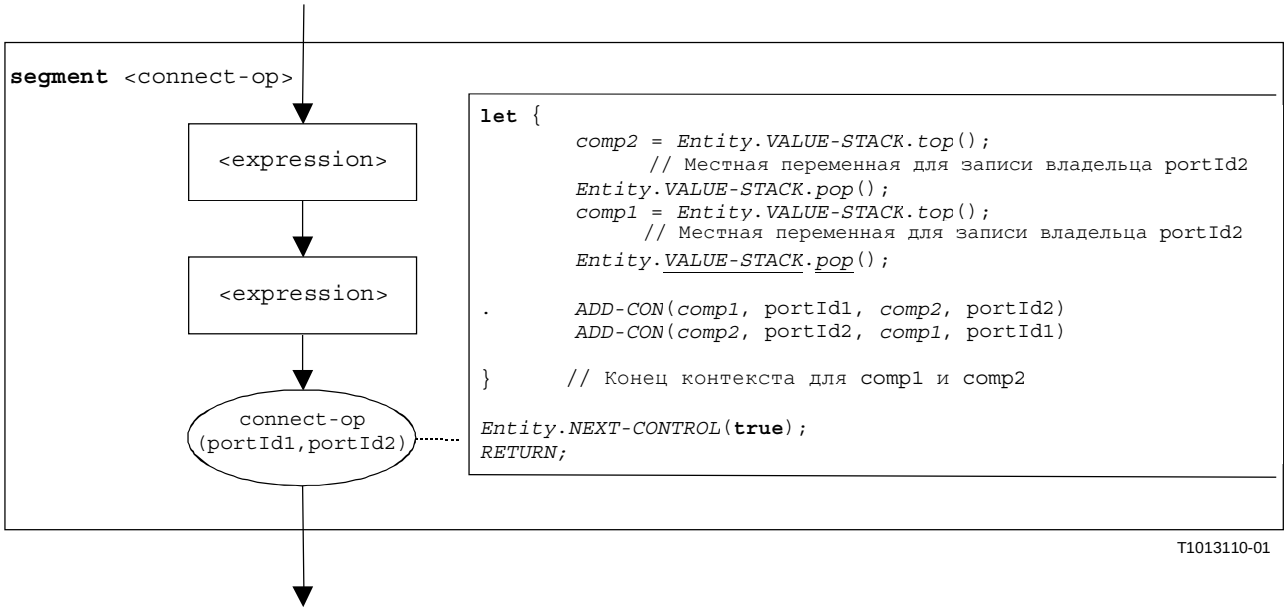


Рисунок B.43/Z.140 – Сегмент <connect-op> потокового графа

B.3.7.7 Объявление константы

Синтаксической структурой объявления константы является:

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

Значение константы рассматривается как выражение, которое определяет значение для типа константы.

ПРИМЕЧАНИЕ. – Глобальные константы заменяются их значениями на предварительной стадии, до применения этой семантики (см. п. B.2.3). Местные константы обрабатываются как объявления переменных с инициализацией. Правильное использование констант, которое означает, что константы никогда не должны появляться в левой части присвоения, проверяется во время статического семантического анализа модуля TTCN-3.

Сегмент <constant-declaration> потокового графа на рис. B.44 определяет выполнение объявления константы, при котором значение константы предоставляется в форме выражения.

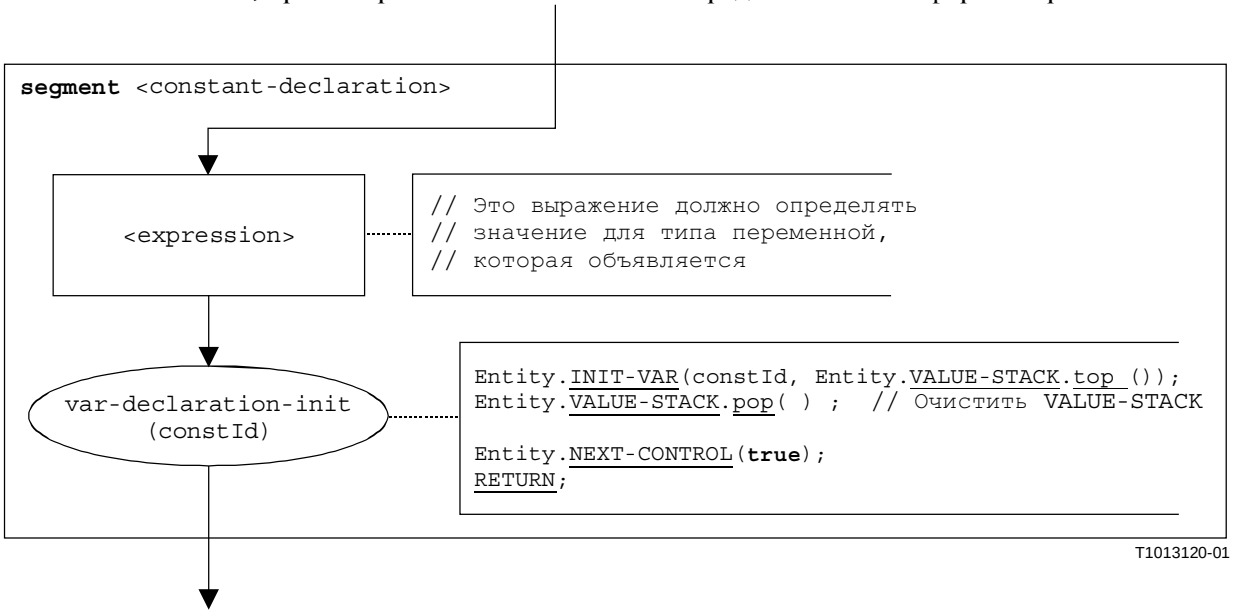


Рисунок B.44/Z.140 – Сегмент <constant-declaration> потокового графа

B.3.7.8 Операция Create

Синтаксической структурой операции **`create`** является:

`<componentTypeId>`**`.create`**

Сегмент `<create-op>` потокового графа на рис. B.45 определяет выполнение операции **`create`**.

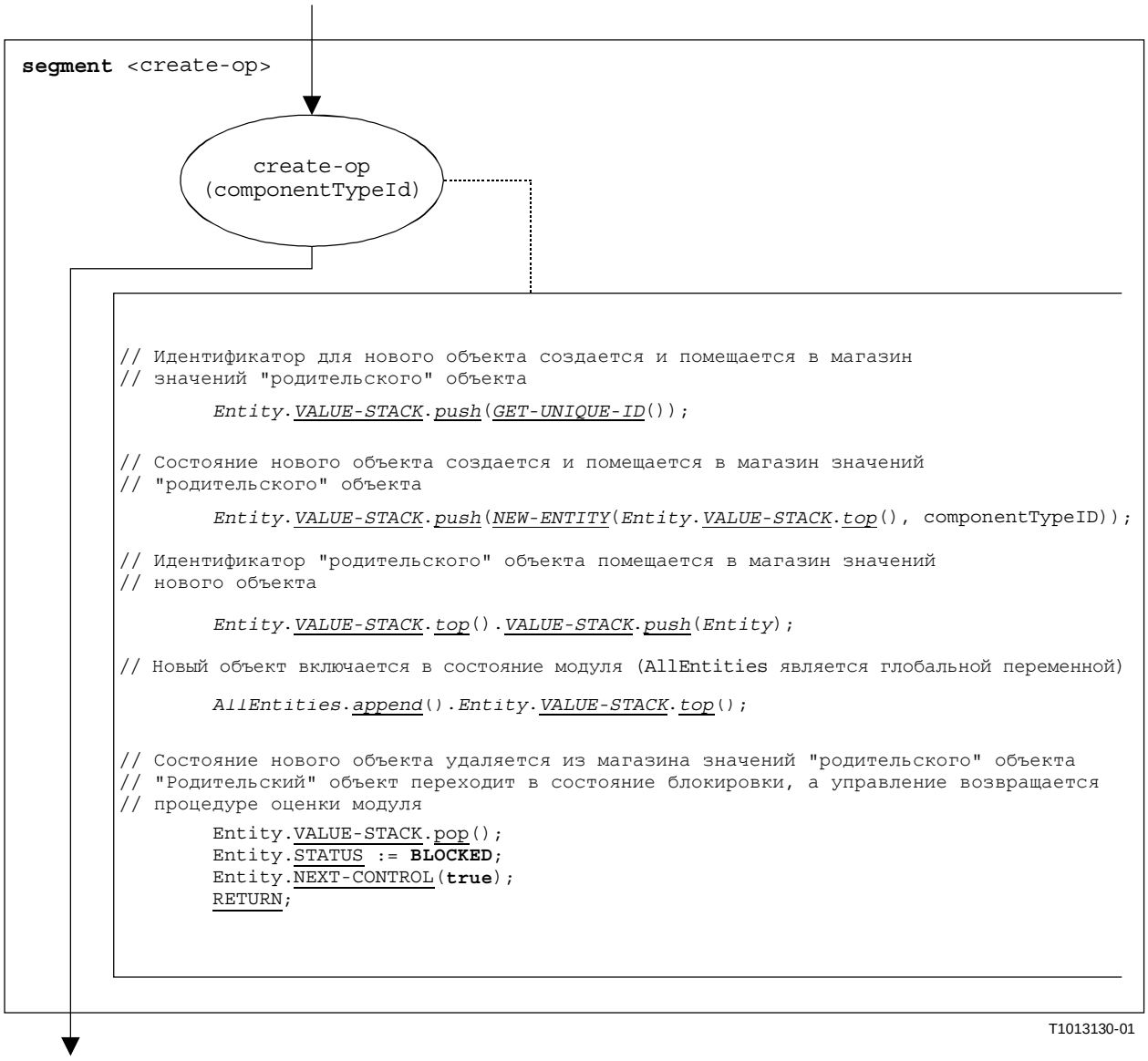


Рисунок B.45/Z.140 – Сегмент <create-op> потокового графа

B.3.7.9 Объявление порта

Синтаксической структурой объявления порта является:

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

Объявление порта можно встретить в определениях типа компонента. Результатом объявления порта является создание нового порта. Сегмент <port-declaration> потокового графа на рис. B.46 определяет выполнение объявления порта.

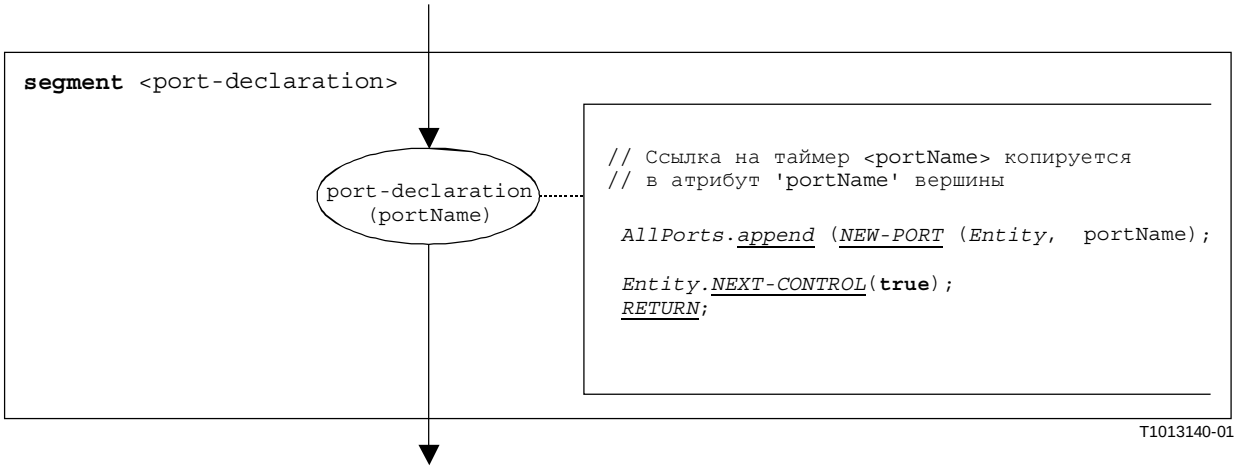


Рисунок B.46/Z.140 – Сегмент <port-declaration> потокового графа

B.3.7.10 Объявление таймера

Синтаксической структурой объявления таймера является:

```
timer <timerId> [:= <float_expression>]
```

Результатом объявления таймера является создание нового связывания таймера. Объявление переменной с выдержкой по умолчанию является факультативным. Значением по умолчанию считается выражение, которое имеет значение типа **float**.

Сегмент <timer-declaration> потокового графа на рис. B.47 определяет выполнение объявления таймера.

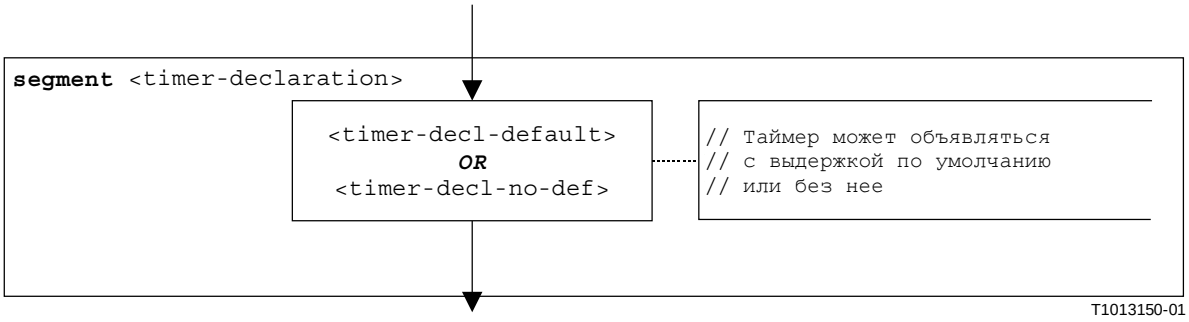


Рисунок B.47/Z.140 – Сегмент <timer-declaration> потокового графа

B.3.7.10.1 Сегмент <timer-decl-default>

Сегмент <timer-decl-default> потокового графа на рис. B.48 определяет выполнение объявления таймера, в котором предусматривается выдержка по умолчанию в форме выражения.

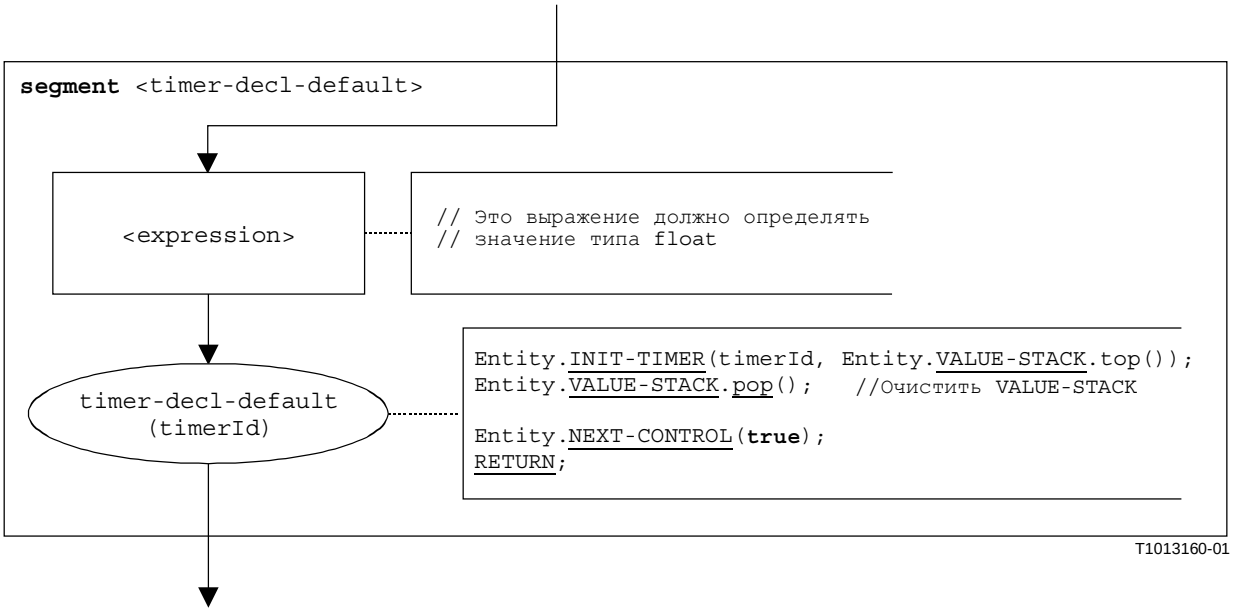


Рисунок B.48/Z.140 – Сегмент <timer-decl-default> потокового графа

B.3.7.10.2 Сегмент <timer-decl-no-def> потокового графа

Сегмент <timer-decl-no-def> потокового графа на рис. B.49 определяет выполнение объявления таймера, в котором не предусматривается выдержка по умолчанию, то есть выдержка по умолчанию таймера неопределенна.

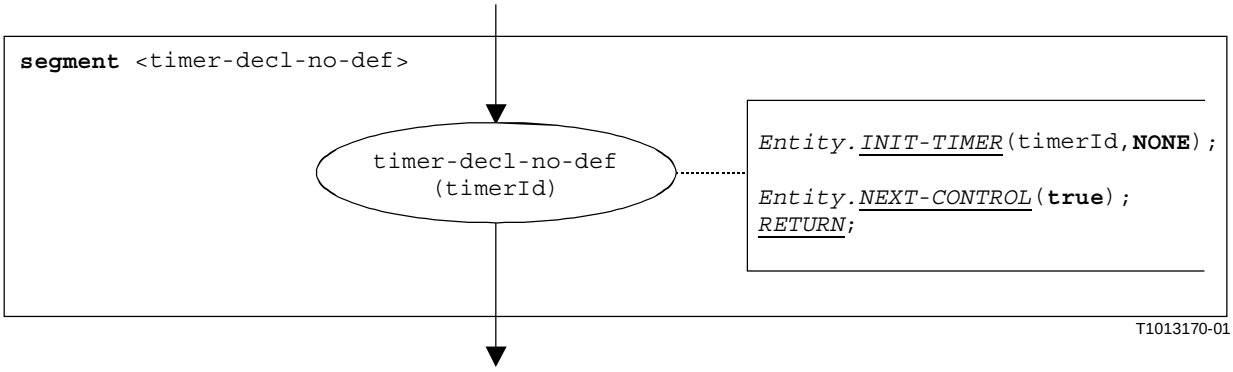


Рисунок B.49/Z.140 – Сегмент <timer-decl-no-def> потокового графа

B.3.7.11 Объявление переменной

Синтаксической структурой объявления переменной является:

```
var <varType> <varId> [:= <varType_expression>]
```

Инициализация переменной путем обеспечения начального значения является факультативной. Начальное значение рассматривается как выражение, которое определяет значение для типа переменной.

Сегмент <variable-declaration> потокового графа на рис. B.50 определяет выполнение объявления переменной.

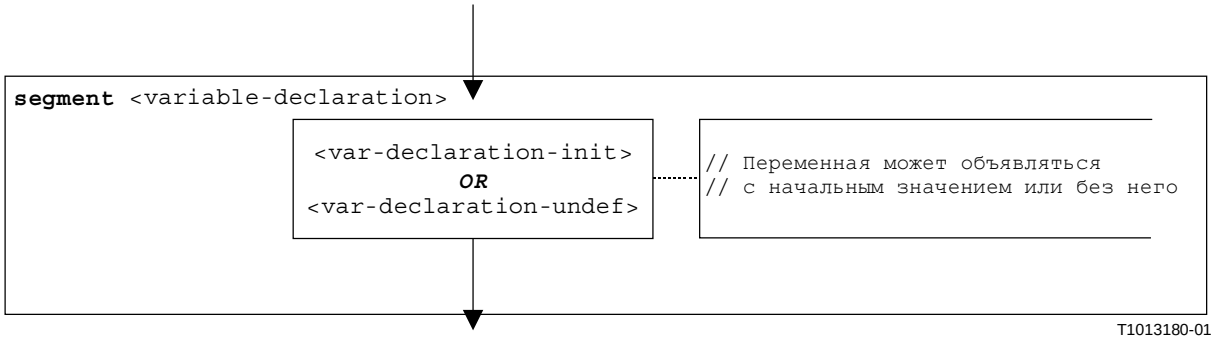


Рисунок B.50/Z.140 – Сегмент <variable-declaration> потокового графа

B.3.7.11.1 Сегмент <var-declaration-init> потокового графа

Сегмент <var-declaration-init> потокового графа на рис. B.51 определяет выполнение объявления переменной, в котором предусматривается начальное значение в форме выражения.

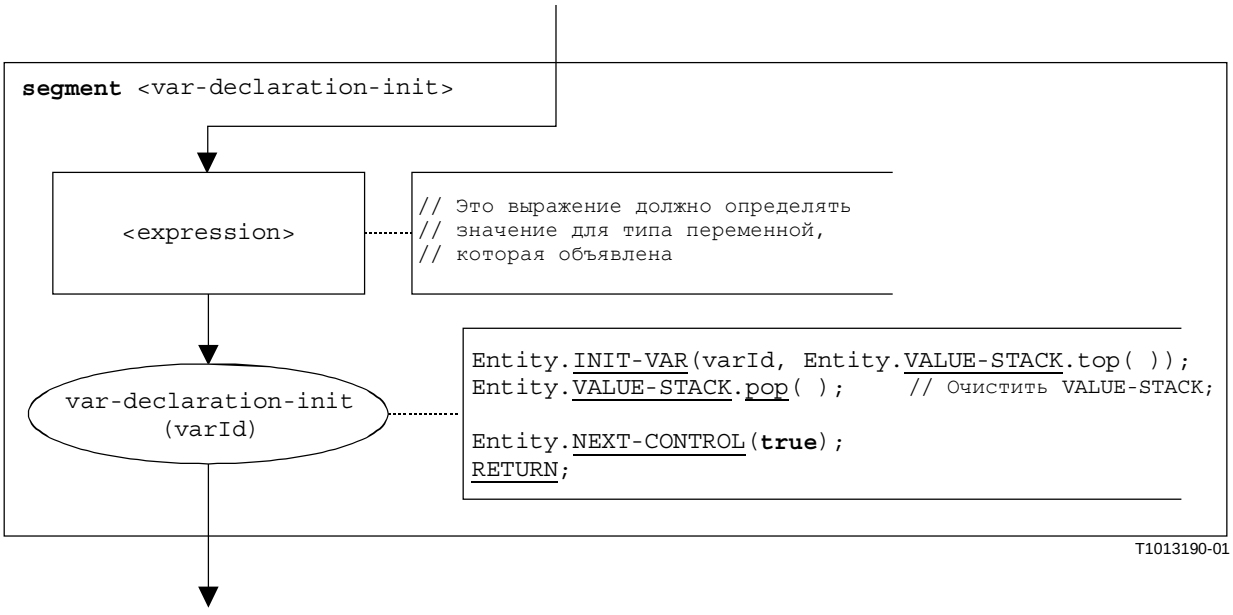


Рисунок B.51/Z.140 – Сегмент <var-declaration-init> потокового графа

B.3.7.11.2 Сегмент <var-declaration-undef> потокового графа

Сегмент <var-declaration-undef> потокового графа на рис. B.52 определяет выполнение объявления переменной, в котором не предусматривается начальное значение, то есть значение переменной неопределенно.

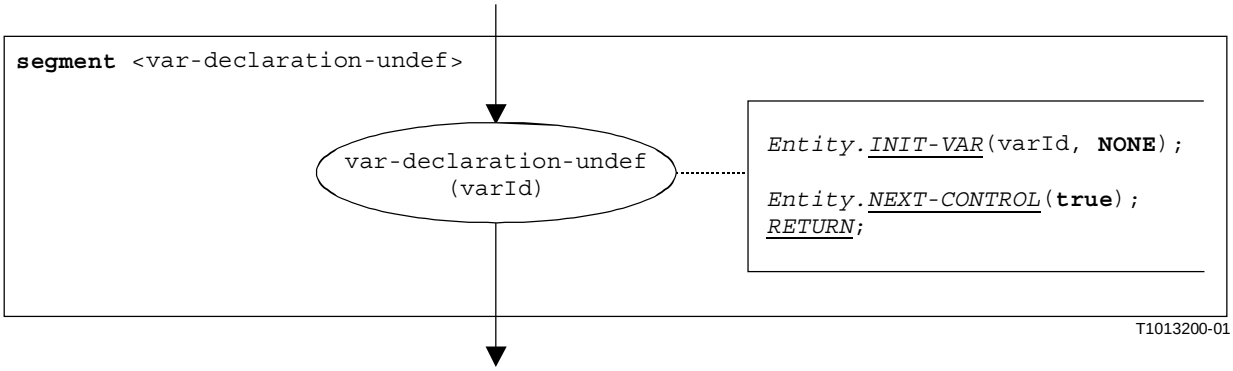


Рисунок B.52/Z.140 – Сегмент <var-declaration-undef> потокового графа

B.3.7.12 Операция Disconnect

Синтаксической структурой операции **disconnect** является:

disconnect(<component_expression$_1$>.<portId1>,<component_expression$_2$>.<portId2>)

Идентификаторы <portId1> и <portId2> рассматриваются как идентификаторы портов в соответствующих тестовых компонентах. Компоненты, которым принадлежат эти порты, указываются ссылками на компоненты <component_expression$_1$> и <component_expression$_2$>. Эти ссылки могут быть записаны в переменных или могут выдаваться функцией. Для простоты мы считаем их выражениями, которые определяют ссылку на компонент. Поэтому для хранения ссылок на компоненты используется магазин значений.

Выполнение операции **disconnect** определяется сегментом <disconnect-op>, показанным на рис. B.53. В этом сегменте потокового графа первое выражение, подлежащее оценке, ссылается на <component_expression$_1$>, а второе выражение – на <component_expression$_2$>, то есть <component_expression$_2$> является верхним значением в магазине значений, когда выполняется вершина disconnect-op.

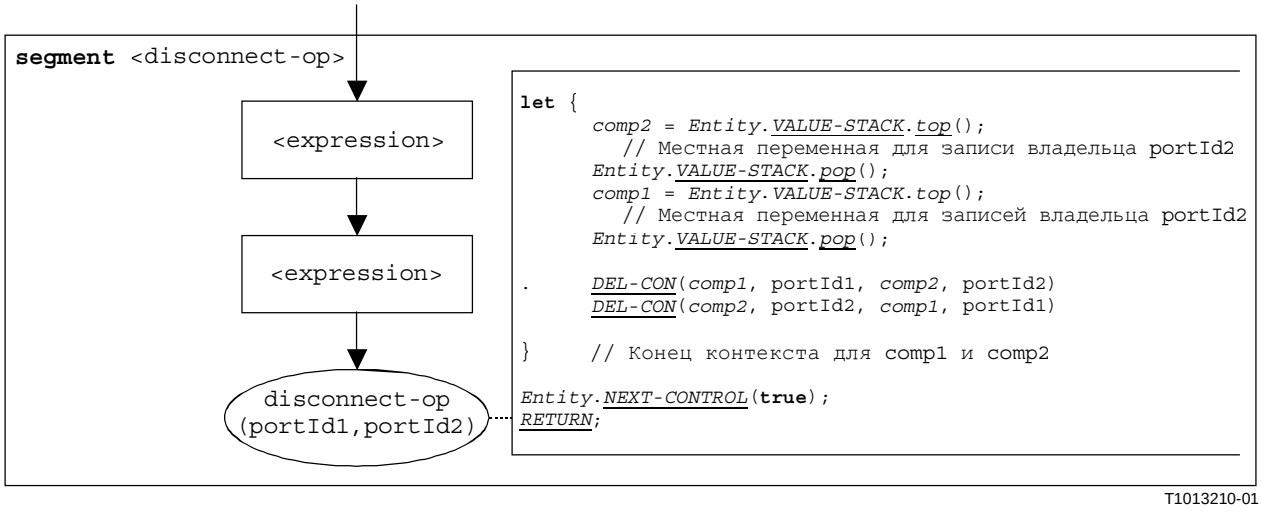


Рисунок B.53/Z.140 – Сегмент <disconnect-op> потокового графа

B.3.7.13 Команда Do-while

Синтаксической структурой команды **do-while** является:

```
do <statement-block>
while (<boolean_expression>)
```

Выполнение команды **do-while** определяется сегментом <do-while-stmt> потокового графа, показанным на рис. B.54.

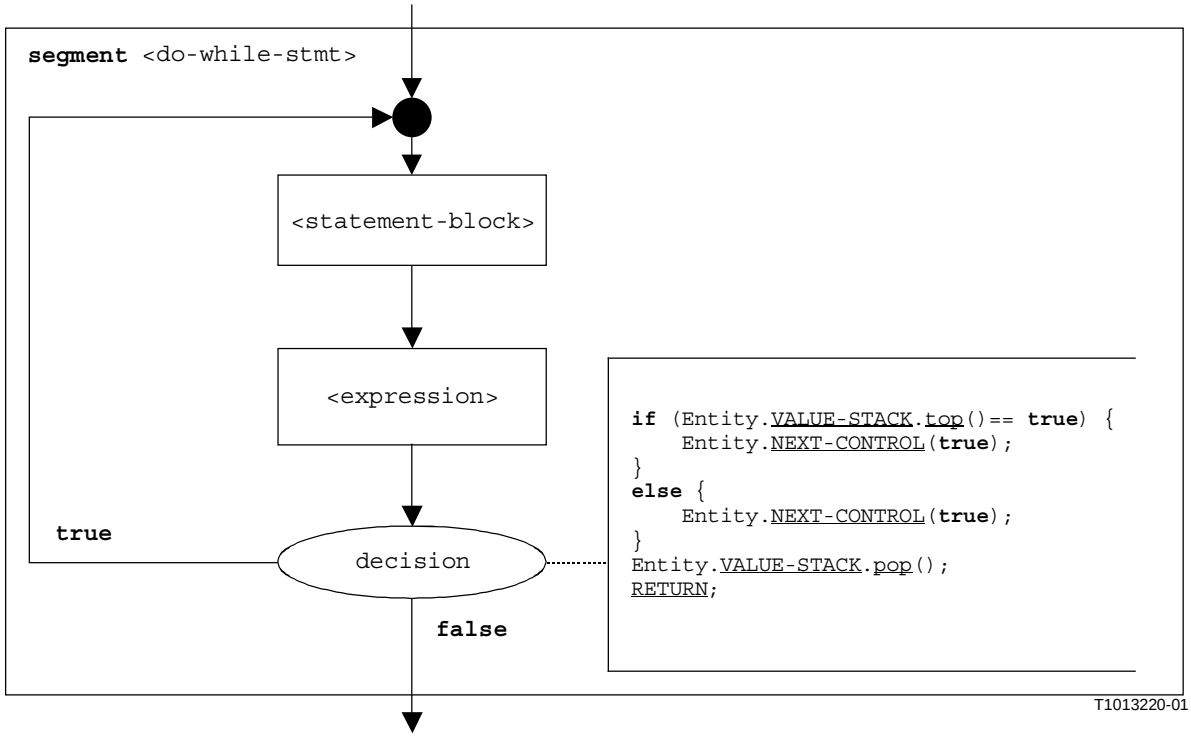


Рисунок B.54/Z.140 – Сегмент <do-while-stmt> потокового графа

B.3.7.14 Операция Done-all-components

Операция **done-all-components** указывает на использование ключевых слов **all component** в операции **done** (см. п. B.7.16). Операция **done-all-components** может вызываться только из **mtc**. Она позволяет проверять, все ли параллельные тестовые компоненты в тестовом примере завершены. Синтаксической структурой операции **done-all-components** является:

```
all component.done;
```

Выполнение операции **done-all-components** определяется сегментом <done-all-comp-op> потокового графа на рис. B.55.

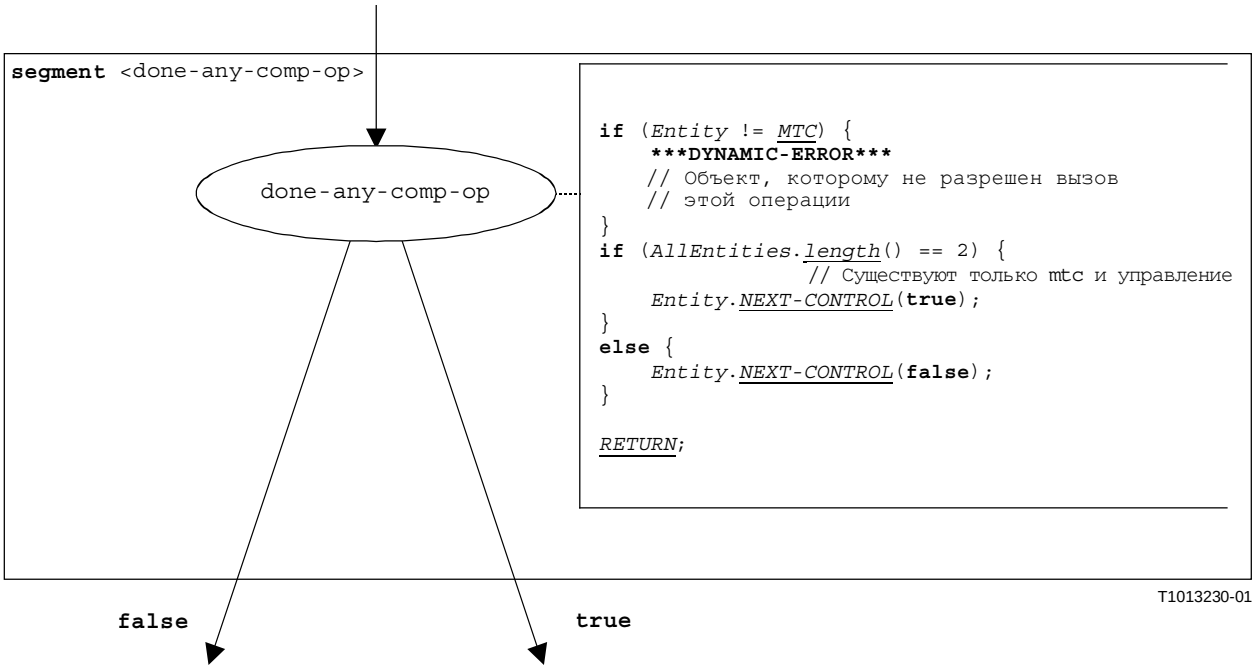


Рисунок B.55/Z.140 – Сегмент <done-all-comp-op> потокового графа

B.3.7.15 Операция Done-any-component

Операция **done-any-component** указывает на использование ключевых слов **any component** в операции done (см. п. B.7.16). Операция **done-any-component** может вызываться только из **mtc**. Она позволяет проверять, завершен ли некоторый параллельный тестовый компонент в тестовом примере. Синтаксической структурой операции **done-any-component** является:

```
any component.done;
```

Выполнение операции **done-any-component** определяется сегментом <done-any-comp-op> потокового графа на рис. 56.

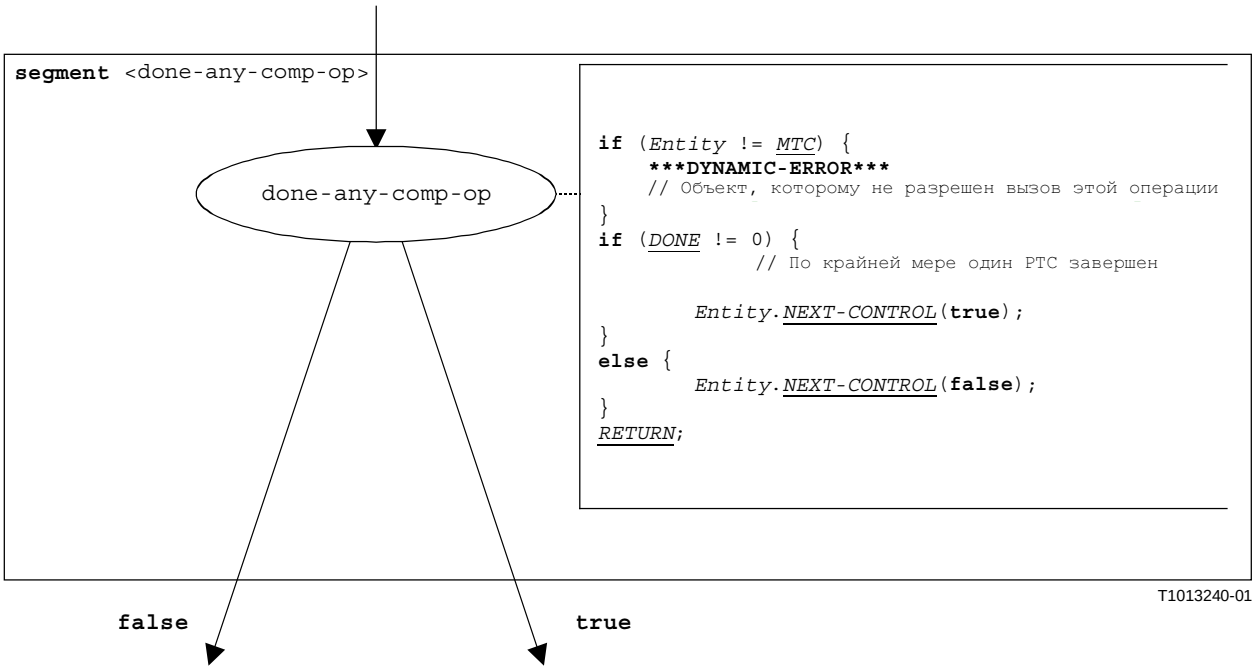


Рисунок B.56/Z.140 – Сегмент <done-any-comp-op> потокового графа

B.3.7.16 Компонентная операция Done

Синтаксической структурой компонентной операции **done** является:

<component_expression>.**done**

Компонентная операция done проверяет, запущен ли компонент или он остановлен. В зависимости от состояния проверяемого компонента (запущен или остановлен) операция **done** решает, как продолжить поток управления. Использование ссылки на компонент определяет компонент, подлежащий проверке. Эта ссылка может быть записана в переменной или может выдаваться функцией. Для простоты считается, что это будет выражение, которое определяет ссылку на компонент.

Сегмент <done-component-op> потокового графа на рис. B.57 определяет выполнение компонентной операции **done**.

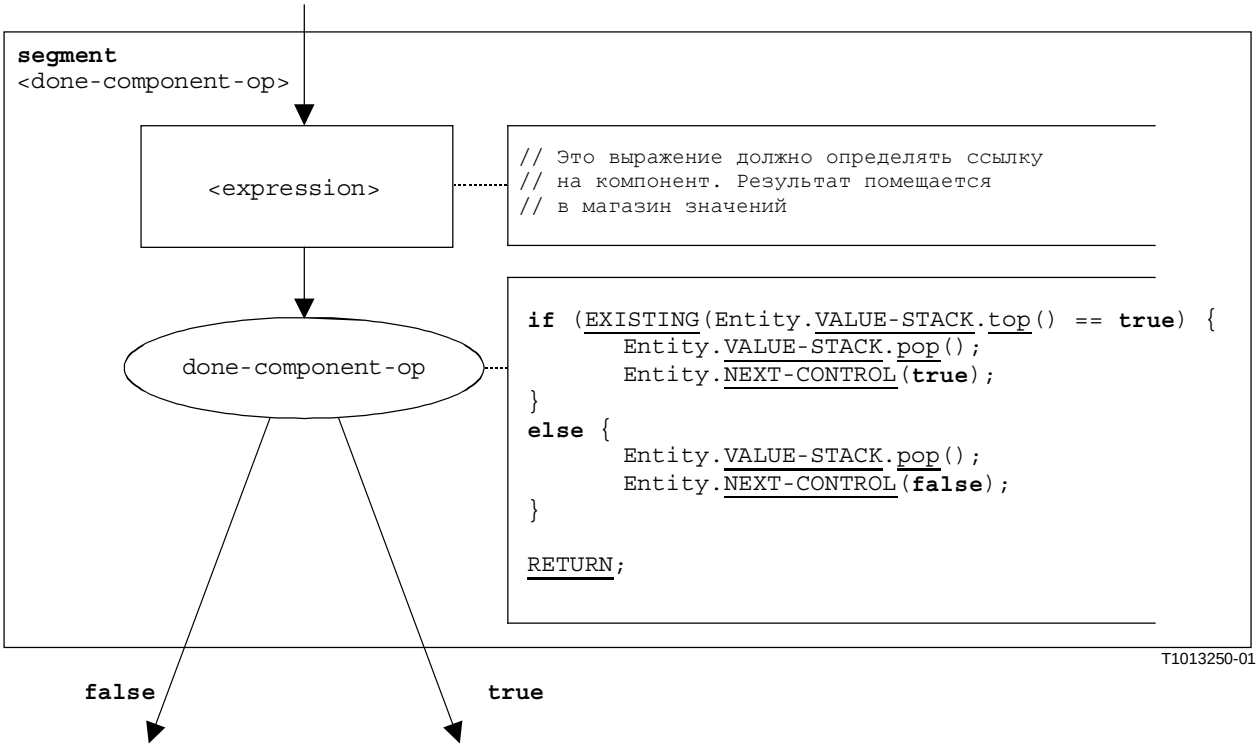


Рисунок B.57/Z.140 – Сегмент <done-component-op> потокового графа

B.3.7.17 Команда Execute

Синтаксической структурой команды **execute** является:

execute(<testCaseId>([<act-par$_1$>, ... , <act-par$_n$>)]) [, <float_expression>])

Команда **execute** описывает выполнение тестового примера <testCaseId> с реальными (факультативными) параметрами <alt-par$_1$>, . . . , <act-par$_n$>. Факультативно команда **execute** может быть защищена выдержкой, указанной в форме выражения, которое определяет некоторую **float**. Если за время указанной выдержки тестовый пример не выдает вердикта, то возникает особое состояние по тайм-ауту, тестовый пример останавливается, и выдается вердикт **error**. Однако TTCN-3 не имеет семантики реального времени, поэтому решение о том, возникло ли особое состояние по тайм-ауту, моделируется в виде недетерминистического выбора.

ПРИМЕЧАНИЕ. – Операционная семантика моделирует только недетерминистический выбор. <float_expression> не определяется.

Если благодаря недетерминистическому выбору не появилось особое состояние по тайм-ауту, то создается **mtc**, экземпляр управления (представляющий управляющую часть модуля TTCN-3) блокируется до окончания тестового примера, а для выполнения следующего тестового примера поток управления передается к **mtc**. Поток управления передается обратно к экземпляру управления, когда закончится **mtc**.

Сегмент <execute-stmt> потокового графа на рис. B.58 определяет выполнение команды **execute**.

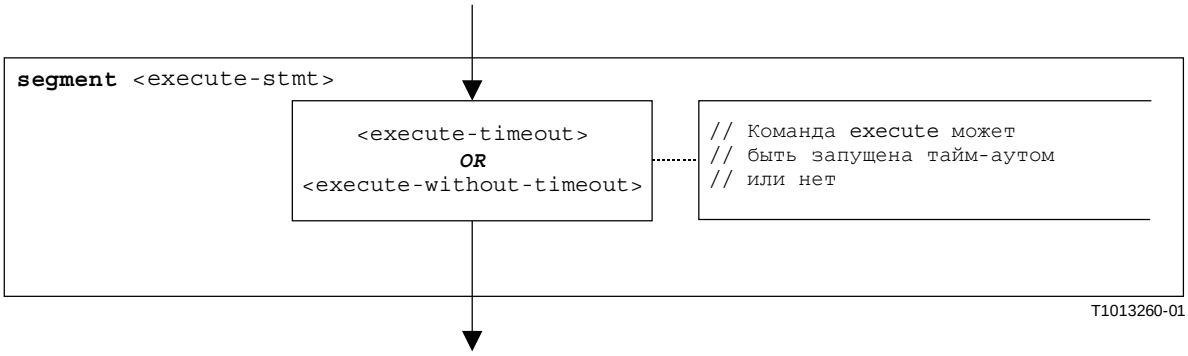


Рисунок B.58/Z.140 – Сегмент <execute-stmt> потокового графа

B.3.7.17.1 Сегмент <execute-timeout> потокового графика

Сегмент <execute-timeout> потокового графа на рис. B.59 определяет выполнение команды **execute**, которое защищено значением тайм-аута.

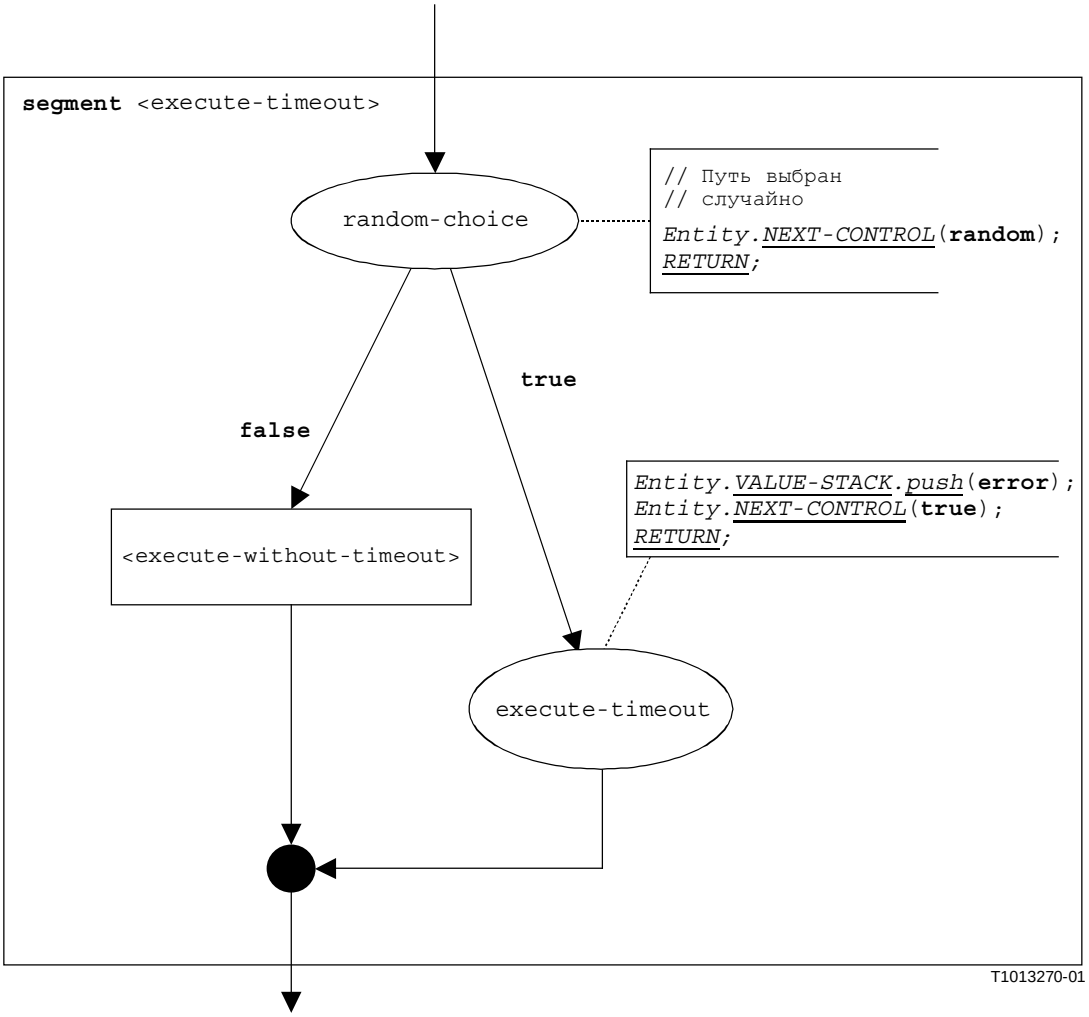


Рисунок B.59/Z.140 – Сегмент <execute-timeout> потокового графа

B.3.7.17.2 Сегмент <execute-without-timeout> потокового графа

Выполнение тестового примера начинается с создания **mtc**. Затем **mtc** запускается с поведением, описанным в определении тестового примера. Потом управление модулем ожидает, пока не закончится тестовый пример. Создание и запуск **mtc** могут описываться командами **create** и **start**:

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

Сегмент <execute-without-timeout> потокового графа на рис. B.60 определяет выполнение команды **execute**, не имеющей особого состояния по тайм-ауту, при помощи сегментов потокового графа с операциями **create** и **start**.

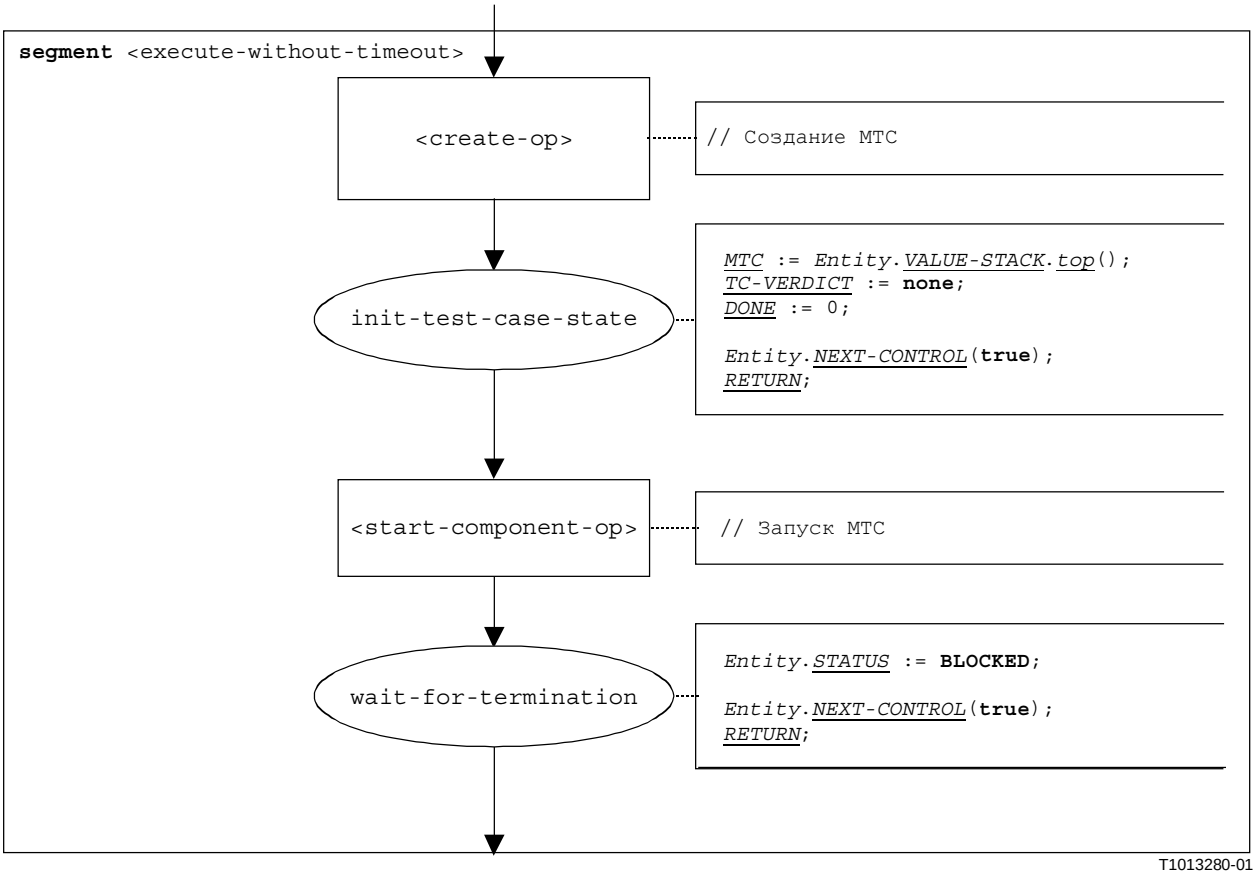


Рисунок B.60/Z.140 – Сегмент <execute-without-timeout> потокового графа

B.3.7.18 Выражение

При обработке выражений различают следующие четыре случая:

a) выражением является буквенное значение (или константа);

b) выражением является переменная;

c) выражением является оператор, приложенный к одному или к нескольким операндам;

d) выражением является функция или вызов операции.

Синтаксической структурой выражения является:

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

где:

<lit-val>	означает буквенное значение;
<var-val>	означает значение переменной;
<func-op-call>	означает функцию или вызов операции;
<operator-appl>	означает применение арифметических операторов, таких как +, –, not и т. п.

Выполнение выражания определяется сегментом <expression>, показанным на рис. B.61.

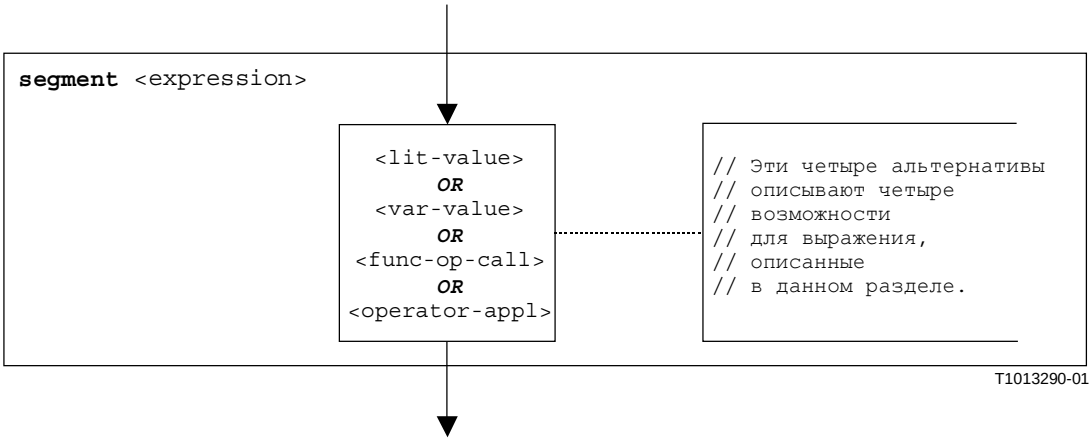


Рисунок B.61/Z.140 – Сегмент <expression> потокового графа

B.3.7.18.1 Сегмент <lit-value> потокового графа

Сегмент <lit-value> потокового графа на рис. B.62 помещает буквенное значение в магазин значений объекта.

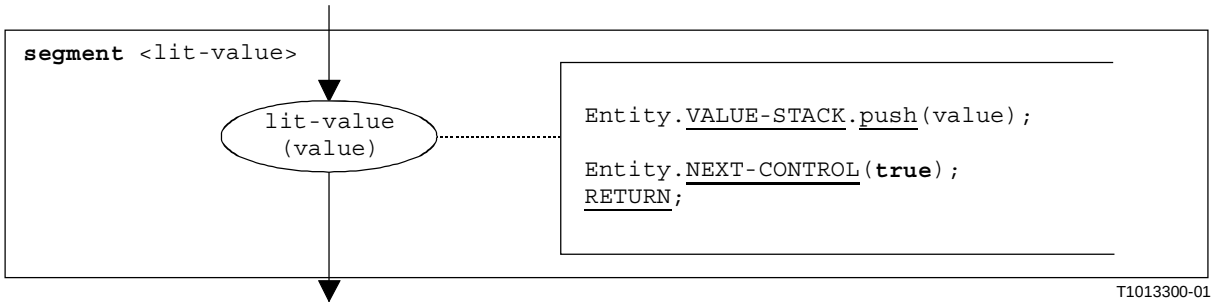


Рисунок B.62/Z.140 – Сегмент <lit-value> потокового графа

B.3.7.18.2 Сегмент <var-value> потокового графа

Сегмент <var-value> потокового графа на рис. B.63 помещает значение переменной в магазин значений объекта.

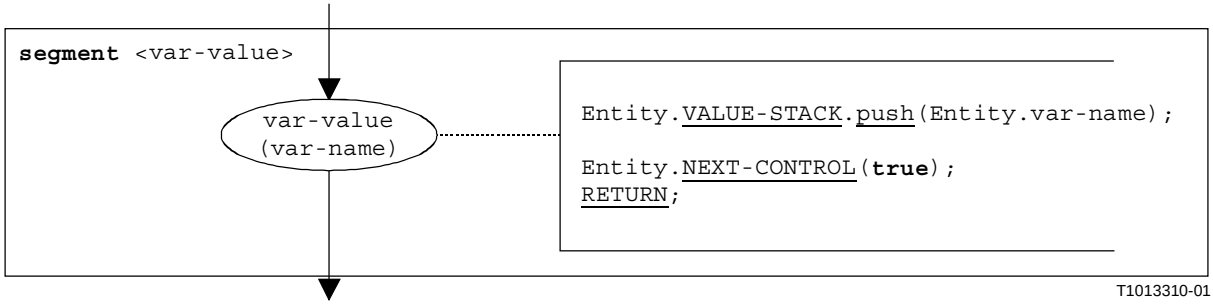


Рисунок B.63/Z.140 – Сегмент <var-value> потокового графа

B.3.7.18.3 Сегмент <func-op-call> потокового графа

Сегмент <func-op-call> потокового графа на рис. B.64 показывает вызовы функций и операций, выдающих значение, которое помещается в магазин значений объекта. Все эти вызовы рассматриваются как выражения.

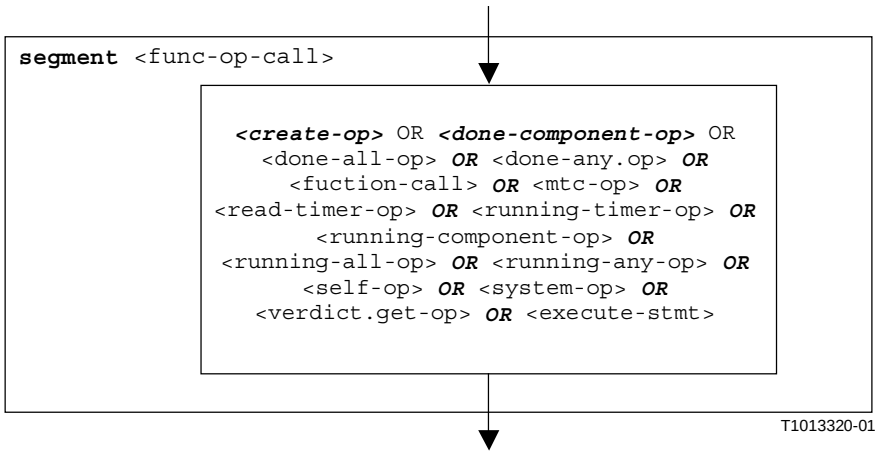


Рисунок B.64/Z.140 – Сегмент <func-op-call> потокового графа

B.3.7.18.4 Сегмент <operator-appl> потокового графа

Представление потокового графа на рис. B.65 прямо отсылает к предположению, что для оценки выражений оператора используется постфиксная (инверсная польская) нотация. Операнды оператора вычисляются и помещаются в магазин оценок. Для применения этого оператора операнды извлекаются из магазина оценок, затем оператор применяется. В конце концов результат применения оператора помещается в этот магазин оценок.

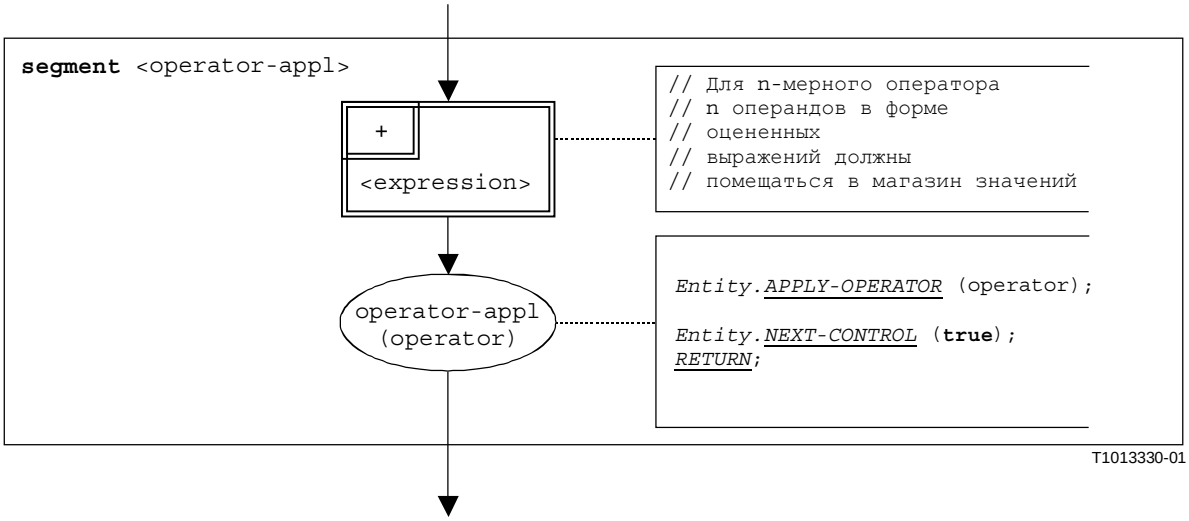


Рисунок B.65/Z.140 – Сегмент <operator-appl> потокового графа

B.3.7.19 Сегмент <finalize-component-init> потокового графа

Сегмент `<finalize-component-init>` потокового графа является частью потокового графа, представляющего поведение определения типа компонента. Его выполнение определяется на рис. B.66.

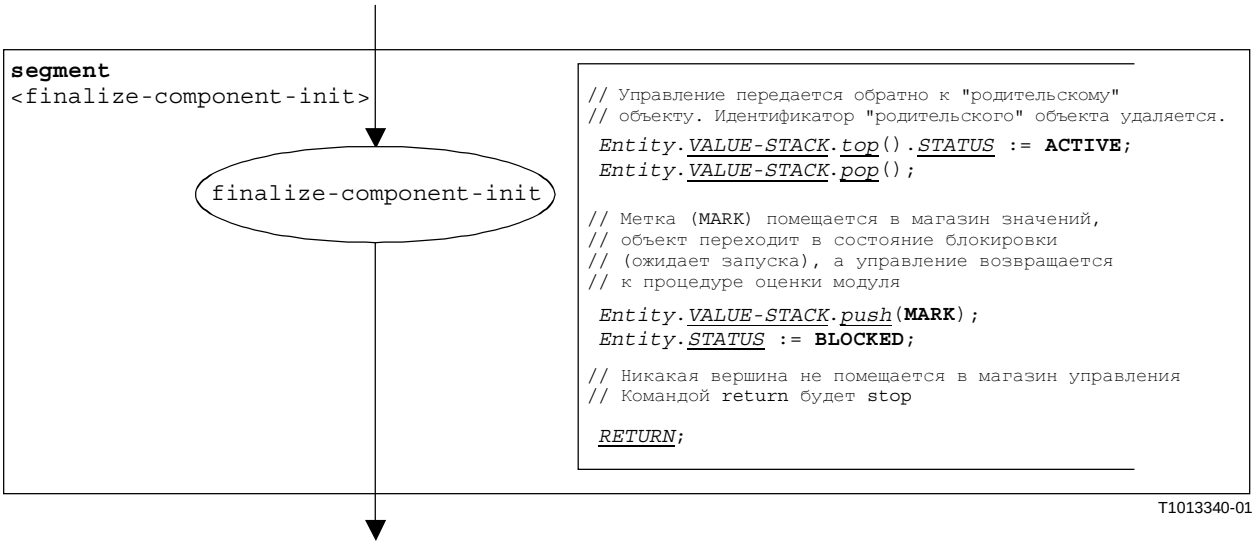


Рисунок B.66/Z.140 – Сегмент <finalize-component-init> потокового графа

B.3.7.20 Сегмент <init-component-scope> потокового графа

Сегмент <init-component-scope> потокового графа является частью потокового графа, представляющего поведение определения типа компонента. Его выполнение определяется на рис. B.67.

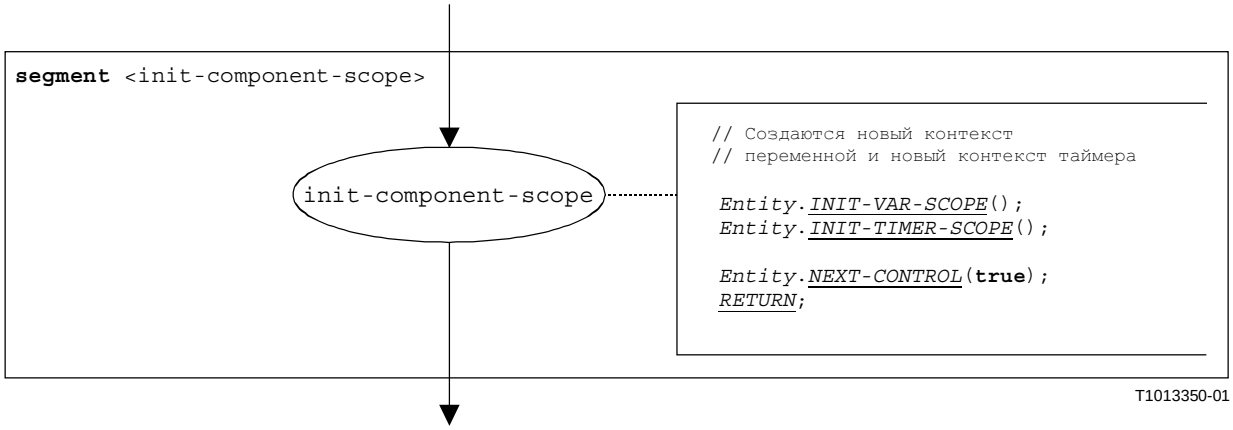


Рисунок B.67/Z.140 – Сегмент <init-component-scope> потокового графа

B.3.7.21 Команда For

Синтаксической структурой команды **for (for-statement)** является:

for (<assignment>, <boolean_expression>, <assignment>) <statement-block>

Инициализация индексной переменной и соответствующая манипуляция с индексной переменной рассматриваются как присвоения этой индексной переменной. Далее <boolean_expression> описывает критерий окончания цикла, указанный этой командой **for**, а <statement-block> описывает тело цикла.

Выполнение команды **for** определяется сегментом <for-stmt>, показанным на рис. B.68. Начальное <assignment> описывает инициализацию индексной переменной. Другое <assignment> в ветви **true**, идущей от вершины decision, описывает манипулирование индексной переменной.

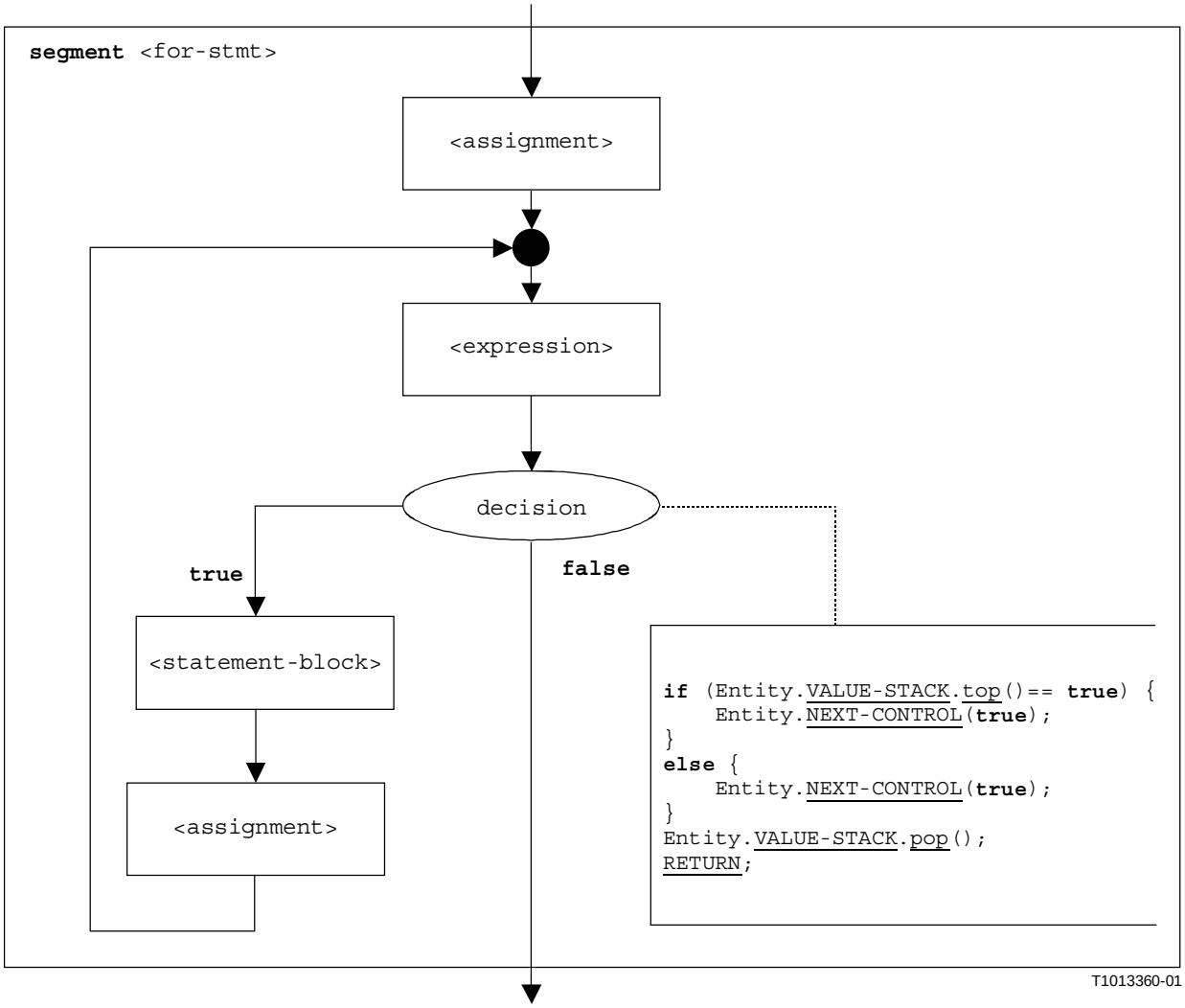


Рисунок B.68/Z.140 – Сегмент <for-stmt> потокового графа

B.3.7.22 Вызов функции

Синтаксической структурой вызова функции является:

<function-name>([<act-par-desc$_1$>, ... , <act-par-desc$_n$>])

Здесь <function-name> указывает имя функции, а <act-par-descr$_1$>, ... , <act-par-descr$_n$> дают описание значений реальных параметров в вызове функции. В случае параметра значения описание реального параметра может обеспечиваться в виде выражения, которое должно быть вычислено до выполнения вызова.

Предполагается, что каждое <act-par-desc$_1$>, соответствующее идентификатору формального параметра <f-par-Id$_1$>, известно, то есть мы можем расширить вышеприведенную синтаксическую структуру до:

<function-name>((<f-par-Id$_1$>, <act-par-desc$_1$>), ... , (<f-par-Id$_n$>,
<act-par-desc$_n$>))

Сегмент <function-call> потокового графа на рис. B.69 определяет выполнение вызова функции. Выполнение разделено на три шага. Первый шаг – это создание записи вызова для <function-name> этой функции. На втором шаге значения реального параметра вычисляются и присваиваются соответствующему полю в записи вызова. На третьем шаге переносится управление поведением, вызывающее функцию.

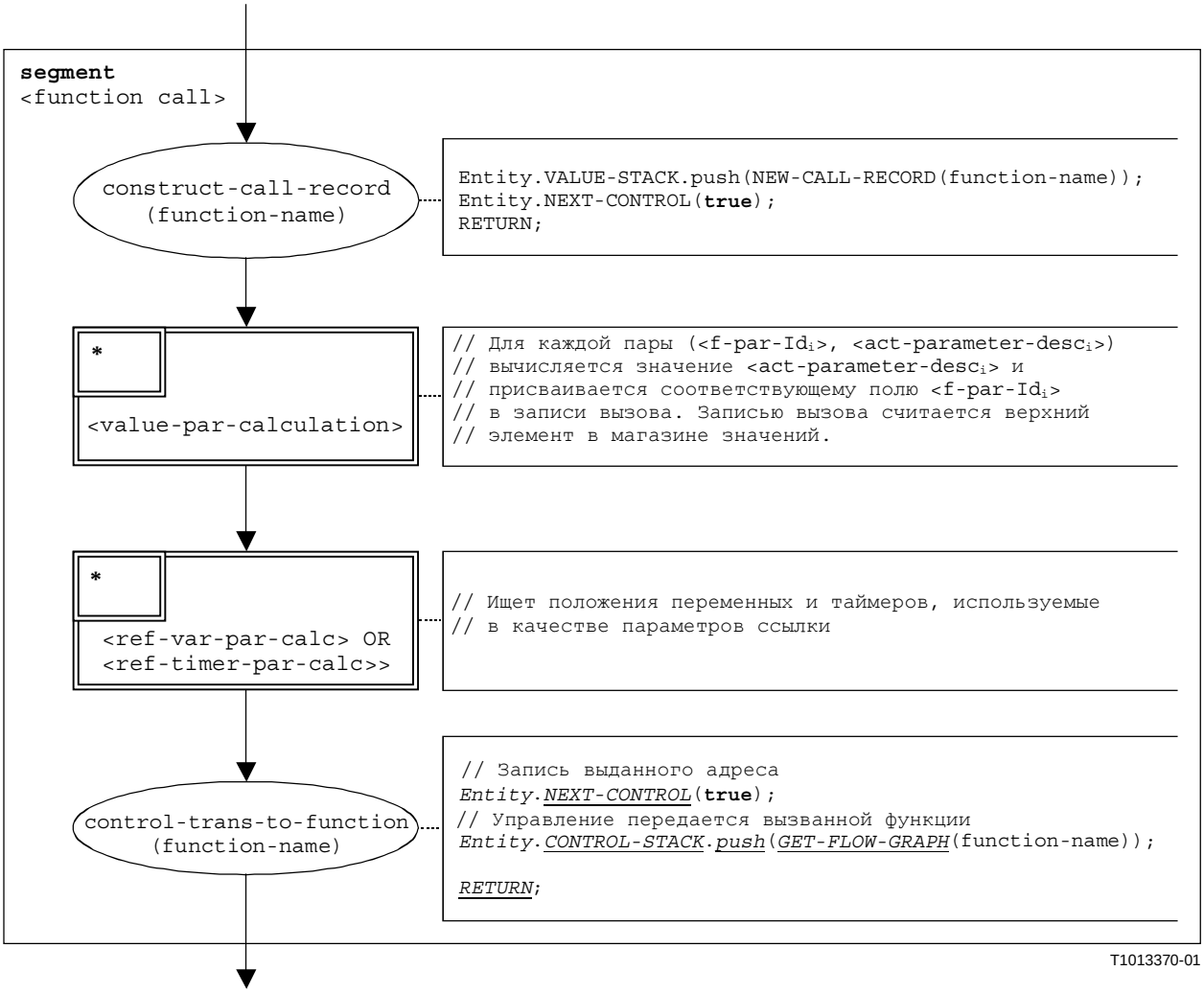


Рисунок B.69/Z.140 – Сегмент <function-call> потокового графа

B.3.7.23 Сегмент <value-par-calculation> потокового графа

Сегмент <value-par-calculation> потокового графа используется для вычисления значений реальных параметров и присвоения их соответствующим полям в записях вызовов функций и тестовых примеров.

Предполагается, что запись вызова является верхним элементом в магазине значений и что должна обрабатываться пара:

(<f-par-Id$_i$>, <act-parameter-desc$_i$>)

Здесь <act-parameter-desc$_i$> – это то, что должно оцениваться, а <f-par-Id$_i$> – это идентификатор формального параметра, имеющего соответствующее поле в записи вызова в магазине значений.

Выполнение сегмента <value-par-calculation> потокового графа показано на рис. B.70.

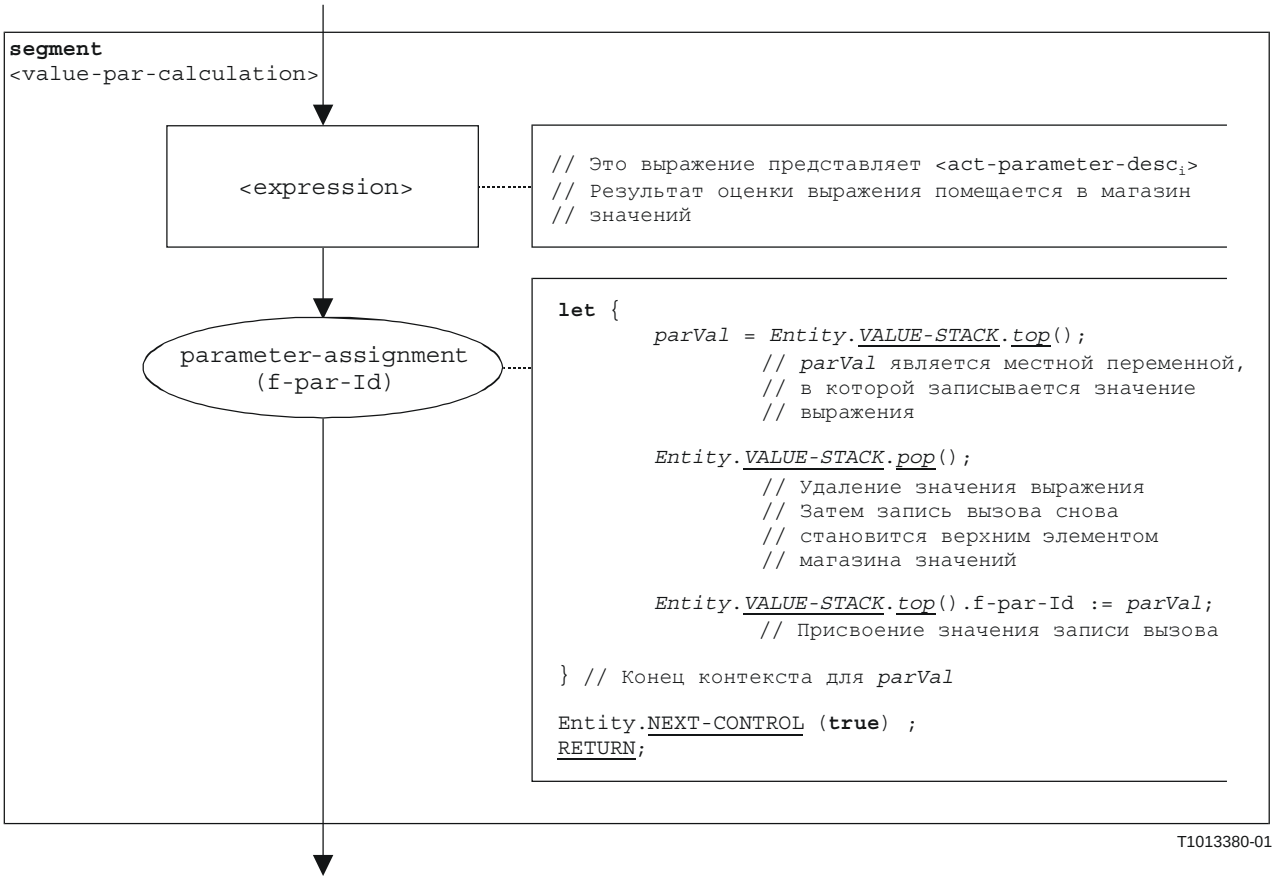


Рисунок B.70/Z.140 – Сегмент <value-par-calculation> потокового графа

B.3.7.24 Сегмент <ref-par-var-calc> потокового графа

Сегмент <ref-par-var-calc> потокового графа используется для поиска положений переменных, используемых в качестве реальных параметров ссылки, и для присвоения их соответствующим полям в записях вызовов для функций и тестовых примеров.

Предполагается, что запись вызова является верхним элементом в магазине значений и что должна обрабатываться пара:

(<f-par-Id$_i$>, <act-par$_i$>)

Здесь <act-par$_i$> – это реальный параметр, для которого необходимо искать положение, а <f-par-Id$_i$> – это идентификатор формального параметра, который имеет соответствующее поле в записи вызова в магазине значений.

Выполнение сегмента <ref-par-var-calc> показано на рис. B.71.

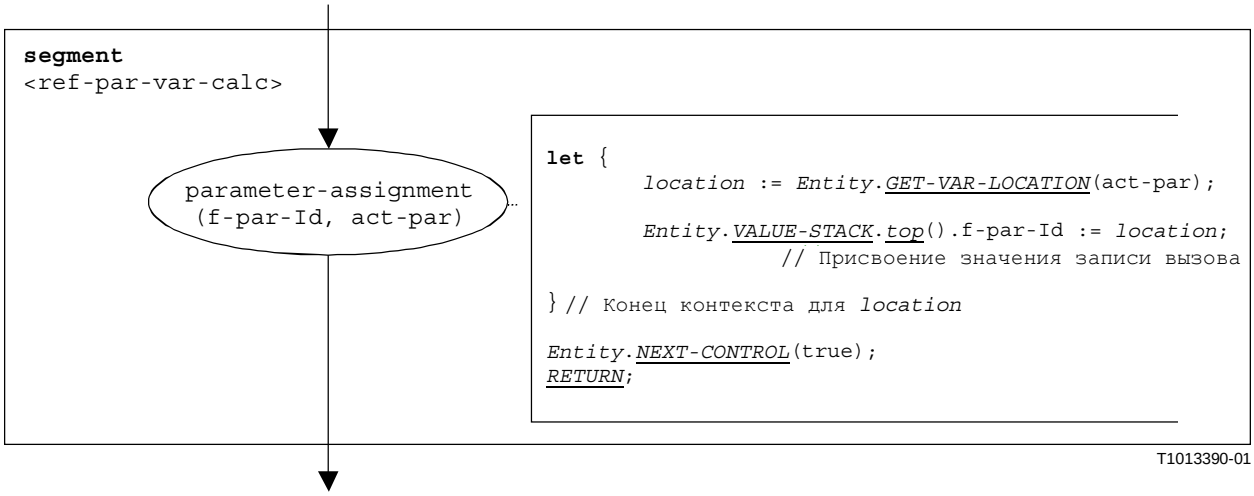


Рисунок B.71/Z.140 – Сегмент <ref-par-var-calc> потокового графа

B.3.7.25 Сегмент <ref-par-timer-calc> потокового графа

Сегмент <ref-par-timer-calc> потокового графа используется для поиска положений таймеров, используемых в качестве реальных параметров ссылок, и для присвоения их соответствующим полям в записях вызовов для функций и тестовых примеров.

Предполагается, что запись вызова является верхним элементом в магазине значений и что должна обрабатываться пара:

(<f-par-Id$_i$>, <act-par$_i$>)

Здесь <act-par$_i$> – это реальный параметр, для которого следует искать положение, а <f-par-Id$_i$> – это идентификатор формального параметра, который имеет соответствующее поле в записи вызова в магазине значений.

Выполнение сегмента <ref-par-timer-calc> потокового графа показано на рис. B.72.

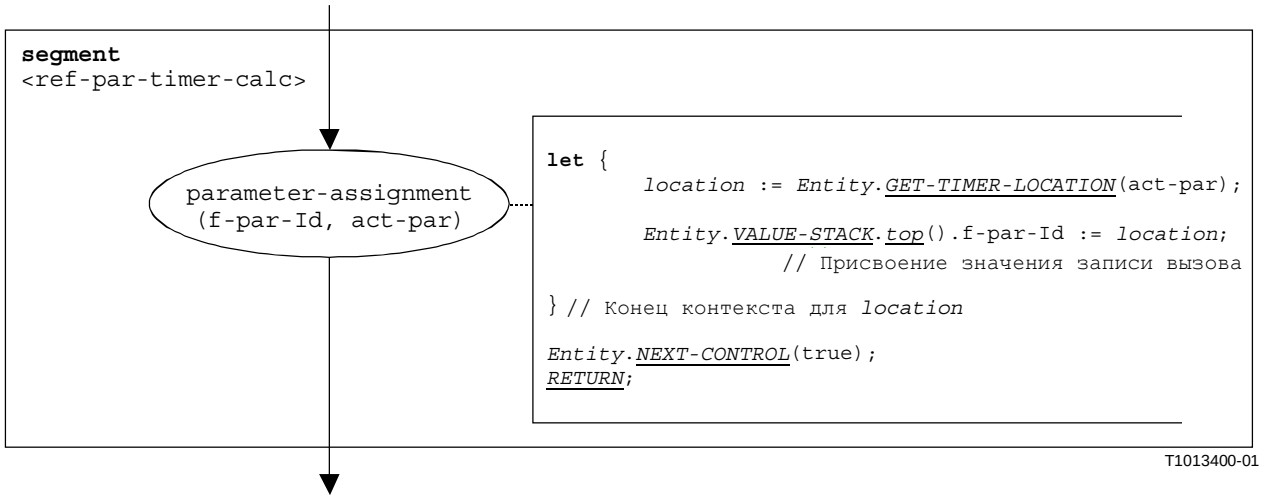


Рисунок B.72/Z.140 – Сегмент <ref-par-timer-calc> потокового графа

B.3.7.26 Сегмент <parameter-handling> потокового графа

Сегмент <parameter-handling> потокового графа используется для начала вызовов функций. Он инициализирует новый контекст и создает переменные и таймеры для обработки параметров. Предполагается, что запись вызова для вызываемой функции является верхним элементом в магазине значений.

Выполнение сегмента <parameter-handling> потокового графа показано на рис. B.73.

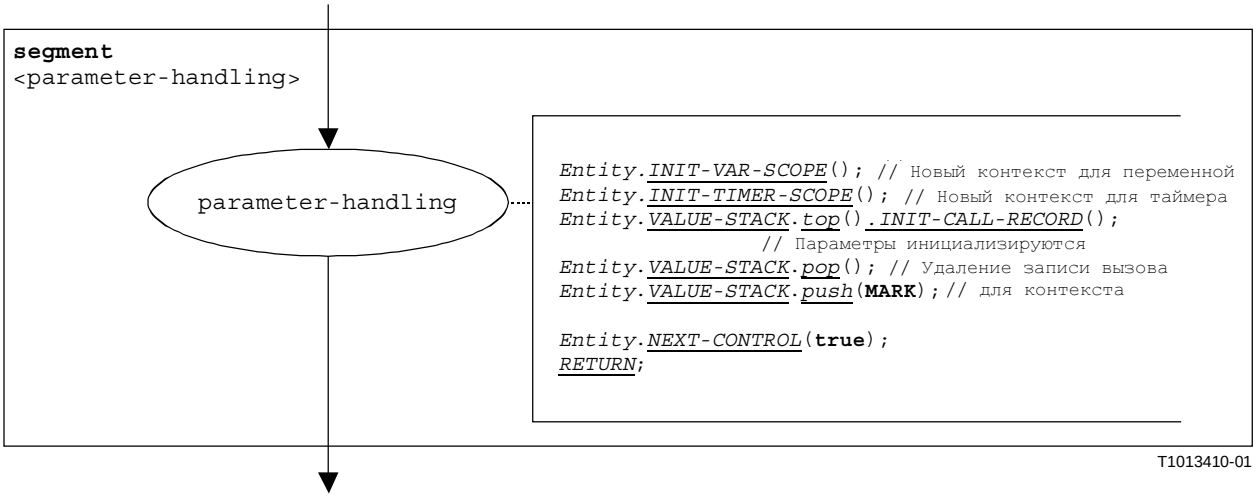


Рисунок B.73/Z.140 – Сегмент <parameter-handling> потокового графа

B.3.7.27 Операция Getcall

Синтаксической структурой операции **getcall** является:

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

Факультативное <component_expression> из раздела **from** указывает передатчик вызова, который обрабатывается операцией **getcall**. Он может обеспечиваться в виде значения переменной или выдаваемого значения функции, то есть он считается выражением. Факультативная <assignmentPart> обозначает присвоение принятой информации, если принятый вызов соответствует спецификации сопоставления <matchingSpec> и разделу **from** (факультативному).

Сегмент <getcall-op> потокового графа на рис. B.74 определяет выполнение операции **getcall**.

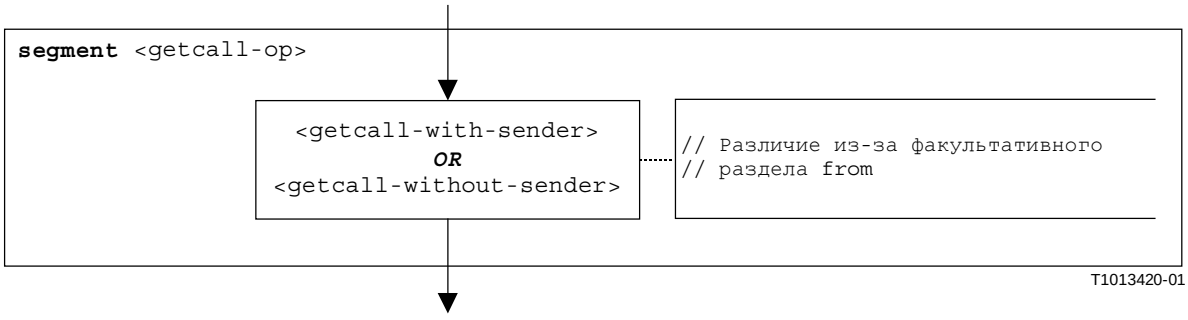


Рисунок B.74/Z.140 – Сегмент <getcall-op> потокового графа

B.3.7.27.1 Сегмент <getcall-with-sender> потокового графа

Сегмент <getcall-with-sender> потокового графа на рис. B.75 определяет выполнение операции getcall, при которой передатчик указывается в виде выражения.

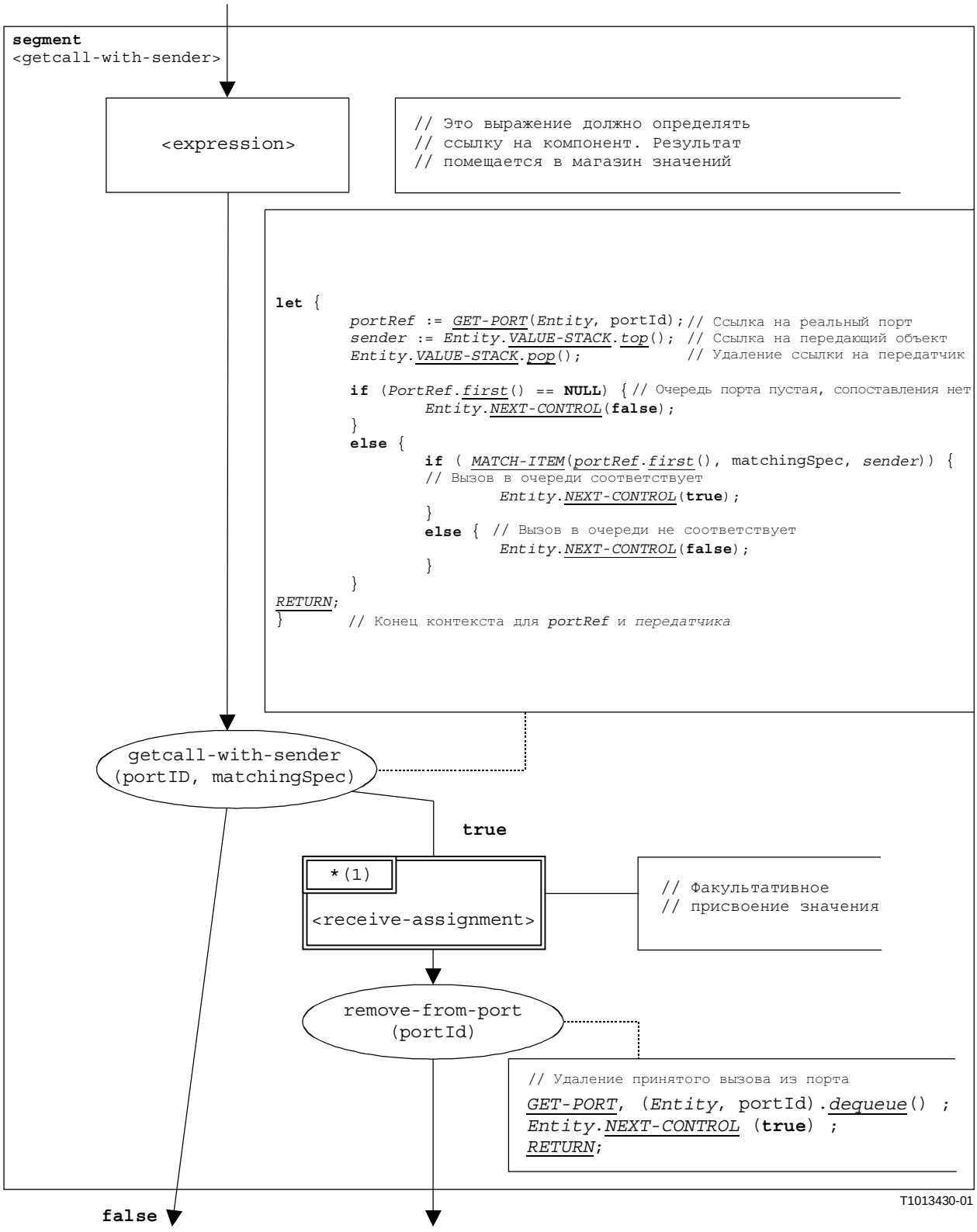


Рисунок B.75/Z.140 – Сегмент <getcall-with-sender> потокового графа

B.3.7.27.2 Сегмент <getcall-without-sender> потокового графа

Сегмент <getcall-without-sender> потокового графа на рис. B.76 определяет выполнение операции **getcall** без раздела **from**.

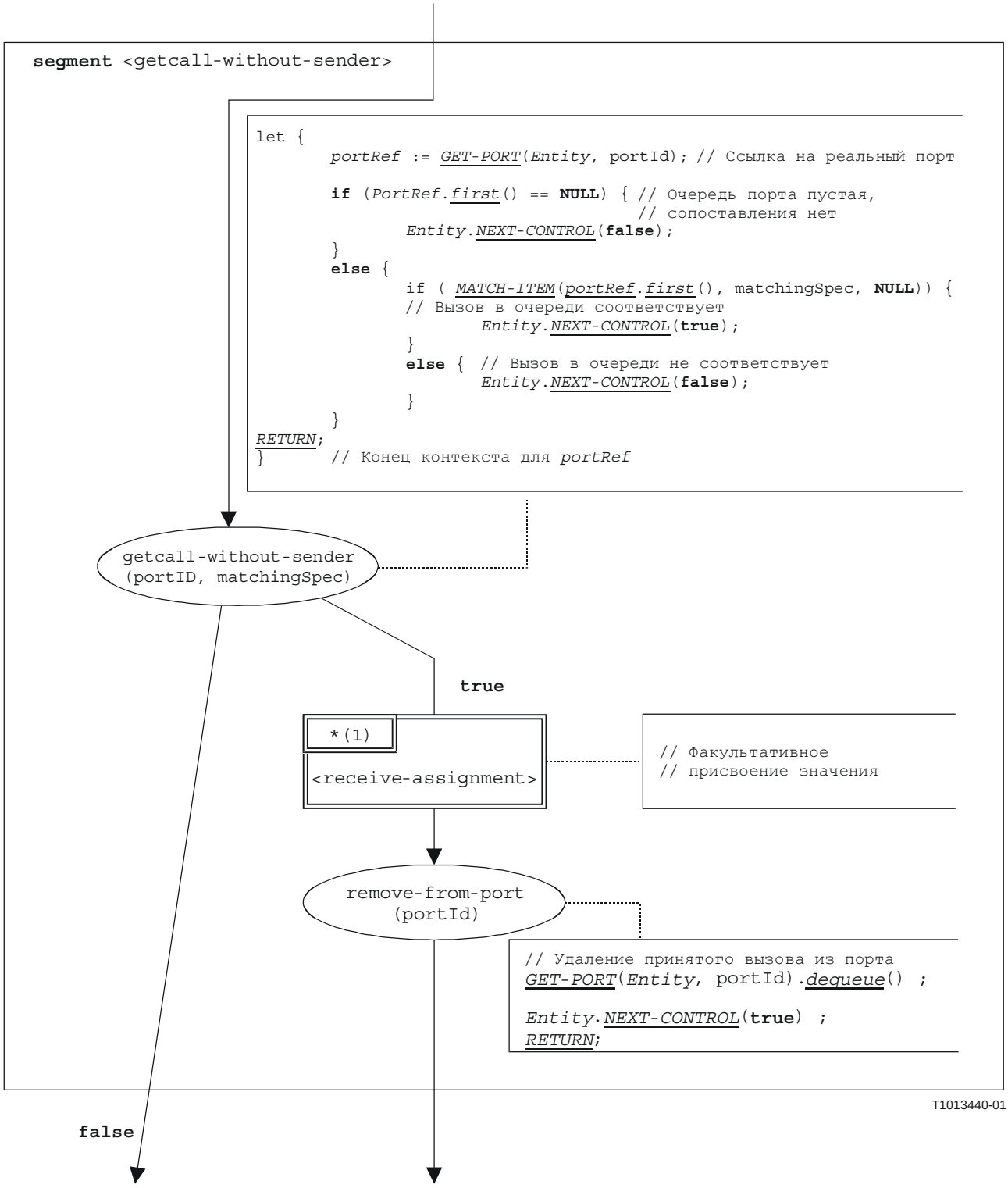


Рисунок B.76/Z.140 – Сегмент <getcall-without-sender> потокового графа

B.3.7.28 Операция Getreply

Синтаксической структурой операции **getreply** является:

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

Факультативное <component_expression> в разделе **from** указывает передатчик ответа, который обрабатывается операцией **getreply**. Он может обеспечиваться в виде значения переменной или выдаваемого значения функции, то есть он считается выражением. Факультативная <assignmentPart> обозначает присвоение принятой информации, если ответ соответствует спецификации сопоставления <matchingSpec> и разделу **from** (факультативному).

Сегмент <getreply-op> потокового графа на рис. B.77 определяет выполнение операции **getreply**.

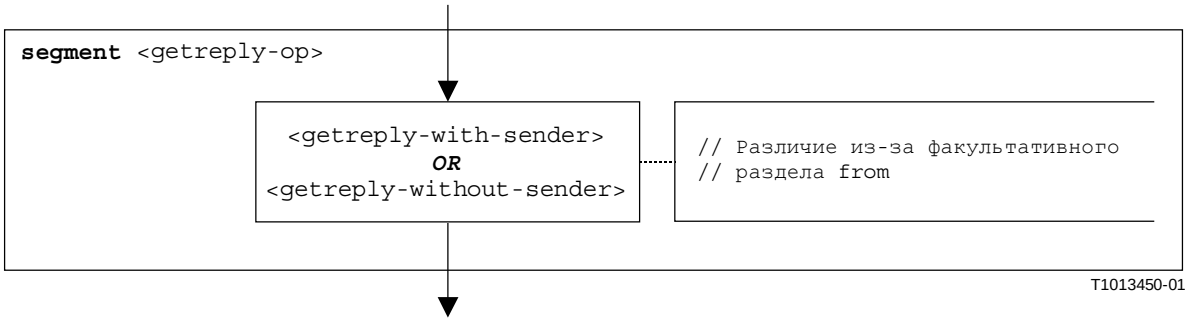


Рисунок B.77/Z.140 – Сегмент <getreply-op> потокового графа

B.3.7.28.1 Сегмент <getreply-with-sender> потокового графа

Сегмент <getreply-with-sender> потокового графа на рис. B.78 определяет выполнение операции **getreply**, при которой передатчик указывается в виде выражения.

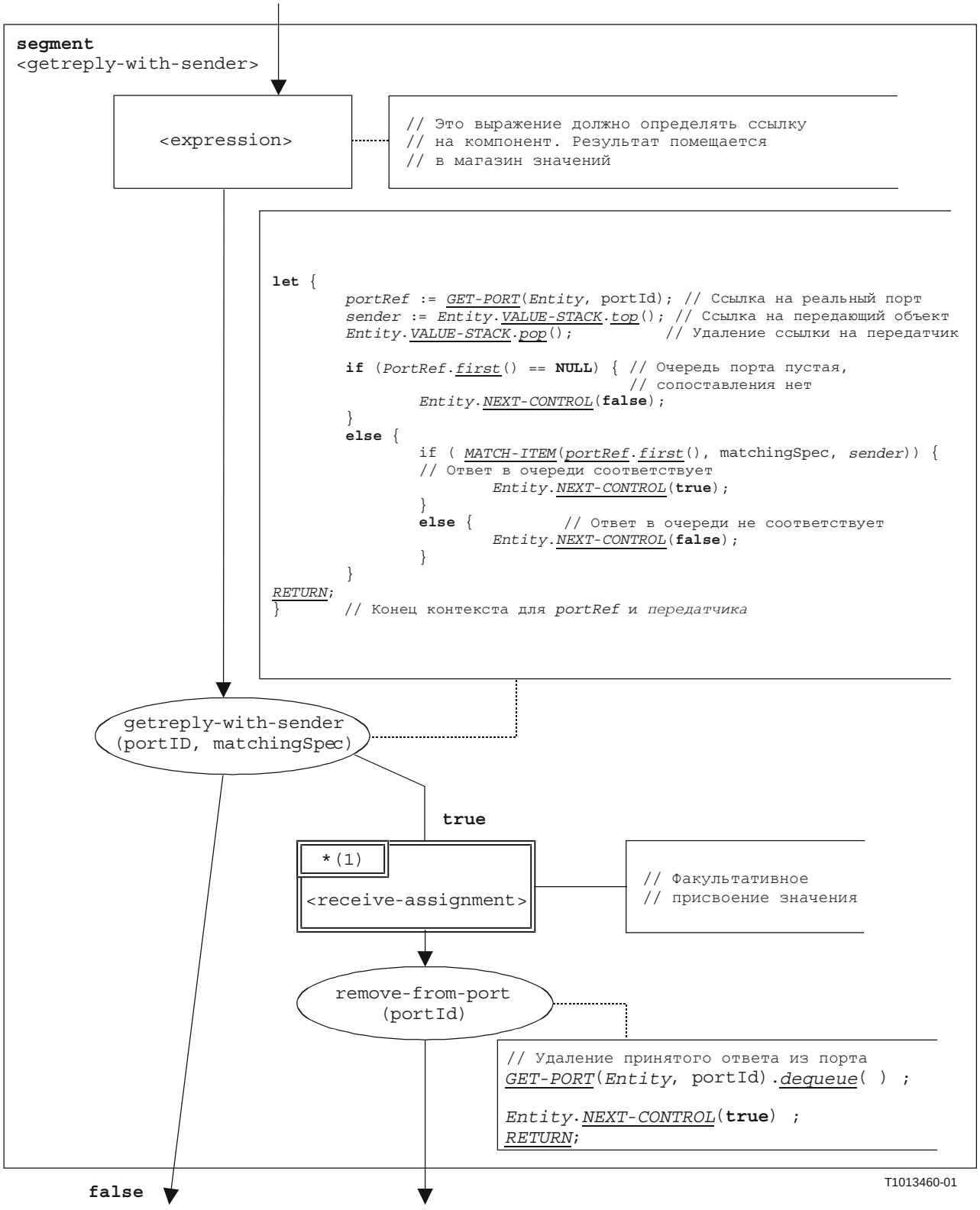


Рисунок B.78/Z.140 – Сегмент <getreply-with-sender> потокового графа

B.3.7.28.2 Сегмент <getreply-without-sender> потокового графа

Сегмент <getreply-without-sender> потокового графа на рис. B.79 определяет выполнение операции **getreply** без раздела **from**.

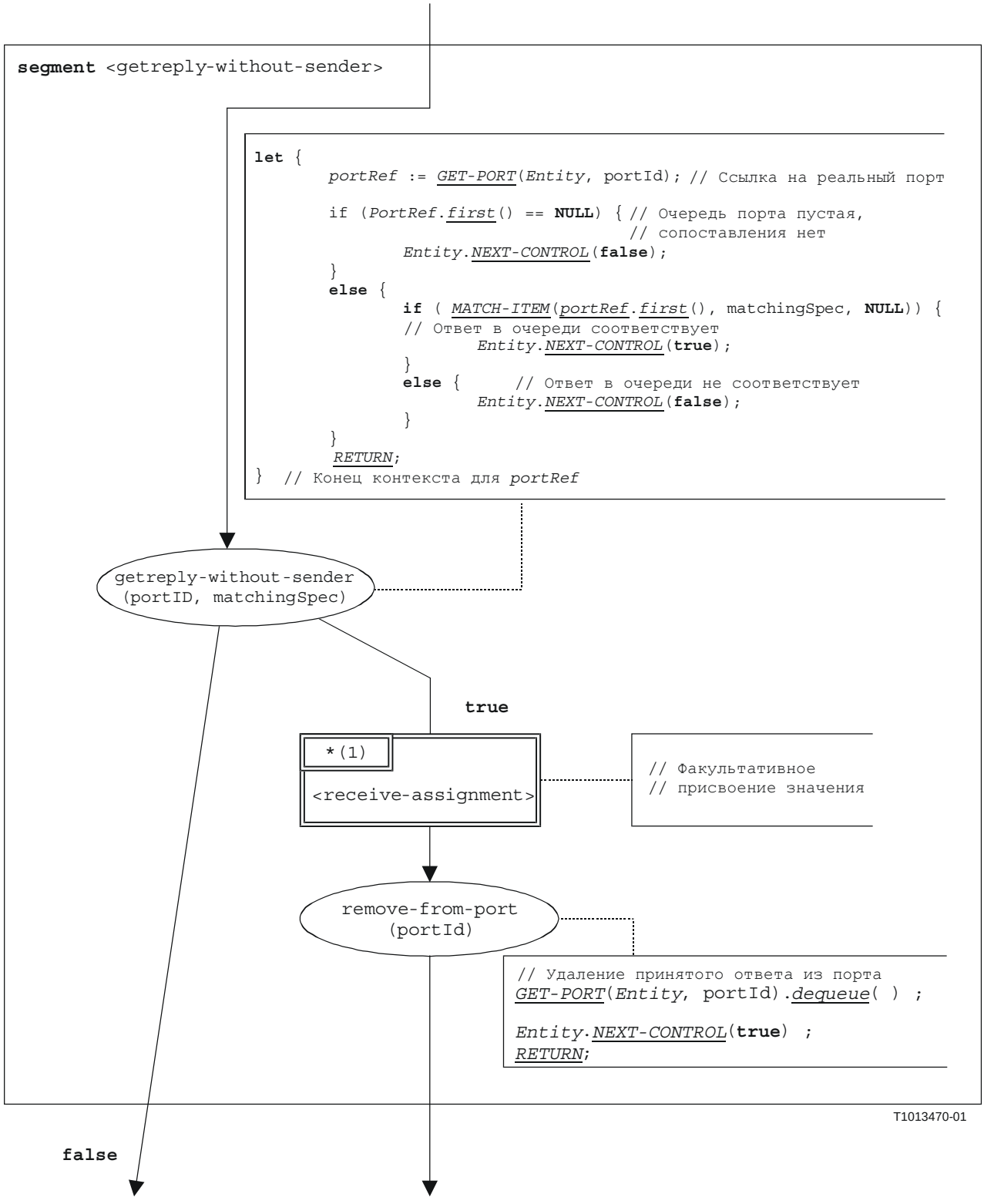


Рисунок B.79/Z.140 – Сегмент <getreply-without-sender> потокового графа

B.3.7.29 Команда Goto

Синтаксической структурой команды **goto** является:

goto <labelId>

Сегмент <goto-stmt> потокового графа на рис. B.80 определяет выполнение команды **goto**.

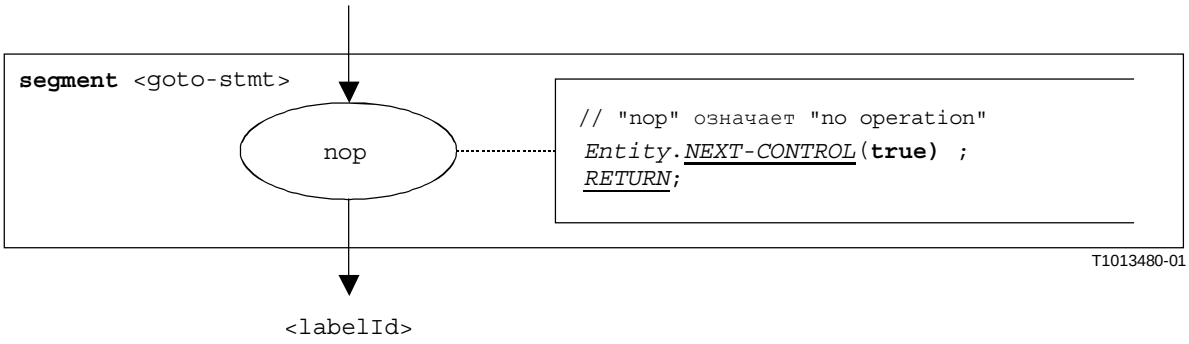


Рисунок B.80/Z.140 – Сегмент <goto-stmt> потокового графа

ПРИМЕЧАНИЕ. – Параметр <labelId> команды **goto** указывает на передачу управления в пункт, в котором определяется метка <*labelId*> (см. также п. B.3.7.31).

B.3.7.30 Команда If-else

Синтаксической структурой команды **if-else** является:

if (<boolean_expression>) <statement-block$_1$>
[**else** <statement-block$_2$>]

Часть else команды **if-else** является факультативной.

Сегмент <if-else-stmt> потокового графа на рис. B.81 определяет выполнение команды **if-else**.

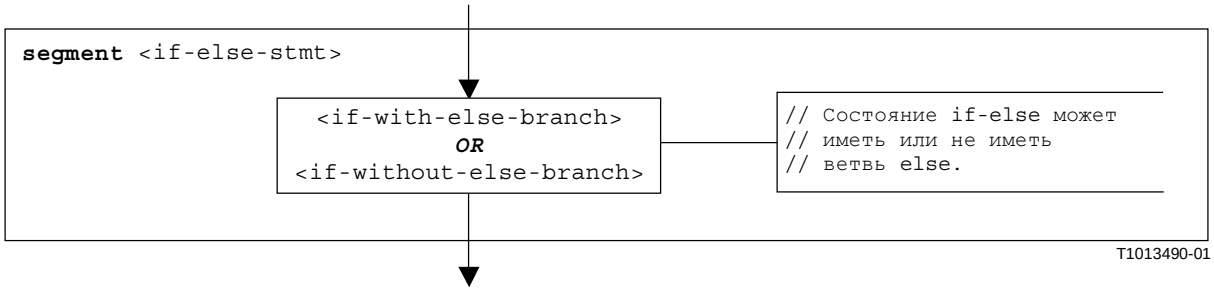


Рисунок B.81/Z.140 – Сегмент <if-else-stmt> потокового графа

B.3.7.30.1 Сегмент <if-with-else-branch> потокового графа

На рис. B.82 описывается выполнение команды **if-else**, которая имеет ветвь **else**. Блок <statement-block> в ветви **true** от вершины **decision** на рис. B.82 соответствует <statement-block$_1$> из вышеприведенной синтаксической структуры. Другой <statement-block> соответствует <statement-block$_2$> из этой структуры.

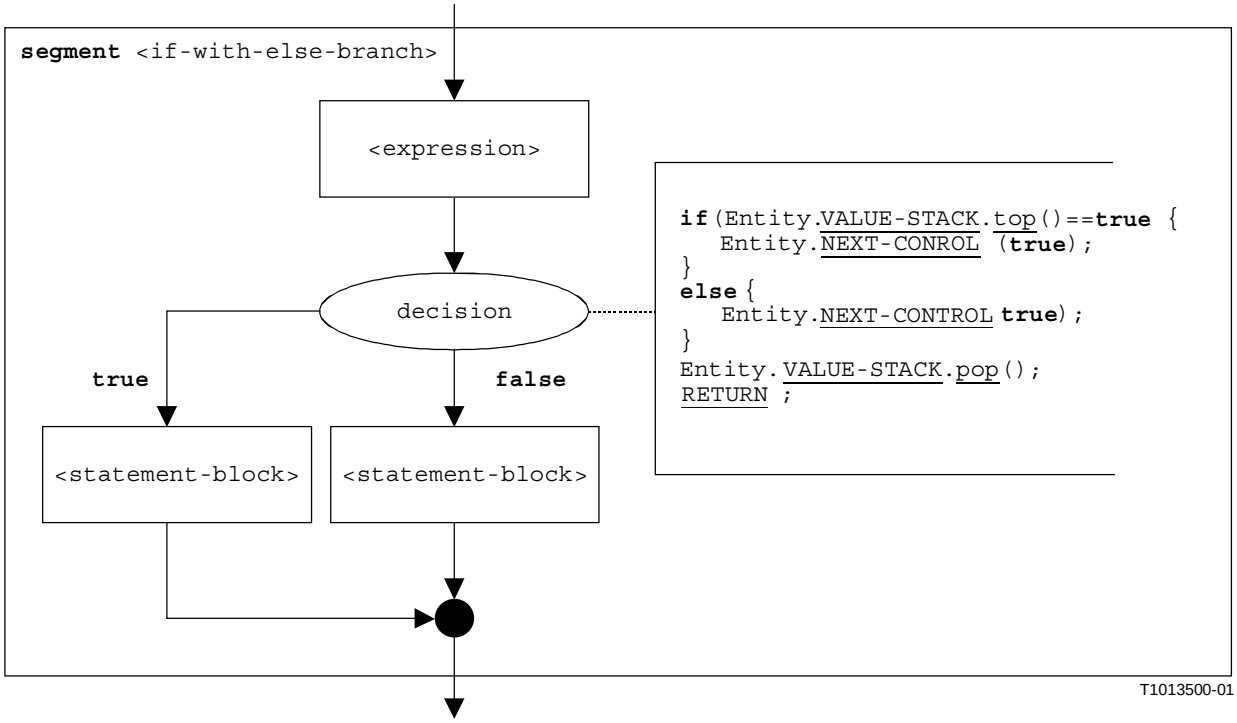


Рисунок B.82/Z.140 – Сегмент <if-with-else-branch> потокового графа

B.3.7.30.2 Сегмент <if-without-else-branch> потокового графа

На рис. B.83 описывается выполнение команды **if-else**, которая не имеет ветви **else**. Блок <statement-block> в ветви **true** от вершины **decision** на рис. B.83 соответствует <statement-block$_1$> из вышеприведенной синтаксической структуры.

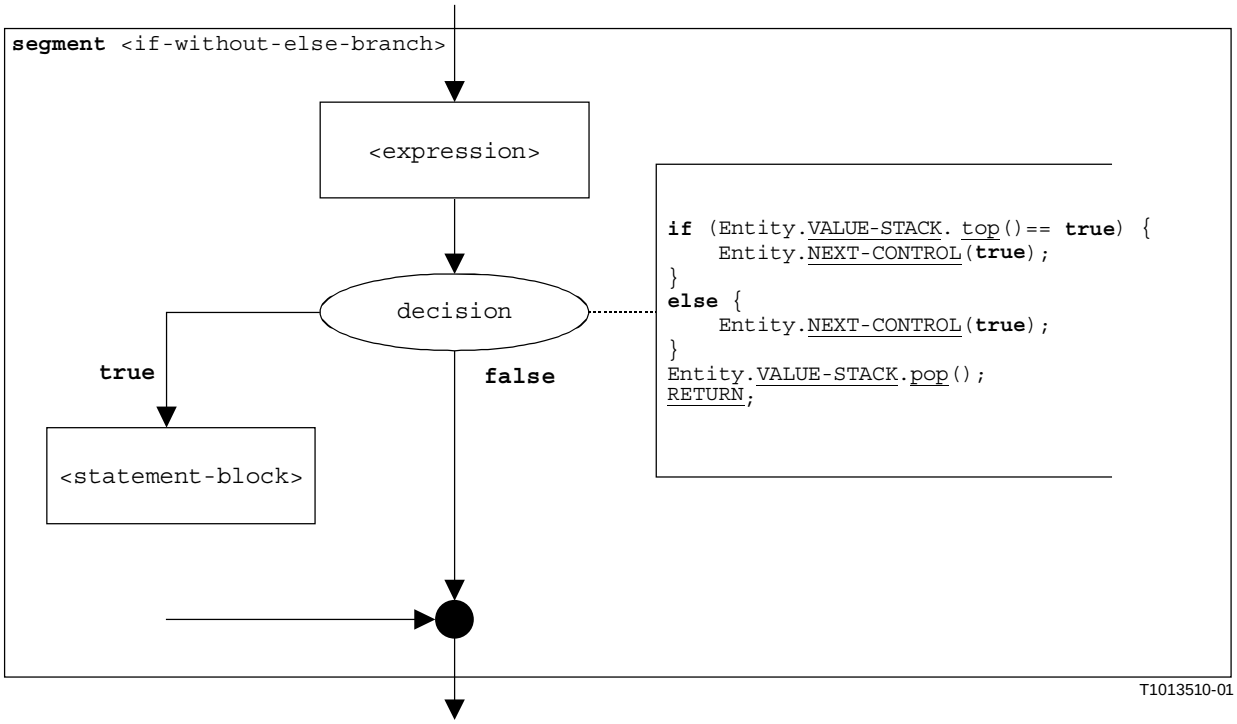


Рисунок B.83/Z.140 – Сегмент <if-without-else-branch> потокового графа

B.3.7.31 Команда Label

Синтаксической структурой команды **label** является:

label <labelId>

Сегмент <label-stmt> потокового графа на рис. B.84 определяет выполнение команды **label**.

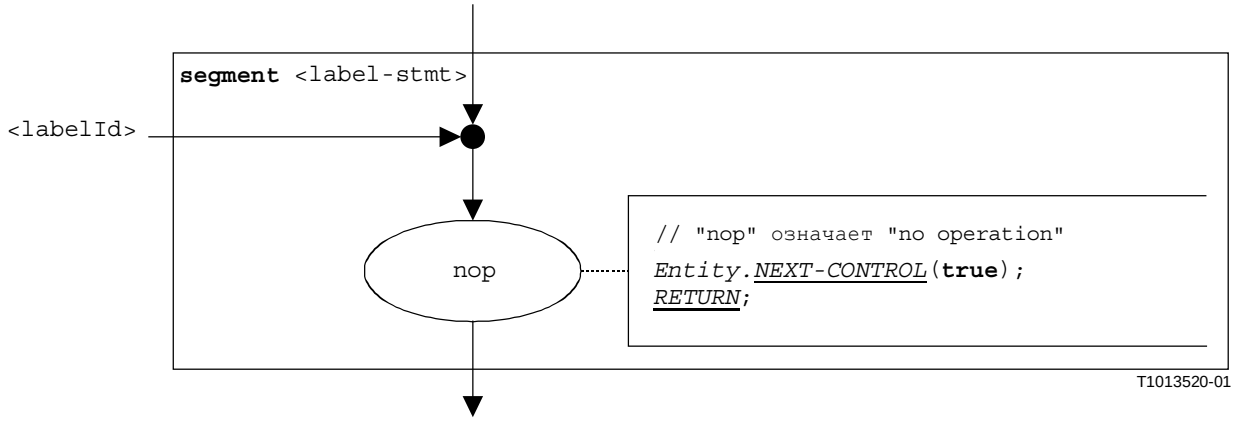


Рисунок B.84/Z.140 – Сегмент <label-stmt> потокового графа

ПРИМЕЧАНИЕ. – Параметр <labelId> команды **label** указывает на возможность того, что метка (label) может быть адресатом для передачи управления с помощью команды **goto** (см. также п. B.3.7.29).

B.3.7.32 Команда Log

Синтаксической структурой команды **log** является:

log (<informal-description>)

Сегмент <log-stmt> потокового графа на рис. B.85 определяет выполнение команды **log**.

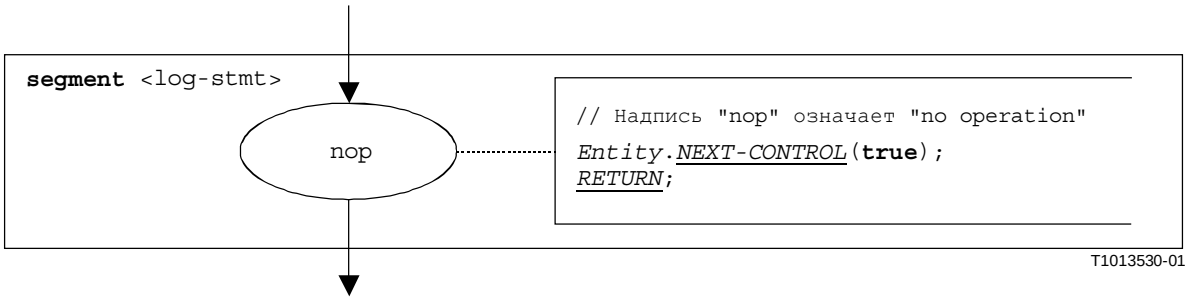


Рисунок B.85/Z.140 – Сегмент <log-stmt> потокового графа

ПРИМЕЧАНИЕ. – Параметр <informal-description> команды **log** не имеет значения для операционной семантики и поэтому не представлен в этом сегменте потокового графа.

B.3.7.33 Операция Map

Синтаксической структурой операции map является:

map(<component_expression>.<portId1>,**system**.<portId2>)

Идентификаторы <portId1> и <portId2> считаются идентификаторами соответствующих интерфейсов тестового компонента и тестовой системы. Компонент, которому принадлежит <portId1>, указывается с помощью ссылки на компонент <component_expression>. Эта ссылка может быть записана в переменных или выдана функцией. Для простоты она считается выражением, которое определяет ссылку на компонент. Поэтому для хранения ссылки на компонент используется магазин значений.

ПРИМЕЧАНИЕ. – Операция **map** не обращает внимания на то, появляется ли команда **system.**<portId> как первый или как второй параметр. Для простоты предполагается, что она является всегда вторым параметром.

Выполнение операции **map** определяется сегментом <map-op> потокового графа, показанного на рис. B.86.

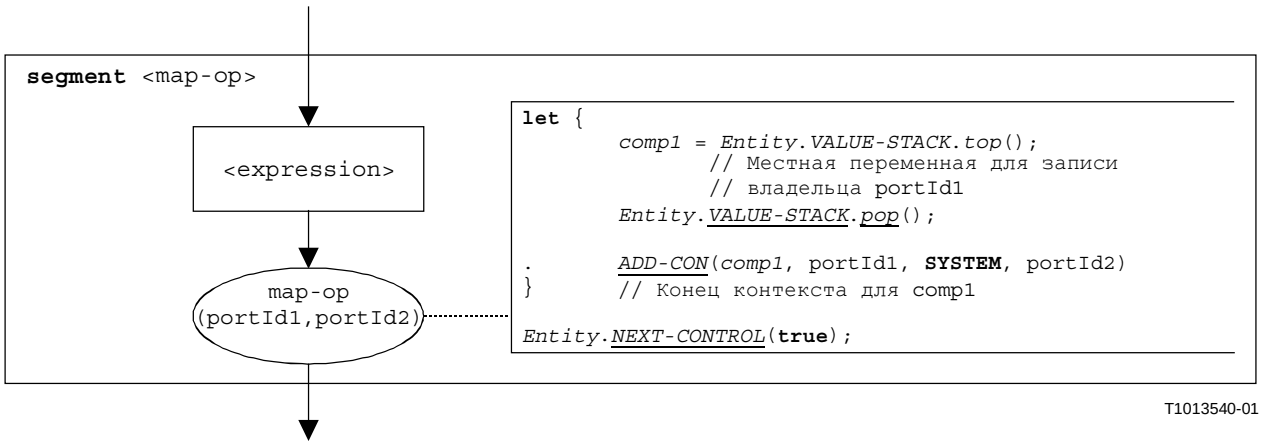


Рисунок B.86/Z.140 – Сегмент <map-op> потокового графа

B.3.7.34 Операция MTC

Синтаксической структурой операции **mtc** является:

mtc

Сегмент <mtc-op> потокового графа на рис. B.87 определяет выполнение операции **mtc**.

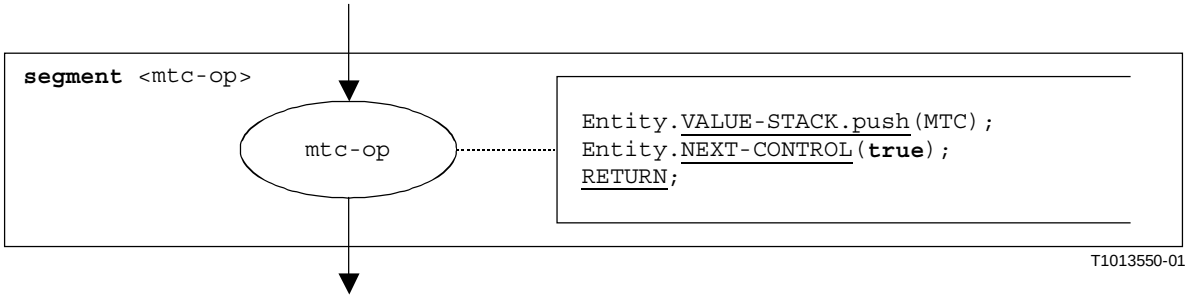


Рисунок B.87/Z.140 – Сегмент <mtc-op> потокового графа

B.3.7.35 Операция Raise

Синтаксической структурой операции **raise** является:

<portId>.**raise** (<exceptSpec>) [**to** <component_expression>]

Факультативное <component_expression> в разделе **to** указывает принимающий объект. Он может быть представлен в виде значения переменной или выданного значения функции.

Сегмент <raise-op> потокового графа на рис. B.88 определяет выполнение операции **raise**.

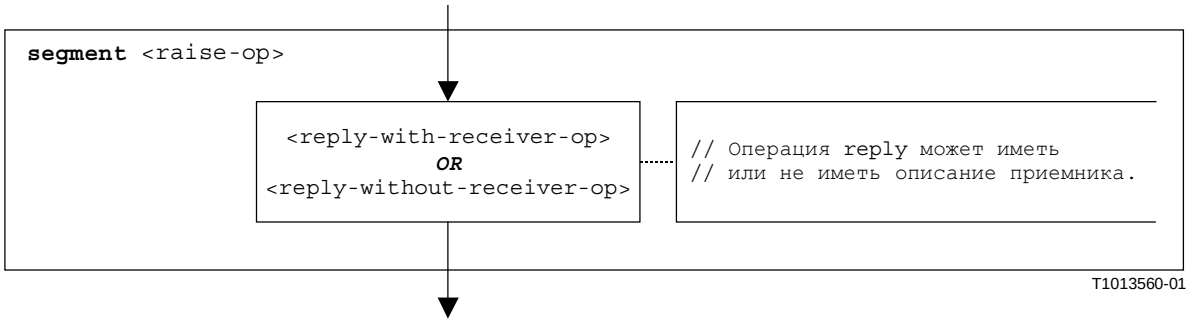


Рисунок B.88/Z.140 – Сегмент <raise-op> потокового графа

B.3.7.35.1 Сегмент <raise-with-receiver-op> потокового графа

Сегмент `<raise-with-receiver-op>` потокового графа на рис. B.89 определяет выполнение операции **`raise`**, при которой приемник указывается в виде выражения.

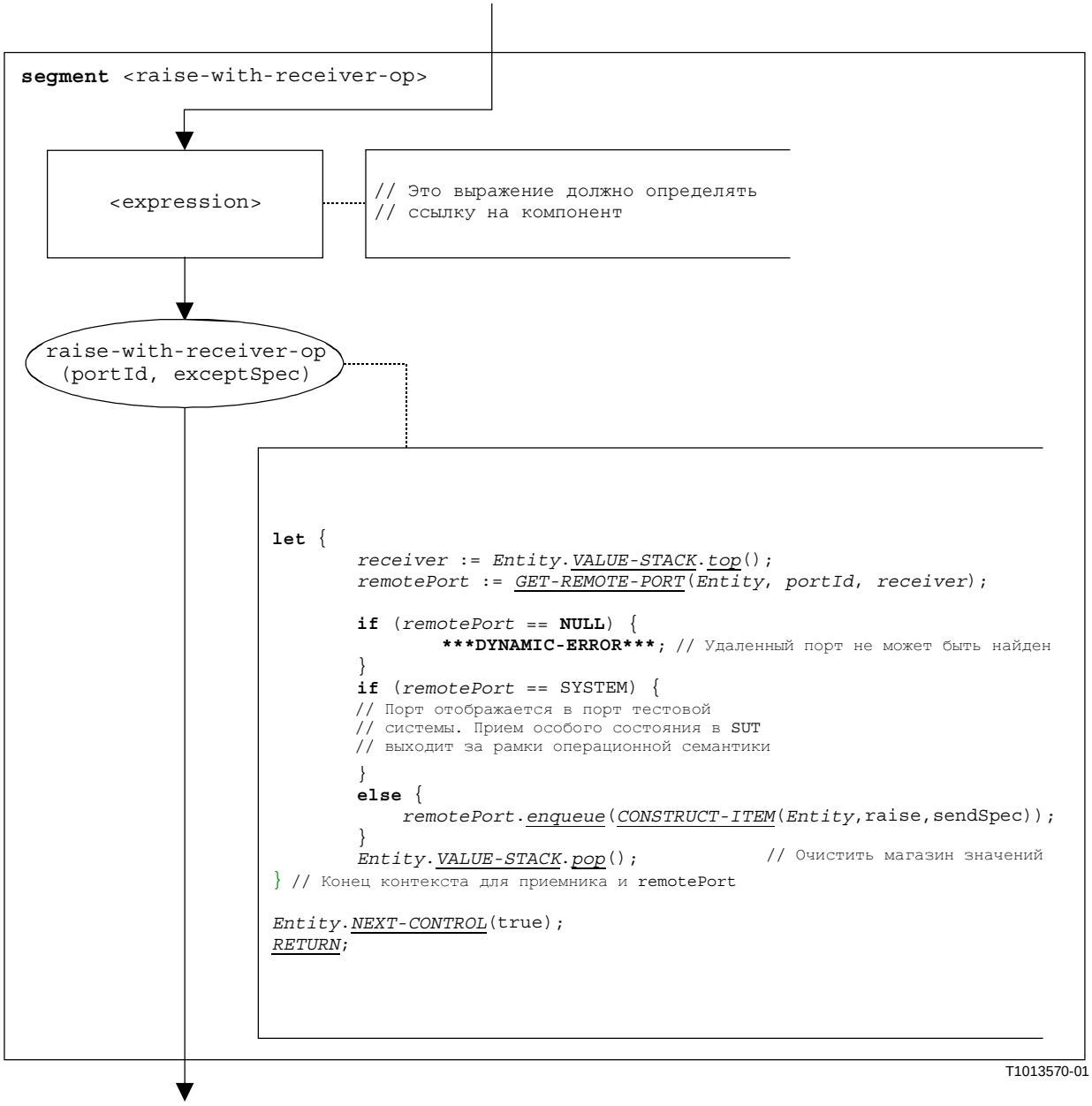


Рисунок B.89/Z.140 – Сегмент <raise-with-receiver-op> потокового графа

B.3.7.35.2 Сегмент <raise-without-receiver-op> потокового графа

Сегмент <raise-without-receiver-op> потокового графа на рис. B.90 определяет выполнение операции **raise** без раздела **to**.

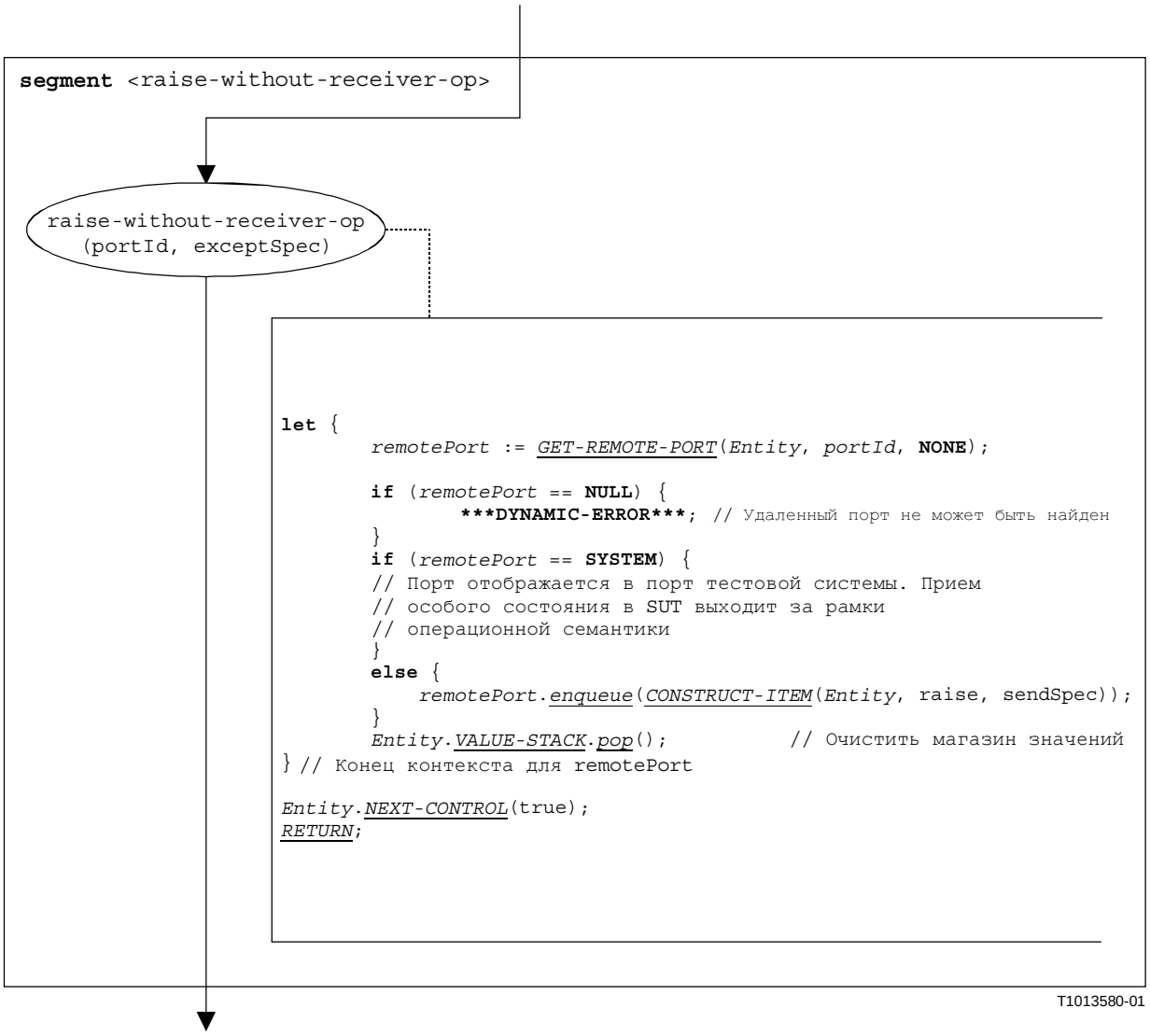


Рисунок B.90/Z.140 – Сегмент <raise-without-receiver-op> потокового графа

B.3.7.36 Таймерная операция Read

Синтаксической структурой таймерной операции **read** является:

```
<timerId>.read
```

Сегмент <read-timer-op> потокового графа на рис. B.91 определяет выполнение таймерной операции **read**.

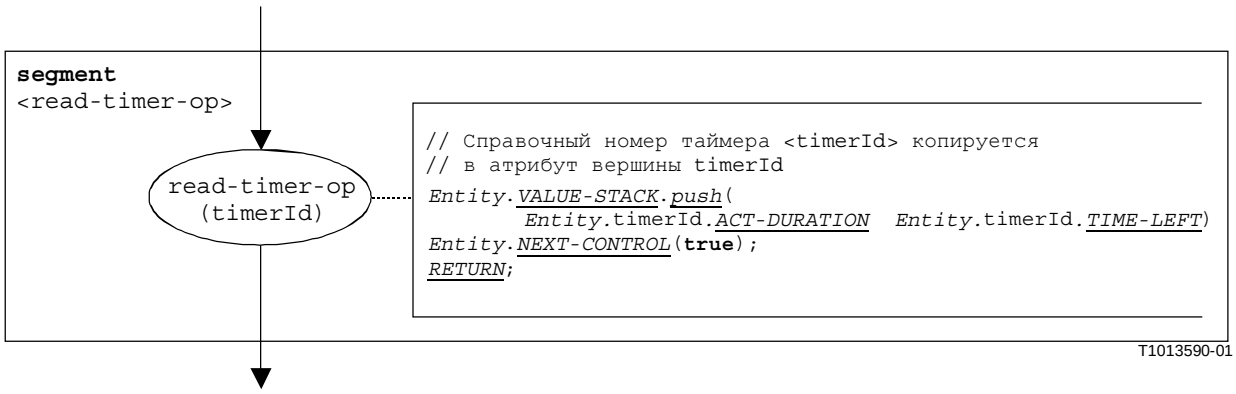


Рисунок B.91/Z.140 – Сегмент <read-timer-op> потокового графа

B.3.7.37 Операция Receive

Синтаксической структурой операции **receive** является:

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

Факультативное <component_expression> в разделе **from** указывает передающий объект. Он может быть представлен в виде значения переменной или выданного значения функции, то есть он считается выражением. Факультативная <assignmentPart> означает присвоение принятой информации, если принятое сообщение соответствует спецификации сопоставления <matchingSpec> и разделу **from** (факультативному).

Сегмент <receive-op> потокового графа на рис. B.92 определяет выполнение операции **receive**.

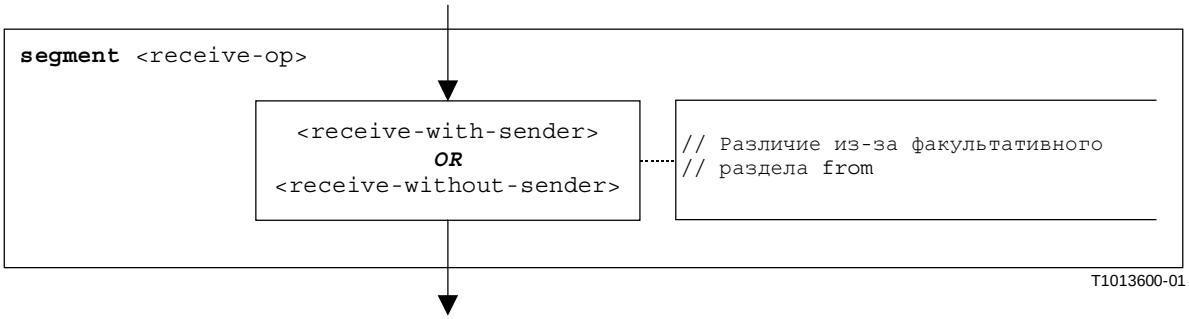


Рисунок B.92/Z.140 – Сегмент <receive-op> потокового графа

B.3.7.37.1 Сегмент <receive-with-sender> потокового графа

Сегмент `<receive-with-sender>` потокового графа на рис. B.93 определяет выполнение операции **`receive`**, при в которой передатчик описывается в виде выражения.

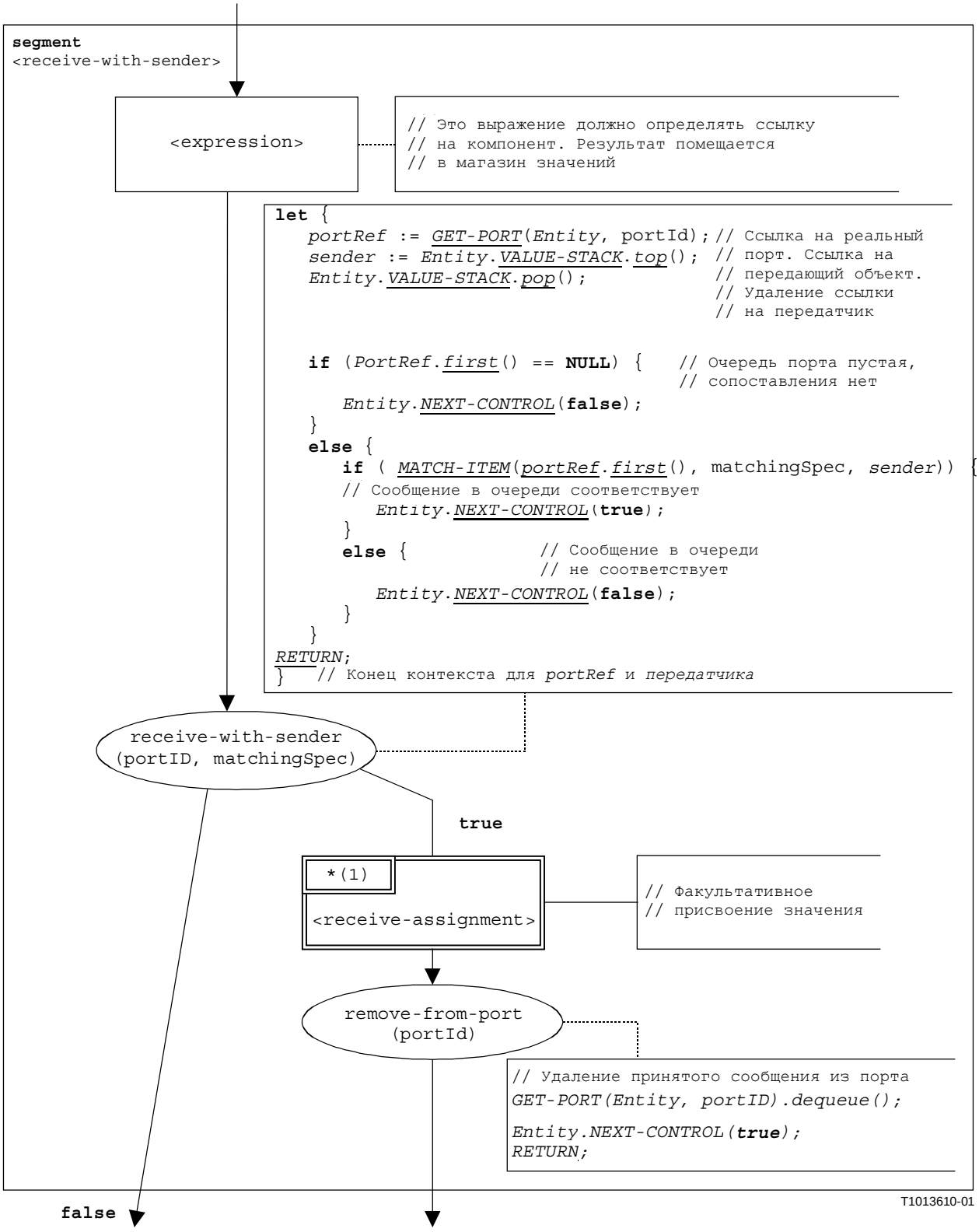


Рисунок B.93/Z.140 – Сегмент <receive-with-sender> потокового графа

B.3.7.37.2 Сегмент <receive-without-sender> потокового графа

Сегмент `<receive-without-sender>` потокового графа на рис. B.94 определяет выполнение операции **`receive`** без раздела **`from`**.

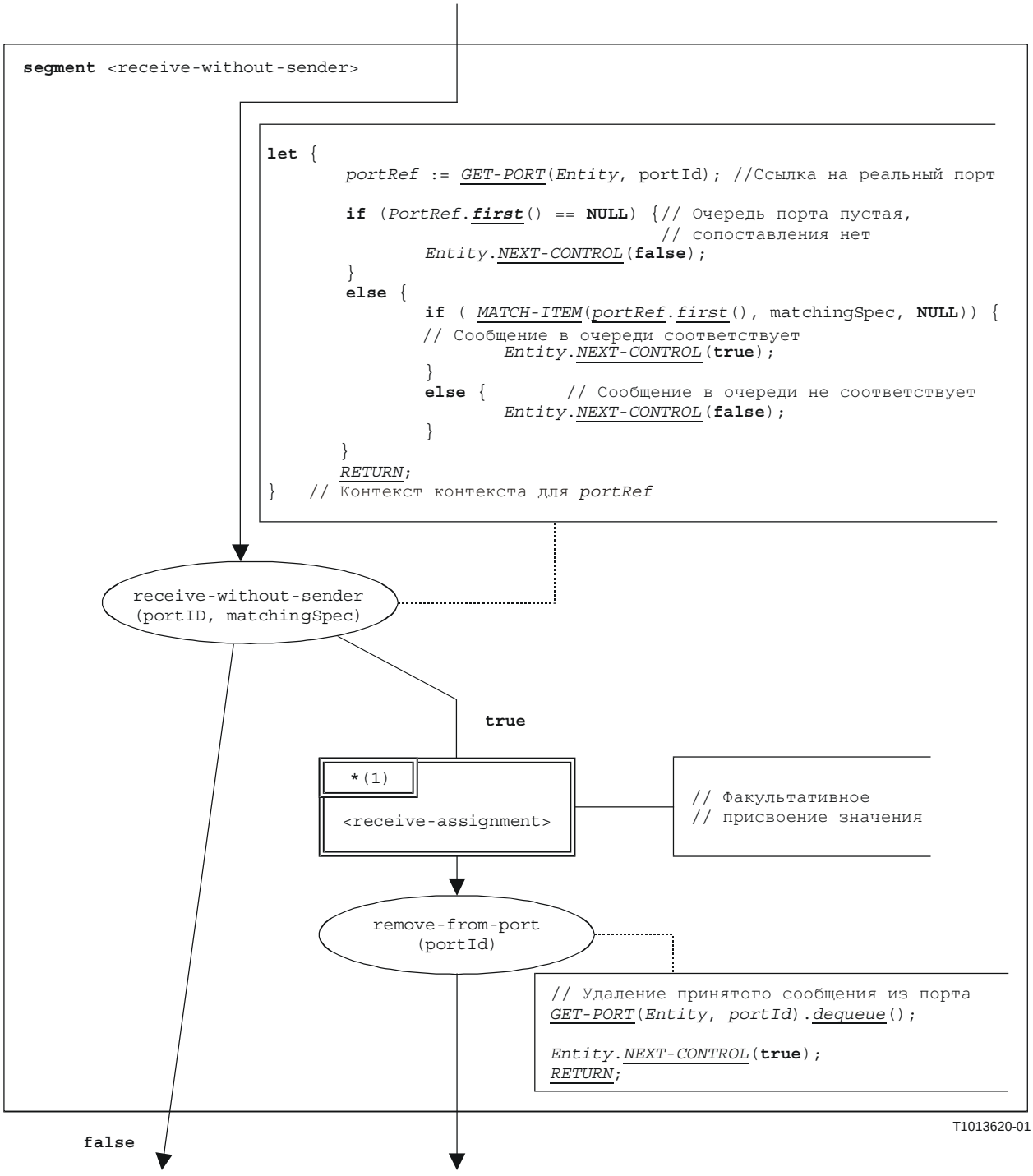


Рисунок B.94/Z.140 – Сегмент <receive-without-sender> потокового графа

B.3.7.37.3 Сегмент <receive-assignment> потокового графа

Сегмент <receive-assignment> потокового графа на рис. B.95 определяет поиск информации в принятом сообщении и ее присвоение переменным.

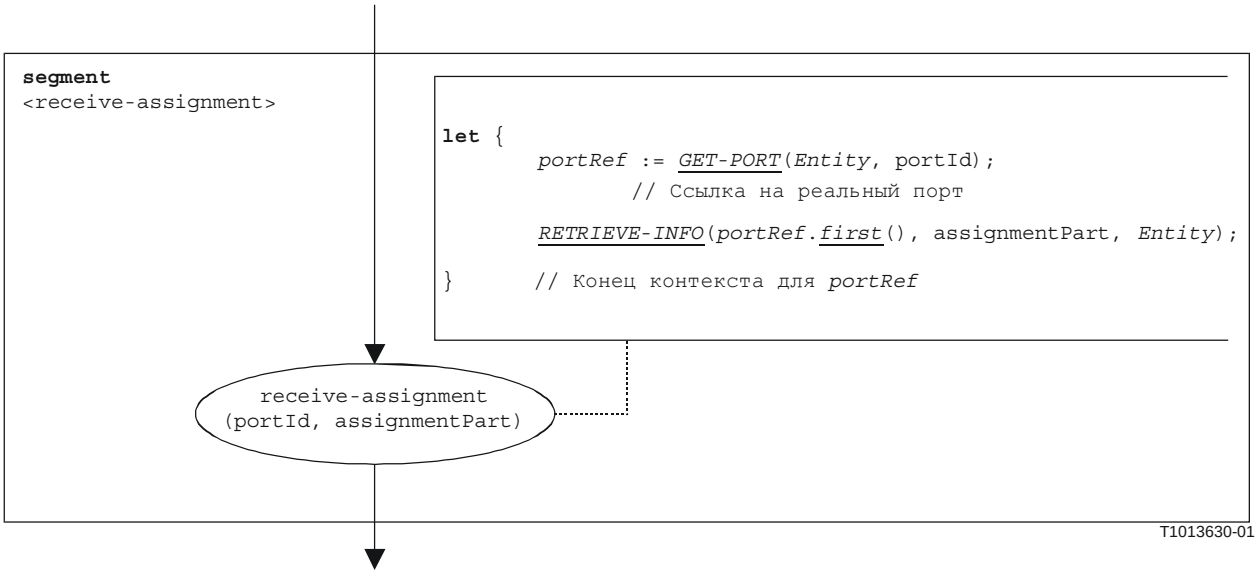


Рисунок B.95/Z.140 – Сегмент <receive-assignment> потокового графа

B.3.7.38 Операция Reply

Синтаксической структурой операции **reply** является:

<portId>.**reply** (<replySpec>) [**to** <component_expression>]

Факультативное <component_expression> в разделе **to** указывает принимающий объект. Он может быть представлен в виде значения переменной или выданного значения функции.

Сегмент <reply-op> потокового графа на рис. B.96 определяет выполнение операции **reply**.

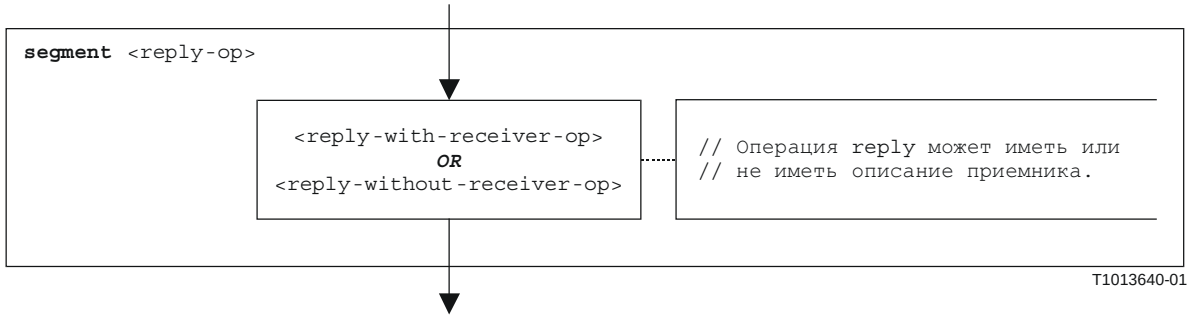


Рисунок B.96/Z.140 – Сегмент <reply-op> потокового графа

B.3.7.38.1 Сегмент <reply-with-receiver-op> потокового графа

Сегмент `<reply-with-receiver-op>` потокового графа на рис. B.97 определяет выполнение операции **`reply`**, при которой приемник описывается в виде выражения.

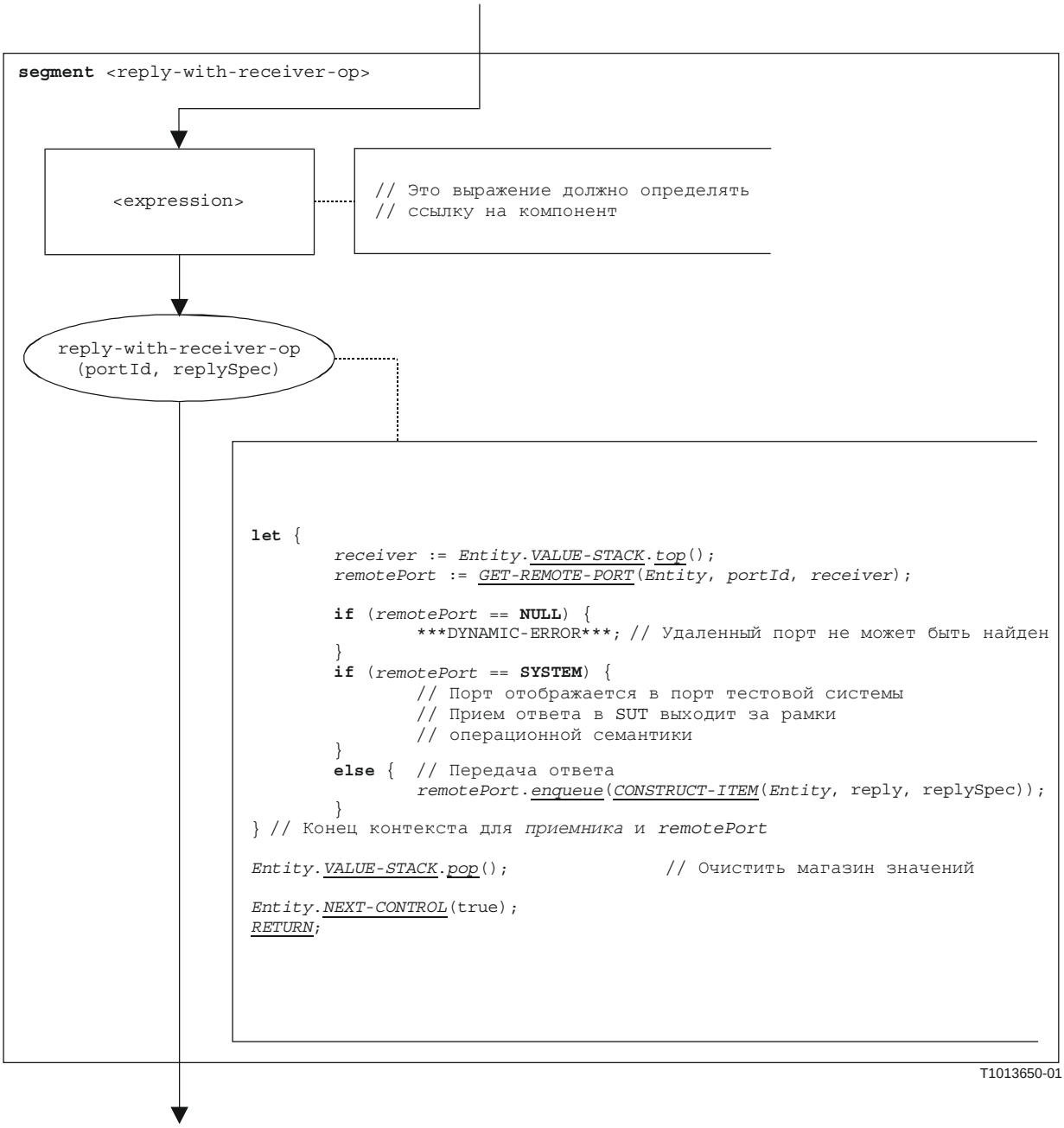


Рисунок B.97/Z.140 – Сегмент <reply-with-receiver-op> потокового графа

B.3.7.38.2 Сегмент <reply-without-receiver-op> потокового графа

Сегмент <reply-without-receiver-op> потокового графа на рис. B.98 определяет выполнение операции **reply** без раздела **to**.

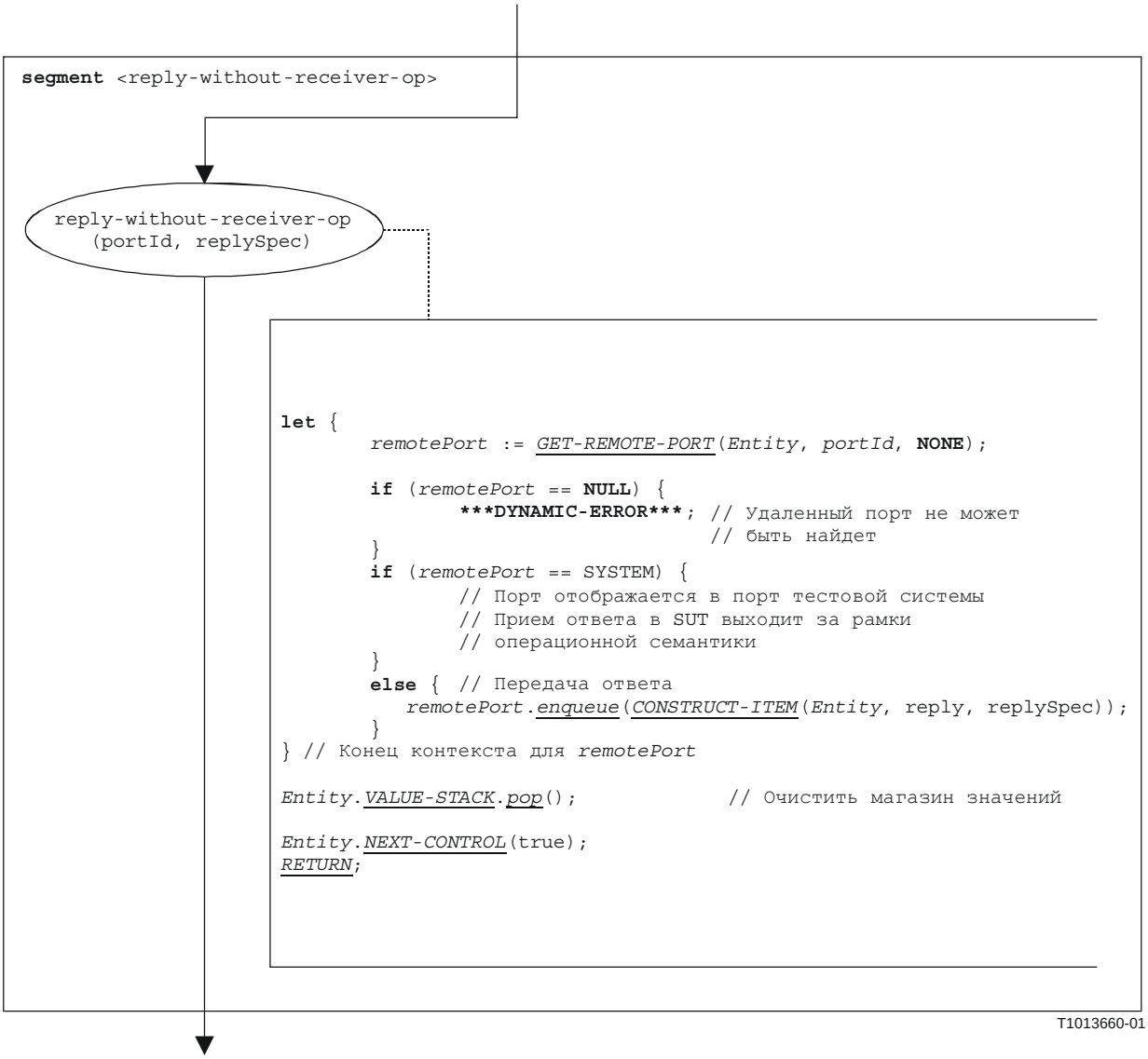


Рисунок B.98/Z.140 – Сегмент <reply-without-receiver-op> потокового графа

B.3.7.39 Команда Return

Синтаксической структурой команды **return** является:

return [<expression>]

Факультативное <expression> описывает возможное выдаваемое значение функции. Выполнение команды **return** означает, что управление выходит из реальной единицы контекста, то есть переменные и таймеры, известные только в этом контексте, должны быть удалены, а магазин значений должен быть обновлен. Результатом команды **return** является операция **stop**, если она является последней командой в описании поведения.

ПРИМЕЧАНИЕ. – Из-за применения кратких нотаций тестовые примеры и управление модулем всегда будут оканчиваться операцией **stop**. Только другие тестовые компоненты могут оканчиваться командой **return**.

Сегмент <return-stmt> потокового графа на рис. B.99 определяет выполнение команды **return**.

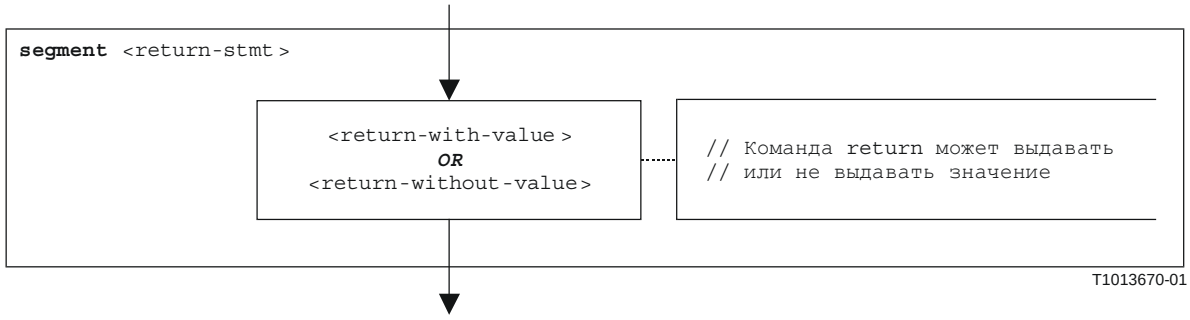


Рисунок B.99/Z.140 – Сегмент <return-stmt> потокового графа

B.3.7.39.1 Сегмент <return-with-value> потокового графа

Сегмент `<return-with-value>` потокового графа на рис. B.100 определяет выполнение команды **`return`**, при которой выдается значение, описанное в виде выражения.

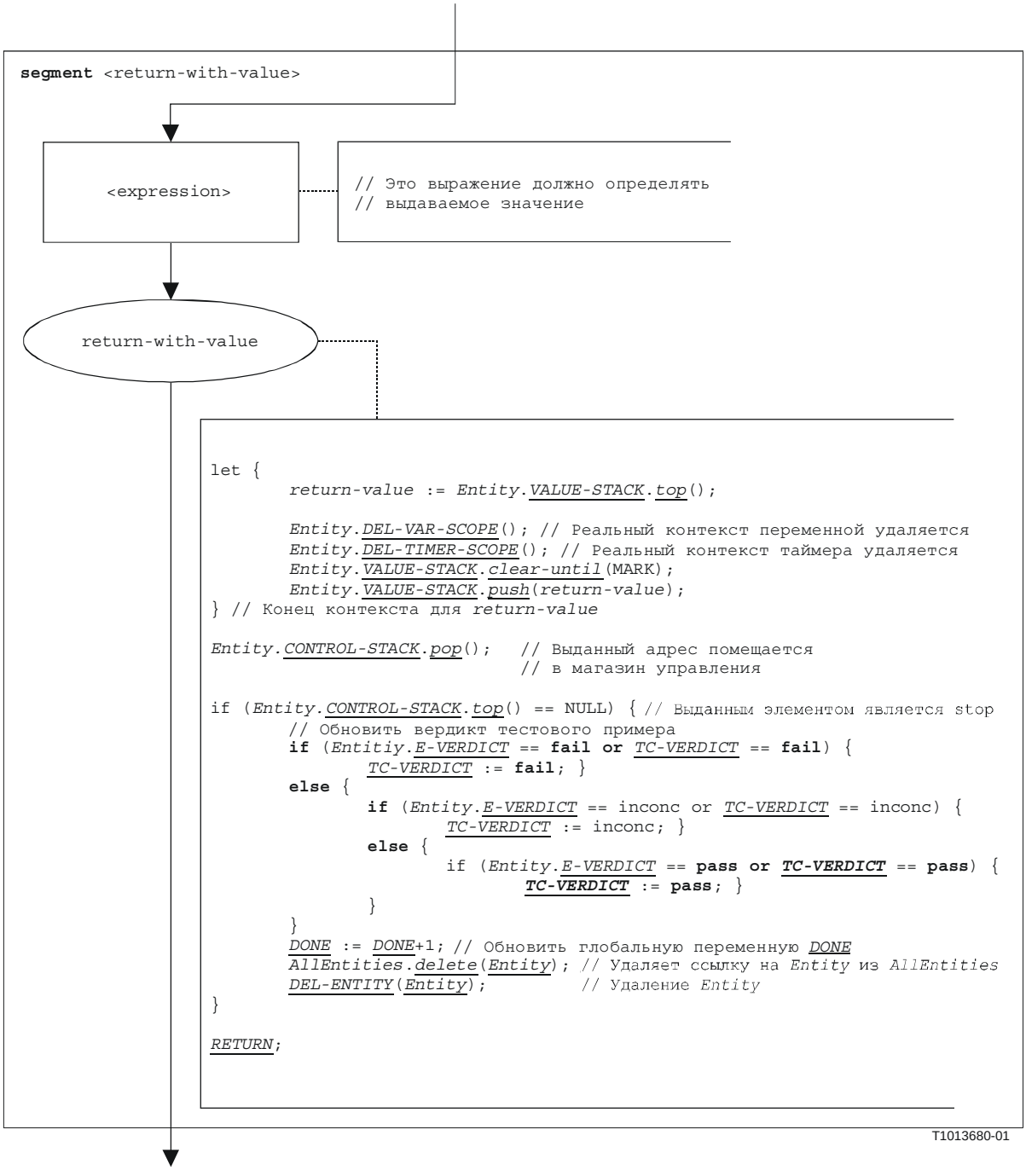


Рисунок B.100/Z.140 – Сегмент <return-with-value> потокового графа

B.3.7.39.2 Сегмент <return-without-value> потокового графа

Сегмент <return-without-value> потокового графа на рис. B.101 определяет выполнение команды **return**, при которой не выдается значение.

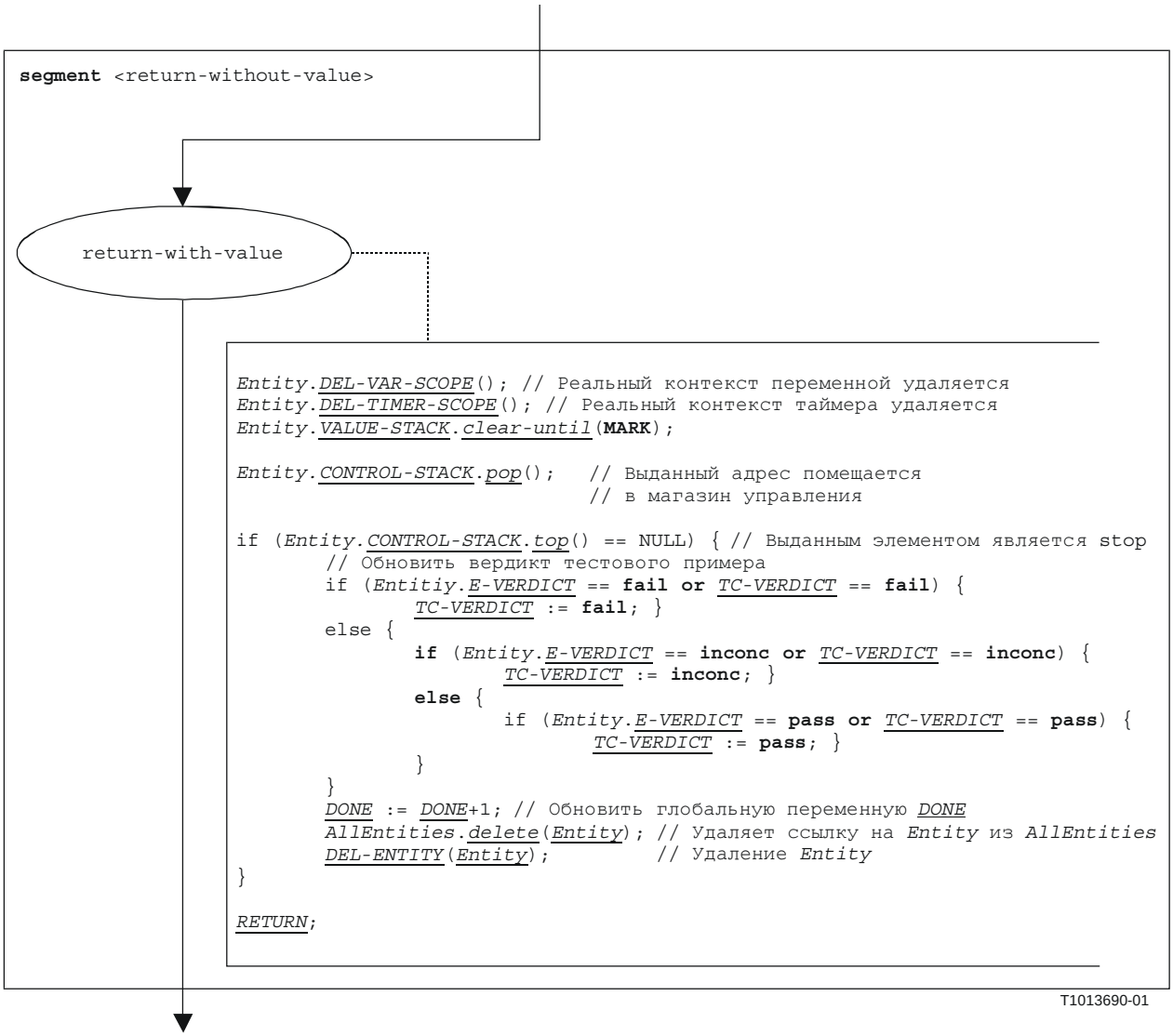


Рисунок B.101/Z.140 – Сегмент <return-without-value> потокового графа

B.3.7.40 Операция Running-all-components

Операция **running-all-components** указывает на использование ключевых слов **all components** в компонентной операции **running** (раздел 42). Операция **running-all-components** может вызываться только из **mtc**. Она позволяет проверить, все ли параллельные тестовые компоненты тестового примера работают. Синтаксической структурой операции **running-all-components** является:

all component.running

Выполнение операции **running-all-components** определяется сегментом <running-all-comp-op> потокового графа, показанным на рис. B.102.

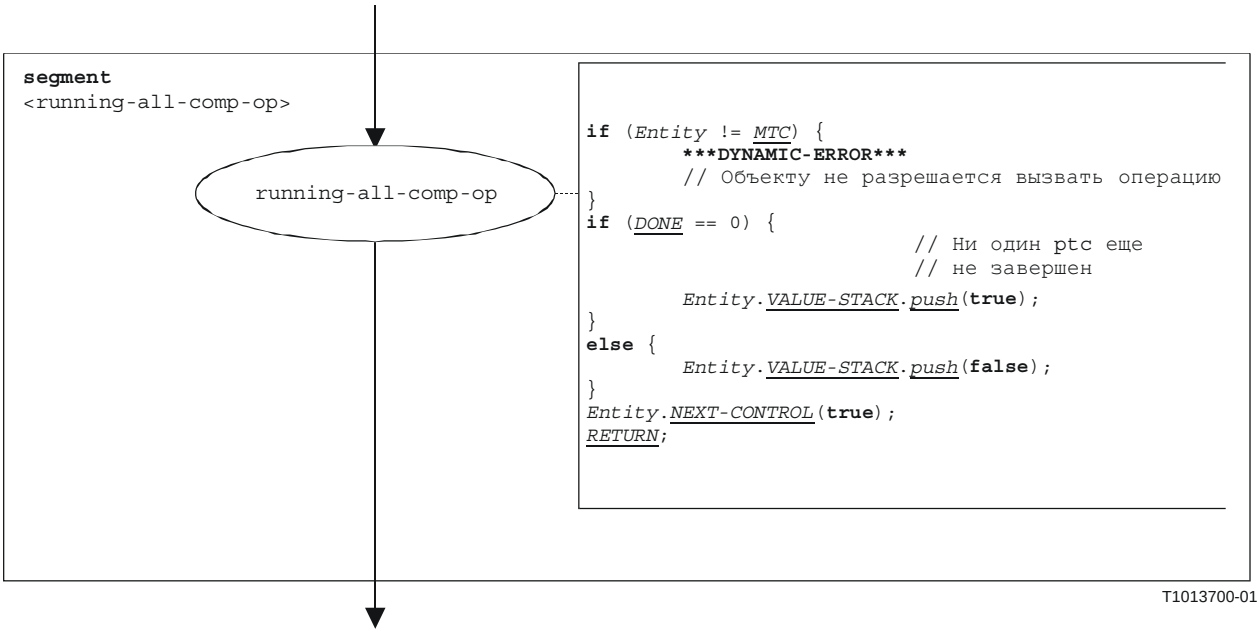


Рисунок B.102/Z.140 – Сегмент <running-all-comp-op> потокового графа

B.3.7.41 Операция Running-any-component

Операция **running-any-component** указывает использование ключевых слов **any component** в компонентной операции **running** (раздел 42). Операция **running-any-component** может вызываться только из **mtc**. Она позволяет проверить, работает ли по крайней мере один из параллельных тестовых компонентов тестового примера. Синтаксической структурой операции **running-any-component** является:

any component.running

Выполнение операции **running-any-component** определяется сегментом <running-any-comp-op> потокового графа, показанного на рис. B.103.

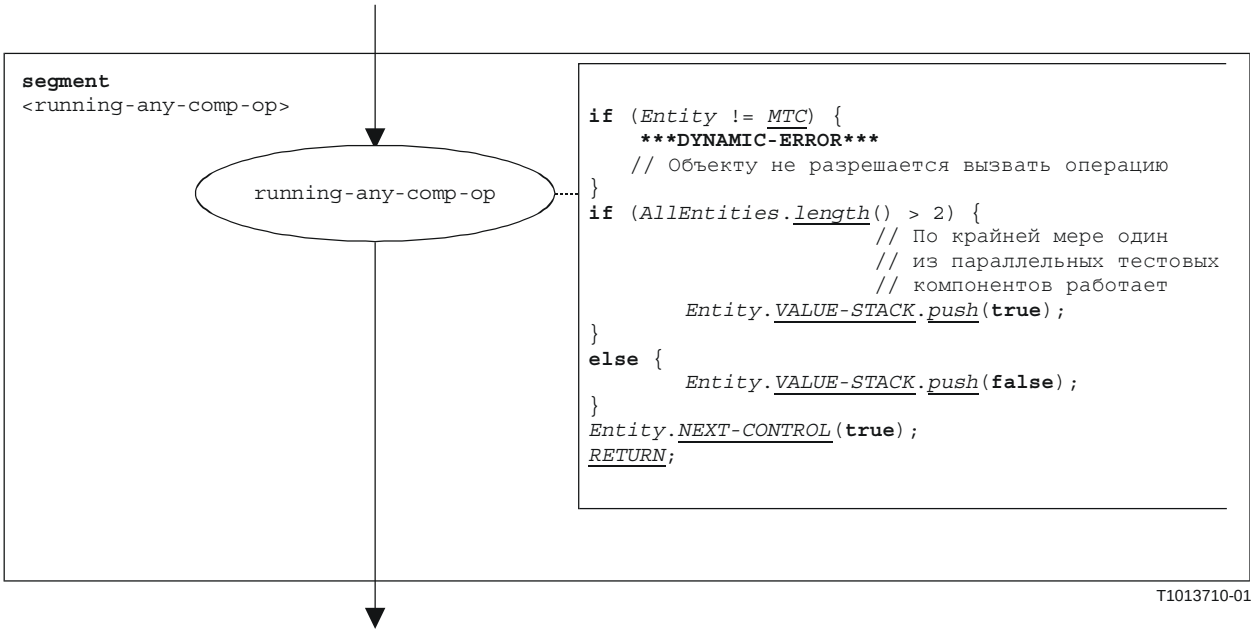


Рисунок B.103/Z.140 – Сегмент <running-any-comp-op> потокового графа

B.3.7.42 Компонентная операция Running

Синтаксической структурой компонентной операции **running** является:

<component_expression>.**running**

Компонентная операция **running** проверяет, работает ли компонент или он остановлен. С помощью ссылки на компонент указывается компонент, который будет проверяться. Ссылка может быть записана в переменной или выдана функцией. Для простоты она считается выражением, которое определяет ссылку на компонент.

Сегмент <running-component-op> потокового графа на рис. B.104 определяет выполнение компонентной операции **running**.

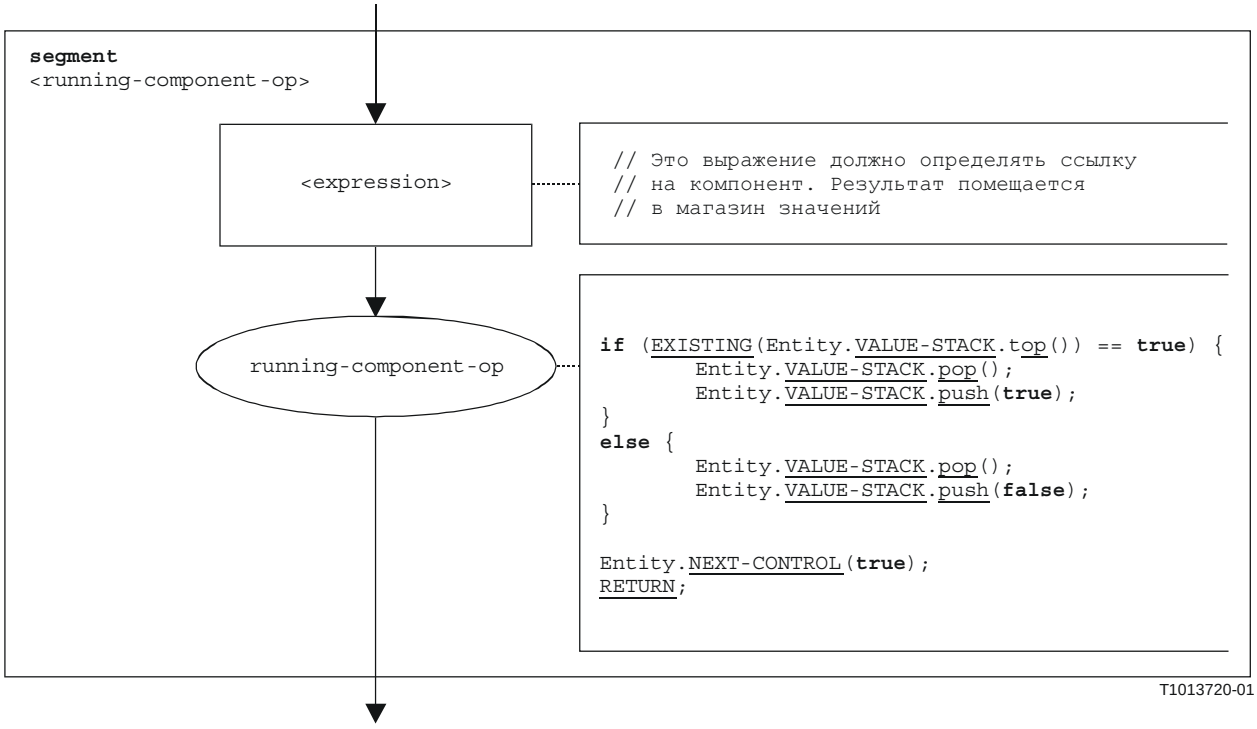


Рисунок B.104/Z.140 – Сегмент <running-component-op> потокового графа

B.3.7.43 Таймерная операция Running

Синтаксической структурой таймерной операции **running** является:

<timerId>.**running**

Сегмент <running-timer-op> потокового графа на рис. B.105 определяет выполнение таймерной операции **running**.

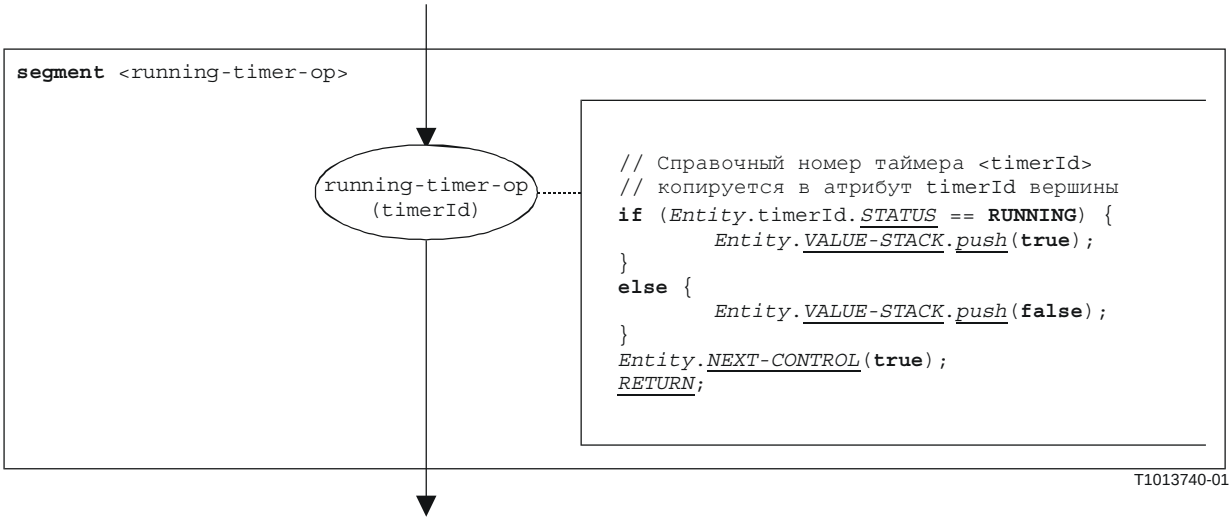


Рисунок B.105/Z.140 – Сегмент <running-timer-op> потокового графа

B.3.7.44 Операция Send

Синтаксической структурой операции **send** является:

<portId>.**send** (<send-spec>) [**to** <component_expression>]

Факультативное <component_expression> в разделе **to** указывает принимающий объект. Он может обеспечиваться в виде значения переменной или выданного значения функции.

Сегмент <send-op> потокового графа на рис. B.106 определяет выполнение операции **send**.

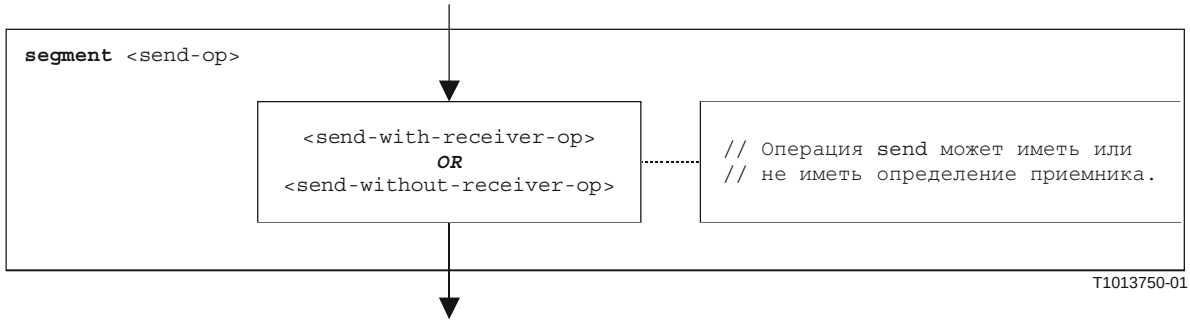


Рисунок B.106/Z.140 – Сегмент <send-op> потокового графа

B.3.7.44.1 Сегмент <send-with-receiver-op> потокового графа

Сегмент `<send-with-receiver-op>` потокового графа на рис. B.107 определяет выполнение операции **`send`**, при которой приемник указывается в виде выражения.

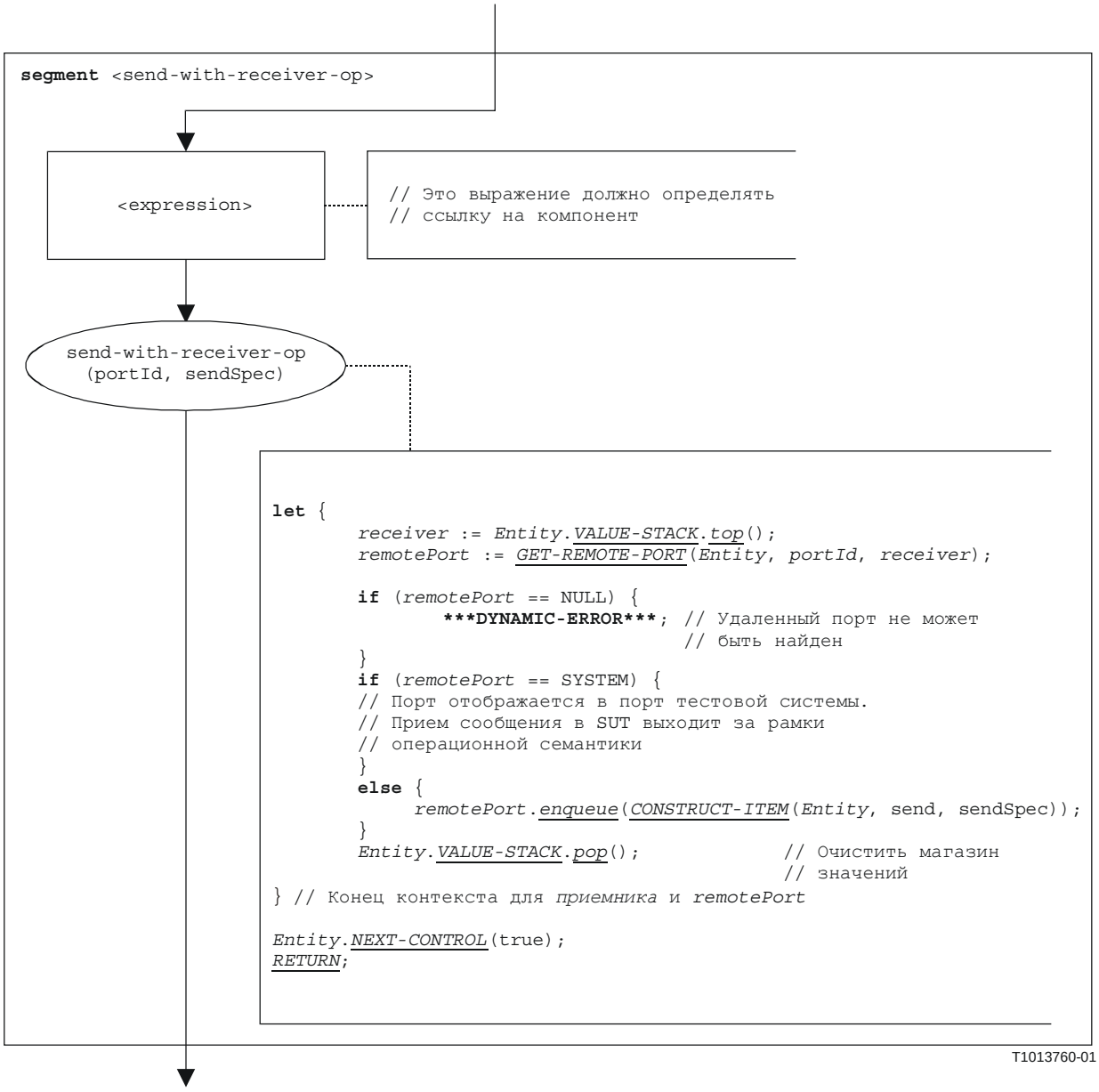


Рисунок B.107/Z.140 – Сегмент <send-with-receiver-op> потокового графа

B.3.7.44.2 Сегмент <send-without-receiver-op> потокового графа

Сегмент <send-without-receiver-op> потокового графа на рис. B.108 определяет выполнение операции **send** без раздела **to**.

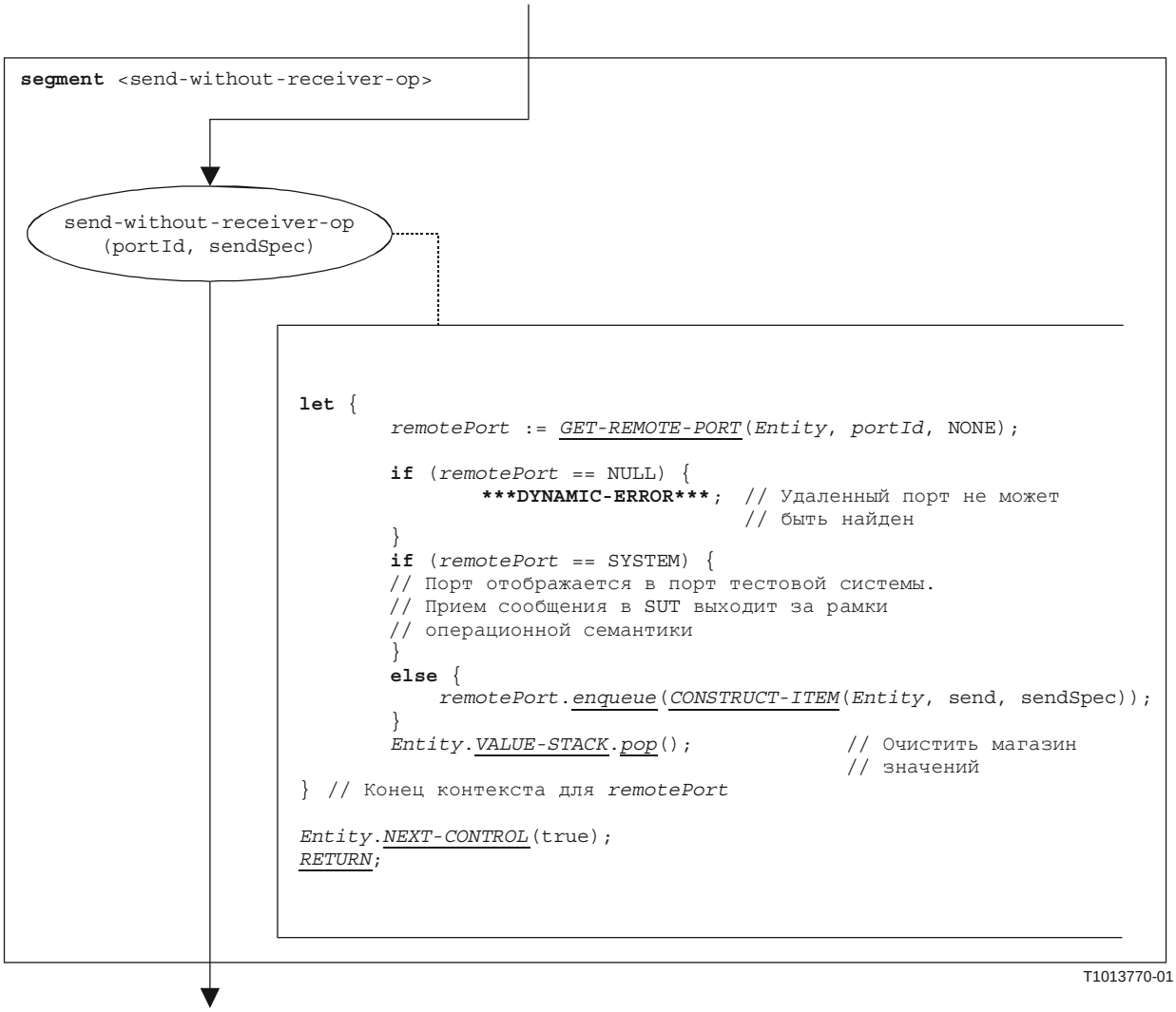


Рисунок B.108/Z.140 – Сегмент <send-without-receiver-op> потокового графа

B.3.7.45 Операция Self

Синтаксической структурой операции **self** является:

self

Сегмент <self-op> потокового графа на рис. B.109 определяет выполнение операции **self**.

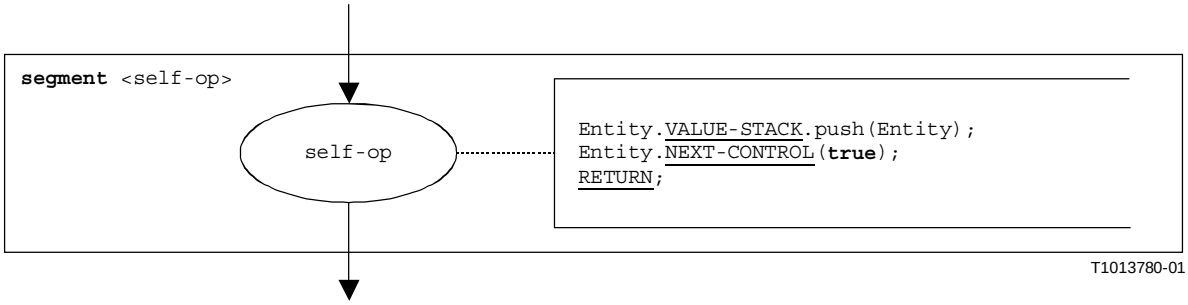


Рисунок B.109/Z.140 – Сегмент <self-op> потокового графа

B.3.7.46 Компонентная операция Start

Синтаксической структурой компонентной операции **start** является:

<component_expression>.**start** (<function-name> (<act-par-desc$_1$>, ... ,
<act-par-descn>))

Компонентная операция **start** запускает только что созданный компонент. Использование ссылки на компонент определяет компонент, который необходимо запустить. Эта ссылка может быть записана в переменной или выдана функцией. Для простоты она считается выражением, которое определяет ссылку на компонент.

Часть <function-name> означает имя функции, которая определяет поведение нового компонента, а <act-par-descr$_1$>, ... , <act-par-descr$_n$> дают описание значений реальных параметров для <function-name>. В случае параметра значения описание реального параметра может обеспечиваться в виде выражения, которое должно быть определено до выполнения вызова. Обработка формальных и реальных параметров аналогична их обработке в вызовах функции (см. п. B.3.7.22).

Сегмент <start-component-op> потокового графа на рис. B.110 определяет выполнение компонентной операции **start**. Операция запуска компонента выполняется в четыре шага. Первый шаг – это создание записи вызова. На втором шаге вычисляются значения реальных параметров. На третьем шаге запрашивается ссылка на компонент, который необходимо запустить, а на четвертом шаге управление и запись вызова переносятся к новому компоненту.

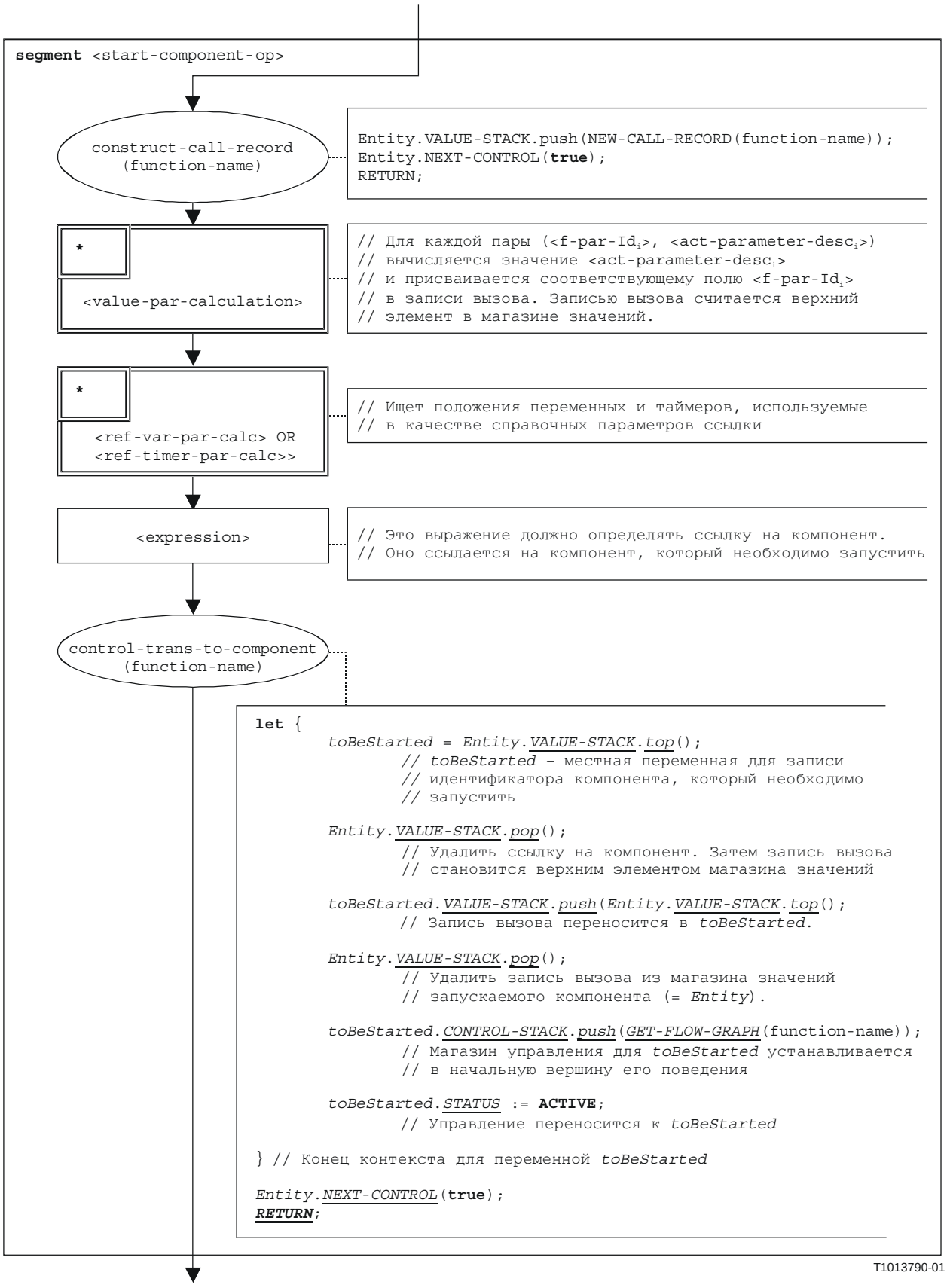


Рисунок B.110/Z.140 – Сегмент <start-component-op> потокового графа

B.3.7.47 Портовая операция Start

Синтаксической структурой портовой операции **start** является:

<portId>.**start**

Сегмент <start-port-op> потокового графа на рис. B.111 определяет выполнение портовой операции **start**.

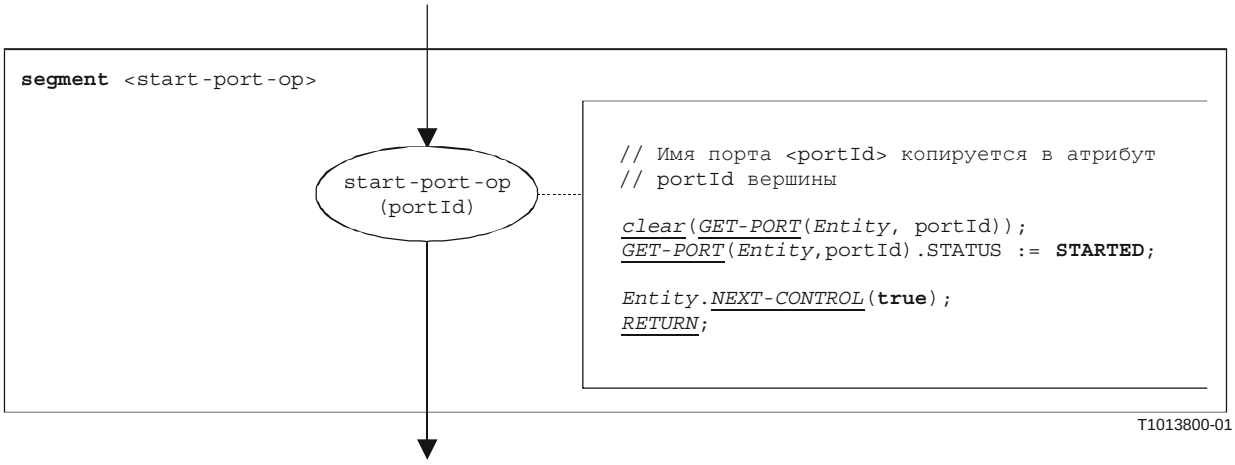


Рисунок B.111/Z.140 – Сегмент <start-port-op> потокового графа

B.3.7.48 Таймерная операция Start

Синтаксической структурой таймерной операции start является:

<timerId>.**start** [(<float_expression>)]

Факультативный параметр <float_expression> таймерной операции **start** означает факультативную выдержку, с которой должен запускаться таймер. Он является выражением, которое определяет значение типа **float**. Если он обеспечивается, то это выражение должно быть определено до применения операции **start**. Результат определения помещается в магазин значений для объекта.

Сегмент <start-timer-op> потокового графа на рис. B.112 определяет выполнение таймерной операции **start**.

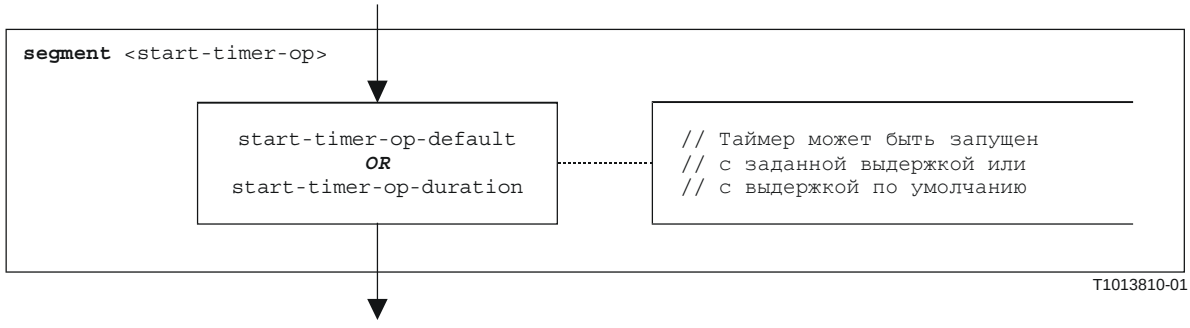


Рисунок B.112/Z.140 – Сегмент <start-timer-op> потокового графа

B.3.7.48.1 Сегмент <start-timer-op-default> потокового графа

Сегмент `<start-timer-op-default>` потокового графа на рис. B.113 определяет выполнение таймерной операции **`start`** со значением по умолчанию.

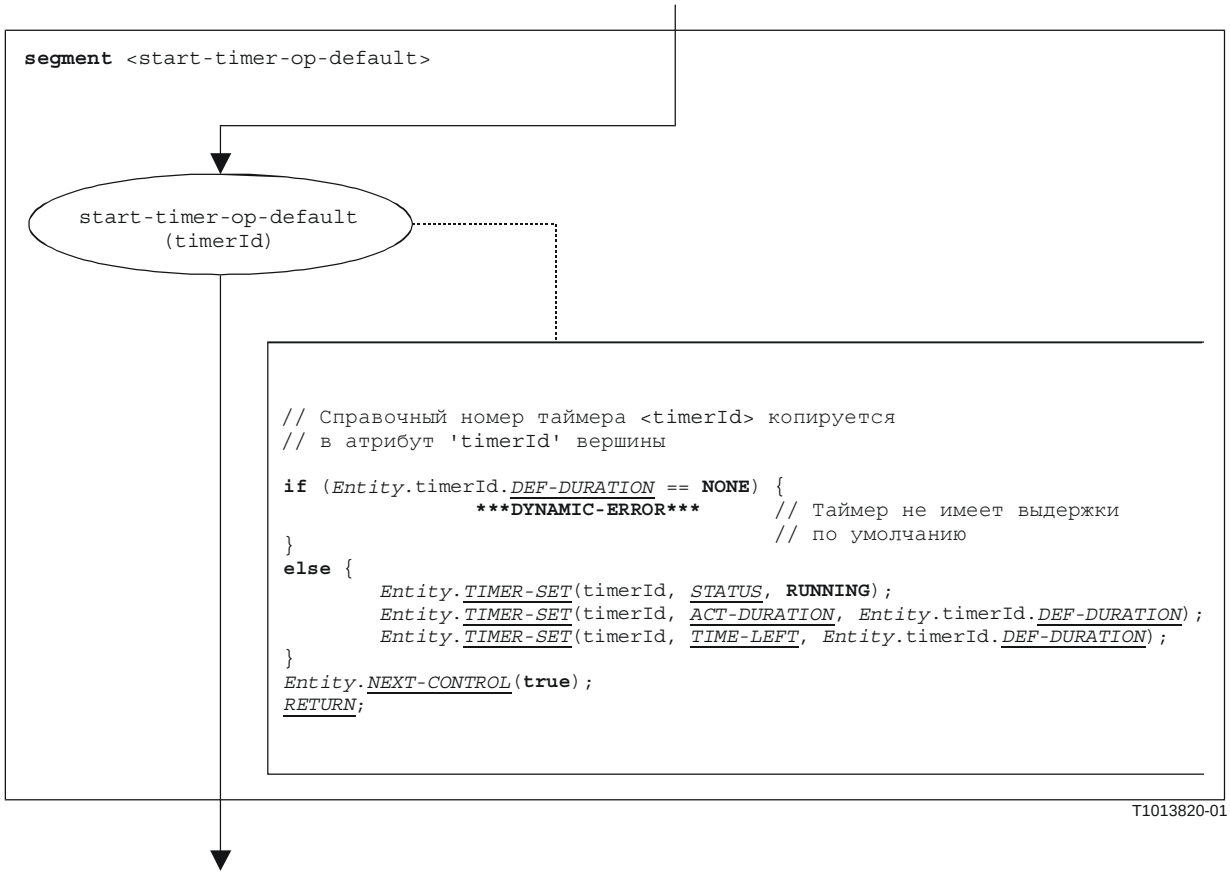


Рисунок B.113/Z.140 – Сегмент <start-timer-op-default> потокового графа

B.3.7.48.2 Сегмент <start-timer-op-duration> потокового графа

Сегмент <start-timer-op-duration> потокового графа на рис. B.114 определяет выполнение таймерной операции **start** с указанной выдержкой.

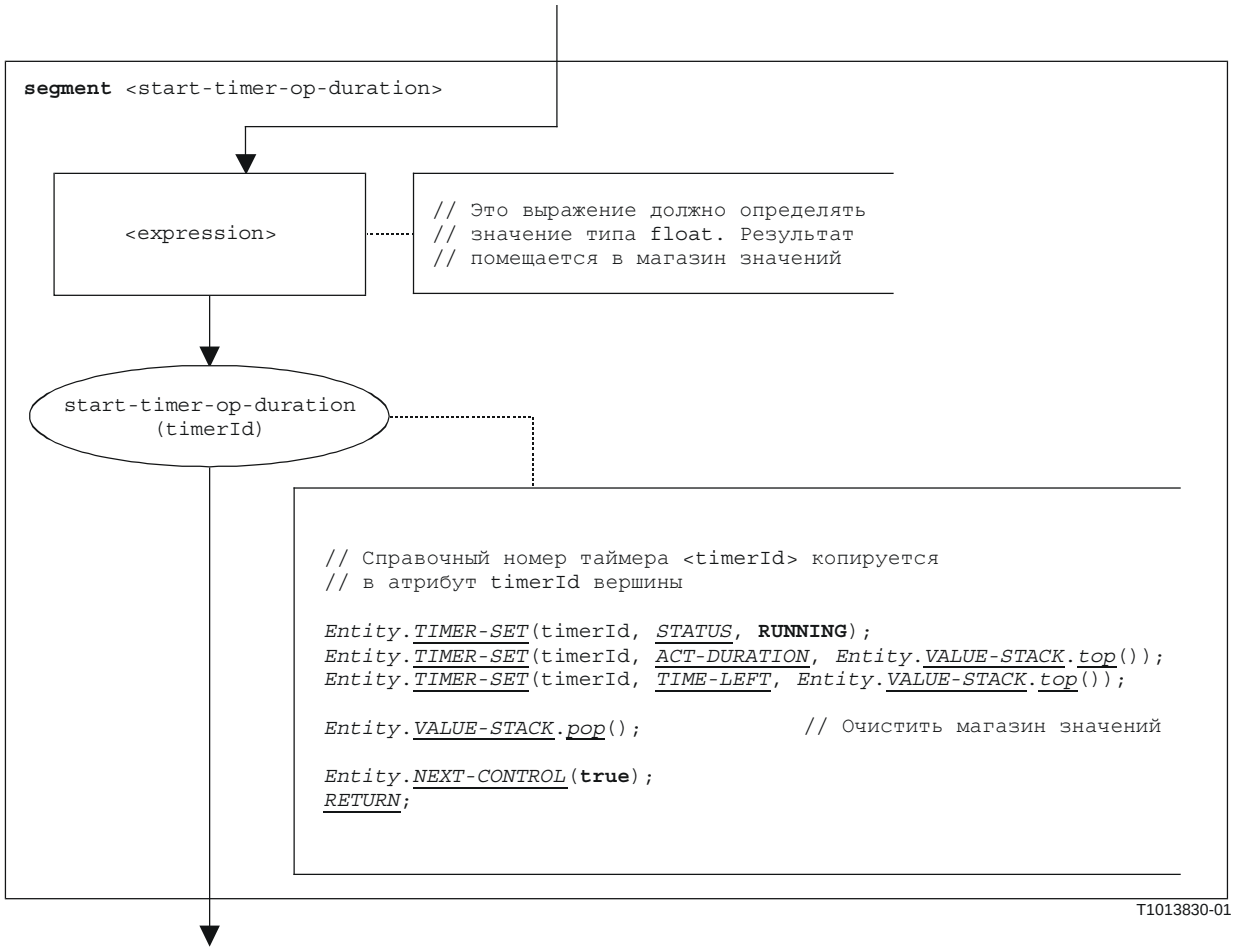


Рисунок B.114/Z.140 – Сегмент <start-timer-op-duration> потокового графа

B.3.7.49 Блок команд

Синтаксической структурой блока команд является:

{<statement$_1$>; . . .; <statement$_n$>}

Блок команд является единицей контекста. При вхождении в единицу контекста должны инициализироваться новые контексты для переменных, таймеров и магазина значений. При выходе из единицы контекста уничтожаются все переменные, таймеры и значения в магазине этого контекста.

Сегмент <statement-block> потокового графа на рис. B.115 определяет выполнение блока команд.

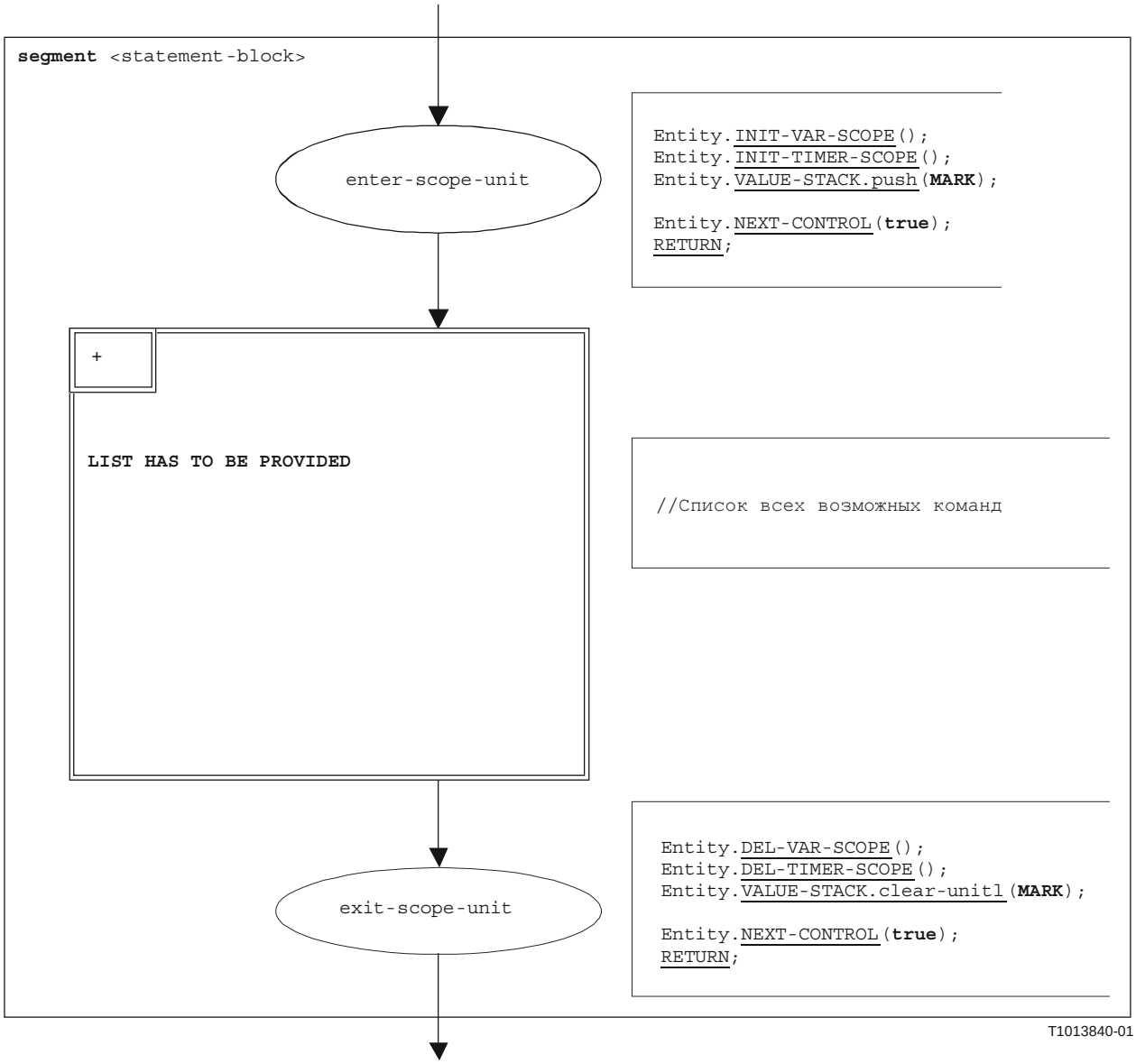


Рисунок B.115/Z.140 – Сегмент <statement-block> потокового графа

B.3.7.50 Операция Stop

Синтаксической структурой объектовой операции **stop** является:

stop

Результат операции **stop** зависит от объекта, который выполняет операцию **stop**:

a) Если **stop** выполняется управлением модуля, то тестовая кампания оканчивается, то есть все тестовые компоненты и управление модулем исчезают из состояния модуля.

b) Если операцию **stop** выполняет **mtc**, то все параллельные тестовые компоненты и **mtc** останавливают выполнение. Глобальный вердикт тестового примера обновляется и помещается в магазин значений из управления модулем. Наконец, управление переносится обратно к управлению модулем, а **mtc** оканчивается.

c) Если операция **stop** выполняется тестовым компонентом, то обновляется глобальный вердикт тестового примера *TC-VERDICT* и глобальная переменная *DONE*. Затем этот компонент полностью исчезает из модуля.

Сегмент <stop-entity-op> потокового графа на рис. B.116 определяет выполнение операции **stop** объекта.

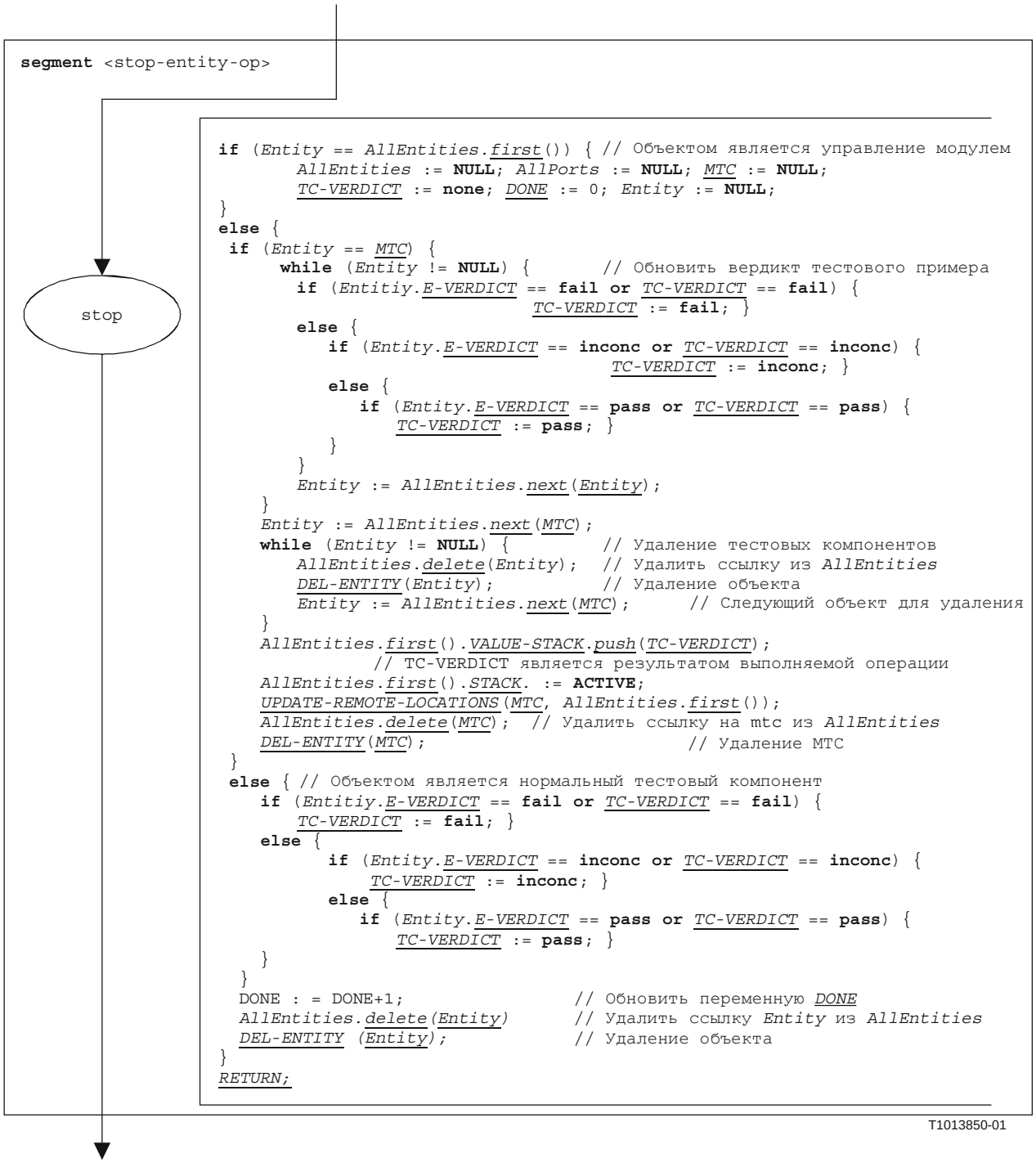


Рисунок B.116/Z.140 – Сегмент <stop-entity-op> потокового графа

B.3.7.51 Портовая операция stop

Синтаксической структурой портовой операции **stop** является:

<portId>.**stop**

Сегмент <stop-port-op> потокового графа на рис. B.117 определяет выполнение портовой операции **stop**.

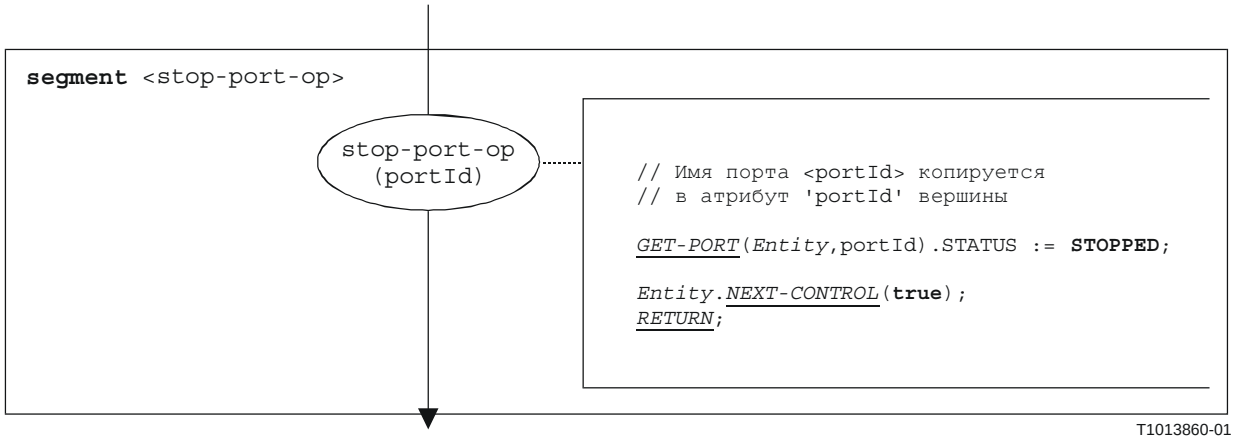


Рисунок B.117/Z.140 – Сегмент <stop-port-op> потокового графа

B.3.7.52 Таймерная операция stop

Синтаксической структурой таймерной операции **stop** является:

<timerId>.**stop**

Сегмент <stop-timer-op> потокового графа на рис. B.118 определяет выполнение таймерной операции **stop**.

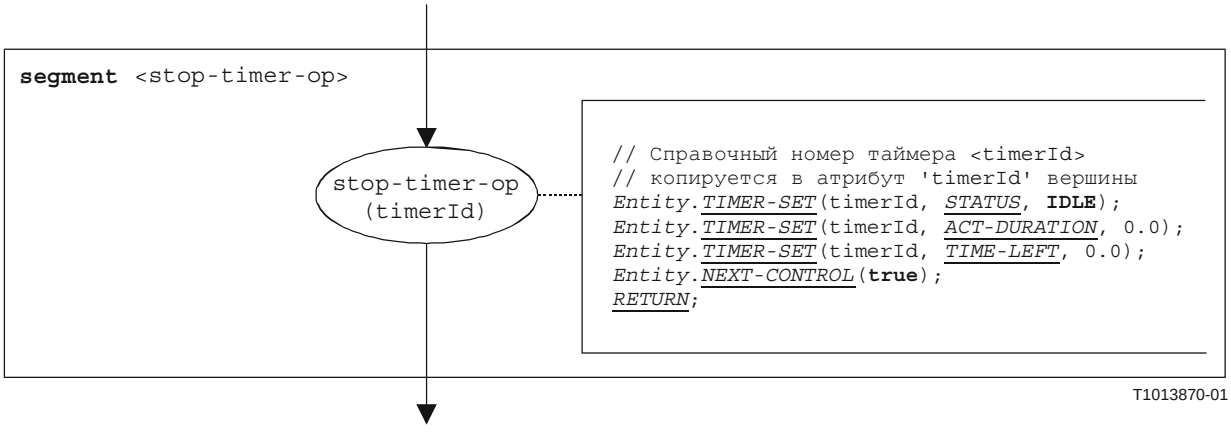


Рисунок B.118/Z.140 – Сегмент <stop-timer-op> потокового графа

B.3.7.53 Операция Sut.action

Синтаксической структурой операции **sut.action** является:

sut.action (<informal description>)

Сегмент <sut.action-op> потокового графа на рис. B.119 определяет выполнение операции **sut.action**.

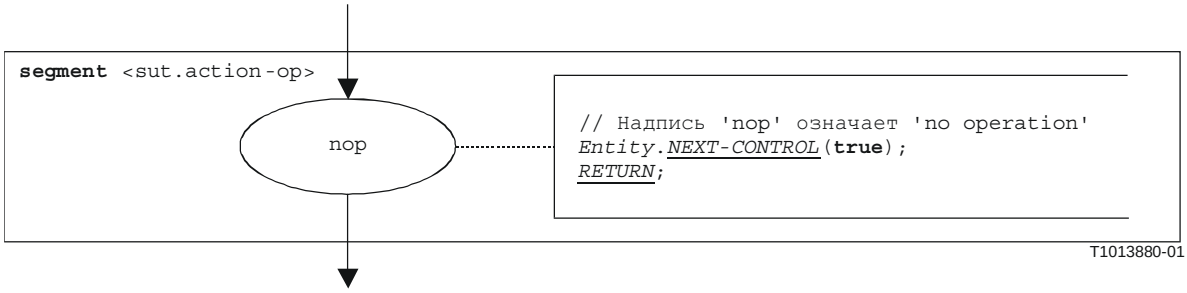


Рисунок B.119/Z.140 – Сегмент <sut.action-op> потокового графа

ПРИМЕЧАНИЕ. – Параметр <informal description> операции **sut.action** не имеет значения для операционной семантики и поэтому не представлен в сегменте потокового графа.

B.3.7.54 Операция System

Синтаксической структурой операции **system** является:

system

Сегмент <system-op> потокового графа на рис. B.120 определяет выполнение операции **system**.

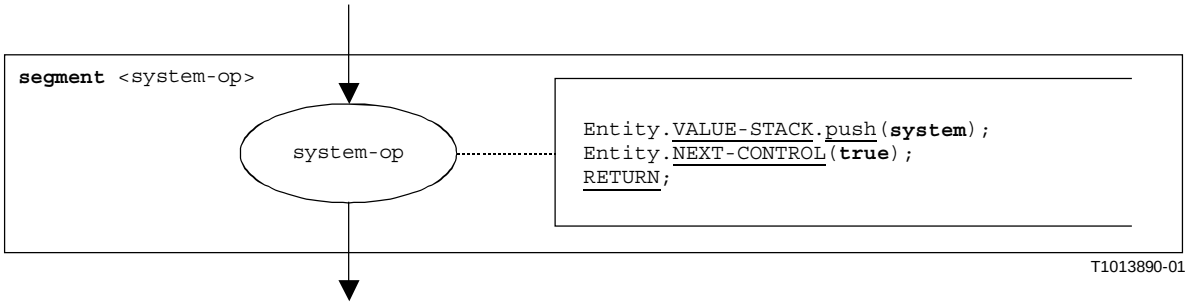


Рисунок B.120/Z.140 – Сегмент <system-op> потокового графа

B.3.7.55 Таймерная операция Timeout

Синтаксической структурой таймерной операции **timeout** является:

<timerId>.**timeout**

Сегмент <timeout-timer-op> потокового графа на рис. B.121 определяет выполнение таймерной операции **timeout**.

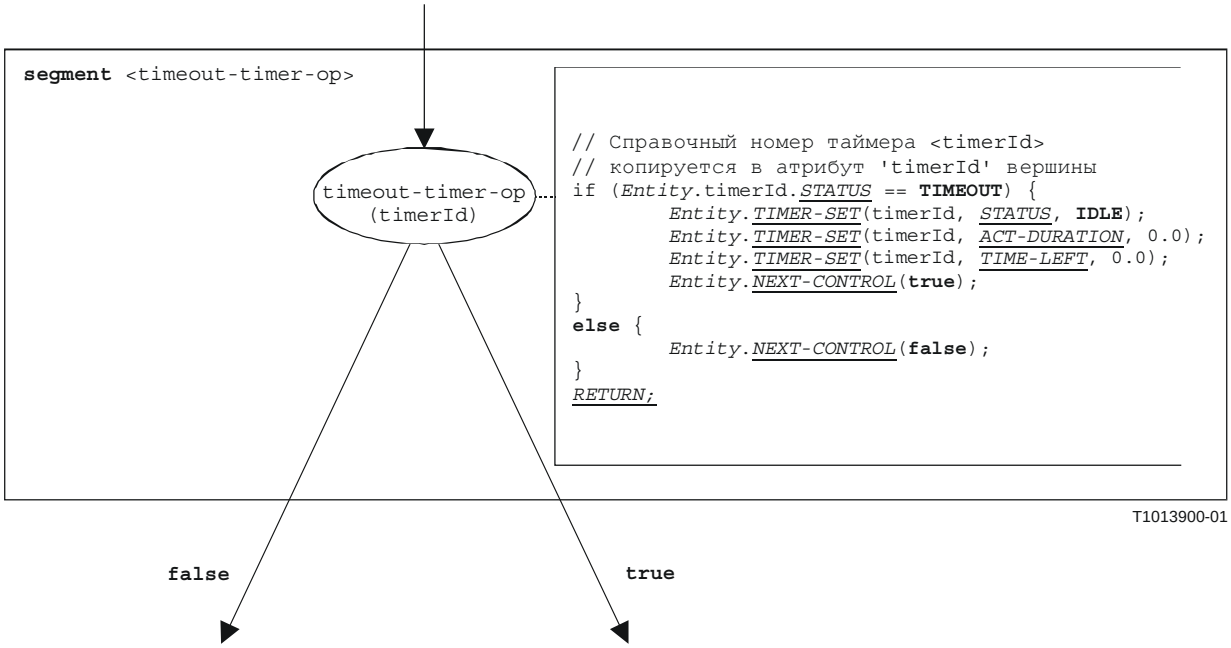


Рисунок B.121/Z.140 – Сегмент <timeout-timer-op> потокового графа

ПРИМЕЧАНИЕ. – Операция **timeout** вложена в команду **alt**. В зависимости от оценки **true** или **false**, вынесенной операцией **timeout**, либо продолжается выполнение с командой, следующей за операцией **timeout** (ветвь **true**), либо должна быть проверена последующая альтернатива в команде **alt** (ветвь **false**).

B.3.7.56 Операция Unmap

Синтаксической структурой операции **unmap** является:

unmap (<component_expression>.<portId1>, **system**.<portId2>)

Идентификаторы <portId1> и <portId2> считаются идентификаторами портов соответствующего тестового компонента и интерфейса тестовой системы. Компонент, которому принадлежит <portId1>, указывается с помощью компонентной ссылки <component_expression>. Эта ссылка может быть записана в переменных или выдана функцией. Для простоты она считается выражением, которое определяет ссылку на компонент. Поэтому для хранения ссылки на компонент используется магазин значений.

ПРИМЕЧАНИЕ. – Операция **unmap** не обращает внимания на то, появляется ли команда **system**.<portId> как первый или как второй параметр. Для простоты предполагается, что она всегда является вторым параметром.

Выполнение операции **unmap** определяется сегментом <unmap-op> потокового графа, показанным на рис. B.122.

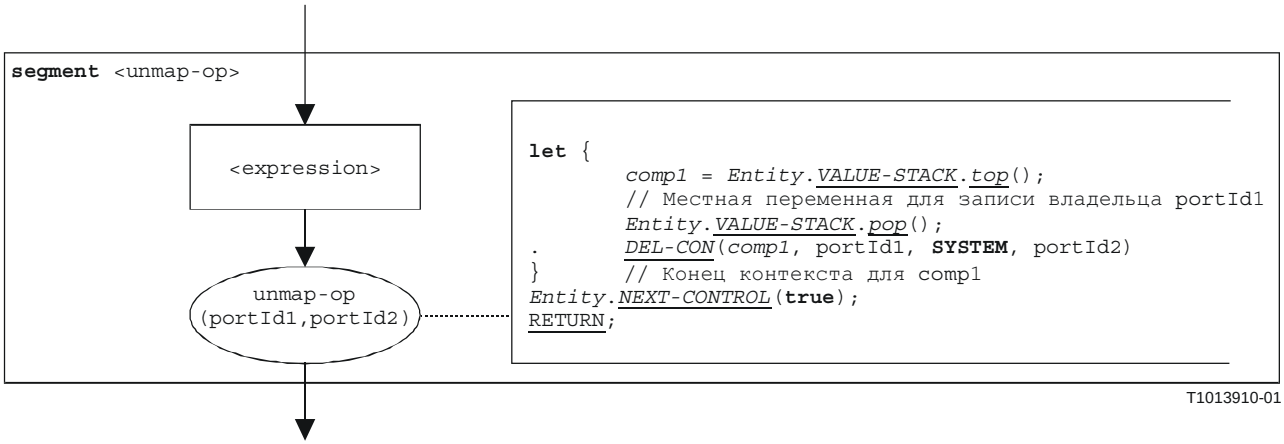


Рисунок B.122/Z.140 – Сегмент <unmap-op> потокового графа

B.3.7.57 Операция Verdict.get

Синтаксической структурой операции **verdict.get** является:

verdict.get

Сегмент <verdict.get-op> потокового графа на рис. B.123 определяет выполнение операции **verdict.get**.

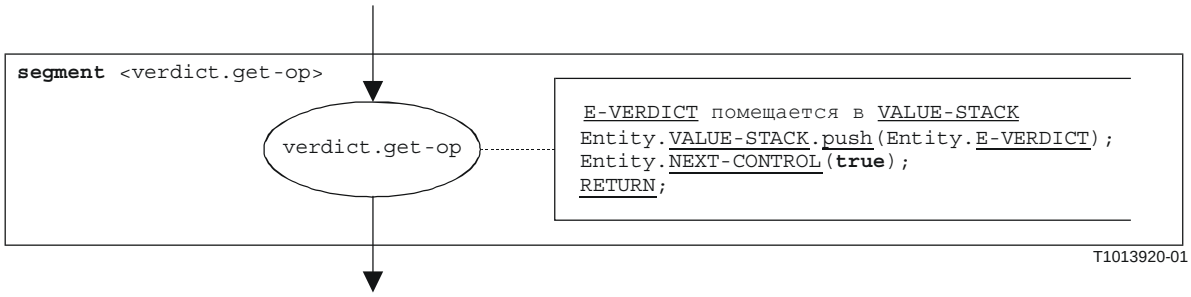


Рисунок B.123/Z.140 – Сегмент <verdict.get-op> потокового графа

B.3.7.58 Операция Verdict.set

Синтаксической структурой операции **verdict.set** является:

verdict.set (<verdicttype_expression>)

ПРИМЕЧАНИЕ. – Параметр <verdicttype_expression> операции **verdict.set** является выражением, которое определяет значение типа **verdicttype**, то есть **none**, **pass**, **income** или **fail**. Это выражение определяется до применения операции **verdict.set**.

Сегмент <verdict.set-op> потокового графа на рис. B.124 определяет выполнение операции **verdict.set**.

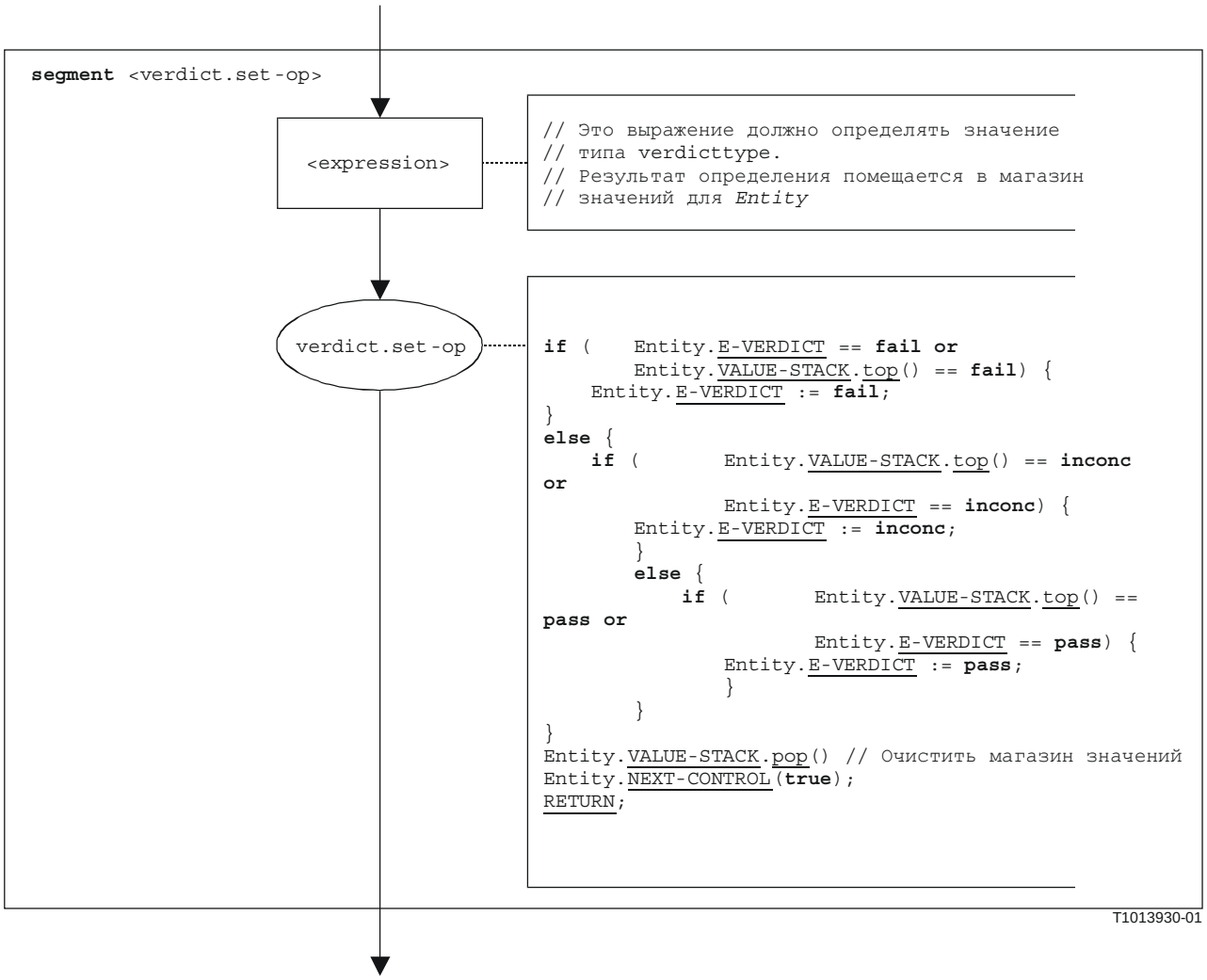


Рисунок B.124/Z.140 – Сегмент <verdict.set-op> потокового графа

B.3.7.59 Команда While

Синтаксической структурой команды **while** является:

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

Выполнение команды **while** определяется сегментом <while-stmt> потокового графа, показанным на рис. B.125.

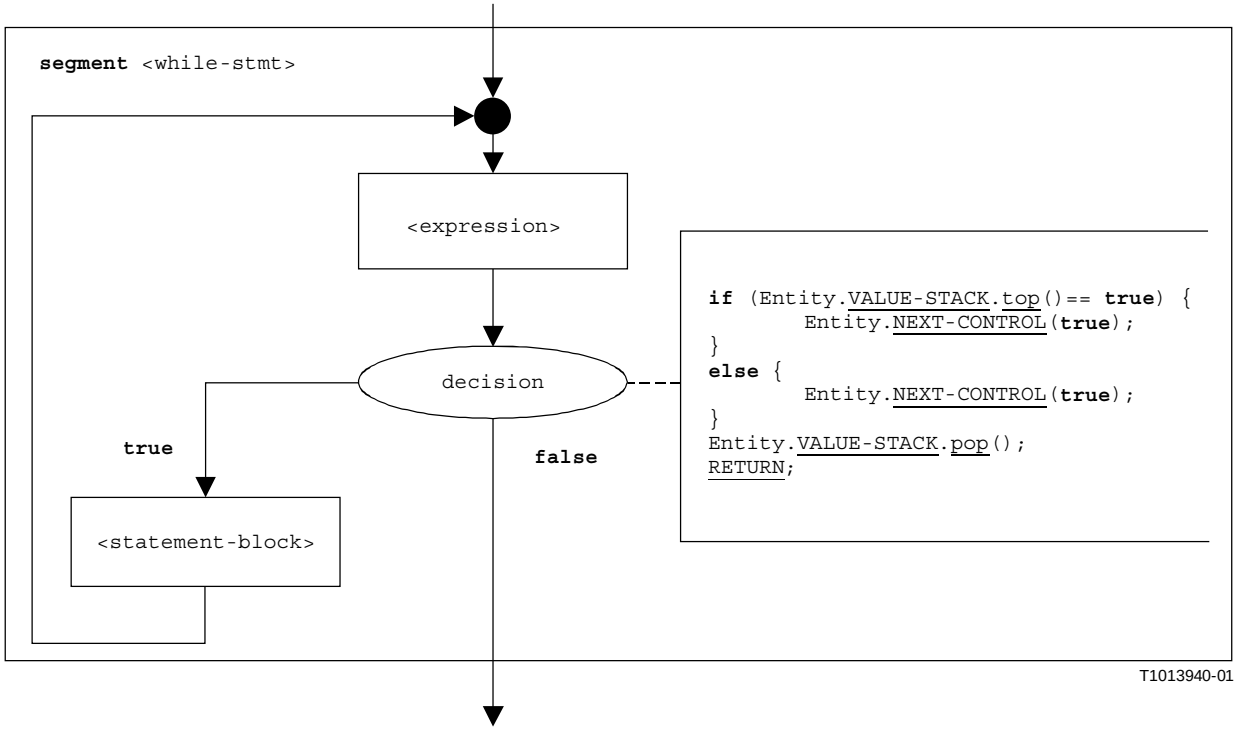


Рисунок B.125/Z.140 – Сегмент <while-stmt> потокового графа

B.3.8 Списки компонентов операционной семантики

B.3.8.1 Функции и состояния

Имя	Описание	Ссылка
NEXT	Выбирает вершину-преемник заданной вершины в потоковом графе	B.3.1.6
GET-FLOW-GRAPH	Выбирает начальную вершину в потоковом графе	B.3.2.6
MTC	Ссылка на mtc в состоянии модуля	B.3.3.1.1
TC-VERDICT	Реальный вердикт тестового примера в состоянии модуля	B.3.3.1.1
DONE	Число оконченных тестовых компонентов (часть состояния модуля)	B.3.3.1.1
append	Операция 'append' списка: добавляет в список элемент в качестве последнего элемента	B.3.3.1.1
delete	Операция 'delete' списка: удаляет элемент из списка	B.3.3.1.1
first	Операция 'first' списка: выдает первый элемент из списка	B.3.3.1.1
	Операция 'first' очереди: выдает первый элемент из очереди	B.3.3.3.2
length	Операция 'length' списка: выдает длину списка	B.3.3.1.1

Имя	Описание	Ссылка
STATUS	Состояние (**ACTIVE** или **BLOCKED**) управления модулем или тестового компонента	B.3.3.2.1
	Состояние (**IDLE, RUNNING** или **TIMEOUT**) таймера	B.3.3.2.4
	Состояние (**STARTED** или **STOPPED**) порта	B.3.3.3.2
E-VERDICT	Местный тестовый вердикт тестового компонента	B.3.3.2.1
CONTROL-STACK	Магазин вершин потокового графа, обозначающих реальное состояние управления у объекта	B.3.3.2.1
VALUE-STACK	Магазин значений для записи результатов выражений, операндов, операций и функций	B.3.3.2.1
push	Магазинная операция 'push': помещает элемент в магазин	B.3.3.2.1
pop	Магазинная операция 'pop': выбирает (выталкивает) элемент из магазина	B.3.3.2.1
top	Магазинная операция 'top': выдает верхний элемент из магазина	B.3.3.2.1
clear	Магазинная операция 'clear': очищает магазин	B.3.3.2.1
	Операция 'clear' очереди: удаляет все элементы из очереди	B.3.3.3.2
clear-until	Магазинная операция 'clear-until': выбирает элементы, пока указанный элемент не станет верхним элементом в магазине	B.3.3.2.1
NEW-ENTITY	Создает новое состояние объекта	B.3.3.2.1
VAR-SET	Установка значения переменной	B.3.3.2.4
TIMER-SET	Установка значений для таймера	B.3.3.2.4
DEF-DURATION	Выдержка таймера по умолчанию	B.3.3.2.4
ACT-DURATION	Выдержка, с которой активный таймер был запущен	B.3.3.2.4
TIME-LEFT	Время, которое считающему таймеру осталось считать до истечения выдержки	B.3.3.2.4
INIT-VAR	Создает новое связывание переменной	B.3.3.2.4
INIT-TIMER	Создает новое связывание таймера	B.3.3.2.4
GET-VAR-LOC	Выбирает положение переменной	B.3.3.2.4
GET-TIMER-LOC	Выбирает положение таймера	B.3.3.2.4
INIT-VAR-LOC	Создает новое связывание переменной с существующим положением	B.3.3.2.4
INIT-TIMER-LOC	Создает новое связывание таймера с существующим положением	B.3.3.2.4
INIT-VAR-SCOPE	Инициирует новый контекст переменной	B.3.3.2.4
INIT-TIMER-SCOPE	Инициирует новый контекст таймера	B.3.3.2.4
DEL-VAR-SCOPE	Удаляет контекст переменной	B.3.3.2.4
DEL-TIMER-SCOPE	Удаляет контекст таймера	B.3.3.2.4
NEW-PORT	Создает новый порт	B.3.3.3.2
GET-PORT	Выбирает справочный номер порта	B.3.3.3.2
GET-REMOTE-PORT	Выбирает справочный номер удаленного порта	B.3.3.3.2
ADD-CON	Добавляет соединение к состоянию порта	B.3.3.3.2
DEL-CON	Удаляет соединение из состояния порта	B.3.3.3.2
enqueue	Операция 'enqueue' очереди: вставляет в очередь элемент в качестве последнего элемента	B.3.3.3.2
dequeue	Операция 'dequeue' очереди: удаляет из очереди первый элемент	B.3.3.3.2
DEL-ENTITY	Удаляет объект из состояния модуля	B.3.3.4
EXISTING	Проверяет, существует тестовый компонент или нет	B.3.3.4

Имя	Описание	Ссылка
UPDATE-REMOTE-REFERENCES	Обновляет таймеры и переменные с одним и тем же положением в разных объектах к одному и тому же значению	B.3.3.4
CONSTRUCT-ITEM	Конструирует элемент, подлежащий передаче	B.3.4.3
MATCH-ITEM	Проверяет, соответствуют ли полученные сообщение, вызов, ответ или особое состояние операции приема	B.3.4.4
RETRIEVE-INFO	Выбирает информацию из полученных сообщения, вызова, ответа или особого состояния	B.3.4.4
NEW-CALL-RECORD	Создает запись вызова для вызова функции	B.3.5.1
INIT-FLOW-GRAPHS	Инициирует обработку потокового графа	B.3.6.1
GET-UNIQUE-ID	Выдает новый уникальный идентификатор, когда он запрошен	B.3.6.1
CONTINUE-COMPONENT	Реальный компонент продолжает свое выполнение	B.3.6.1
RETURN	Возвращает управление к процедуре оценки модуля, определенной в п. B.3.6	B.3.6.1
****DYNAMIC-ERROR****	Описывает появление динамической ошибки	B.3.6.1

B.3.8.2 Специальные ключевые слова

Ключевое слово	Описание	Ссылка
MARK	Используется в качестве метки для *VALUE-STACK*	B.3.3.2
ACTIVE	*STATUS* в состоянии объекта	B.3.3.2
BLOCKED	*STATUS* в состоянии объекта	B.3.3.2
NULL	Символическое значение в указателе и в подобных ему типах, указывающее на отсутствие адресации к чему-либо	
IDLE	*STATUS* в состоянии таймера	B.3.3.2.4
RUNNING	*STATUS* в состоянии таймера	B.3.3.2.4
TIMEOUT	*STATUS* в состоянии таймера	B.3.3.2.4
STARTED	*STATUS* для порта	B.3.3.2.4
STOPPED	*STATUS* для порта	B.3.3.2.4
NONE	Используется для описания неопределенного значения	

B.3.8.3 Сегменты потокового графа

Идентификатор	Относится к конструкции TTCN-3	Ссылка	
		Рисунок	Раздел
`<alt-stmt>`	Команда `alt`	B.25	B.3.7.1
`<alt-with-else>`	Команда `alt`	B.26	B.3.7.1
`<alt-without-else>`	Команда `alt`	B.27	B.3.7.1
`<assignment-stmt>`	Команда присвоения	B.29	B.3.7.2
`<b-call-with-receiver>`	`call`	B.35	B.3.7.3.3
`<b-call-without-receiver>`	`call`	B.36	B.3.7.3.4
`<b-call-with-rec-dur>`	`call`	B.37	B.3.7.3.5

Идентификатор	Относится к конструкции TTCN-3	Ссылка	
		Рисунок	Раздел
<b-call-without-rec-dur>	call	B.38	B.3.7.3.6
<blocking-call-op>	call	B.31	B.3.7.3
<call-op>	call	B.30	B.3.7.3
<catch-op>	catch	B.39	B.3.7.4
<catch-with-sender>	Используется в операции catch	B.40	B.3.7.4.1
<catch-without-sender>	Используется в операции catch	B.41	B.3.7.4.2
<clear-port-op>	Порт clear	B.42	B.3.7.5
<constant-declaration>	Объявление константы	B.44	B.3.7.7
<connect-op>	connect	B.43	B.3.7.6
<create-op>	create	B.45	B.3.7.8
<disconnect-op>	disconnect	B.53	B.3.7.12
<do-while-stmt>	Команда do-while	B.54	B.3.7.13
<done-all-comp-op>	all component.done	B.55	B.3.7.14
<done-any-comp-op>	any component.done	B.56	B.3.7.15
<done-component-op>	Компонент done	B.57	B.3.7.16
<execute-stmt>	execute	B.58	B.3.7.17
<execute-timeout>>	execute	B.59	B.3.7.17
<execute-without-timeout>>	execute	B.60	B.3.7.17
<expression>	Выражение	B.61	B.3.7.18
<finalize-component-init>	Используется в поведении для определений типов компонентов	B.66	B.3.7.19
<for-stmt>>	Команда for	B.68	B.3.7.21
<function-call>	Вызов определяемых пользователем функций	B.69	B.3.7.22
<func-op-call>	Используется в <expression>	B.64	B.3.7.18.3
<getcall-op>	**getcall**	B.74	B.3.7.27
<getcall-with-sender>	Используется в операции **getcall**	B.75	B.3.7.27.1
<getcall-without-sender>	Используется в операции **getcall**	B.76	B.3.7.27.2
<getreply-op>	**getreply**	B.76	B.3.7.28
<getreply-with-sender>	Используется в операции **getreply**	B.78	B.3.7.28.1
<getreply-without-sender>	Используется в операции **getreply**	B.79	B.3.7.28.2
<goto-stmt>	**goto**	B.80	B.3.7.29
<if-else-stmt>	**if-else**	B.80	B.3.7.30
<if-with-else-branch>	**if-else**	B.82	B.3.7.30.1
<if-without-else-branch>	**if-else**	B.83	B.3.7.30.2
<init-component-scope>	Используется в поведении для определений типов компонентов	B.67	B.3.7.20
<label-stmt>	**label**	B.84	B.3.7.31
<lit-value>	Используется в <expression>	B.62	B.3.7.18.1
<log-stmt>	**Log**	B.85	B.3.7.32
<map-op>	Операция **map**	B.86	B.3.7.33

Идентификатор	Относится к конструкции TTCN-3	Ссылка	
		Рисунок	Раздел
<mtc-op>	**mtc**	B.87	B.3.7.34
<nb-call-with-receiver>	**call**	B.33	B.3.7.3.1
<nb-call-without-receiver>	**call**	B.34	B.3.7.3.2
<non-blocking-call-op>	**call**	B.32	B.3.7.3
<operator-appl>	Используется в <expression>	B.65	B.3.7.18.4
<parameter-handling>	Создание объектов, вызовов функции	B.73	B.3.7.26
<port-declaration>	Объявление порта	B.46	B.3.7.9
<raise-op>	**raise**	B.88	B.3.7.35
<raise-with-receiver-op>	**raise**	B.89	B.3.7.35.1
<raise-without-receiver-op>	**raise**	B.90	B.3.7.35.2
<read-timer-op>	Таймер **read**	B.91	B.3.7.36
<receive-assignment>	Используется в операции **receive**	B.95	B.3.7.37.3
<receive-op>	**receive**	B.92	B.3.7.37
<receive-with-sender>	Используется в операции **receive**	B.93	B.3.7.37.1
<receive-without-sender>	Используется в операции **receive**	B.94	B.3.7.37.2
<receiving-branch>	Команда **alt**	B.28	B.3.7.1.1
<reply-op>	**reply**	B.96	B.3.7.38
<reply-with-receiver-op>	**reply**	B.97	B.3.7.38.1
<reply-without-receiver-op>	**reply**	B.98	B.3.7.38.2
<ref-par-var-calc>	Создание объектов, вызовов функции	B.71	B.3.7.24
<ref-par-timer-calc>	Создание объектов, вызовов функции	B.72	B.3.7.25
<return-stmt>	**return**	B.99	B.3.7.39
<return-with-value>	**return**	B.100	B.3.7.39.1
<return-without-value>	**return**	B.101	B.3.7.39.2
<running-all comp-op>	**all component.running**	B.102	B.3.7.40
<running-any comp-op>	**any component.running**	B.103	B.3.7.41
<running-component-op>	Компонент **running**	B.104	B.3.7.42
<running-timer-op>	Таймер **running**	B.105	B.3.7.43
<self-op>	**self**	B.109	B.3.7.45
<send-op>	**send**	B.106	B.3.7.44
<send-with-receiver-op>	**send**	B.107	B.3.7.44.1
<send-without-receiver-op>	**send**	B.108	B.3.7.44.2
<start-component-op>	Компонент **start**	B.110	B.3.7.46
<start-port-op>	Порт **start**	B.111	B.3.7.47
<start-timer-op>	Таймер **start**	B.112	B.3.7.48
<start-timer-op-default>	Таймер **start**	B.113	B.3.7.48.1
<start-timer-op-duration>	Таймер **start**	B.114	B.3.7.48.2
<stop-entity-op>	Выполнение **stop** управления модулем, mtc или тестового компонента	B.116	B.3.7.50

Идентификатор	Относится к конструкции TTCN-3	Ссылка	
		Рисунок	Раздел
<stop-port-op>	Порт **stop**	B.117	B.3.7.51
<statement-block>	Блок команд	B.115	B.3.7.49
<stop-timer-op>	Таймер **stop**	B.118	B.3.7.52
<sut.action-op>	**sut.action-op**	B.119	B.3.7.53
<system-op>	**system**	B.120	B.3.7.54
<timeout-timer-op>	Таймер **timeout**	B.121	B.3.7.55
<timer-declaration>	Объявление таймера	B.47	B.3.7.10
<timer-decl-default>	Объявление таймера с выдержкой по умолчанию	B.48	B.3.7.10.1
<timer-decl-no-def>	Объявление таймера без выдержки по умолчанию	B.49	B.3.7.10.2
<unmap-op>	Операция **unmap**	B.122	B.3.7.56
<value-par-calculation>	Создание объектов, вызовов функции	B.70	B.3.7.23
<variable-declaration>	Объявление переменной	B.50	B.3.7.11
<variable-declaration-init>	Объявление переменной с начальным значением	B.51	B.3.7.11.1
<variable-declaration-undef>	Объявление переменной без начального значения	B.52	B.3.7.11.2
<var-value>	Используется в <expression>	B.63	B.3.7.18.2
<verdit.get-op>	**verdict.get**	B.123	B.3.57
<verdit.set-op>	**verdict.set**	B.124	B.3.7.58
<while-stmt>	Команда **while**	B.125	B.3.7.59

Приложение C

Сопоставление входящих значений

C.1 Механизмы сопоставления шаблона

В данном Приложении определяются механизмы сопоставления, которые могут использоваться в шаблонах TTCN-3 (и только в шаблонах).

C.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. Например:

```
// 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,          // Специфичное значение типа integer
  field2 : = "My string",    // Специфичное значение типа charstring
  field3 : = true,           // Специфичное значение типа boolean
  field4 : = {1, 2, 3}       // Специфическое значение массива integer
}
```

C.1.2 Механизмы сопоставления вместо значений

C.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. Например:

```
template Mymessage MyTemplate : =
{
  field1 : = (2, 4, 6),               // Список значений integer
  field2 : = ("String1", "String2"),  // Список значений charstring
  :
  :
}
```

C.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
  :
}
```

C.1.2.3 Пропущенные значения

Ключевое слово **omit** указывает, что какое-либо факультативное поле шаблона должно отсутствовать. Оно может использоваться со значениями любых типов, учитывая, что рассматриваемое поле шаблона является факультативным. Например:

```
template Mymessage:MyTemplate : =
{  :
   :
   field3 := omit,           // Пропустить это поле
   :
}
```

C.1.2.4 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.

Например:

```
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
}
```

C.1.2.5 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. Например:

```
template Mymessage:MyTemplate : =
{  :
   field3 : = *,  // Будет соответствовать true или false, или "пропущенное поле"
   :
}
```

C.1.2.6 Диапазон значений

Диапазон обозначает ограниченную область приемлемых значений. Он используется только со значениями типа **integer** (и подтипов **integer**). 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.

Например:

```
template Mymessage:MyTemplate : =
{
   field1 : = (1 . . 6),     // Диапазон типа integer
   :
   :
   :
}
// Другими записями для field1 могли быть (–∞…8) и (12…∞)
```

C.1.3 Механизмы сопоставления внутри значений

C.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.

Например:

```
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.

C.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. C.1.5). Например, "`abcdxyz`", "`abccxyz`", "`abcxxyz`" и т. д. будут соответствовать **`pattern`** "`abc?xyz`". Однако "`abcxyz`", "`abcdefxyz`" и т. д. не будут соответствовать.

C.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. Он используется только в рамках значений типов string или значений массива. Символ "*" соответствует максимально длинной последовательности элементов согласно комбинации, определяемой символами, окружающими "*". Например:

```
template Mymessage MyTemplate : =
{ :
  field2 : = "abcxyz",
  field3 : = '10*11'B, // Где "*" может быть любой последовательностью битов
                       // (возможно, пустой)
  field4 : = {*, 2, 3} // Где первый элемент 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.

C.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`".

C.1.4 Сопоставление атрибутов значений

C.1.4.1 Ограничения длины

Атрибут ограничения длины используется для ограничения длины значений цепочки и числа элементов в структуре **`set of`** или **`record of`**. Он используется только как атрибут в следующих механизмах: дополнение (Complement), *AnyValue, AnyValueOrNone, AnyElement* и *AnyElementOrNone*. Он может использоваться также в сочетании с атрибутом **`ifpresent`**. Синтаксис **`length`** можно найти в пп. 6.2.3 и 6.3.3.

Единицы длины в случае значений цепочки должны интерпретироваться согласно таблице 4. Для типов **`set of`** и **`record of`** 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.

В случае опущенного поля атрибут **length** всегда считается соответствующим (то есть при наличии **omit** он будет избыточным). При наличии *AnyValueOrNone* и **ifpresent** он содержит ограничение на входящее значение, если оно имеется. Например:

```
template Mymessage MyTemplate : =
{
   field1 : = complement (4, 5) length (1 . . 6),  // •••••••••••• (1, 2, 3, 6)
   field2 : = "ab*ab" length (13)  // •••••••••••• ••••• •••••••
                                   // AnyElementOrNone •••••••••• 9 ••••••
```

C.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. Например:

```
template Mymessage:MyTemplate:=
{  :
   field2 : = "abcd" ifpresent,  // Сопоставляется "abcd", если не опущено
   :
   :
}
```

ПРИМЕЧАНИЕ. – Смысл *AnyValueOrName* точно такой же, как у ? **ifpresent**.

C.1.5 Сопоставление комбинации знаков

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 метазнаков ? и * для обозначений "любой знак" и "любое число любых знаков" соответственно. Например:

```
template charstring MyTemplate : = pattern "ab??xyz*";
```

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.

Если потребуется считать любой метазнак обычным знаком, то перед ним должен стоять метазнак '\'. Например:

```
template charstring MyTemplate : = pattern "ab?\?xyz*";
```

Этот шаблон будет соответствовать любой цепочке знаков, которая содержит знаки 'ab', за которыми следуют любые знаки, за которыми в свою очередь следуют знаки '?xyz', а затем любое число любых знаков.

В дополнение к прямым значениям цепочки внутри команды **pattern** можно также использовать ссылки на существующие шаблоны, константы или переменные. Ссылки могут сводиться к одному или нескольким типам цепочек знаков. Например:

```
const charstring MyString : = "ab?";
template charstring MyTemplate : = pattern MyString;
```

Этот шаблон будет соответствовать любой цепочке знаков, которая содержит знаки 'ab', за которыми следуют любые знаки. Любая цепочка знаков, следующая за ключевым словом **pattern** явно или с помощью ссылки, фактически будет интерпретироваться согласно правилам, определенным в этом подразделе.

Команда **pattern** позволяет также использовать оператор связывания (конкатенации), а в случае universal charstring использовать счетверенную продукцию для определения одиночного знака. Например:

```
const charstring MyString : = "ab?";
template universal charstring MyTemplate : = pattern MyString
  & "de" & (1, 1, 13, 7);
```

Этот шаблон будет соответствовать любой цепочке знаков, которая содержит знаки 'ab', за которыми следуют любые знаки, за которыми в свою очередь следуют знаки 'de', а затем знак из стандарта ИСО/МЭК 10646-1 с группой = 1, матрицей = 1, рядом = 65 и ячейкой = 7.

Приложение D

Предопределенные функции TTCN-3

D.1 Предопределенные функции TTCN-3

В данном Приложении описываются предопределенные функции TTCN-3.

D.1.1 Целое число в знак

```
int2char (integer value) return char
```

Эта функция преобразует значение **integer** в диапазоне 0…127 (8-битовое кодирование) в значение знака из Рекомендации МСЭ-Т Т.50 и стандарта ИСО/МЭК 646 [5]. Значение **integer** описывает 8-битовое кодирование знака.

Результатом функции будет –1, если значение аргумента отрицательно или превышает 127.

D.1.2 Знак в целое число

```
char2int (char value) return integer
```

Эта функция преобразует значение **char** из Рекомендации МСЭ-Т Т.50 и стандарта ИСО/МЭК 646 [5] в значение **integer** в диапазоне 0…127. Значение **integer** описывает 8-битовое кодирование знака.

D.1.3 Целое число в универсальный знак

```
int2unichar (integer value) return universal char
```

Эта функция преобразует значение **integer** в диапазоне 0…268435455 (32-битовое кодирование) в значение знака из стандарта ИСО/МЭК 10646-1 [6]. Значение **integer** описывает 32-битовое кодирование знака.

Результатом функции будет –1, если значение аргумента отрицательно или превышает 268435455.

D.1.4 Универсальный знак в целое число

unichar2int (**universal char** value) **return integer**

Эта функция преобразует значение **universal char** из стандарта ИСО/МЭК 10646 [6] в значение **integer** в диапазоне 0…268435455. Значение integer описывает 32-битовое кодирование знака.

D.1.5 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.

D.1.6 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 0…F представляют десятичные значения 0…15 соответственно.

D.1.7 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.

D.1.8 Цепочка знаков в целое число

str2int (**charstring** value) **return integer**

Эта функция преобразует **charstring**, представляющую значение **integer**, в эквивалентное **integer**. Если цепочка не представляет действительное целочисленное значение, то функция выдает значение НУЛЬ (0).

Примеры:

str2int ("66") будет выдавать **integer** со значением 66
str2int ("-66") будет выдавать **integer** со значением -66
str2int ("abc") будет выдавать **integer** со значением 0
str2int ("0") будет выдавать **integer** со значением 0

D.1.9 Целое число в цепочку битов

int2bit (**integer** value, 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. Ошибка тестового примера появляется в случае, когда value отрицательно или когда полученная **bitstring** содержит больше битов, чем указано в параметре length.

D.1.10 Целое число в шестнадцатеричную цепочку

int2hex (**integer** value, 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 младшим разрядом, а самая левая – старшим. Шестнадцатеричные цифры 0…F представляют десятичные значения 0…15 соответственно. Если преобразование выдает значение с числом шестнадцатеричных цифр, которое меньше указанного в параметре length, то hexstring должна заполняться слева нулями. Ошибка тестового примера появляется в случае, когда value отрицательно или когда полученная **hexstring** содержит больше шестнадцатеричных цифр, чем указано в параметре length.

D.1.11 Целое число в цепочку октетов

int2oct (**integer** value, 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 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. Если преобразование выдает значение с числом шестнадцатеричных цифр, которое меньше указанного в параметре length, то **hexstring** должна заполняться слева нулями. Ошибка тестового примера появляется в случае, когда value отрицательно или когда полученная **hexstring** содержит больше шестнадцатеричных цифр, чем указано в параметре length.

D.1.12 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).

Примеры:

```
int2str ("66") будет выдавать charstring со значением 66
int2str ("-66") будет выдавать charstring со значением -66
int2str ("0") будет выдавать integer со значением 0
```

D.1.13 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.

Примеры:

```
lengthof ('010'B) // выдает 3
lengthof ('F3'H)  // выдает 2
lengthof ('F2'O)  // выдает 1
lengthof ("Length_of_Example")   // выдает 17
```

D.1.14 Число элементов в структурированном типе

```
sizeof (structured_type value) return integer
```

Эта функция выдает реальное число элементов в типе **record, record of, set, set of, template** или в массиве.

```
// Дано
type record MyPDU
   {    boolean field1,
        integer field2
   }
// тогда
sizeof (MyPDU)
// выдает 2
```

D.1.15 Функция IsPresent

```
ispresent (any_type value) return boolean
```

Эта функция выдает 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 внутри объекта данных, который определен как факультативный объект.

```
// 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
```

D.1.16 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
```

Приложение E

Использование других типов данных совместно с TTCN-3

E.1 Использование ASN.1 с TTCN-3

В данном Приложении описывается оптимальное использование языка ASN.1 совместно с TTCN-3.

TTCN-3 предоставляет прозрачный интерфейс для использования ASN.1 версии 1997 года (определенной в Рекомендациях МСЭ-Т серии X.680 [7], [8], [9] и [10]) в модулях TTCN-3. Идентификатор языка для ASN.1 версии 1997 года при импортировании в модуль TTCN-3 должен обозначаться как "ASN.1:1997".

При использовании ASN.1 с TTCN-3 ключевые слова, перечисленные в таблице E.1, не должны применяться в качестве идентификаторов в модулях TTCN-3. Ключевые слова ASN.1 должны соответствовать требованиям Рекомендации МСЭ-Т X.680 [7].

Таблица E.1/Z.140 – Список ключевых слов ASN.1

ABSENT	**EMBEDDED**	**INTERSECTION**	**SEQUENCE**
ABSTRACT-SYNTAX	**END**	**Iso10646string**	**SET**
ALL	**ENUMERATED**	**MAX**	**SIZE**
APPLICATION	**EXCEPT**	**MIN**	**STRING**
AUTOMATIC	**EXPLICIT**	**MINUS-INFINITY**	**SYNTAX**
BEGIN	**EXPORTS**	**NULL**	**T61String**
BIT	**EXTERNAL**	**NumericString**	**TAGS**
BMPSTRING	**FALSE**	**OBJECT**	**TeletexString**
BOOLEAN	**FROM**	**ObjectDescriptor**	**TRUE**
BY	**GeneralizedTime**	**OCTET**	**TYPE-IDENTIFIER**
CHARACTER	**GeneralString**	**OF**	**UNION**
CHOICE	**IA5String**	**OPTIONAL**	**UNIQUE**
CLASS	**IDENTIFIER**	**PDV**	**UNIVERSAL**
COMPONENT	**IMPLICIT**	**PLUS-INFINITY**	**UniversalString**
COMPONENTS	**IMPORTS**	**PRESENT**	**UTCTime**
CONSTRAINED	**INCLUDES**	**PrintableString**	**VideotexString**
DEFAULT	**INSTANCE**	**PRIVATE**	**VisibleString**
DEFINITIONS	**INTEGER**	**REAL**	**WITH**

E.1.1 Эквивалентные типы ASN.1 и TTCN-3

Типы ASN.1, перечисленные в таблице E2, считаются эквивалентами своих аналогов из TTCN-3.

Таблица E.2/Z.140 – Список эквивалентов ASN.1 и TTCN-3

Тип ASN.1	Отображается в эквивалент TTCN-3
BOOLEAN	boolean
INTEGER	integer
REAL	float
OBJECT IDENTIFIER	objid
BIT STRING	bitstring
OCTET STRING	octetstring
SEQUENCE	record

Таблица E.2/Z.140 – Список эквивалентов ASN.1 и TTCN-3

Тип ASN.1	Отображается в эквивалент TTCN-3
SEQUENCE OF	record of
SET	set
SET OF	set of
ENUMERATED	enumerated
CHOICE	union

Во всех операторах, функциях, механизмах сопоставления, обозначениях значений TTCN-3 и т. п., в которых могут использоваться типы TTCN-3, приведенные в таблице E2, могут также использоваться соответствующие типы ASN.1.

E.1.2 Типы и значения данных ASN.1

Типы и значения ASN.1 могут применяться в модулях TTCN-3. Определения ASN.1 строятся с использованием отдельного модуля ASN.1.

Пример:

```
MyASN1module DEFINITIONS     : : =
BEGIN
       Z::=   INTEGER              -- Определение простого типа

       BMessage::= SET             -- Определение типа ASN.1
       {
         name Name,
         title VisibleString,
         date Date
       }

       johnValues Bmessage ::=     -- Определение типа ASN.1
       {
         name "John Doe",
         title "Mr",
         date "April 12th"
       }
END
```

Модуль ASN.1 должен записываться согласно синтаксису из Рекомендаций МСЭ-Т серии X.680 [7], [8], [9] и [10]. Когда типы и значения ASN.1 объявлены, они могут использоваться в модулях TTCN-3 точно так же, как используются обычные типы и значения TTCN-3 из других модулей TTCN-3 (то есть требуемые определения должны импортироваться).

Пример:

```
module MyTTCNModule
{
       import all from MyASN1module language "ASN.1:1997";

       const Bmessage MyTTCNConst:= johnValues;
}
```

ПРИМЕЧАНИЕ. – Определения ASN.1, не являющиеся типами и значениями (то есть классы информационных объектов или наборы информационных объектов), не доступны непосредственно из нотации TTCN-3. Такие определения внутри модуля ASN.1 должны сводиться к типу или значению, после чего на них можно будет ссылаться из модуля TTCN-3.

E.1.2.1 Контекст идентификаторов ASN.1

Импортируемые идентификаторы ASN.1 следуют тем же контекстным правилам, которым следуют типы и значения TTCN-3 (см. п. 5.4).

E.1.3 Параметризация в ASN.1

Из модуля TTCN-3 разрешается ссылаться на параметризованные определения типа и значения ASN.1. Однако все параметризованные определения ASN.1, используемые в модуле TTCN-3, должны представляться с реальными параметрами (не разрешаются открытые типы или значения), а представленные реальные параметры должны быть решаемыми за время трансляции.

Базовый язык TTCN-3 не поддерживает параметризацию уникальных конкретных объектов ASN.1. Поэтому специфичная для ASN.1 параметризация, охватывающая объекты, которые не могут быть определены прямо на базовом языке TTCN-3, должна решаться в части с ASN.1 до использования в TTCN-3. Специфичными объектами ASN.1 являются:

a) множество значений;

b) классы информационных объектов;

c) информационные объекты;

d) наборы информационных объектов.

Приведенный ниже пример является незаконным, так как в нем определяется тип TTCN-3, в котором набор объектов ASN.1 взят в качестве реального параметра.

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Определение модуля ASN.1
     -- Определение класса информационных объектов
     MESSAGE ::= CLASS {&msgTypeValueINTEGER UNIQUE,
                        &MsgFields}
     -- Определение информационного объекта
     setupMessage MESSAGE ::=          {    &msgTypeValue 1,
                  &MsgFields         OCTET STRING}
        setupAckMessage MESSAGE ::=    {    &msgTypeValue 2,
                  &MsgFields         BOOLEAN}
     -- Определение набора информационных объектов
     MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

     -- Тип ASN.1, ограниченный набором объектов
     MyMessage {MESSAGE : MsgSet} ::= SEQUENCE
     {
        code  MESSAGE.&msgTypeValue ({MsgSet}),
        Type  MESSAGE.&MsgFields ({MsgSet})
}
END

module MyTTCNModule
{
    // Определение модуля TTCN-3
    import all from MyASN1module language "ASN.1:1997";

    // Незаконный тип TTCN-3 с набором объектов в качестве параметра
    type record Q (MESSAGE MyMsgSet) ::= { Z         field1,
                                MyMessage (MyMsgSet) field2}
```

Чтобы сделать это определение законным, следует определить дополнительный тип ASN.1 MyMessage1, как показано ниже. Это решает параметризацию набора информационных объектов, поэтому ее можно прямо использовать в модуле TTCN-3.

```
MyASN1module DEFINITIONS ::=

BEGIN
     -- Определение модуля ASN.1

     ...

     MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

     -- Дополнительный тип ASN.1 для устранения параметризации с набором объектов
     MyMessage1 ::= MyMessage{ MyProtocol}
END

module MyTTCNModule
{
   // Определение модуля TTCN-3
   import all from MyASN1module language "ASN.1:1997";

   // Законный тип TTCN-3 без набора объектов в качестве параметра
   type record Q := {  Z           field1,
                       MyMessage1  field2}
}
```

E.1.4 Определение типов сообщений с помощью ASN.1

В ASN.1 сообщения определяются с помощью SEQUENCE (или, возможно, SET).

Пример:

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Определение модуля ASN.1

     MyMessageType ::= SEQUENCE
     {
          field1         Field1Type,
          field2         Field2Type    OPTIONAL, -- Это поле пожет быть опущено
          :
          fieldN         FieldNType
     }
END
```

Сообщения, определенные с помощью ASN.1, безусловно, могут быть субструктурированы с использованием SEQUENCE, SET и т. п.

E.1.5 Определение шаблонов сообщений ASN.1

Если сообщения определяются в ASN.1 с использованием, например, SEQUENCE (или, возможно, SET), то реальные сообщения как для события **send**, так и для события **receive** могут описываться с использованием синтаксиса значений ASN.1.

Пример:

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Определение модуля ASN.1

     -- Определение сообщения
MyMessageType::= SEQUENCE
     {    field1    [1] IA5STRING,          // Как цепочка знаков TTCN-3
          field2    [2] INTEGER OPTIONAL,   // Как целое число TTCN-3
          field3    [4] Field3Type,         // Как записьTTCN-3
          field4    [5] Field4Type          // Как массив TTCN-3
     }
```

```
     Field3Type::= SEQUENCE {field31 BIT STRING, field32 INTEGER, field33 OCTET
          STRING},
     Field4Type::= SEQUENCE OF BOOLEAN


     -- Может иметь следующее значение
     myValue MyMessageType::=
     {
          field1         "A string",
          field2         123,
          field3         {field31 '11011'B, field32 456789, field33 'FF'O},
          field4         {true, false}
     }
END
```

E.1.5.1 Прием сообщений ASN.1 с использованием синтаксиса шаблонов TTCN-3

Механизмы сопоставления не поддерживаются в стандартном синтаксисе ASN.1. Поэтому, если желательно использовать механизмы сопоставления с принимаемым сообщением ASN.1, то взамен должен использоваться синтаксис TTCN-3 для приемных шаблонов. Отметим, что этот синтаксис содержит справочные номера компонентов, позволяющие ссылаться на отдельные компоненты ASN.1 в SEQUENCE, SET и т. п.

Пример:

```
import type myMessageType from MyASN1module language "ASN.1:1997";

// Шаблоном сообщения с использованием механизмов сопоставления в TTCN-3 может быть
template myMessageType MyValue:=
{
   field1 :=         "A"<?>"tr"<*>"g",
   field2 :=         *,
   field3.field31 :='110??'B,
   field3.field32 :=?,
   field3.field33 :='F?'O,
   field4.[0] :=     true,
   field4.[1] :=     false
}

// Следующий синтаксис также действителен
template myMessageType MyValue:=
{
   field1 := "A"<?>"tr"<*>"g",     // Цепочка с символом подстановки
   field2 := *,                    // Любое целое число или полное отсутствие
   field3 := {'110??'B, ?, 'F?'O},
   field4 := {?, false}
}
```

E.1.5.2 Порядок расположения полей шаблона

Когда в шаблонах TTCN-3 используются типы ASN.1, важность порядка расположения полей в шаблоне будет зависеть от типа конструкции ASN.1, используемой для определения типа сообщения. Например, если используется SEQUENCE или SEQUENCE OF, то поля сообщения должны передаваться или сопоставляться в порядке, указанном в шаблоне. Если используется SET или SET OF, то поля сообщения могут передаваться или сопоставляться в любом порядке.

E.1.6 Информация о кодировании

TTCN-3 позволяет связывать ссылки на правила кодирования и варианты в рамках правил кодирования с различными элементами языка TTCN-3. Возможно также указать недействительные методы кодирования. Эта информация о кодировании описывается с помощью команды **with** согласно следующему синтаксису:

Пример:

```
module MyModule
{  :
   import type myMessageType from MyASN1module language "ASN.1:1997" with
       {encode : = "PER:1997"}
               // Все экземпляры MyMessageType должны кодироваться с
               // использованием PER:1997

{  with {encode "BER:1997"}  // Безусловным (по умолчанию) кодированием
                             // для всего модуля (тестового примера) является
                             // BER:1997
```

E.1.6.1 Атрибуты кодирования ASN.1

Предопределенными (стандартизованными) атрибутами кодирования для ASN.1 являются следующие цепочки:

a) "BER:1997" означает кодирование согласно Рекомендации МСЭ-Т X.690 (BER) [1997];

b) "CER:1997" означает кодирование согласно Рекомендации МСЭ-Т X.690 (CER) [1997];

c) "DER:1997" означает кодирование согласно Рекомендации МСЭ-Т X.690 (DER) [1997];

d) "PER-BASIC-UNALIGNED:1997" означает кодирование согласно Рекомендации МСЭ-Т X.691 (Невыровненное PER) [1997];

e) "PER-BASIC-ALIGNED:1997" означает кодирование согласно Рекомендации МСЭ-Т X.691 (Выровненное PER) [1997];

f) "PER-CANONICAL-UNALIGNED:1997" означает кодирование согласно Рекомендации МСЭ-Т X.691 (Невыровненное каноническое PER) [1997];

g) "PER-CANONICAL-ALIGNED:1997" означает кодирование согласно Рекомендации МСЭ-Т X.691 (Выровненное каноническое PER) [1997].

СЕРИИ РЕКОМЕНДАЦИЙ МСЭ-Т

Серия A	Организация работы МСЭ-Т
Серия B	Средства выражения: определения, символы, классификация
Серия C	Общая статистика электросвязи
Серия D	Общие принципы тарификации
Серия E	Общая эксплуатация сети, телефонная служба, функционирование служб и человеческие факторы
Серия F	Нетелефонные службы связи
Серия G	Системы и средства 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	TMN и техническая эксплуатация сети: международные системы передачи, телефонные, телеграфные, факсимильные и арендованные каналы
Серия N	Техническая эксплуатация: 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 и взаимосвязь открытых систем
Серия Y	Глобальная информационная инфраструктура и аспекты межсетевого протокола (IP)
Серия Z	**Языки и общие аспекты программного обеспечения для систем электросвязи**

Отпечатано в Швейцарии
Женева, 2004 г.