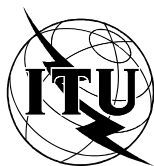


МЕЖДУНАРОДНЫЙ СОЮЗ ЭЛЕКТРОСВЯЗИ

МСЭ-Т

СЕКТОР СТАНДАРТИЗАЦИИ
ЭЛЕКТРОСВЯЗИ МСЭ

Z.143

(03/2006)

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

Методы формального описания (FDT) – Нотация тестирования и управления тестом (TTCN)

Нотация тестирования и управления тестом версии 3 (TTCN-3): Операционная семантика

Рекомендация МСЭ-Т Z.143

РЕКОМЕНДАЦИИ МСЭ-Т СЕРИИ Z

ЯЗЫКИ И ОБЩИЕ АСПЕКТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ СИСТЕМ ЭЛЕКТРОСВЯЗИ

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

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

Рекомендация МСЭ-Т Z.143

Нотация тестирования и управления тестом версии 3 (TTCN-3): Операционная семантика

Резюме

В настоящей Рекомендации дается определение операционной семантики TTCN-3 (*Нотация тестирования и управления тестом версии 3*). Операционная семантика необходима для однозначной интерпретации спецификаций, разработанных с использованием TTCN-3. Данная Рекомендация опирается на базовый язык TTCN-3, который определен в Рекомендации МСЭ-Т Z.140.

Источник

Рекомендация МСЭ-Т Z.143 утверждена 16 марта 2006 года 17-й Исследовательской комиссией МСЭ-Т (2005–2008 гг.) в соответствии с процедурой, изложенной в Рекомендации МСЭ-Т A.8.

ПРЕДИСЛОВИЕ

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

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

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

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

ПРИМЕЧАНИЕ

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

Соблюдение положений данной Рекомендации носит добровольный характер. Однако в Рекомендации могут содержаться определенные обязательные положения (например, для обеспечения возможности взаимодействия или применимости), и соблюдение положений данной Рекомендации достигается в случае выполнения всех этих обязательных положений. Для выражения необходимости выполнения требований используется синтаксис долженствования и соответствующие слова (такие, как "должен" и т. п.), а также их отрицательные эквиваленты. Использование этих слов не предполагает, что соблюдение положений данной Рекомендации является обязательным для какой-либо из сторон.

ПРАВА ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

МСЭ обращает внимание на вероятность того, что практическое применение или реализация этой Рекомендации может включать 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 получил извещение об интеллектуальной собственности, защищенной патентами, которые могут потребоваться для выполнения этой 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 по адресу: http://www.itu.int/ITU-T/ipr/.

© ITU 2006

Все права сохранены. Никакая часть данной публикации не может быть воспроизведена с помощью каких-либо средств без предварительного письменного разрешения МСЭ.

СОДЕРЖАНИЕ

Стр.

1 Сфера применения ... 1
2 Справочные документы ... 1
3 Определения и сокращения ... 1
3.1 Определения ... 1
3.2 Сокращения ... 1
4 Введение ... 1
5 Структура настоящей Рекомендации ... 2
6 Ограничения ... 2
7 Замена кратких форм ... 2
7.1 Порядок следования шагов замены ... 3
7.2 Замена глобальных констант и параметров модуля ... 3
7.3 Включение одиночных операций приема в операторы alt ... 3
7.4 Включение автономных вызовов альтернативных шагов в операторы alt ... 4
7.5 Замена операторов interleave ... 4
7.6 Замена операций trigger ... 16
8 Семантика потокового графа в TTCN-3 ... 17
8.1 Потоковые графы ... 17
8.2 Представление поведения TTCN-3 в виде потоковых графов ... 22
8.3 Определения состояний для модулей TTCN-3 ... 27
8.4 Сообщения, вызовы процедур, ответы и особые состояния ... 35
8.5 Записи вызовов для функций, альтернативных шагов и тестовых примеров ... 37
8.6 Процедура оценки для модуля TTCN-3 ... 38
9 Сегменты потокового графа для конструкций TTCN-3 ... 40
9.1 Оператор action ... 40
9.2 Оператор activate ... 40
9.3 Оператор alt ... 41
9.4 Вызов альтернативного шага ... 47
9.5 Оператор assignment ... 47
9.6 Операция call ... 47
9.7 Операция catch ... 53
9.8 Операция check ... 54
9.9 Операция clear порта ... 57
9.10 Операция connect ... 57
9.11 Определение constant ... 58
9.12 Операция create ... 59
9.13 Оператор deactivate ... 59
9.14 Операция disconnect ... 61
9.15 Оператор do-while ... 62
9.16 Компонентная операция done ... 63
9.17 Оператор execute ... 64
9.18 Выражение ... 67
9.18b Сегмент <dynamic-error> потокового графа ... 69
9.19 Сегмент <finalize-component-init> потокового графа ... 70
9.20 Сегмент <init-component-scope> потокового графа ... 70
9.21 Сегмент <parameter-handling> потокового графа ... 71
9.22 Сегмент <statement-block> потокового графа ... 71
9.23 Оператор for ... 72
9.24 Вызов функции ... 73
9.25 Операция getcall ... 77

Стр.

9.26 Операция getreply ... 78
9.27 Операция getverdict ... 78
9.28 Оператор goto ... 79
9.29 Оператор if-else ... 79
9.30 Оператор label ... 80
9.31 Оператор log ... 80
9.32 Операция map ... 81
9.33 Операция mtc ... 81
9.34 Объявление port ... 82
9.35 Операция raise ... 82
9.36 Операция read таймера ... 84
9.37 Операция receive ... 85
9.38 Оператор repeat ... 88
9.39 Операция reply ... 88
9.40 Оператор return ... 90
9.41 Компонентная операция running ... 93
9.42 Операция running таймера ... 96
9.43 Операция self ... 97
9.44 Операция send ... 97
9.45 Операция setverdict ... 100
9.46 Компонентная операция start ... 100
9.47 Операция start порта ... 102
9.48 Операция start таймера ... 102
9.49 Компонентная операция stop ... 104
9.50 Оператор stop выполнения ... 108
9.51 Операция stop порта ... 110
9.52 Операция stop таймера ... 110
9.53 Операция system ... 111
9.54 Объявление timer ... 111
9.55 Операция timeout таймера ... 113
9.56 Операция unmap ... 113
9.57 Объявление variable ... 114
9.58 Оператор while ... 116

10 Списки компонентов операционной семантики ... 116

10.1 Функции и состояния ... 116
10.2 Специальные ключевые слова ... 118
10.3 Потоковые графы описаний поведения TTCN-3 ... 119
10.4 Сегменты потокового графа ... 119

Рекомендация МСЭ-Т Z.143

Нотация тестирования и управления тестом версии 3 (TTCN-3): Операционная семантика

1 Сфера применения

В настоящей Рекомендации дается определение операционной семантики TTCN-3. Данная Рекомендация опирается на базовый язык TTCN-3, который определен в Рекомендации МСЭ-Т Z.140 [1].

2 Справочные документы

Указанные ниже Рекомендации МСЭ-Т и другие источники содержат положения, которые путем ссылки на них в данном тексте составляют положения настоящей Рекомендации. На момент публикации указанные издания были действующими. Все Рекомендации и другие источники могут подвергаться пересмотру; поэтому всем пользователям данной Рекомендации предлагается изучить возможность применения последнего издания Рекомендаций и других источников, перечисленных ниже. Список действующих в настоящее время Рекомендаций МСЭ-Т регулярно публикуется. Ссылка на документ в данной Рекомендации не придает ему как отдельному документу статус Рекомендации.

[1] Рекомендация МСЭ-Т Z.140 (2006), *Нотация тестирования и управления тестом версии 3 (TTCN-3): Базовый язык*.

3 Определения и сокращения

3.1 Определения

В настоящей Рекомендации используются термины и определения, приведенные в Рекомендации МСЭ-Т Z.140 [1].

3.2 Сокращения

В настоящей Рекомендации используются следующие сокращения:

ASN.1	Abstract Syntax Notation One	Абстрактно-синтаксическая нотация 1
BNF	Backus-Nauer Form	Форма Бэкуса-Наура
IDL	Interface Description Language	Язык описания интерфейса
MTC	Master Test Component	Главный тестовый компонент
SUT	System Under Test	Тестируемая система
TTCN	Testing and Test Control Notation	Нотация тестирования и управления тестом

4 Введение

В данном разделе смысл поведения TTCN-3 определяется интуитивно-понятным и однозначным способом. Это не означает, что операционная семантика носит формальный характер, и, следовательно, возможности проводить математические доказательства на основе данной семантики весьма ограничены.

В рассматриваемой операционной семантике обеспечивается представление о работе модуля TTCN, ориентированное на понятие состояний. Вводятся состояния разных видов, и смысл различных конструкций TTCN-3 описывается путем:

1) использования информации о состоянии в целях определения предусловий для выполнения конструкции; и
2) определения того, каким образом выполнение конструкции будет изменять состояние.

Операционная семантика ограничивается смыслом поведения в рамках нотации TTCN-3, т. е. функциями, альтернативными шагами, тестовыми примерами, управлением модулем и языковыми конструкциями для определения тестового поведения, например, операциями **send** и **receive**, операторами **if-else**- или **while-**. Смысл некоторых конструкций TTCN-3 можно объяснить путем их замены другими языковыми конструкциями. Например, операторы **interleave** являются краткой формой серии вложенных операторов **alt,** а смысл каждого оператора **interleave** объясняется его заменой соответствующей серией вложенных операторов **alt.**

Определение семантики языка в большинстве случаев опирается на дерево абстрактного синтаксиса для кода, который подлежит описанию. Эта семантика не функционирует на дереве абстрактного синтаксиса, но ей требуется графическое представление описаний поведения TTCN-3 в виде потоковых графов. Потоковый граф описывает поток управления в функции, альтернативном шаге, тестовом примере или управлении модулем. Отображение описаний поведения TTCN-3 на потоковые графы осуществляется простым и понятным способом.

ПРИМЕЧАНИЕ. – Отображение операторов TTCN-3 на потоковые графы является неформальным шагом и не определяется путем использования правил BNF в Рекомендации МСЭ-Т Z. 140 [1]. Причина этого состоит в том, что правила BNF являются неоптимальными для интуитивно-понятного отображения, поскольку в правила BNF закодированы несколько статических семантических правил, чтобы иметь возможность в ходе синтаксической проверки проводить статические семантические проверки.

5 Структура настоящей Рекомендации

Структуру настоящей Рекомендации образуют следующие четыре части:

1) В первой части (см. раздел 6) дается описание ограничений операционной семантики, т. е. относящихся к семантике вопросов, которые не охватываются данной Рекомендацией.
2) Во второй части (см. раздел 7) определяется смысл сокращенных нотаций и макронотаций TTCN-3 путем их замены другими языковыми конструкциями TTCN-3. Эти замены в модуле TTCN-3 можно рассматривать как ступень предварительной обработки перед возможной интерпретацией модуля согласно последующему описанию операционной семантики.
3) В третьей части (см. раздел 8) дается описание операционной семантики TTCN-3 с помощью интерпретации потокового графа и модификации состояний.
4) В четвертой части (см. раздел 9) описывается отображение различных операторов TTCN-3 на сегменты потокового графа, образующие компоновочные блоки для потоковых графов, представляющих функции, альтернативные шаги, тестовые примеры и управление модулем.

6 Ограничения

Операционная семантика охватывает только поведенческие аспекты TTCN-3, т. е. она описывает смысл операторов и операций. В ней не предусматривается:

a) Семантика для аспектов данных TTCN-3. Сюда входят такие вопросы, как кодирование, декодирование и использование данных, импортируемых из спецификаций не-TTCN-3.
b) Семантика для механизма группировки. Группировка относится к части определений модуля TTCN-3 и не имеет поведенческих аспектов.
c) Семантика для оператора **`import`**. Импорт определений должен выполняться в части определений модуля TTCN-3. Операционная семантика обрабатывает импортируемые определения, как если бы они определялись в импортирующем модуле.
d) Семантика для параметризации портов.

7 Замена кратких форм

Краткие формы должны расширяться с помощью соответствующих полных определений на текстовом уровне, прежде чем эта операционная семантика может быть использована для объяснения поведения TTCN-3.

Краткими формами TTCN-3 являются:

- списки объявлений параметров модуля, констант и переменных одного и того же типа, а также списки объявлений таймеров;
- автономные операции приема;
- автономные вызовы альтернативных шагов;
- операции **`trigger`**;
- пропуск операторов **`return`** и **`stop`** в конце определений функции и тестового примера;
- пропуск операторов **`stop`** выполнения; и
- операторы **`interleave`**.

В дополнение к обработке кратких форм, операционной семантике требуется специальная обработка для параметров модуля и глобальных констант, т. е. констант, которые описываются в части определений модуля. Все ссылки на параметры модуля и глобальные константы будут заменяться конкретными значениями. Это предполагает, что значение параметров модуля и глобальных констант может быть определено до того, как операционная семантика станет релевантной.

ПРИМЕЧАНИЕ 1. – Обработка параметров модуля и глобальных констант в операционной семантике будет отличаться от их обработки в компиляторе TTCN-3. Операционная семантика дает описание поведенческого смысла TTCN-3 и не служит руководством для реализации компилятора TTCN-3.

ПРИМЕЧАНИЕ 2. – Операционная семантика обрабатывает параметры и локальные константы в тестовых компонентах, тестовых примерах, функциях и управлении модулем подобно обработке переменных. Ошибочное использование локальных констант или параметров **`in`**, **`out`** и **`inout`** должно проверяться статически.

7.1 Порядок следования шагов замены

Текстовые замены кратких форм, глобальных констант и параметров модуля должны выполняться в следующем порядке:

1) замена списков объявлений параметров модуля, констант, переменных и таймеров индивидуальными объявлениями;
2) замена глобальных констант и параметров модуля конкретными значениями;
3) включение автономных операций приема в операторы **alt**;
4) включение автономных вызовов альтернативных шагов в операторы **alt**;
5) расширение операторов **interleave**;
6) замена всех операций **trigger** на эквивалентные операции **receive** и операторы **repeat**;
7) добавление **return** в конце функций без оператора **return**, добавление операций **self.stop** в конце определений тестового примера без оператора останова;
8) добавление **stop** в конце части управления модулем без оператора останова.

ПРИМЕЧАНИЕ. – Если не придерживаться данного порядка следования шагов замены, то результат замен не будет соответствовать определенному поведению.

7.2 Замена глобальных констант и параметров модуля

Константы, объявленные в части определений модуля, являются глобальными для управления модулем и для всех тестовых компонентов, созданных во время выполнения модуля TCN-3. Параметры модуля считаются глобальными константами на этапе выполнения.

До начала интерпретации модуля в операционной семантике все ссылки на глобальные константы и параметры модуля будут заменяться на фактические значения. Если значение константы или параметра модуля задано в виде выражения, то это выражение должно быть оценено. Тогда результат этой оценки будет заменять все ссылки на константы или параметр модуля.

7.3 Включение одиночных операций приема в операторы alt

Операциями приема в 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 осуществлять ветвление, обусловленное приемом сообщений, вызовами процедур, ответами и особыми состояниями. Операции **timeout** и **done** не являются в действительности операциями приема, но их можно использовать таким же образом в качестве операций приема, т. е. как альтернативы в операторах **alt**. Поэтому в операционной семантике операции **timeout** и **done** обрабатываются подобно операциям приема.

В функции, в альтернативном шаге или тестовом примере операция приема может использоваться в качестве автономного оператора. Операция **timeout** также может использоваться при управлении модулем как автономный оператор. В таком случае операция приема считается сокращением оператора **alt** только с одной альтернативой, определяемой операцией приема. Что касается операционной семантики, то оператор **alt**, в котором оператор приема является вложенным, будет заменять все автономные вхождения операций приема.

ПРИМЕР:

```
// Автономное вхождение
    :
   MyCL.trigger(MyType:?);
    :

// будет заменено на
    :
   alt {
      []   MyCL.trigger (MyType:?) { }
   }
    :

// или
    :
   MyPTC.done;
    :

// будет заменено на
    :
   alt {
      []   MyPTC.done { }
   }
    :
```

7.4 Включение автономных вызовов альтернативных шагов в операторы alt

TTCN-3 позволяет вызывать альтернативные шаги, такие как функции в функциях, альтернативных шагах, тестовых примерах и управлении модулем. Смысл автономного вызова альтернативного шага задается оператором **alt** только с одной ветвью, которая вызывает альтернативный шаг. Оператор **alt** отвечает за фиксацию мгновенного состояния процесса ("снимок"), которая оценивается в пределах альтернативного шага, и за инициирование механизма обработки по умолчанию, если на альтернативном шаге не может быть выбрана ни одна из альтернатив.

ПРИМЕЧАНИЕ. – Используемый в управлении модулем альтернативный шаг может включать только альтернативы с операциями **timeout** и ветвь **else**.

ПРИМЕР:

```
// Автономное вхождение
    :
   myAltstep(MyPar1Val);
    :

// будет заменено на
    :
   alt {
      []   myAltstep(MyPar1Val) { }
   }
    :
```

7.5 Замена операторов interleave

Смысл оператора **interleave** (чередование) определяется его заменой на серию вложенных операторов **alt** с таким же смыслом. В данном разделе приводится описание алгоритма конструкции замены для оператора **interleave**. Замена должна осуществляться на синтаксическом уровне.

В самом операторе **interleave** не разрешается:

1) использовать операторы передачи управления **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **repeat** и **return**;
2) вызывать альтернативные шаги;
3) вызывать определяемые пользователем функции, включающие операции связи;
4) защищать ветви оператора **interleave** булевыми выражениями; и
5) задавать ветви **else**.

Ввиду этих ограничений все неупомянутые автономные операторы (например, присвоение, **log**, **send** или **reply**), *блокирующие* операции **call** и составные операторы **interleave**, **if-else** и **alt** могут быть использованы в операторе **interleave**.

ПРИМЕЧАНИЕ 1. – Блокирующие операции **call** и операторы **if-else** могут рассматриваться в качестве автономных операторов, если они не имеют вложенных операторов **alt**. В случае вложенных операторов **alt** альтернативы взаимодействуют с оператором **interleave** и требуют специальной обработки. Для простоты в приведенном ниже алгоритме не делается различий между этими двумя случаями.

ПРИМЕЧАНИЕ 2. – Неблокирующие операции **call** также допускаются в операторах чередования; они рассматриваются как автономные операторы.

Алгоритм, описанный в данном разделе, управляет только операторами **interleave** без вложенных операторов **interleave**. В случае оператора **interleave**, имеющего вложенные операторы **interleave**, последние должны быть заменены до применения алгоритма.

ПРИМЕЧАНИЕ 3. – Из-за ограничений 1–5 всегда можно найти конечные замены для гнездовых вложений операторов **interleave**.

Алгоритм замены функционирует на графическом представлении оператора чередования и преобразует его в семантически эквивалентную древовидную структуру, описывающую серию вложенных операторов **alt**. Для этого требуется графическое представление автономных операторов, составных операторов **if-else**, *blocking* **call**, **alt** и **interleave**.

Автономный оператор описывается узлом с оператором в качестве надписи. Последовательность автономных операторов описывается множеством узлов, связанных потоковой линией. Это показано на рисунке 1.

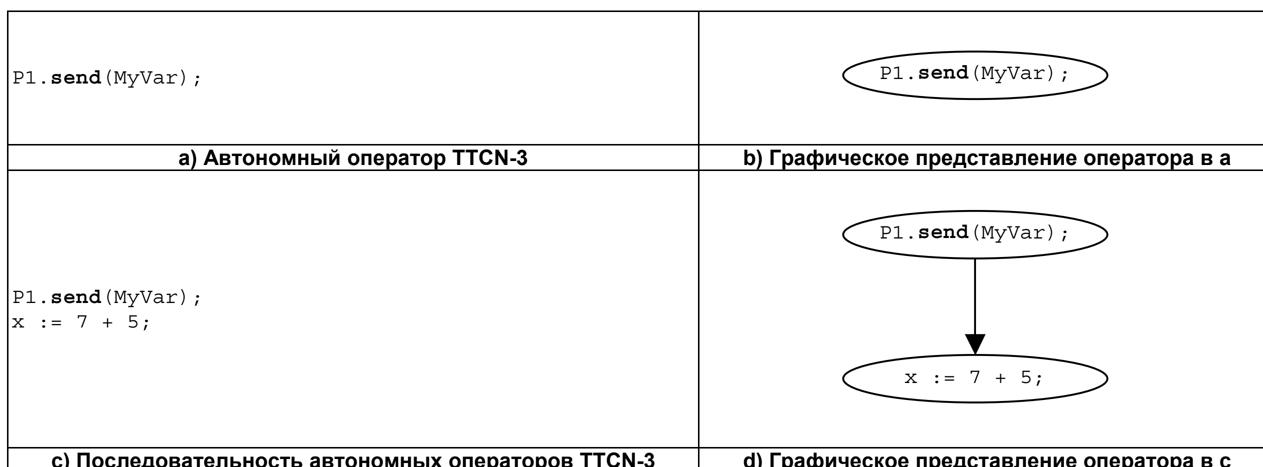


Рисунок 1/Z.143 – Графическое представление автономных операторов в TTCN-3

Графическое представление оператора **`if-else`** показано на рисунке 2. Оператор **`if-else`** представлен узлом IF с двумя потоковыми линиями, соединенными с первым оператором в двух альтернативах. Оператор **`if-else`** без ветви ELSE представлен таким же образом, если имеются операторы, следующие за оператором **`if-else`**. В этом случае потоковая линия, представляющая ветвь *else*, соединяется с первым оператором, следующим за оператором **`if-else`**. Оператор **`if-else`** без ветви ELSE и без последующих операторов представлен узлом IF только с одной потоковой линией.

ПРИМЕЧАНИЕ 4. – Надписи на потоковых линиях на рисунке 1 даны лишь для целей удобочитаемости. В алгоритме используется только связь, выраженная потоковой линией, а не надписью.

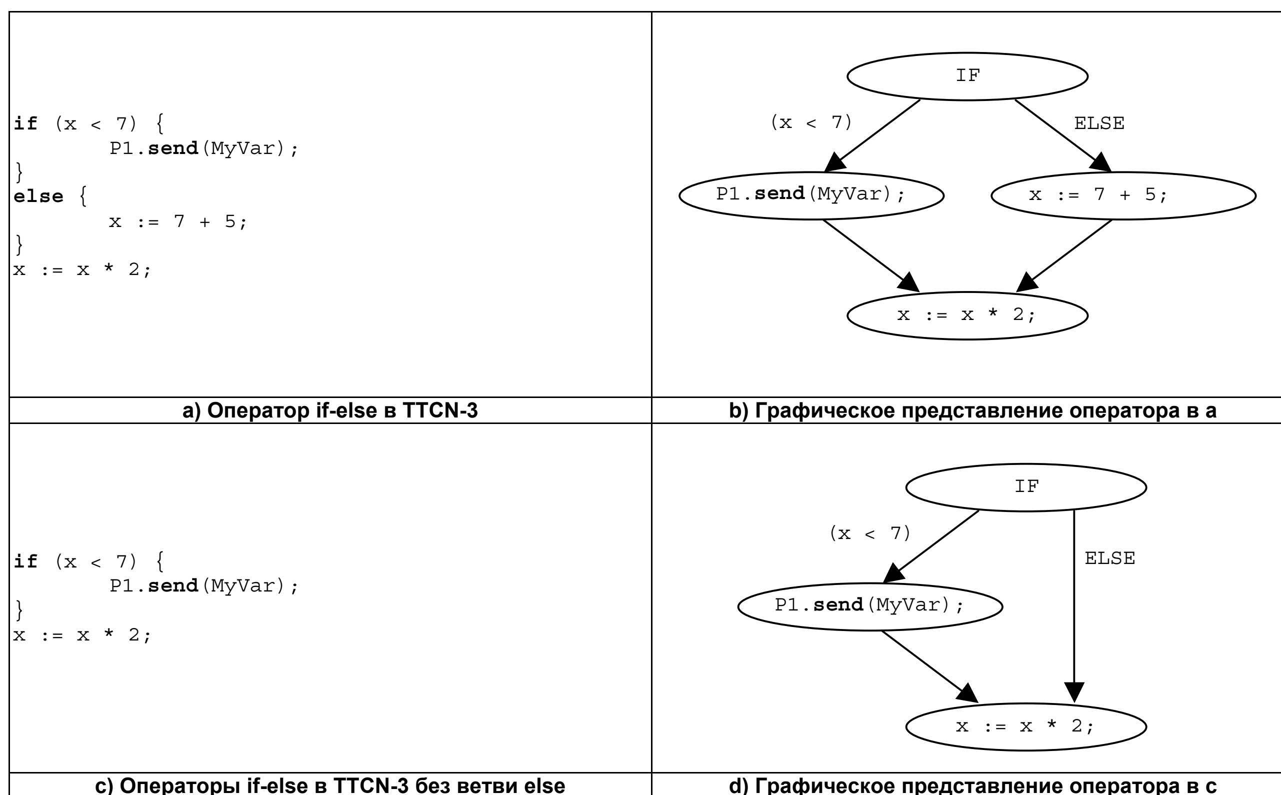
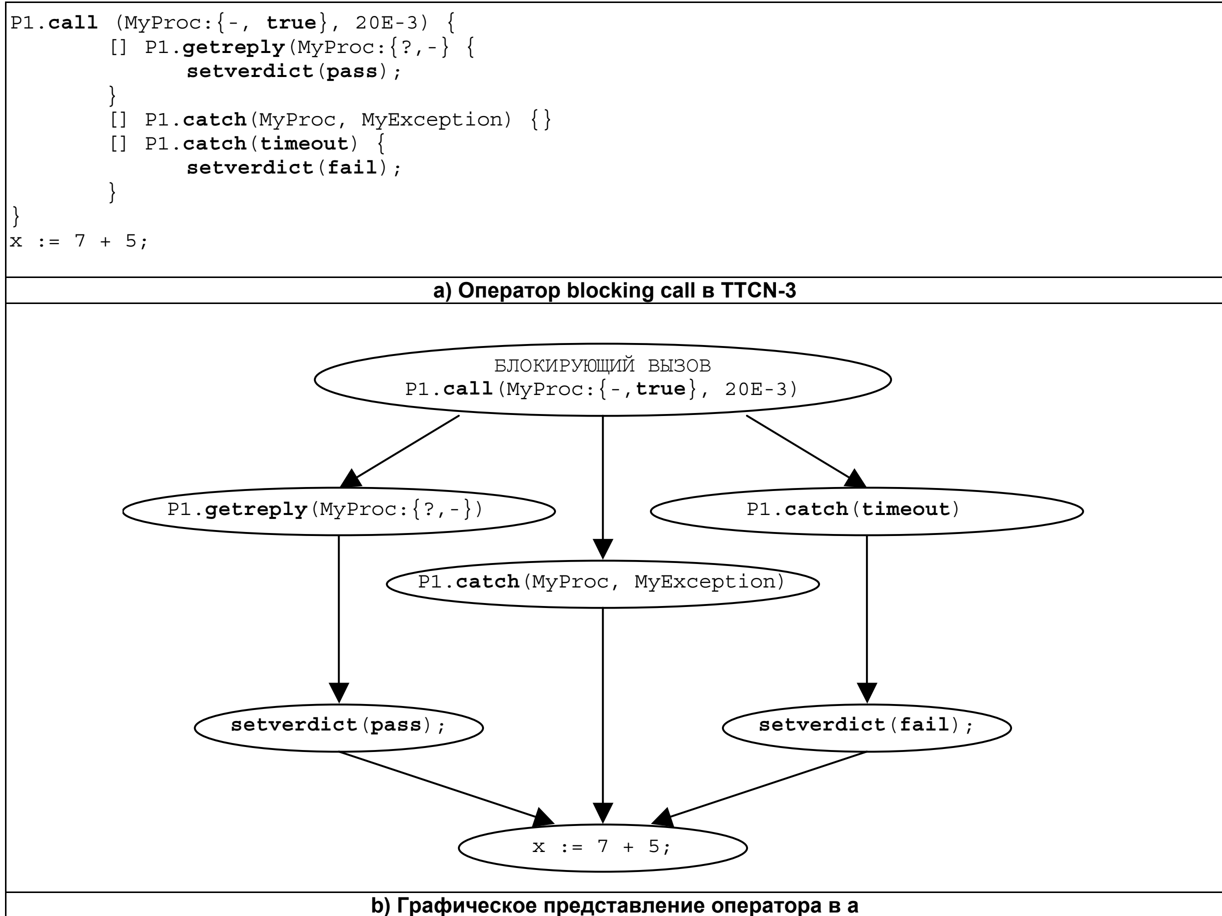


Рисунок 2/Z.143 – Графическое представление операторов if-else в TTCN-3

Графическое представление оператора blocking **`call`** показано на рисунке 3. Оператор blocking **`call`** представлен узлом BLOCKING-CALL с потоковыми линиями, связанными с операторами **`getreply`** и **`catch`** разных альтернатив.



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

a) Оператор blocking call в TTCN-3

b) Графическое представление оператора в a

Рисунок 3/Z.143 – Графическое представление оператора blocking call в TTCN-3

Графическое представление оператора **alt** показано на рисунке 4. Оператор **alt** представлен *alt*-узлом с несколькими потоковыми линиями, соединенными с разными альтернативами.

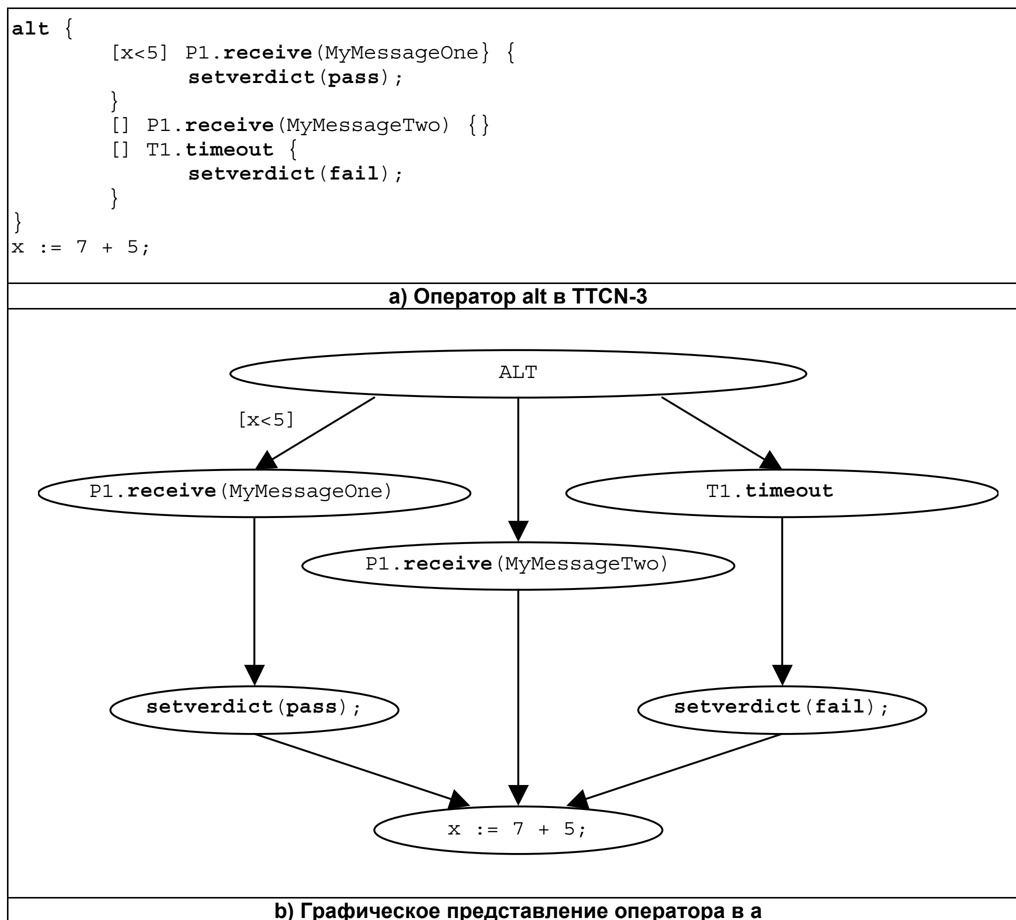
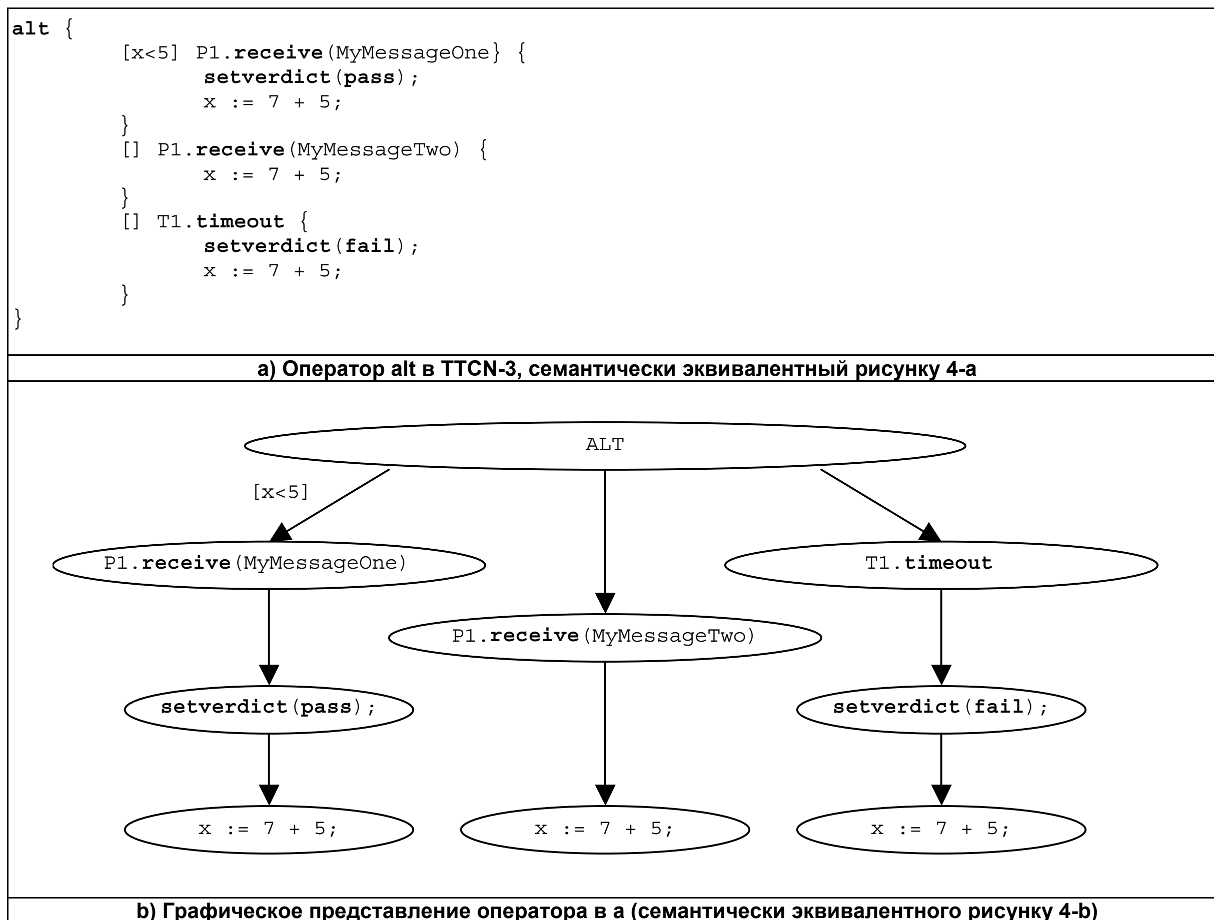


Рисунок 4/Z.143 – Графическое представление оператора alt в TTCN-3

Вообще, графические представления операторов **if-else**, *blocking* **call** и **alt** являются ориентированными графами без петель, где потоковые линии разных альтернатив объединяются при выходе из оператора. Можно с помощью дублирования преобразовать такие ориентированные графы в семантически эквивалентные древовидные представления. Это показано на рисунке 5 для оператора alt на рисунке 4. Такие древовидные представления будут создаваться по приведенному ниже алгоритму.



а) Оператор alt в TTCN-3, семантически эквивалентный рисунку 4-a

b) Графическое представление оператора в а (семантически эквивалентного рисунку 4-b)

Рисунок 5/Z.143 – Графическое представление оператора alt в TTCN-3

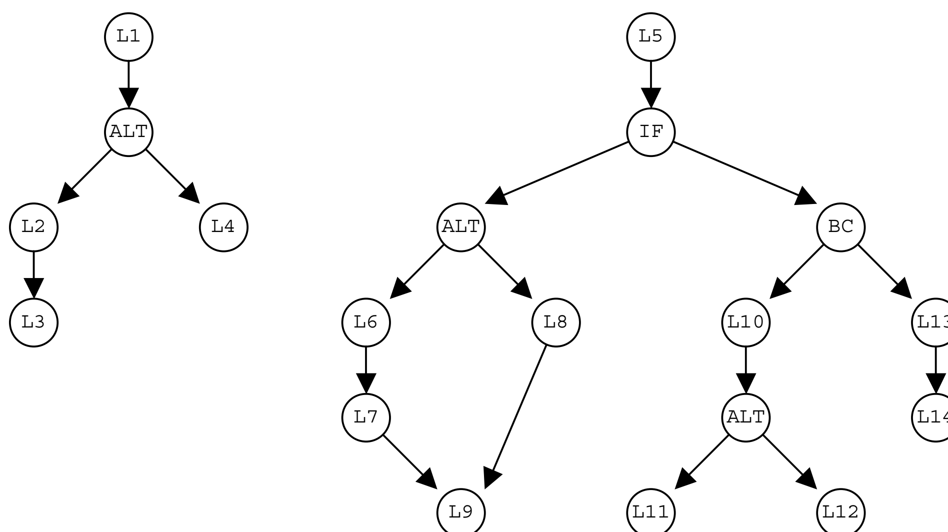
Оператор **interleave** может быть описан с помощью графа, состоящего из множества ориентированных субграфов, каждый из которых образуется посредством автономных операторов и составных операторов **if-else**, *blocking* **call** и **alt**. Ориентированные субграфы описывают чередующиеся потоки управления. Пример представлен на рисунке 6. Надписи в узлах на рисунке 6-b относятся к меткам операторов TTCN-3 на рисунке 6-a.

```
interleave {
        [] P1.receive(M1} {                               // L1
             alt {                                        // ALT
             [] P1.receive(M3) {                          // L2
                  setverdict(pass);                       // L3
             }
             [] T1.timeout { }                            // L4
        }

        }
        [] P2.receive(M2) {                               // L5
             if (x < 5 ) {                                // IF
             alt {                                        // ALT
                   [] P2.receive(M4) {                    // L6
                      setverdict(pass);                   // L7
                   }
                   [] Comp1.done { }                      // L8
                   }
                   x := 7 + 5;                            // L9
                   }
             else {
                   P3.call(MyProcTempl, 20E-3) {          // BC (= БЛОКИРУЮЩИЙ ВЫЗОВ)

                   [] P3.getreply(ReplyTempl) {           // L10
                      alt {                               // ALT
                      [] P2.receive(M5) { }               // L11
                      [] P2.receive(M6) { }               // L12
                      }
                      }
                   [] P3.catch(timeout) {                 // L13
                           setverdict(fail);              // L14
                           }
                      }
                   }
             }
}
```

a) Оператор interleave в TTCN-3



b) Графическое представление оператора в a

Рисунок 6/Z.143 – Графическое представление оператора interleave в TTCN-3

Формально оператор **`interleave`** может быть описан графом $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$, где:

$\boldsymbol{St}$	является множеством допустимых операторов TTCN-3; а
$\boldsymbol{F} \subseteq (\boldsymbol{St} \times \boldsymbol{St})$	описывает потоковую связь.

Название *допустимые операторы TTCN-3* относится к статическим ограничениям 1–5, выше.

Для алгоритма построения конструкций необходимо определить следующие функции:

- Функция *REACHABLE* возвращает все достижимые операторы из оператора $\boldsymbol{s}$ в графе $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$:

  REACHABLE $(\boldsymbol{s}, \boldsymbol{GI}) = \{ \boldsymbol{s} \} \cup \{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge \exists(\boldsymbol{s} = \boldsymbol{x_1}, \boldsymbol{x_2}, \ldots , \boldsymbol{x_n} = \boldsymbol{stmt})$, где $\boldsymbol{x_i} \in \boldsymbol{St}$, $i \in \{1...n\} \wedge (\boldsymbol{x_i}, \boldsymbol{x_{i+1}}) \in \boldsymbol{F}\}$

- Функция *SUCCESSORS* возвращает все последующие элементы оператора $\boldsymbol{s}$ в графе $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$:

  SUCCESSORS $(\boldsymbol{s}, \boldsymbol{GI}) = \{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge (\boldsymbol{s}, \boldsymbol{stmt}) \in \boldsymbol{F}\}$

- Функция *ENABLED* возвращает все операторы графа $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$, не имеющие предшествующих элементов:

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

- Функция *KIND* возвращает вид или тип оператора TTCN-3 в графе, представляющем оператор **`interleave`**.

- Функция *DISCARD* устраняет оператор $\boldsymbol{s}$ или множество операторов $\boldsymbol{S}$ из графа $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$ и возвращает полученный в результате граф $\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$:

  Для одиночных узлов:

  DISCARD $(\boldsymbol{s}, \boldsymbol{GI}) = \boldsymbol{GI'}$ где: $\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$, при $\boldsymbol{St'} = \boldsymbol{St}\backslash\{s\}$ и $\boldsymbol{F'} = \boldsymbol{F} \cap (\boldsymbol{St}\backslash\{s\} \times \boldsymbol{St}\backslash\{s\})$.

  Для множеств узлов:

  DISCARD $(\boldsymbol{S}, \boldsymbol{GI}) = \boldsymbol{GI'}$, где: $\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$ при $\boldsymbol{St'} = \boldsymbol{St}\backslash\boldsymbol{S}$ и $\boldsymbol{F'} = \boldsymbol{F} \cap (\boldsymbol{St}\backslash\boldsymbol{S} \times \boldsymbol{St}\backslash\boldsymbol{S})$.

- Функция *RECEIVING* получает множество операторов графа $\boldsymbol{GI}$ и возвращает все операторы приема:

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

- Функция *RANDOM* выбирает случайным образом элемент $\boldsymbol{s}$ из данного множества $\boldsymbol{S}$ и возвращает $\boldsymbol{s}$.

  RANDOM $(\boldsymbol{S}) = \boldsymbol{s}$, где $\boldsymbol{s} \in \boldsymbol{S}$

Алгоритм построения конструкций (см. рисунок 7) дерева является рекурсивной процедурой, где в каждом рекурсивном вызове образуются последующие узлы для данного узла. Процедура представлена в нотации псевдокода, напоминающего язык C, где используются описанные выше функции и некоторая дополнительная математическая нотация.

```
CONSTRUCT-SUCCESSORS (statementType *predecessor, graphType GI) {
    // - statementType относится к типу узла создаваемого дерева
    // - *predecessor относится к последнему созданному узлу
    // - graphType обозначает тип графа операторов TTCN-3
    // - GI вызывается значением и относится к субграфу, состоящему из всех остальных операторов
    // TTCN-3, которые необходимо принимать во внимание

    var graphType myGraph;
    var statementType i, myStmt;
    var statementType *newStmt, *firstInBranch; // указатели для новых операторных узлов в
                                                // рекурсивно создаваемом дереве

    // Поиск множеств операторов TTCN-3 без предшествующих элементов в 'GI'
    var statementSet enabStmts := ENABLED(GI);//все операторы без предшествующего элемента
    var statementSet enabRecStmts := RECEIVING(enabStmts);//операторы приема в 'enabStmts'
    var statementSet enabNonRecStmts := enabStmts\enabRecStmts;
                                                // операторы неприема в 'enabStmts'

    if (GI.St == ∅) {   // Предполагается, что GI.St относится к набору операторов в GI
        return;         // Никакие операторы не потеряны,критерий завершения рекурсии
    }
    elseif (enabNonRecStmts != ∅) { // Обработка операторов неприема в 'enabStmts'

        myStmt := RANDOM(enabNonRecStmts);
                // Только один оператор может быть в'enabNonRec', поскольку Алгоритм
                // продолжает конструкцию до тех пор, пока не появится ветвь для
                // оператора чередования.
        newStmt := create(myStmt, predecessor);
                // Создание нового узла дерева, представляющего'myStmt' в дереве, и обновление
                // указателей в 'newStmt' и 'predecessor'.

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

                firstInBranch := create(i, newStmt);
                    // Создание второго узла для первого оператора в ветви из-за
                    // оператора if-else.
                    // Заметим, что этот оператор создания будет использован для создания узлов
                    // дерева, представляющих операторы приема в блокирующих операциях call.

                myGraph := DISCARD({i, myStmt} ∪ REACHABLE(myStmt, GI)\REACHABLE(i, GI))
                    // Удаление i, myStmt и всех операторов, достижимых из
                    // myStmt , но недостижимых из i. Последний учитывает ветвление потока
                    // управления в субграфе графа GI.

                CONSTRUCT-SUCCESSORS(firstInBranch, myGraph);// СЛЕДУЮЩИЙ РЕКУРСИВНЫЙ ШАГ
            }
        }
        elseif (KIND(myStmt) == ALT) {
            for each (i in SUCCESSORS(myStmt, GI) {

            CONSTRUCT-SUCCESSORS(mystmt, DISCARD(REACHABLE(myStmt, GI)\REACHABLE(i, GI)));
                // СЛЕДУЮЩИЙ РЕКУРСИВНЫЙ ШАГ, the DISCARD(REACHABLE(myStmt, GI)\REACHABLE(i, GI))
                        // аргумент учитывает ветвление потока управления из-за
                        // различных событий приема.
            }
        }
        else {      // mystmt является автономным оператором
            CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, GI));
                // СЛЕДУЮЩИЙ РЕКУРСИВНЫЙ ШАГ
        }
    }
    else {  // Обработка событий приема,которые чередуются
        if (KIND(predecessor) != ALT) { // alt-узел отсутствует и должен быть создан, если
                                        // на чередование не влияет вложенный оператор alt
            predecessor := create(ALT, predecessor);
        }

        for each i in enabRecStmts) {
            newStmt := create(i, predecessor);              // Новый узел дерева
            CONSTRUCT-SUCCESSORS(newStmt, DISCARD(i, GI));  // СЛЕДУЮЩИЙ РЕКУРСИВНЫЙ ШАГ(S)
        }
    }
}
```

Рисунок 7/Z.143 – Алгоритм замены для операторов interleave в TTCN-3

Сначала функция CONSTRUCT-SUCCESSORS (см. рисунок 7) будет вызываться с *корневым узлом* пустого дерева и графом операторов TTCN-3, описывающих оператор **`interleave`**, который будет заменен. По завершении, для доступа к созданному дереву может быть использован *корневой узел*.

Применение функции CONSTRUCT-SUCCESSORS к оператору **`interleave,`** показанное на рисунке 6, приводит к дереву, показанному на рисунке 8. Метки относятся к операторам на рисунке 6-a. Множественные метки являются результатом дублирования кода. Код TTCN-3, соответствующий дереву на рисунке 8, представлен на рисунке 9.

ПРИМЕЧАНИЕ 5. – Пример для применения алгоритма на рисунке 7 (см. рисунки 6, 8 и 9) является весьма содержательным. Этот пример приведен для того, чтобы показать множество особых ситуаций, т. е. ветвление и слияние потоковых линий, вложенный оператор **`alt`**, оператор *blocking* **`call`** и оператор **`if-else`**.

Рисунок 8/Z.143 – Результат применения алгоритма на рисунке 7 к оператору interleave на рисунке 6

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

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

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

Рисунок 9/Z.143 – Семантически эквивалентный код TTCN-3 для оператора interleave на рисунке 6

7.6 Замена операций trigger

Операция **trigger** фильтрует сообщения по определенному критерию сопоставления из потока сообщений на данном порте. Семантика операции **trigger** может быть описана путем ее замены на две операции **receive** и оператор **goto**. В операционной семантике предполагается, что эта замена осуществляется на синтаксическом уровне.

ПРИМЕР 1:

```
// Следующая операция trigger ...

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

// будет заменена на ...

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

Если оператор **`trigger`** используется в более сложном операторе **`alt,`** то замена выполняется таким же образом.

ПРИМЕР 2:

```
// Следующий оператор alt включает оператор trigger ...

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

// который будет заменен на

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

8 Семантика потокового графа в TTCN-3

Операционная семантика TTCN-3 базируется на интерпретации потоковых графов. В данном разделе вводятся потоковые графы (см. п. 8.1), объясняется конструкция потоковых графов, представляющих определения управления модулем TTCN-3, тестовых примеров, альтернативных шагов, функций и компонентных типов (см. п. 8.2), определяются состояния модуля и компонентов для описания состояний выполнения модуля TTCN-3 (см. п. 8.3), приводится описание обработки сообщений, удаленных вызовов процедур, ответов на удаленные вызовы процедур и особых состояний (см. п. 8.4) и объясняется оценочная процедура управления модулем и тестовых примеров (см. п. 8.6).

8.1 Потоковые графы

Потоковый граф – это ориентированный граф, состоящий из помеченных узлов и помеченных ребер. При обходе потокового графа описывается возможный поток управления при выполнении представленного описания поведения.

8.1.1 Рамка потокового графа

Потоковый граф помещается в рамку, определяющую границу потокового графа. Имя потокового графа следует за ключевыми словами **flow graph** (это не ключевые слова базового языка TTCN-3) и помещается в левый верхний угол потокового графа. В качестве соглашения предполагается, что имя потокового графа относится к описанию поведения TTCN, представленному потоковым графом. На рисунке 10 показан простой потоковый граф.

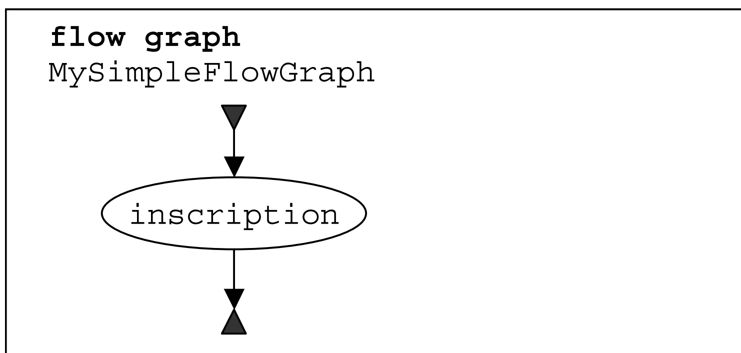


Рисунок 10/Z.143 – Простой потоковый граф

8.1.2 Узлы потокового графа

Потоковые графы состоят из *начальных узлов, конечных узлов, базовых узлов* и *ссылочных узлов.*

8.1.2.1 Начальные узлы

Начальные узлы описывают начальную точку потокового графа. Потоковый граф имеет только один начальный узел. На рисунке 11-a показан начальный узел.

a) Начальный узел потокового графа	b) Конечный узел потокового графа

Рисунок 11/Z.143 – Начальный и конечный узлы

8.1.2.2 Конечные узлы

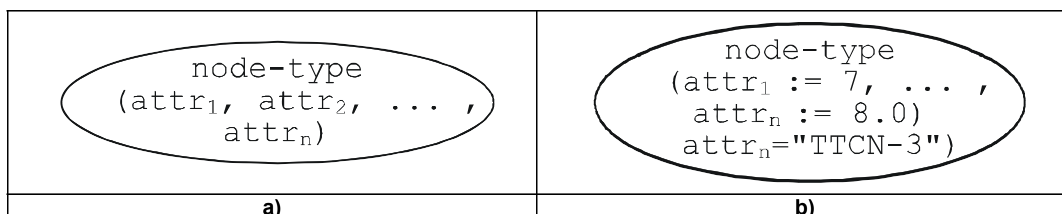
Конечные узлы описывают конечные точки потокового графа. Потоковый граф может иметь несколько конечных узлов или, в случае петель, ни одного конечного узла. Базовые узлы (см. п. 8.1.2.3) и ссылочные узлы (см. п. 8.1.2.4), не имеющие последующих узлов, соединяются с конечным узлом для указания на то, что они описывают последнее действие на пути через потоковый граф. На рисунке 11-b показан конечный узел.

8.1.2.3 Базовые узлы

Базовый узел описывает единицу выполнения, т. е. она выполняется за один шаг. Базовый узел имеет тот или иной тип и, в зависимости от этого типа, может иметь соответствующий список атрибутов. На рисунке 12 показаны два базовых узла.

В надписи базового узла атрибуты узла следуют за типом узла и помещаются в круглые скобки. Тип и атрибуты используются для определения действия, которое необходимо произвести во время выполнения представленной языковой конструкции. Атрибуты описывают информацию, запрашиваемую из соответствующей конструкции TTCN-3.

Атрибуты имеют значения, и операционная семантика будет запрашивать эти значения путем ссылок на имя атрибута. Если требуется, в базовых узлах можно присваивать явные значения с помощью присвоения ':='. Пример показан на рисунке 12-b.



a)	b)

Рисунок 12/Z.143 – Базовые узлы с атрибутами

8.1.2.4 Ссылочные узлы

Ссылочные узлы отсылают к сегментам потокового графа (см. п. 8.1.4), которые являются потоковыми субграфами. Смысл ссылочного узла определяется путем его замены в потоковом графе на указанный сегмент потокового графа. Надпись узла в ссылочном узле дает ссылку на сегмент потокового графа. На рисунке 13-а показан ссылочный узел.

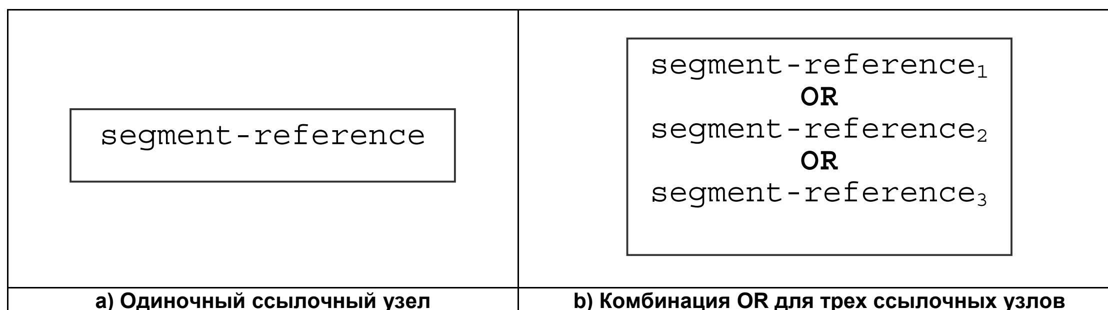


Рисунок 13/Z.143 – Ссылочный узел

8.1.2.4.1 Комбинация OR для ссылочных узлов

В ряде случаев несколько сегментов потокового графа могут заменить ссылочный узел. В этих случаях для ссылки на несколько сегментов потокового графа может быть использован оператор **OR** (см. рисунок 13-b). В реальном потоковом графе, представляющем управление модулем, тестовый пример или функцию, представленной конструкцией определяется одна альтернатива.

8.1.2.4.2 Множественные вхождения ссылочных узлов

В ряде случаев ссылочный узел одного и того же типа может появиться в потоковом графе нуль, один или несколько раз. Что касается регулярных выражений, то возможное повторение частей регулярного выражения описывается с помощью символов оператора '+' (одно или несколько повторений) и '*' (нуль или несколько повторений). Как показано на рисунке 14, эти операторы были приспособлены к потоковым графам путем введения ссылочных узлов с двойной рамкой, содержащей соответствующие символы операторов. Одиночная потоковая линия (см. п. 8.1.3) заменяет ссылочный узел в случае нулевого числа вхождений (используя ссылочный узел с двойной рамкой и с оператором '*').

Рисунок 14/Z.143 – Повторение ссылочных узлов

Верхняя граница возможных повторений ссылочного узла может быть задана в виде целого числа в круглых скобках после символа '*' или '+' в ссылочном узле с двойной рамкой. Ссылка на сегмент, показанная на рисунке 15, может появиться от нуля до пяти раз.

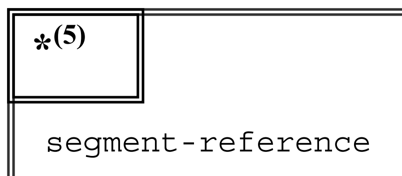
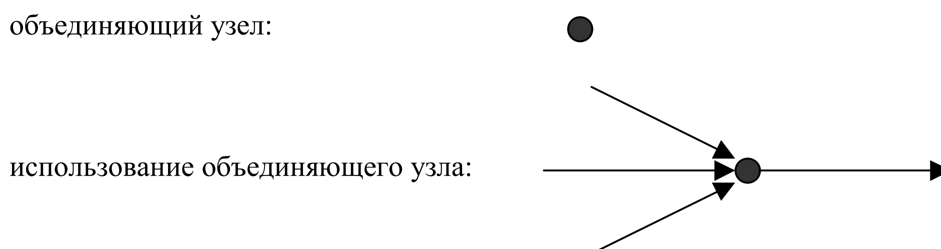


Рисунок 15/Z.143 – Ограниченное число повторений ссылочного узла

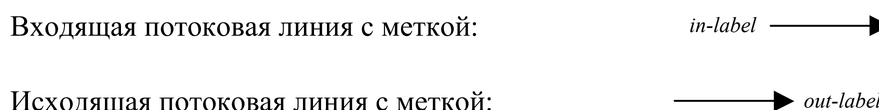
8.1.3 Потоковые линии

Потоковые линии представляются с помощью стрелок. Потоковая линия имеет надпись *true* или *false*, которая указывает на состояние, при котором происходит выбор этой потоковой линии во время интерпретации потокового графа. Допускается в качестве краткой нотации опускать надпись true. Примеры потоковых линий приведены ниже:

Для поддержки объединения нескольких потоковых линий в одну потоковую линию на графическом уровне вводится специальный объединяющий узел. Объединяющий узел и пример его использования показаны ниже:



Вычерчивание длинных потоковых линий на больших диаграммах, необходимых, например, для моделирования конструкций **`goto`** и **`label`** в TTCN-3, затруднительно. С этой целью для исходящих и входящих потоковых линий могут использоваться метки. Примеры показаны ниже.



Исходящая потоковая линия с меткой соединяется с входящей потоковой линией с меткой, если эти метки идентичны. Метки для входящих потоковых линий являются уникальными. Если есть несколько исходящих потоковых линий с одной и той же меткой, то это рассматривается как объединение линий с входящей потоковой линией, имеющей идентичную метку.

8.1.4 Сегменты потокового графа

Сегменты потокового графа – это потоковые субграфы. На них ссылаются в ссылочных узлах, и они определяют смысл этого ссылочного узла. Сегменты потокового графа могут содержать дальнейшие ссылочные узлы.

Как показано на рисунке 16, сегменты потокового графа имеют определенные интерфейсы, состоящие из входящих и исходящих потоковых линий. 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. Например, входящие и исходящие потоковые линии с метками нужны для описания смысла операторов **`goto`** и **`alt`** в TTCN-3.

Сегменты потокового графа заключаются в рамку, а имя сегмента потокового графа предваряется ключевым словом **`segment,`** за которым следует имя сегмента в верхнем левом углу рамки. Потоковые линии, описывающие интерфейс сегмента потокового графа, пересекают рамку сегмента потокового графа.

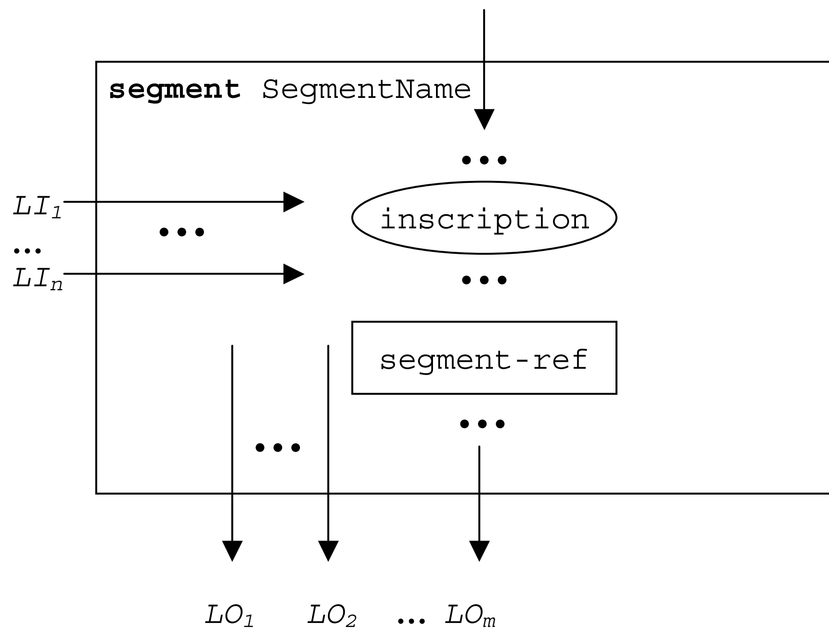


Рисунок 16/Z.143 – Структура описания сегмента потокового графа

8.1.5 Комментарии

Для улучшения согласованности и удобочитаемости может использоваться специальный символ комментария, который связывает комментарии с узлами потокового графа и потоковыми линиями. Символ комментария и его использование показаны на рисунке 17.

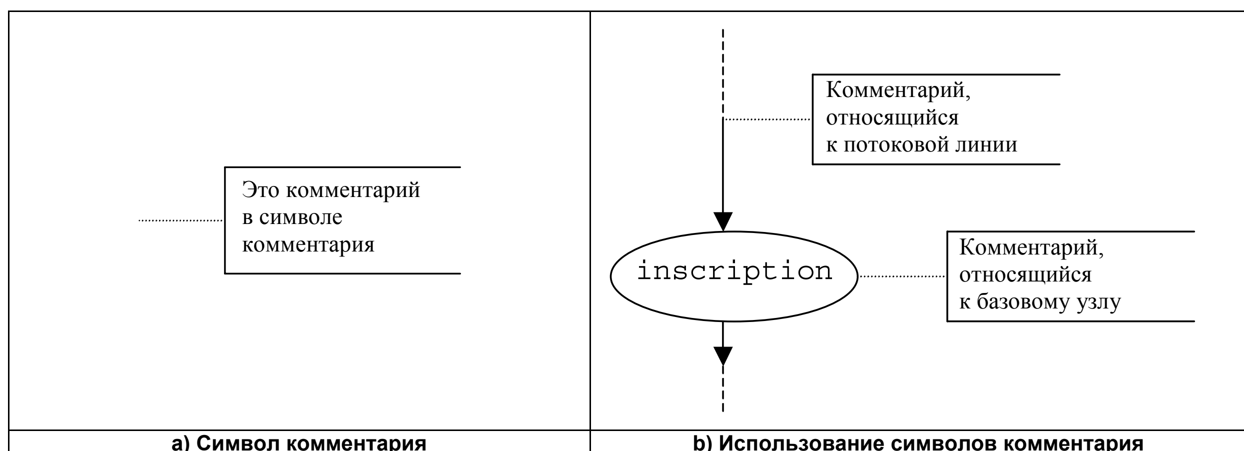


Рисунок 17/Z.143 – Представление комментариев в виде потокового графа

8.1.6 Обработка описаний потокового графа

Процедура оценки в операционной семантике предполагает просмотр потоковых графов, содержащих только базовые узлы, т. е. все ссылочные узлы должны быть дополнены соответствующими определениями сегментов потокового графа. Для поддержки такого просмотра требуется функция <u>*NEXT*</u>, которая определяется следующим образом:

actualNodeRef.<u>*NEXT*</u>(*bool*) := `successorNodeRef`, где:

- *actualNodeRef* является ссылкой базового узла потокового графа;
- *successorNodeRef* является ссылкой последующего узла для узла, на который ссылаются с помощью *actualNodeRef*;
- *bool* является булевой функцией, определяющей, возвращается ли последующее значение *true* или *false* (см. п. 8.1.3).

8.2 Представление поведения TTCN-3 в виде потоковых графов

В операционной семантике предполагается, что описания поведения TTCN-3 даны в виде множества потоковых графов, т. е. для каждого описания поведения TTCN-3 должен быть сконструирован отдельный потоковый граф.

В операционной семантике в качестве описаний поведения интерпретируются следующие виды определений TTCN-3:

a) управление модулем;

b) определения тестовых примеров;

c) определения функций;

d) определения альтернативных шагов;

e) определения компонентных типов.

Управление модулем определяет проведение тестирования, т. е. порядок выполнения (возможно, повторного) фактических тестовых примеров. Определения тестовых примеров характеризуют поведение MTC. Функции формируют структуру поведения. Они выполняются с помощью управления модулем или тестовых компонентов. Альтернативные шаги используются для определения поведения по умолчанию или для структурирования поведения способом, подобным применению функций. Предполагается, что определения компонентных типов являются описаниями поведения, поскольку они задают создание, объявление и инициализацию портов, констант, переменных и таймеров во время создания экземпляра того или иного компонентного типа.

8.2.1 Процедура конструкции потокового графа

Потоковые графы, представленные на рисунках 18–22, и сегменты потокового графа, представленные в разделе 8, являются только шаблонами. Они содержат *заполнители* для информации, которая должна быть предоставлена для построения конкретного потокового графа или сегмента потокового графа. Заполнители отмечаются скобками '<' и '>.

Конструкция для представления потокового графа модуля TTCN-3 выполняется за три шага:

1) Для каждого оператора TTCN-3 в определениях управления модулем, тестовых примеров, альтернативных шагов, функций и компонентных типов конструируется конкретный сегмент потокового графа.

2) Для управления модулем и для каждого определения тестового примера, альтернативного шага, функции и компонентного типа конструируется конкретный потоковый граф (со ссылочными узлами).

3) В пошаговой процедуре все ссылочные узлы в конкретных потоковых графах заменяются на соответствующие определения сегментов потокового графа, до тех пор пока все потоковые графы не будут содержать только один начальный узел, оконечные узлы и базовые узлы потокового графа.

ПРИМЕЧАНИЕ 1. – Базовые узлы потокового графа описывают базовые неделимые исполнительные модули. Операционная семантика для поведения TTCN-3 базируется на интерпретации базовых узлов потокового графа. В разделе 8.6 представлены методы выполнения только для базовых узлов потокового графа.

Замена ссылочного узла на соответствующее определение сегмента потокового графа может привести к несвязанным частям в потоковом графе, т. е. к частям, которых нельзя достичь из начального узла путем обхода потокового графа по потоковым линиям. В операционной семантике несвязанные части потокового графа будут игнорироваться.

ПРИМЕЧАНИЕ 2. – Несвязанная часть потокового графа является результатом процедуры механической замены. Для конструкции оптимального представления в виде потокового графа должны приниматься во внимание также различные комбинации операторов TTCN-3. Однако целью настоящей Рекомендации является предоставление корректной и полной семантики, а не оптимального представления в виде потокового графа.

8.2.2 Представление управления модулем в виде потокового графа

Синтаксическая структура модуля TTCN-3 схематически имеет следующий вид:

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

Для представления поведения потокового графа существенна только следующая информация:

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

Это сравнимо с определением функции, и поэтому представление управления модулем в виде потокового графа подобно представлению функции в виде потокового графа (см. п. 8.2.4). В семантике имеет место доступ к потоковому графу, представляющему управление модулем, путем использования имени модуля.

ПРИМЕЧАНИЕ. – Смысл части определений модуля выходит за рамки данной операционной семантики. Параметры модуля определяются как глобальные константы на этапе выполнения. Ссылки на параметры модуля должны заменяться на их конкретные значения на синтаксическом уровне (см. п. 8.3).

На рисунке 18 показана схема представления управления модулем в виде потокового графа. Имя потокового графа control определяет потоковый граф, представляющий управление модулем. Узлы потокового графа имеют соответствующие комментарии, описывающие смысл различных узлов. Ссылочный узел <stop-entity-op> относится к случаю, когда операция **stop** явно не задана, т. е. в операционной семантике предполагается, что операция **stop** добавляется неявным образом.

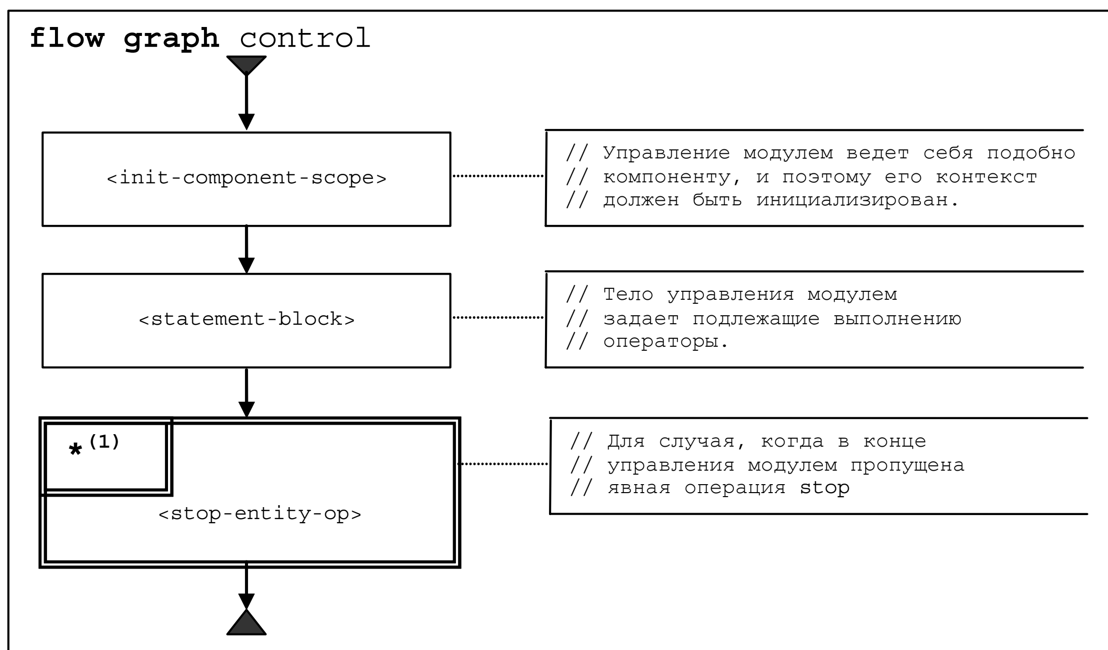


Рисунок 18/Z.143 – Представление управления модулем в виде потокового графа

8.2.3 Представление тестовых примеров в виде потокового графа

Синтаксическая структура определения тестового примера TTCN-3 схематически имеет следующий вид:

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

Приведенный выше <testcase-interface> относится к разделу **runs on** (обязательному) и разделу **system** (факультативному) в определении тестового примера. В описании тестового примера в виде потокового графа приводится поведение MTC. Информация, предоставляемая в <testcase-interface>, несущественна для MTC. Она будет использована оператором **execute**, но нет необходимости указывать ее в представлении тестового примера в виде потокового графа. Таким образом, для представления в виде потокового графа существенна только следующая информация:

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

На рисунке 19 показана схема представления тестового примера в виде потокового графа. Имя потокового графа <identifier> относится к имени представленного тестового примера. Узлы потокового графа имеют соответствующие комментарии, описывающие смысл различных узлов. Ссылочный узел <stop-entity-op> относится к случаю, когда операция stop для MTC явно не задана, т. е. в операционной семантике предполагается, что операция **stop** добавляется неявным образом.

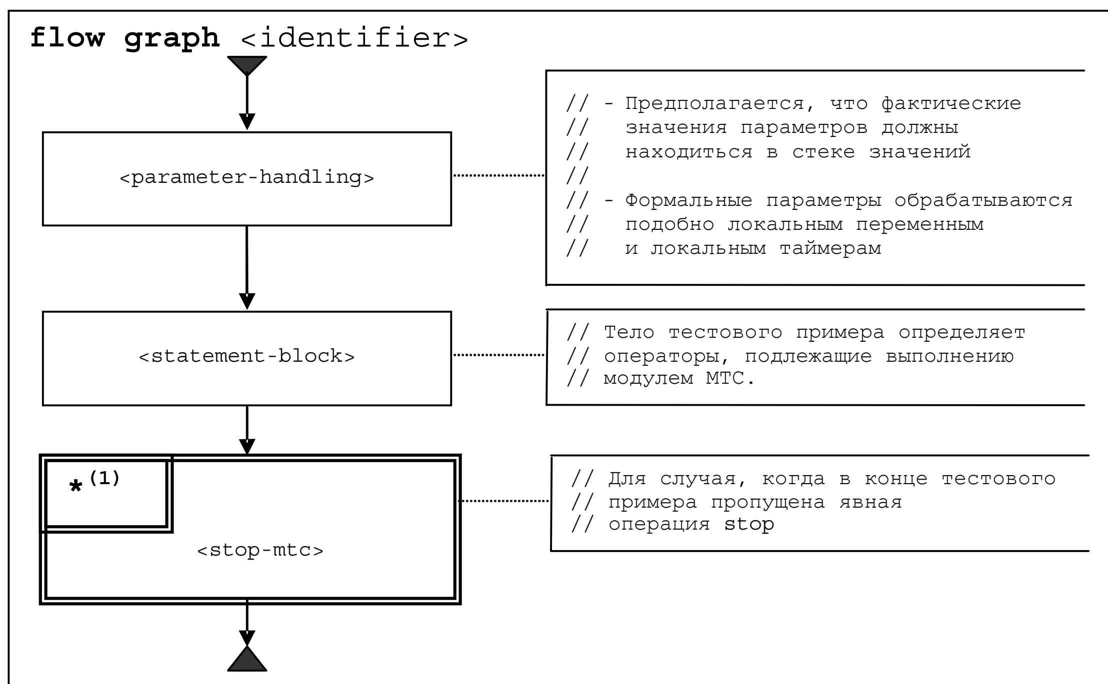


Рисунок 19/Z.143 – Представление тестовых примеров в виде потокового графа

8.2.4 Представление функций в виде потоковых графов

Синтаксическая структура функции TTCN-3 схематически имеет следующий вид:

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

Приведенный выше факультативный <function-interface> относится к разделам **runs on** и **return** в определении функции. Информация, предоставляемая посредством в <function-interface>, несущественна для описания поведения. Она будет использована для проверок статической семантики и не требует своего представления в потоковом графе. Таким образом, для представления в виде потокового графа существенна только следующая информация:

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

В семантике будет осуществляться доступ к потоковым графам, представляющим функции путем использования имен функций.

На рисунке 20 показано представление функции в виде потокового графа. Имя потокового графа `<identifier>` относится к имени представляемой функции. Ссылочный узел `<return-without-value>` относится к случаю, когда не задан явный оператор **`return`**, т. е. в операционной семантике предполагается, что оператор **`return`** добавляется неявным образом.

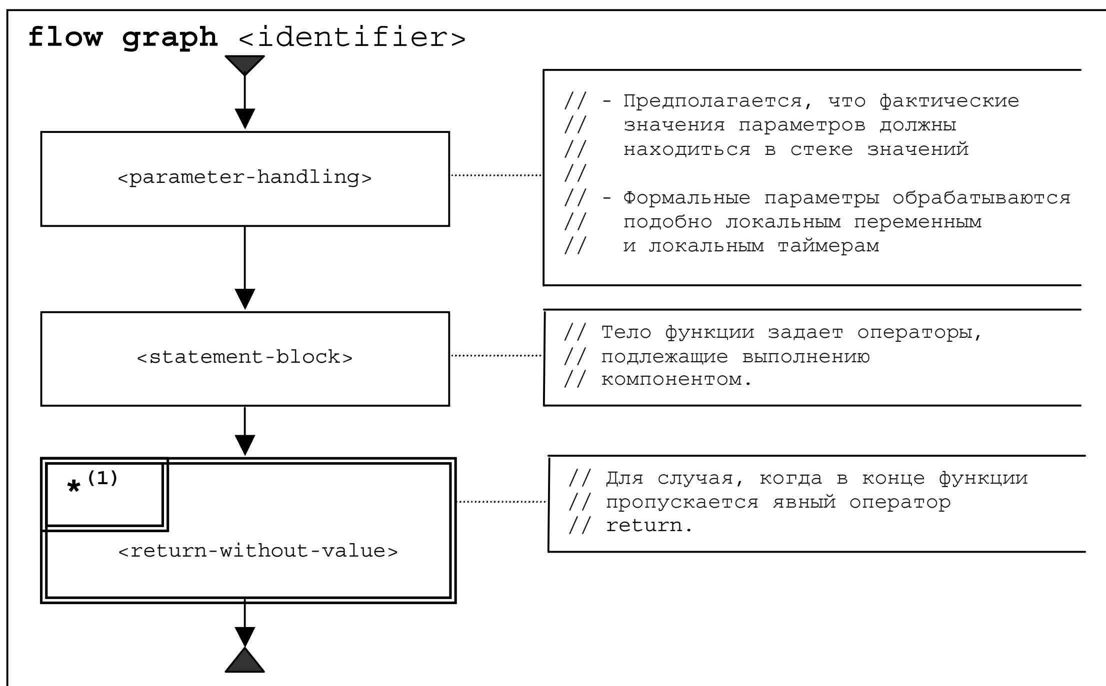


Рисунок 20/Z.143 – Представление функций в виде потоковых графов

8.2.5 Представление альтернативных шагов в виде потоковых графов

Синтаксическая структура альтернативного шага TTCN-3 схематически имеет следующий вид:

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

Приведенный выше факультативный `<altstep-interface>` относится к разделу **`runs on`** в определении альтернативного шага. Информация, предоставляемая посредством `<altstep-interface>`, несущественна для описания поведения. Она будет использована для проверок статической семантики, и ее не надо представлять в потоковом графе. Таким образом, для представления в виде потокового графа существенна только следующая информация:

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

ПРИМЕЧАНИЕ. – Принимаются во внимание только альтернативы до первой ветви else и первая ветвь else. Ветви, следующие за первой ветвью else, являются недостижимыми.

В семантике будет осуществляться доступ к потоковым графам, представляющим альтернативные шаги, путем использования имен альтернативных шагов.

На рисунке 21 показана схема представления альтернативного шага в виде потокового графа. Имя потокового графа `<identifier>` относится к имени представляемого альтернативного шага. Ссылочный узел `<successful-altstep-termination>` учитывает случай, когда альтернативный шаг завершается после выбора и выполнения альтернативы. Ссылочный узел `<unsuccessful-altstep-termination>` определяет случай, когда не выполнялась никакая альтернатива альтернативного шага.

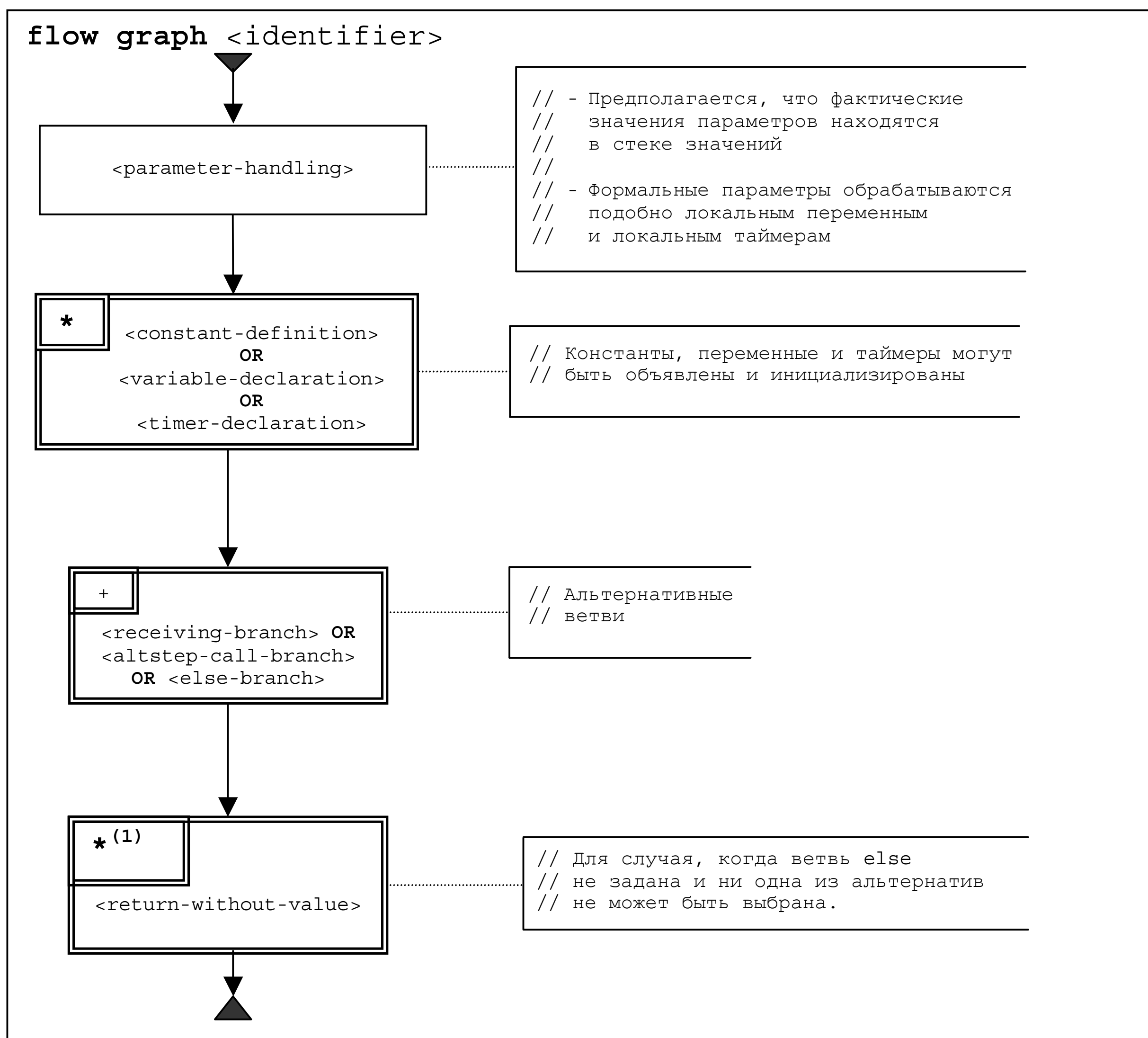


Рисунок 21/Z.143 – Представление альтернативных шагов в виде потоковых графов

8.2.6 Представление определений компонентных типов в виде потоковых графов

Синтаксическая структура определения компонентного типа TTCN-3 схематически имеет следующий вид:

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

В семантике будет осуществляться доступ к потоковым графам, представляющим типы, путем использования имен компонентных типов.

На рисунке 22 показана схема представления того или иного определения компонентного типа в виде потокового графа. Имя потокового графа `<identifier>` относится к имени представляемого компонентного типа.

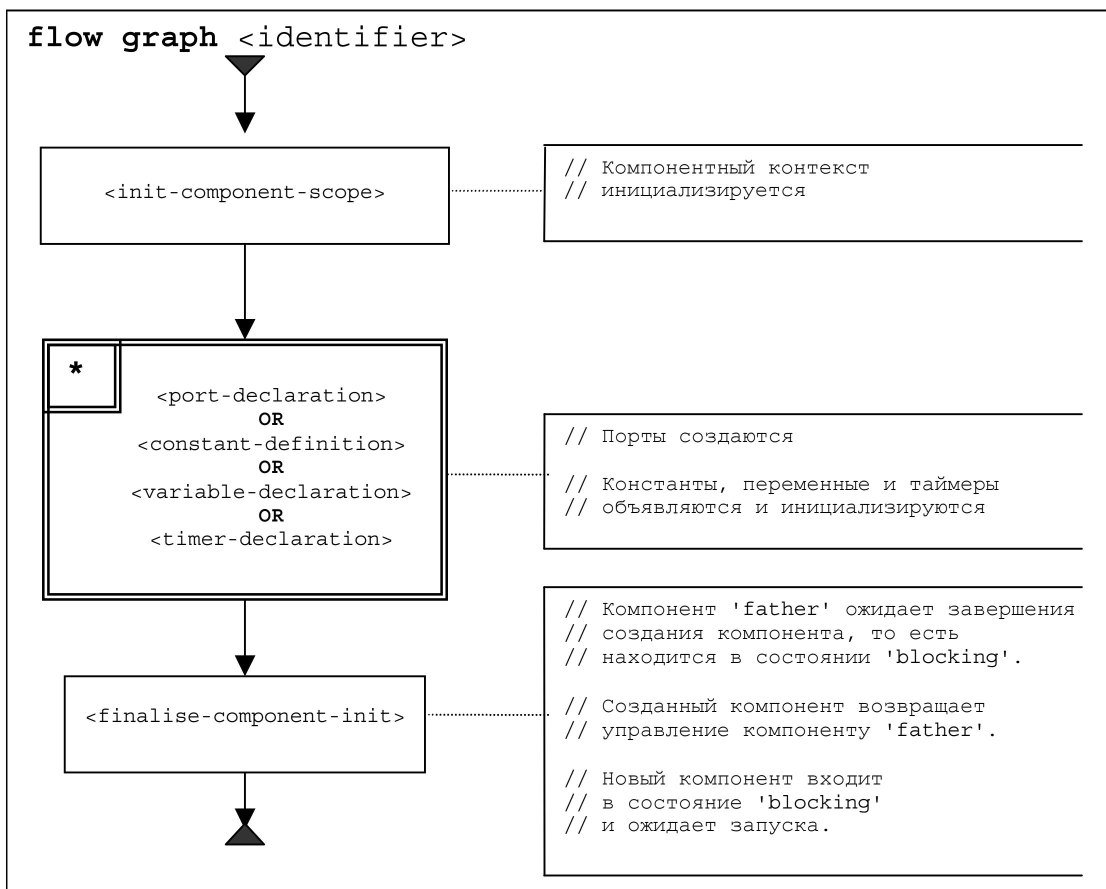


Рисунок 22/Z.143 – Представление определений компонентных типов в виде потоковых графов

8.2.7 Поиск начальных узлов потоковых графов

Для поиска ссылки на начальный узел потокового графа необходима следующая функция:

Функция <u>GET-FLOW-GRAPH</u>: <u>GET-FLOW-GRAPH</u> (flow-graph-identifier)

Эта функция возвращает ссылку на начальный узел потокового графа с именем *flow-graph-identifier*. Этот *flow-graph-identifier* указывает на имя модуля для управления, имена тестовых примеров, имена функций и определения компонентных типов.

8.3 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 включает в себя несколько подсостояний, описывающих состояния тестовых компонент и портов. В данном разделе вводятся состояния модуля, состояния компонентов и состояния порта. Кроме того, определяются функции вывода информации от манипулируемых состояний и к ним.

8.3.1 Состояние модуля

Как показано на рисунке 23, состояние модуля подразделяется на <u>*ALL-ENTITY-STATES*</u>, <u>*ALL-PORT-STATES*</u>, <u>*MTC*</u>, <u>*TC-VERDICT*</u>, <u>*DONE*</u> и <u>*SNAP-ACTIVE*</u>. <u>*ALL-ENTITY-STATES*</u> описывает состояние управления модулем и состояния создаваемых тестовых компонентов во время выполнения тестового примера. <u>*ALL-PORT-STATES*</u>, ссылка на <u>*MTC*</u> и <u>*TC-VERDICT*</u> действуют только во время выполнения тестового примера. <u>*ALL-PORT-STATES*</u> описывает состояния различных портов. <u>*MTC*</u> выдает ссылку на главный тестовый компонент (MTC). <u>*TC-VERDICT*</u> хранит фактический глобальный тестовый вердикт тестового примера, <u>*DONE*</u> – это список всех тестовых компонентов, остановленных во время выполнения тестового примера, а <u>*SNAP-ACTIVE*</u> используется как часть процедуры фиксации мгновенного состояния MTC. <u>*SNAP-ACTIVE*</u> хранит число активных тестовых компонентов, когда MTC фиксирует мгновенное состояние. Это используется для оценки операций **`all component`**.**`done`** и **`all component`**.**`running`**.

ПРИМЕЧАНИЕ 1. – Число обновлений TC-VERDICT равно числу тестовых компонентов, которые были завершены.

Поведение управления модулем (*M-CONTROL* на рисунке 23) обрабатывается так же, как обычный тестовый компонент, а его состояние является первым элементом в *ALL-ENTITY-STATES* состояния модуля.

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

Рисунок 23/Z.143 – 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 в данной операционной семантике порты обрабатываются на верхнем уровне состояния модуля.

8.3.1.1 Доступ к состоянию модуля

MTC, *TC-VERDICT* и *SNAP-ACTIVE* являются частями состояния модуля и обрабатываются подобно глобальным переменным, т. е. ключевые слова *MTC* и *TC-VERDICT* могут использоваться для вывода 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 глобально определенные уникальные идентификаторы для процедуры оценки.

Для обработки списков *ALL-ENTITY-STATES*, *ALL-PORT-STATES* и *DONE* может использоваться список операций *add*, *append*, *delete*, *member*, *first*, *length*, *next*, *random* и *change*. Они имеют следующий смысл:

- *myList.add*(*item*) добавляет *item* в качестве первого элемента в список *myList*;
- *myList.append*(*item*) добавляет *item* в качестве последнего элемента в список *myList*;
- *myList.delete(item) удаляет item из списка myList;*
- *myList.member*(*item*) возвращает **true,** если *item* является элементом списка *myList*, в противном случае возвращает **false**;
- *myList.first*() возвращает первый элемент из *myList*;
- *myList.length*() возвращает длину из *myList*;
- *myList.next*(*item*) возвращает элемент, который следует за *item* в *myList*, или **NULL**, если *item* является последним элементом в *myList*;
- *MyList.random*(<*condition*>) возвращает случайным образом элемент из *myList*, который выполняет булево условие <*condition*> или **NULL**, если никакой элемент из *myList* не выполняет <*condition*>;
- *MyList.change*(<*operation*>) позволяет применить <*operation*> ко всем элементам из *myList*.

ПРИМЕЧАНИЕ 2. – Операции random и change не являются общими списочными операциями. Они вводятся для объяснения смысла ключевых слов **all** и **any** в операциях TTCN-3.

8.3.2 Состояния объектов

Состояния объектов используются для описания фактических состояний управления модулем и тестовых компонентов. В состоянии модуля состояния объекта обрабатываются в списке *ALL-ENTITY-STATES*. На рисунке 24 показана структура состояния объекта.

<identifier>	STATUS	CONTROL-STACK	DEFAULT-LIST	DEFAULT-POINTER	VALUE-STACK	E-VERDICT	TIMER-GUARD	DATA-STATE	TIMER-STATE	SNAP-DONE

Рисунок 24/Z.143 – Структура состояния объекта

Здесь <*identifier*> является уникальным идентификатором объекта, т. е. управления модулем для тестового компонента в тестовой системе. Такие уникальные идентификаторы создаются неявным образом для управления модулем, для **mtc** и тестовой системы **system**, когда модуль начинает выполнение или тестовый пример выполняется с помощью оператора **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, находится ли управление модулем или тестовый компонент в состоянии **ACTIVE**, **SNAPSHOT**, **REPEAT** или **BLOCKED**. Управление модулем блокируется во время выполнения тестового примера. Тестовые компоненты могут блокироваться во время создания других тестовых компонентов, т. е. во время выполнения операции **create**. Состояние **SNAPSHOT** указывает на то, что компонент активен, но находится в фазе оценки фиксации мгновенного состояния. Состояние **REPEAT** означает, что компонент активен и находится в операторе alt, который должен быть оценен повторно из-за оператора **repeat**.

Часть *CONTROL-STACK* является стеком ссылок на узлы потокового графа. Верхним элементом в *CONTROL-STACK* является узел потокового графа, который должен быть интерпретирован следующим. Этот стек требуется для моделирования соответствующим способом вызовов функции.

Часть DEFAULT-LIST является списком активированных значений по умолчанию, т. е. это список указателей, которые указывают на начальные узлы активированных значений по умолчанию. Этот список составлен в обратном порядке активации, т. е. значение, которое было активировано первым, является последним элементом в списке.

Во время использования механизма по умолчанию DEFAULT-POINTER указывает на следующее значение по умолчанию, которое должно быть оценено, если фактическая оценка значения по умолчанию завершилась неуспешно.

Часть VALUE-STACK является стеком значений всех возможных типов, допускающим промежуточное хранение конечных или промежуточных результатов операций, функций или операторов. Например, результат оценки выражения или результат операции **mtc** будет занесен в VALUE-STACK. В дополнение к значениям всех типов данных, известных в модуле, определяется особое значение **MARK**, являющееся частью стекового алфавита. Значение **MARK** используется при выходе из контекстного блока для очистки VALUE-STACK.

Часть E-VERDICT хранит фактический локальный вердикт тестового компонента. Если состояние объекта представляет управление модулем, то E-VERDICT игнорируется.

Часть TIMER-GUARD представляет специальный таймер, который необходим для защиты времени выполнения тестовых примеров и продолжительности операций вызова. TIMER-GUARD моделируется как связывание таймеров (см. п. 8.3.2.4 и рисунок 28).

Часть DATA-STATE рассматривается как перечень списков связываний переменных. Структура перечня списков отражает вложенные контекстные блоки, вызванные вложенными вызовами функций. Каждый список из перечня списков связываний переменных описывает известные переменные и их значения в определенном контекстном блоке. Вход в контекстный блок или выход из него соответствует прибавлению или удалению списка связываний переменных из DATA-STATE. Описание части DATA-STATE состояния объекта можно найти в п. 8.3.2.2.

Часть TIMER-STATE рассматривается как перечень списков связываний таймеров. Структура перечня списков отражает вложенные контекстные блоки, вызванные вложенными вызовами функций. Каждый список из перечня списков связываний таймеров описывает известные таймеры и их состояние в определенном контекстном блоке. Вход в контекстный блок или выход из него соответствует прибавлению или удалению списка состояний таймера из части *timer state* (состояние таймера). Описание части *timer state* состояния объекта можно найти в п. 8.3.2.4.

Часть SNAP-DONE поддерживает семантику фиксации мгновенного состояния для тестовых компонентов. Когда фиксируется мгновенное состояние, копия списка DONE состояния модуля присваивается части SNAP-DONE, т. е. SNAP-DONE является списком компонентных идентификаторов остановленных компонентов.

8.3.2.1 Доступ к состояниям объекта

Здесь <*identifier*> является уникальным идентификатором состояния объекта, который может использоваться для доступа к компоненту, представленному состоянием объекта и различными частями состояния объекта.

Части STATUS, DEFAULT-POINTER, E-VERDICT и TIMER-GUARD состояния объекта обрабатываются подобно глобально видимым переменным, т. е. значения частей STATUS, DEFAULT-POINTER и E-VERDICT могут быть выведены или изменены путем использования нотации "dot", например, *myEntity*.STATUS, *myEntity*.DEFAULT-POINTER и *myEntity*.E-VERDICT, где *myEntity* указывает на состояние объекта.

ПРИМЕЧАНИЕ. – В дальнейшем предполагается, что нотацию "dot" можно применять путем использования ссылок и уникальных идентификаторов. Например, в *myEntity*.STATUS, *myEntityState* может быть указатель на состояние объекта или может быть значение поля <*identifier*>.

Части CONTROL-STACK, DEFAULT-LIST и VALUE-STACK состояния объекта *myEntity* могут быть адресованы путем использования 'dot'-нотации *myEntity*.CONTROL-STACK, *myEntity*.DEFAULT-LIST и *myEntity*.VALUE-STACK.

К частям CONTROL-STACK и VALUE-STACK может быть обеспечен доступ, и манипулировать ими можно путем использования стековых операций *push*, *pop*, *top*, *clear* и *clear-until*. Стековые операции имеют следующий смысл:

- myStack.push(item) помещает элемент в myStack;
- myStack.pop() выбирает верхний элемент из myStack;
- myStack.top() возвращает верхний элемент из myStack или **NULL**, если myStack пустой;
- myStack.*clear*() очищает myStack, т. е. выбирает все элементы из myStack;
- myStack.*clear-until*(item) выбирает элементы из myStack, пока не будет найден элемент, или пока myStack не будет пуст.

К части DEFAULT-LIST может быть обеспечен доступ, и манипулировать ею можно путем использования списочных операций *add*, *append*, *delete*, *member*, *first*, *length*, *next*, *random* и *change*. Смысл этих списочных операций определен в п. 8.3.1.1.

Для создания нового состояния объекта предполагается использовать функцию NEW-ENTITY :

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

Она создает новое состояние объекта и возвращает его ссылку. Компоненты нового состояния объекта имеют следующие значения:

- <identifier> устанавливается на *entityIdentifier* и будет глобально определяемым уникальным идентификатором;
- *STATUS* устанавливается на **ACTIVE**;
- *flow-graph-node-reference* является единственным (верхним) элементом в *CONTROL-STACK*;
- *DEFAULT-LIST* является пустым списком;
- *DEFAULT-POINTER* имеет значение **NULL**;
- *VALUE-STACK* является пустым стеком;
- *E-VERDICT* устанавливается на **none**;
- *TIMER-GUARD* является новым связыванием таймеров (см. п. 8.3.2.4) с именем GUARD, состоянием **IDLE** и без периода времени по умолчанию;
- *DATA-STATE* является пустым списком;
- *TIMER-STATE* является пустым списком;
- *SNAP-DONE* является пустым списком.

При обходе потокового графа *CONTROL-STACK* часто меняет свое значение одним и тем же образом: верхний элемент выбирается из этого стека, а последующий узел по отношению к выбранному узлу заносится в *CONTROL-STACK*. Эта последовательность операций инкапсулируется в функции *NEXT-CONTROL*:

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

8.3.2.2 Состояние данных и связывание переменной

Как показано на рисунке 25, состояние данных *DATA-STATE* состояния объекта является перечнем списков связываний переменных. Каждый список связываний переменных определяет связывание переменных в определенном контекстном блоке. Прибавление нового списка связываний переменных соответствует входу в новый контекстный блок, например, вызывается функция. Удаление списка связываний переменных соответствует выходу из контекстного блока, например, функция выполняет оператор **return**.

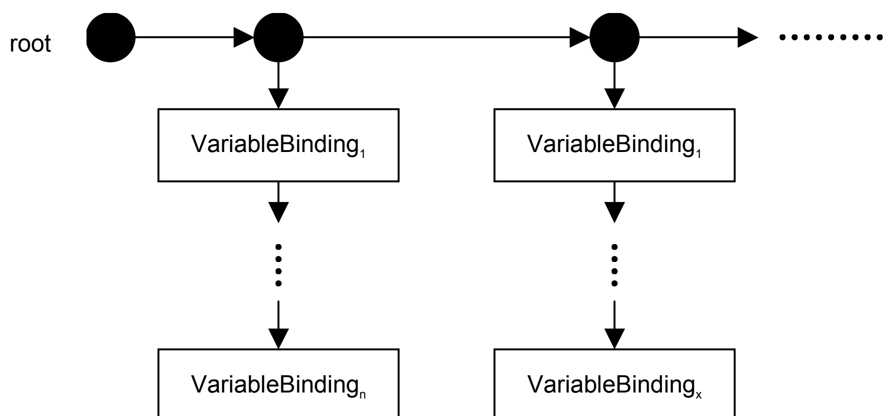


Рисунок 25/Z.143 – Структура части *DATA-STATE* состояния объекта

На рисунке 26 показана структура связывания переменной. Переменная имеет имя, <*location*> и *VALUE*. *VAR NAME* определяет переменную в контекстном блоке. Здесь <*location*> является уникальным идентификатором местоположения в памяти значения переменной. Часть *VALUE* связывания переменной описывает фактическое значение переменной.

ПРИМЕЧАНИЕ. – Уникальные идентификаторы местоположения должны предоставляться автоматически при объявлении переменной.

Рисунок 26/Z.143 – Структура связывания переменной

Различие между именем переменной и местоположением было введено для моделирования вызовов функции и выполнения надлежащим образом тестовых примеров с параметризацией значения и ссылки:

a) Параметр, передаваемый по значению, обрабатывается подобно объявлению новой переменной, т. е. связывание новой переменной добавляется к списку связываний переменных контекста вызываемой функции или выполняемого тестового примера. В новом связывании переменной используется имя формального параметра *VAR-NAME*, принимается новое местоположение и значение, которое пересылается в функцию или тестовый пример.

b) Параметр, передаваемый по ссылке, также приводит к новому связыванию переменной в контексте вызываемой функции или выполняемого тестового примера. В новом связывании переменной также используется имя формального параметра *VAR-NAME*, но принимается не новое значение и не новое местоположение. В новом связывании переменной принимается копия *<location>* и *VALUE* переменной, передаваемой по ссылке.

При обновлении значения переменной, например, в случае присвоения для переменной, имя переменной используется для идентификации местоположения, и в то же время обновляются все связывания переменной с одним и тем же местоположением. Таким образом, при выходе из контекстного блока список переменных, принадлежащих данному контекстному блоку, может быть удален без дальнейшего обновления. Благодаря процедуре обновления переменные, передаваемые по ссылке, автоматически будут иметь правильное значение.

8.3.2.3 Доступ к состояниям данных

Значение переменной может быть вызвано путем использования "dot"-нотации *myEntity.myVar.VALUE*, где *myEntity* указывает на состояние объекта, а *myVar* является именем переменной.

Считается, что для обработки переменных и контекста переменной должны быть определены следующие функции:

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

устанавливает часть *VALUE* переменной *myVar* в действительном контексте объекта *myEntity* на *myVal*. Кроме того, часть *VALUE* всех переменных с тем же местоположением, что и переменная *myVar*, будет также установлена на *myVal*.

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

создает новое связывание переменной для переменной *myVar* с начальным значением *myVal* в фактическом контекстном блоке объекта *myEntity*. Использование ключевого слова **NONE** в качестве *myVal* означает, что создается переменная с неопределенным начальным значением. Автоматически создается новое и уникальное значение *<location>*.

c) Функция GET-VAR-LOC: myEntity.GET-VAR-LOC (myVar)

выводит местоположение переменной *myVar,* которой владеет *myEntity*.

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

создает новое связывание переменной для переменной *myVar* с местоположением *myLoc* в фактическом контекстном блоке объекта *myEntity*. Переменная будет инициализирована значением другой переменной с местоположением *myLoc*.

ПРИМЕЧАНИЕ. – Переменные с одним и тем же местоположением являются результатом параметризации по ссылке. Как описано в п. 8.3.2.2, благодаря обработке параметров-ссылок все переменные с одним и тем же местоположением будут иметь одинаковые значения в течение их времени существования.

e) Функция INIT-VAR-SCOPE: myEntity.INIT-VAR-SCOPE ()

инициализирует новый контекст переменной в состоянии данных объекта *myEntity*, т. е. добавляется пустой список в качестве первого списка в перечне списков связываний переменных.

f) Функция DEL-VAR-SCOPE: myEntity.DEL-VAR-SCOPE ()

удаляет переменный контекст состояния данных объекта *myEntity*, т. е. удаляется первый список в перечне списков связываний переменных.

8.3.2.4 Состояние таймера и связывание таймера

Как показано на рисунках 27 и 25, состояние таймера TIMER-STATE и состояние данных DATA-STATE состояния объекта сопоставимы. Оба эти состояния являются перечнями списков связываний, и каждый список связываний определяет допустимые связывания в конкретном контексте. Добавление нового списка соответствует входу в новый контекстный блок, а удаление списка связываний соответствует выходу из контекстного блока.

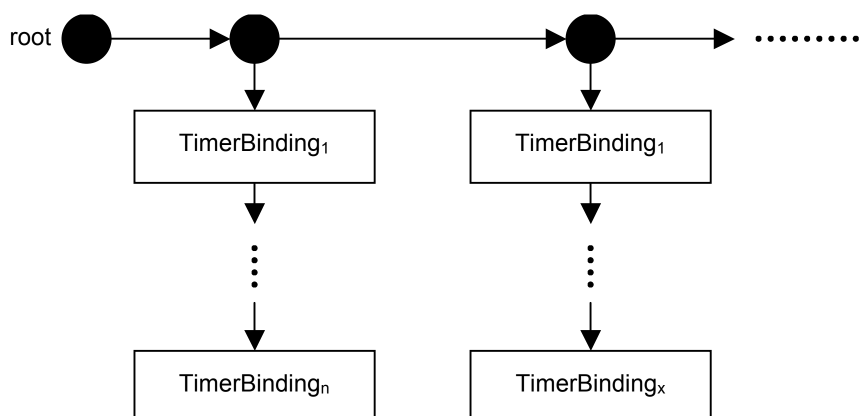


Рисунок 27/Z.143 – Структура части TIMER-STATE состояния объекта

На рисунке 28 показана структура связывания таймера. Смысл TIMER-NAME и <location> похож на смысл VAR-NAME и <location> для связывания переменной (рисунок 26).

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

Рисунок 28/Z.143 – Структура связывания таймера

STATUS обозначает, является ли таймер активным, неактивным или у него истекло время ожидания. Соответствующими значениями для STATUS являются **IDLE**, **RUNNING** и **TIMEOUT**. DEF-DURATION описывает период времени таймера по умолчанию. ACT-DURATION сохраняет фактический период времени, с которым был запущен работающий таймер. TIME-LEFT описывает фактический период времени, который должен отработать таймер до того, как у него истечет время ожидания.

ПРИМЕЧАНИЕ. – Время DEF-DURATION является неопределенным, если таймер объявлен без установки периода времени по умолчанию. ACT-DURATION и TIME-LEFT устанавливаются на 0.0, если таймер останавливается или у него истекает время ожидания. Если таймер запускается без установки периода времени, то значение DEF-DURATION копируется в ACT-DURATION. Если таймер запускается без установки определенного периода времени, то имеет место динамическая ошибка.

Необходимо, чтобы SNAP-VALUE и SNAP-STATUS поддерживали семантику фиксации мгновенного состояния процесса в TTCN-3. При фиксации мгновенного состояния процесса SNAP-VALUE получает фактическое значение ACT-DURATION – TIME-LEFT, а SNAP-STATUS получает то же значение, что и STATUS. Оценка фиксации мгновенного состояния процесса будет опираться только на значения в SNAP-VALUE и SNAP-STATUS.

Значение таймера может быть передано в функции только по ссылке, т. е. механизм похож на механизм для переменных, описанный в п. 8.3.2.2. Это означает создание нового связывания таймера (с именем формального параметра), которое получает копии <location>, STATUS, DEF-DURATION, ACT-DURATION, TIME-LEFT, SNAP-VALUE и SNAP-STATUS от таймера, переданного по ссылке. При обновлении таймера все связывания таймера с одинаковым значением <location> обновляются в одно и то же время.

8.3.2.5 Доступ к состояниям таймера

Значения частей STATUS, DEF-DURATION, ACT-DURATION, TIME-LEFT, SNAP-VALUE и SNAP-STATUS таймера *myTimer* могут быть получены путем использования dot-нотации:

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

- myEntity.myTimer.SNAP-STATUS.

Объект *myEntity* в dot-нотации указывает на состояние объекта, представляющее состояние тестового компонента или управления модулем, которое относится к таймеру *myTimer*.

Для изменения значений частей *STATUS*, *DEF-DURATION*, *ACT-DURATION*, *TIME-LEFT*, *SNAP-VALUE* и *SNAP-STATUS* имени *timer-name* таймера должна использоваться обобщенная операция *TIMER-SET*, например:

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

устанавливает значение *STATUS* таймера *myTimer* в фактическом контексте объекта *myEntity* на значение *myVal*. Кроме того, значение *STATUS* всех таймеров с тем же местоположением, что и таймер *myTimer,* будет также устанавливаться на *myVal*. Для изменения значений частей *DEF-DURATION*, *ACT-DURATION*, *TIME-LEFT*, *SNAP-VALUE* и *SNAP-STATUS* можно также использовать функцию *TIMER-SET*.

Для обработки таймеров, контекста таймера и фиксации мгновенного состояния процесса должны быть определены следующие функции:

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

   создает новое связывание таймера для таймера *myTimer* с периодом времени по умолчанию *myDuration* в фактическом контексте объекта *myEntity*. Использование ключевого слова **NONE** в качестве *myDuration* означает, что создается таймер без указания периода времени по умолчанию.

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

   выводит местоположение таймера *myTimer,* которым владеет объект *myEntity*.

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

   создает новое связывание таймера для таймера *myTimer* с местоположением *myLocation* в фактическом контекстном блоке объекта *myEntity*. Таймер будет инициализирован значениями *STATUS*, *DEF-DURATION*, *ACT-DURATION* и *TIME-LEFT* другого таймера с местоположением <*location*>.

   ПРИМЕЧАНИЕ. – Таймеры с одним и тем же местоположением являются результатом параметризации по ссылке. Благодаря обработке ссылочных параметров таймера, как описано в п. 8.3.2.3, все таймеры с одним и тем же местоположением будут иметь одинаковые значения для *STATUS*, *DEF-DURATION*, *ACT-DURATION* и *TIME-LEFT* в течение их времени существования.

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

   инициализирует новый контекст таймера в состоянии таймера объекта *myEntity*, т. е. в качестве первого списка в перечне списков связываний таймера добавляется пустой список.

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

   удаляет контекст таймера состояния таймера в объекте *myEntity*, т. е. удаляется первый список в перечне списков связываний таймера.

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

   осуществляет обновление *SNAP-VALUE* и *SNAP-STATUS* во всех таймерах, которыми владеет *myEntity*, т. е.:

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

8.3.3 Состояния порта

Состояния порта используются для описания фактических состояний портов. Состояния портов обрабатываются в списке *ALL-PORT-STATES* (см. рисунок 23) в пределах состояния модуля. На рисунке 29 показана структура состояния порта. Часть *PORT-NAME* указывает на имя порта, которое используется для идентификации порта тестовым компонентом *OWNER*, владеющим этим портом. Часть *STATUS* выдает фактическое состояние порта. Порт может быть в состоянии либо **STARTED** (запущен), либо **STOPPED** (остановлен).

ПРИМЕЧАНИЕ. – Порт в тестовой системе однозначно определяется владеющим тестовым компонентом <owner> и именем порта <port-name>, локальным по отношению к <owner>.

В списке *CONNECTIONS-LIST* состояния порта осуществляется слежение за соединениями между различными портами в тестовой системе. Механизм этих операций объясняется в п. 8.3.3.1.

В части *VALUE-QUEUE* состояния порта хранятся сообщения, вызовы, ответы и особые состояния, которые получены на этом порте, но еще не использованы.

Часть *SNAP-VALUE* поддерживает механизм фиксации мгновенного состояния процесса в TTCN-3. Когда происходит фиксация мгновенного состояния процесса, то первый элемент в *VALUE-QUEUE* копируется в *SNAP-VALUE*. *SNAP-VALUE* примет значение **NULL,** если *VALUE-QUEUE* пуст или *STATUS* находится в состоянии **STOPPED**.

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

Рисунок 29/Z.143 – Структура состояния порта

8.3.3.1 Обработка соединений между портами

Соединение между двумя тестовыми компонентами выполняется путем соединения двух из этих портов с помощью операции **`connect`**. Таким образом, компонент может затем использовать имя своего локального порта для адресации удаленной очереди. Как показано на рисунке 30, соединение *connection* представляется в состояниях обеих соединяемых очередей посредством пары объектов *REMOTE-ENTITY* и *REMOTE-PORT-NAME*. Объект *REMOTE-ENTITY* является уникальным идентификатором тестового компонента, владеющего удаленным портом. Имя *REMOTE-PORT-NAME* указывает на локальное имя, используемое объектом *REMOTE-ENTITY* для адресации очереди. В TTCN-3 поддерживаются соединения портов типа "один со многими", и поэтому все соединения порта организуются по списку.

ПРИМЕЧАНИЕ 1. – Соединения с помощью операций **`map`** также обрабатываются в списке соединений. Операция **`map`**: **`map`**(*PTC1*:*MyPort*, **`system`**.*PCO1*) приводит к новому соединению (**`system`**, *PCO1*) в состоянии порта *MyPort,* которым владеет *PTC1*. Удаленная сторона, с которой соединяется *PCO1*, располагается внутри системы SUT. Ее поведение выходит за пределы действия данной семантики.

ПРИМЕЧАНИЕ 2. – В операционной семантике обрабатывается ключевое слово **`system`** в качестве символического адреса. Соединение (**`system`**, *myPort*) в списке соединений порта указывает на то, что порт отображается на порт *myPort* в интерфейсе тестовой системы.

REMOTE-ENTITY	REMOTE-PORT-NAME

Рисунок 30/Z.143 – Структура соединения

8.3.3.2 Обработка состояний портов

К очереди значений в состоянии порта может быть обеспечен доступ и можно манипулировать ею путем использования известных операций с очередями *enqueue*, *dequeue*, *first* и *clear*. При использовании функции *GET-PORT* или *GET-REMOTE* происходят ссылки на очередь, к которой должен обеспечиваться доступ.

ПРИМЕЧАНИЕ 1. – Операции с очередями *enqueue*, *dequeue*, *first* и *clear* имеют следующий смысл:

- *myQueue*.*enqueue*(*item*) помещает *item* (элемент) в качестве последнего элемента в *myQueue*;
- *myQueue*.*dequeue*() удаляет первый элемент из *myQueue*;
- *myQueue*.*first*() возвращает первый элемент в *myQueue* или **`NULL`**, если *myQueue* пуста;
- *myQueue*.*clear*() удаляет все элементы из *myQueue*.

Обработку состояний портов поддерживают следующие функции:

a) Функция *NEW-PORT*: *NEW-PORT* (*myEntity, myPort*)

создает новый порт и возвращает его ссылку. Новым портом владеет *myEntity,* и порт имеет имя *myPort* для порта, определяемого тестовым компонентом *myEntity* и именем порта *myPort*. Состоянием нового порта является **STARTED**. Списки *CONNECTIONS-LIST* и *VALUE-QUEUE* пусты. *SNAP-VALUE* имеет значение **`NULL`** (т. е. очередь на входе нового порта пуста).

b) Функция *GET-PORT*: *GET-PORT* (*myEntity, myPort*)

возвращает ссылку на порт, определяемый тестовым компонентом *myEntity,* владеющим портом и именем порта *myPort*.

c) Функция *GET-REMOTE-PORT*: *GET-REMOTE-PORT* (*myEntity, myPort, myRemoteEntity*)

возвращает ссылку на порт, которым владеет тестовый компонент *myRemoteEntity* и который соединен с портом, определяемым компонентами *myEntity* и *myPort*. Если удаленный порт отображается на порт в интерфейсе тестовой системы, то возвращается символический адрес **SYSTEM**.

ПРИМЕЧАНИЕ 2. – *GET-REMOTE-PORT* возвращает **`NULL`**, если удаленный порт отсутствует или если удаленный порт не может быть однозначно идентифицирован. В качестве значения для параметра *myRemoteEntity* может использоваться специальное значение **`NONE`**, если удаленный объект неизвестен или не требуется, т. е. для данного порта существует только соединение типа "один с одним".

d) Состояние *STATUS* порта обрабатывается подобно переменной. Оно может быть адресовано путем уточнения *STATUS* вызовом *GET-PORT*:

GET-PORT(*myEntity*, *myPort*).*STATUS*

e) Функция *ADD-CON*: *ADD-CON* (*myEntity, myPort, myRemoteEntity*, *myRemotePort*)

добавляет соединение (*myRemoteEntity*, *myRemotePort*) к списку соединений порта *myPort,* которым владеет *myEntity*.

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

удаляет соединение (*myRemoteEntity*, *myRemotePort*) из списка соединений порта *myPort,* которым владеет *myEntity*.

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

обновляет SNAP-VALUE для всех портов, которыми владеет *myEntity*, т. е.:

```
SNAP-PORTS (myEntity) {
    for all ports p          /* в состоянии module state */ {
        if (p.OWNER == myEntity) {
            if (p.STATUS == STOPPED) {
                p. SNAP-VALUE := NULL;
            }
            else {
                p. SNAP-VALUE := p.first()
            }
        }
    }
}
```

8.3.4 Основные функции для обработки состояний модуля

В операционной семантике предполагается существование следующих функций для обработки состояний модуля.

ПРИМЕЧАНИЕ 1. – Во время интерпретации модуля TTCN-3 существует только одно состояние модуля. Предполагается, что компоненты состояния модуля хранятся в глобальных переменных, а не в сложном объекте данных. Таким образом, можно считать, что следующие функции работают с глобальными переменными и не обращаются к конкретному объекту состояний модуля.

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

удаляет объект с уникальным идентификатором *myEntity*. Удаление включает в себя:

– удаление состояния *myEntity* объекта;

– удаление всех портов, которыми владеет *myEntity*;

– удаление всех соединений, в которые включен *myEntity*.

b) Функция UPDATE-REMOTE-REFERENCES:

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

функция UPDATE-REMOTE-REFERENCES обновляет переменные и таймеры с одинаковым местоположением в обоих объектах. Значения, которые будут использоваться для обновления, являются значениями переменных и таймеров, которыми владеет *source* (источник).

ПРИМЕЧАНИЕ 2. – UPDATE-REMOTE-REFERENCES используется во время завершения тестовых примеров. Эта функция позволяет обновлять переменные управления модулем, которые пересылаются тестовым примерам как ссылочные параметры.

8.4 Сообщения, вызовы процедур, ответы и особые состояния

Обмен информацией между тестовыми компонентами и между тестовыми компонентами и системой SUT связан с *сообщениями, вызовами процедур, ответами на вызовы процедур и особыми состояниями*. В целях обеспечения связи эти элементы должны быть сконструированы, закодированы и декодированы. Конкретное кодирование, т. е. преобразование типов данных TTCN-3 в биты и байты, и декодирование, т. е. преобразование битов и байтов в типы данных TTCN-3, выходит за рамки операционной семантики. В настоящей Рекомендации *сообщения, вызовы процедур, ответы на вызовы процедур, и особые состояния* обрабатываются на концептуальном уровне.

8.4.1 Сообщения

Сообщения относятся к связи на базе сообщений. Значениями всех типов данных (предопределенных и определенных пользователем) могут обмениваться объекты, участвующие в связи. Как показано на рисунке 31, в операционной семантике сообщение обрабатывается как структурный объект, в который входят части *sender* (отправитель), *type* (тип) и *value* (значение). Часть *sender* идентифицирует объект отправителя сообщения, часть *type* задает тип сообщения, а часть *value* определяет значение сообщения.

Рисунок 31/Z.143 – Структура сообщения

ПРИМЕЧАНИЕ. – В операционной семантике представлена только модель для концепций TTCN-3. А передается ли или должна ли передаваться и/или приниматься информация *отправителя* и каким образом это осуществляется, зависит от реализации тестовой системы, например, в ряде случаев информация отправителя может быть элементом части значения сообщения и поэтому не является отдельной частью структуры сообщения.

8.4.2 Вызовы процедур и ответы

Вызовы процедур и ответы на вызовы процедур относятся к связи на базе процедур. Они определяются подобно значениям записи с компонентами, представляющими параметры. В операционной семантике также обрабатываются вызовы процедур и ответы на вызовы процедур подобно значениям в структурных типах. На рисунках 32 и 33 представлены структура вызова процедуры и структура ответа.

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

Рисунок 32/Z.143 – Структура вызова процедуры

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

Рисунок 33/Z.143 – Структура ответа на вызов процедуры

Части *sender* и *procedure-reference* имеют одинаковый смысл на обоих рисунках. Часть *sender* указывает на объект отправителя вызова или на ответ на вызов процедуры. Часть *procedure-reference* указывает на процедуру, к которой принадлежат вызов и ответ. Часть *parameter-part* вызова процедуры на рисунке 32 указывает на параметры **in** и параметры **inout,** а часть *parameter-part* ответа на рисунке 33 указывает на параметры **inout** и параметры **out** процедуры, к которой принадлежат вызов и ответ. Кроме того, ответ имеет часть *value* для возврата значений в ответе на процедуру.

ПРИМЕЧАНИЕ 1. – Как указано в предыдущем примечании (см. п. 8.4.1), в операционной семантике происходит только представление модели для концепций TTCN-3. А передается ли или должна ли передаваться и/или приниматься информация, описанная на рисунках 32 и 33, и каким образом это осуществляется, зависит от реализации тестовой системы.

ПРИМЕЧАНИЕ 2. – Для вызова процедуры параметры **out** неактуальны и на рисунке 32 они опущены. Для ответа на вызов процедуры параметры **in** неактуальны и на рисунке 33 они опущены.

ПРИМЕЧАНИЕ 3. – Типы параметров и тип возвращаемого значения можно всегда получить одновременно из соответствующего определения сигнатуры.

8.4.3 Особые состояния

Особые состояния также относятся к связи на базе процедур. На рисунке 34 показана структура особого состояния. Она состоит из четырех частей. Часть *sender* определяет отправителя особого состояния; часть *procedure-reference* указывает на процедуру, к которой принадлежит особое состояние; часть *type* определяет тип особого состояния, а часть *value* обеспечивает значение особого состояния. Сигнатура процедуры, на которую ссылаются в части ссылки на процедуру, определяет список допустимых типов особых состояний. Полученное особое состояние должно соответствовать одному из перечисленных типов. Вообще, оно может быть любым из предопределенных или определяемых пользователем типов данных TTCN-3.

sender	procedure-reference	type	value

Рисунок 34/Z.143 – Структура особого состояния

8.4.4 Конструкция сообщений, вызовов процедур, ответов и особых состояний

Операциями для передачи сообщения, вызова процедуры, ответа на вызов процедуры или особого состояния являются **send**, **call**, **reply** и **raise**. Все эти операции передачи строятся по одному и тому же способу:

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

Части <port-name> и <sending-operation> определяют порт и операцию, которые используются для передачи элемента. В случае соединений типа "один со многими" объект <receiver> необходимо определять. Элемент, подлежащий передаче, конструируется путем использования <send-specification>. В спецификации передачи могут использоваться конкретные значения ссылки на шаблоны, значения переменных, константы, выражения, функции и т. д. для конструирования и кодирования элемента, подлежащего передаче.

В операционной семантике предполагается, что существует обобщенная функция <u>*CONSTRUCT-ITEM*</u>:

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

> возвращает сообщение, вызов процедуры, ответ на вызов процедуры или особое состояние в зависимости от `<sending-operation>` и `<send-specification>` (оба `<sending-operation>` и `<send-specification>` указывают на соответствующие части в операции передачи TTCN-3). Ссылка на объект <u>*myEntity*</u> является отправителем подлежащего передаче элемента. Также предполагается, что эта информация <u>*sender*</u> является частью подлежащего передаче элемента (рисунки 31–34).

8.4.5 Сопоставление сообщений, вызовов процедур, ответов и особых состояний

Операциями для приема сообщения, вызова процедуры, ответа на вызов процедуры или особого состояния являются **`receive`**, **`getcall`**, **`getreply`** и **`catch`**. Все эти операции приема образуются одним и тем же способом:

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

Части `<port-name>` и `<receiving-operation>` определяют порт и операцию, которые используются для приема элемента. В случае соединений типа "один со многими" может быть использован раздел **`from`** для выбора конкретного объекта отправителя `<sender>`. Подлежащий приему объект должен выполнять условия, заданные в `<matching-part>`, т. е. он должен сопоставляться. В `<matching-part>` могут использоваться конкретные значения, ссылки на шаблоны, значения переменных, константы выражения, функции и т. д. для описания условий сопоставления.

В операционной семантике предполагается, что существует обобщенная функция <u>*MATCH-ITEM*</u>:

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

> возвращает **`true`**, если *myItem* выполняет условия части `<matching-part>` и если *myItem* был отправлен объектом `<sender>`; в противном случае эта функция возвращает **`false`**.

8.4.6 Вызов информации из полученных элементов

Информация из полученных сообщений, вызовов процедур, ответов на вызовы процедур и особых состояний может быть выведена в части `<assignment-part>` (см. п. 8.4.5) функций приема **`receive`**, **`getcall`**, **`getreply`** и **`catch`**. Часть `<assignment-part>` описывает, каким образом параметры вызовов процедур и ответов, возвращаемые значения, кодируемые в ответах, сообщениях, особых состояниях, и идентификатор объекта `<sender>` присваиваются переменным.

В операционной семантике предполагается, что существует обобщенная функция <u>*RETRIEVE-INFO*</u>:

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

> все значения, подлежащие выводу согласно части `<assignment-part>`, выводятся и присваиваются переменным, перечисленным в части присвоения. Присвоения выполняются с помощью операции <u>*VAR-SET*</u>, т. е. переменные с одинаковым местоположением обновляются в одно и то же время.

8.5 Записи вызовов для функций, альтернативных шагов и тестовых примеров

Функции, альтернативные шаги и тестовые примеры вызываются (или выполняются) по их имени и списку фактических параметров. Фактические параметры предоставляют ссылки для ссылочного параметра и конкретные значения для параметра-значения, как это устанавливается формальными параметрами в определении функции и тестового примера. Как показано на рисунке 35, в операционной семантике обрабатываются вызовы функций, альтернативные шаги и тестовые примеры путем использования *call records* (записей вызовов). Значение <u>*BEHAVIOUR-ID*</u> является именем функции или тестового примера, а параметры-значения предоставляют конкретные значения <$parId_1$> ... <$parId_n$> для формальных параметров <<u>$parId_1$</u>> ... <<u>$parId_n$</u>>. Ссылочные параметры обеспечивают ссылки на местоположения существующих переменных и таймеров. Прежде чем может быть выполнена функция или тестовый пример, должна быть сконструирована соответствующая запись вызова.

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

Рисунок 35/Z.143 – Структура записи вызова

8.5.1 Обработка записей вызовов

Имя функции или тестового примера, а также фактические значения параметров можно вывести путем использования dot-нотации, например, *myCallRecord.parId$_n$* или *myCallRecord.*<u>*behaviour-id*</u>, где *myCallRecord* является указателем на запись вызова.

Предполагается, что для конструкции вызова можно использовать функцию *NEW-CALL-RECORD*:

NEW-CALL-RECORD(*myBehaviour*)

> создает новую запись вызова для функции или тестового примера *myBehaviour* и возвращает указатель на новую запись. Поля параметров новой записи вызова имеют неопределенные значения.

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

> создает переменные и таймеры для обработки значения и параметров-ссылок в фактическом контексте тестового компонента или управления модулем *myEntity*. Переменные для обработки параметров-значений инициализируются соответствующими значениями, содержащимися в записи вызова. Переменные и таймеры для обработки параметров-ссылок получают предоставленное местоположение. Кроме того, они получают значение существующей переменной или таймера в другом контекстном блоке компонента, в котором была создана запись вызова.

8.6 Процедура оценки для модуля TTCN-3

8.6.1 Фазы оценки

В процедуру оценки для модуля TTCN-3 входят:

1) *фаза инициализации*;
2) *фаза обновления*;
3) *фаза выбора*; и
4) *фаза выполнения*.

Фазы 2), 3) и 4) повторяются до тех пор, пока не закончится управление модулем. Процедура оценки описывается с помощью смешанного использования неформального текста, псевдокода и функций, содержащихся в предыдущих разделах.

8.6.1.1 Фаза I: Инициализация

В фазу инициализации входят следующие действия:

a) **Объявление и инициализация переменных:**

- *INIT-FLOW-GRAPHS*(); // Инициализация обработки потокового графа. *INIT-FLOW-GRAPHS* // объясняется в п. 8.6.2
- *Entity* := **NULL**; // *Entity* (объект) будет использован для ссылки на состояние // объекта. Состояние объекта представляет либо управление // модулем, либо тестовый компонент.

ПРИМЕЧАНИЕ. – Следующие глобальные переменные *ALL-ENTITY-STATES*, *ALL-PORT-STATES*, *MTC*, *TC-VERDICT* и *DONE* образуют состояние модуля, которым манипулируют во время интерпретации модуля TTCN-3 (см. п. 8.3.1).

- *ALL-ENTITY-STATES* := **NULL**;
- *ALL-PORT-STATES* := **NULL**;
- *MTC* := **NULL**;
- *TC-VERDICT* := **none**;
- *DONE* := **NULL**;
- *SNAP-DONE* := 0;

b) **Создание и инициализация управления модулем**

- *Entity:=* *NEW-ENTITY* (*GET-UNIQUE-ID*(),*GET-FLOW-GRAPH* (<*moduleId*>)); // Новое состояние объекта создается и инициализируется // начальным узлом потокового графа, представляющим // поведение управления модуля с именем <*moduleId*>. // *GET-UNIQUE-ID* объясняется в п. 8.6.2.
- *Entity*.*INIT-VAR-SCOPE*(); // Новый контекст переменной
- *Entity*.*INIT-TIMER-SCOPE*(); // Новый контекст таймера
- Entity.VALUE-STACK.push(MARK); // Маркер заносится в стек значений
- ALL-ENTITY-STATES.append(Entity); // Новый объект заносится в состояние модуля.

8.6.1.2 Фаза II: Обновление

Фаза обновления относится ко всем действиям, которые выходят за рамки операционной семантики, но влияют на интерпретацию модуля TTCN-3. В фазу обновления входят следующие действия:

a) **Ход времени:** Все работающие таймеры обновляются, т. е. значения *TIME-LEFT* работающих таймеров (возможно) уменьшаются, и если период времени таймера истекает из-за обновления, то обновляются соответствующие связывания таймера, т. е. *TIME-LEFT* устанавливается на 0.0, а *STATUS* устанавливается на **`TIMEOUT`**;

ПРИМЕЧАНИЕ 1. – В обновление таймеров входит обновление всех работающих таймеров *TIMER-GUARD* в состояниях модуля. Таймеры *TIMER-GUARD* используются для защиты особого состояния тестовых примеров и операций вызова.

b) **Поведение системы SUT:** Сообщения, удаленные вызовы процедур, ответы на удаленные вызовы процедур и особые состояния, (возможно) принимаемые от системы SUT, заносятся в очереди порта, на котором будут иметь место соответствующие приемы.

ПРИМЕЧАНИЕ 2. – В данной операционной семантике не делается никаких предположений относительно хода времени и поведения системы SUT.

8.6.1.3 Фаза III: Выбор

В фазу выбора входят два следующих действия:

a) **Выбор:** Выбирается незаблокированный объект, т. е. объект, имеющий значение *STATUS* либо **ACTIVE,** либо **SNAPSHOT**;

b) **Хранение:** В глобальной переменной *Entity* хранится идентификатор выбранного объекта.

8.6.1.4 Фаза IV: Выполнение

В фазу выполнения входят следующие два действия:

a) **Шаг выполнения выбранного объекта:** Выполните верхний узел потокового графа в *CONTROL-STACK* объекта *Entity*;

b) **Критерий окончания проверки:** Остановите выполнение, если управление модулем завершено, т. е. список состояний объекта пуст; в противном случае имеет место продолжение с фазы II.

ПРИМЕЧАНИЕ. – Шаг выполнения для выбранного объекта можно рассматривать как вызов процедуры. Проверка критерия окончания выполняется, когда завершается шаг выполнения, т. е. имеет место возврат управления.

8.6.2 Глобальные функции

В процедуре оценки используются глобальные функции *INIT-FLOW-GRAPHS* и *GET-UNIQUE-ID*:

a) Предполагается, что *INIT-FLOW-GRAPHS* является функцией, которая инициализирует обработку потокового графа. Эта обработка может включать в себя создание потоковых графов и обработку указателей на потоковые графы и узлы потоковых графов.

b) Предполагается, что *GET-UNIQUE-ID* является функцией, возвращающей уникальный идентификатор всякий раз, когда он вызывается. Этот уникальный идентификатор может быть реализован в виде переменной счетчика, значение которой возрастает и возвращается всякий раз, когда вызывается *GET-UNIQUE-ID*.

В псевдокоде, используемом в последующих разделах для описания выполнения узлов потокового графа, применяются функции *CONTINUE-COMPONENT*, *RETURN*, **`***DYNAMIC-ERROR***`**:

a) *CONTINUE-COMPONENT:* фактический тестовый компонент продолжает свое выполнение с узлом, находящимся вверху управляющего стека, т. е. управление не передается обратно к процедуре оценки модуля, описанной в данном разделе.

b) *RETURN* возвращает управление обратно к процедуре оценки модуля, описанной в данном разделе. *RETURN* является последним действием '*шага выполнения для выбранного объекта*' фазы *выполнения*.

c) **`***DYNAMIC-ERROR***`** указывает на появление динамической ошибки. Сама процедура обработки ошибок выходит за рамки операционной семантики. Если имеет место динамическая ошибка, то все последующее поведение тестового примера считается неопределенным. В этом случае ресурсы, выделенные для этого тестового примера, освобождаются, а вердикт **error** присваивается тестовому примеру. Управление передается оператору в управляющей части, следующей за оператором выполнения, в котором имела место ошибка. Это моделируется сегментом потокового графа <dynamic-error> (раздел 9.18b)

ПРИМЕЧАНИЕ. – Появление динамической ошибки связано с тестовым поведением. Динамическая ошибка согласно операционной семантике указывает на проблему в использовании нотации TTCN-3, например, на неправильное использование или на состояние конкуренции (race condition).

d) *`APPLY-OPERATOR`* используется в качестве обобщенной функции для описания оценки операций (например, e.g., +. *, / или –) в выражениях (см. п. 9.18.4).

9 Сегменты потокового графа для конструкций TTCN-3

Поведение TTCN-3 в операционной семантике представляется в виде потоковых графов. Алгоритм построения конструкций для потоковых графов, представляющих поведение, описан в п. 8.2. Он базируется на шаблонах для потоковых графов и сегментов потоковых графов, которые должны использоваться при построении конструкции конкретных потоковых графов для описаний управления модулем, тестовых примеров, альтернативных шагов, функций и компонентных типов, определенных в модуле TTCN-3. В данном разделе можно найти определения шаблонов для сегментов потокового графа. Они представлены в алфавитном, а не в логическом порядке.

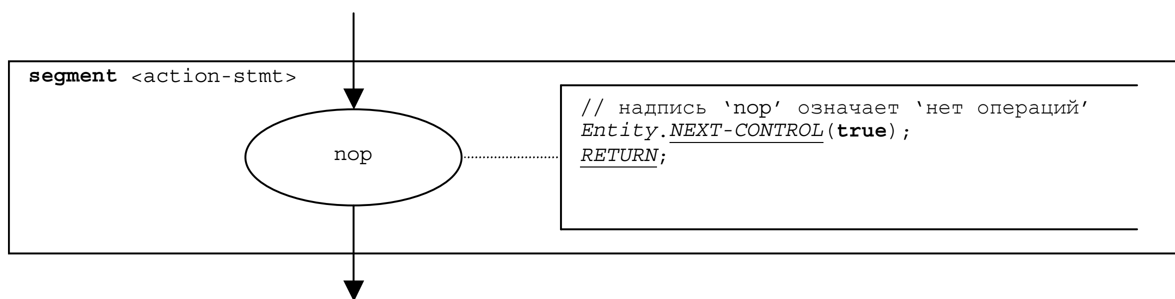
Определения сегментов потокового графа даны в виде рисунков. Узлы потокового графа представлены на левой стороне рисунков, а комментарии, связанные с узлами и потоковыми линиями, показаны на правой стороне. Описательные комментарии представлены для ссылочных узлов, а комментарии в виде псевдокода связаны с базовыми узлами. Псевдокод описывает, как интерпретируется базовый узел, т. е. как он изменяет состояние модуля. В нем используются функции, определенные в разделе 8, и глобальные переменные, объявленные и инициализированные в процедуре оценки для модулей TTCN-3 (см. п. 8.6). В разделе 8 можно найти общий вид всех функций и ключевых слов, используемых в псевдокоде.

9.1 Оператор action

Синтаксическая структура оператора **action** (действие) имеет следующий вид:

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

Сегмент <action-stmt> потокового графа на рисунке 36 определяет выполнение оператора **action**.



ПРИМЕЧАНИЕ. – Параметр <informal description> оператора **action** не имеет смысла для операционной семантики и поэтому не представлен в сегменте потокового графа.

Рисунок 36/Z.143 – Сегмент <action-stmt> потокового графа

9.2 Оператор activate

Синтаксическая структура оператора **activate** (активация) имеет следующий вид:

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

Имя <altstep-name> обозначает имя альтернативного шага, который активируется как поведение по умолчанию, а `<act-par–descr1>, ... , <act-par-descrn>` описывает фактические значения параметров альтернативного шага во время его активации.

Предполагается, что для каждого `<act-par-desc1>` известен соответствующий идентификатор формального параметра `<f-par-Id1>`, т. е. приведенную выше синтаксическую структуру можно расширить до:

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

Сегмент `<activate-stmt>` потокового графа на рисунке 37 определяет выполнение оператора активации. Это выполнение подразделяется на три шага. На первом шаге создается запись вызова для альтернативного шага `<function-name>`. На втором шаге вычисляются значения фактического параметра, которые присваиваются соответствующему полю в записи вызова. На третьем шаге запись вызова вносится в качестве первого элемента в список *DEFAULT-LIST* объекта, который активирует значение по умолчанию.

ПРИМЕЧАНИЕ. – Для альтернативных шагов, которые активируются как поведение по умолчанию, допускаются только параметры-значения. Обработка параметров-значений на рисунке 37 описывается с помощью сегмента <value-par-calculation> потокового графа, который определен в п. 9.24.1.

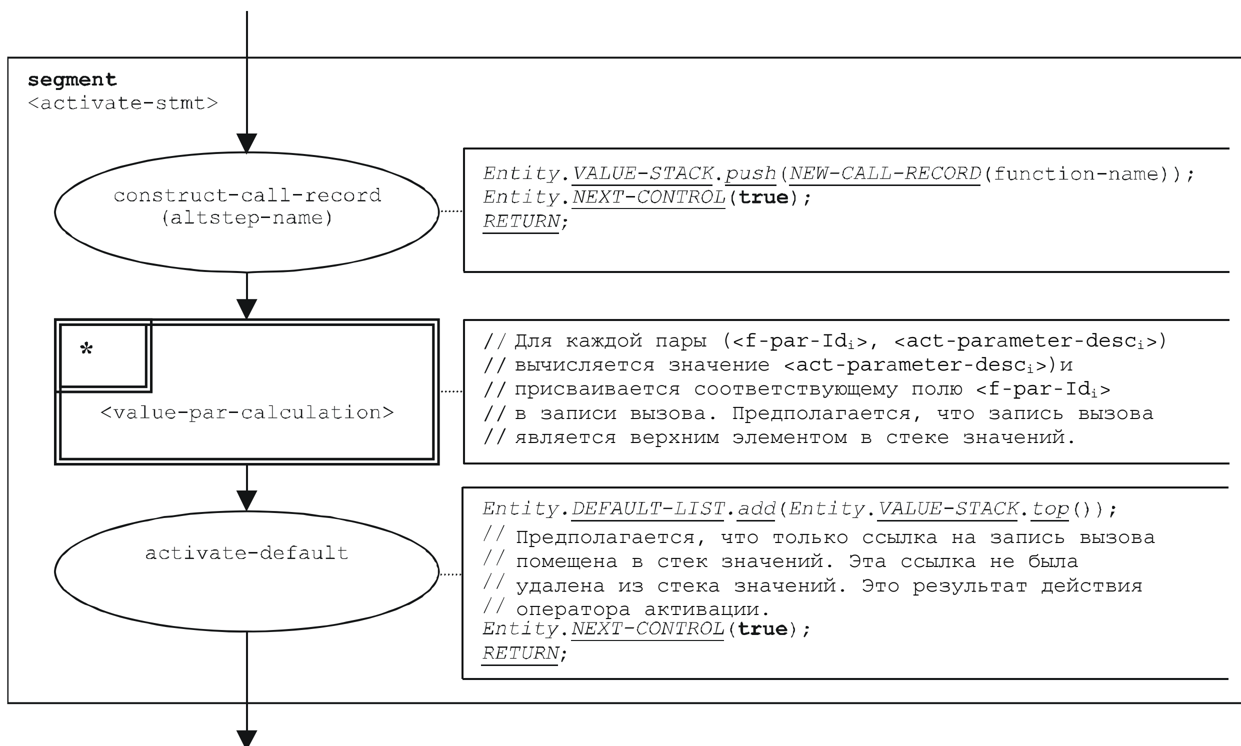


Рисунок 37/Z.143 – Сегмент <activate-stmt> потокового графа

9.3 Оператор alt

Оператор **alt** – это наиболее сложный и важный оператор в TTCN-3. Он реализует семантику "снимка", т. е. фиксации мгновенного состояния процесса, и определяет процедуру ветвления, вызванную приемом сообщений, ответов, вызовов и особых состояний, появлением значений тайм-аута и завершением компонентов. Кроме того, вызов механизма действий по умолчанию в TTCN-3 также связан с оператором **alt**.

На рисунке 38 дано представление оператора **alt** в виде потокового графа. В сегменте <receiving-branch> потокового графа содержатся различные альтернативы, вызванные приемом сообщений, ответов, вызовов и особых ситуаций, появлением значений тайм-аутов и завершением компонентов.

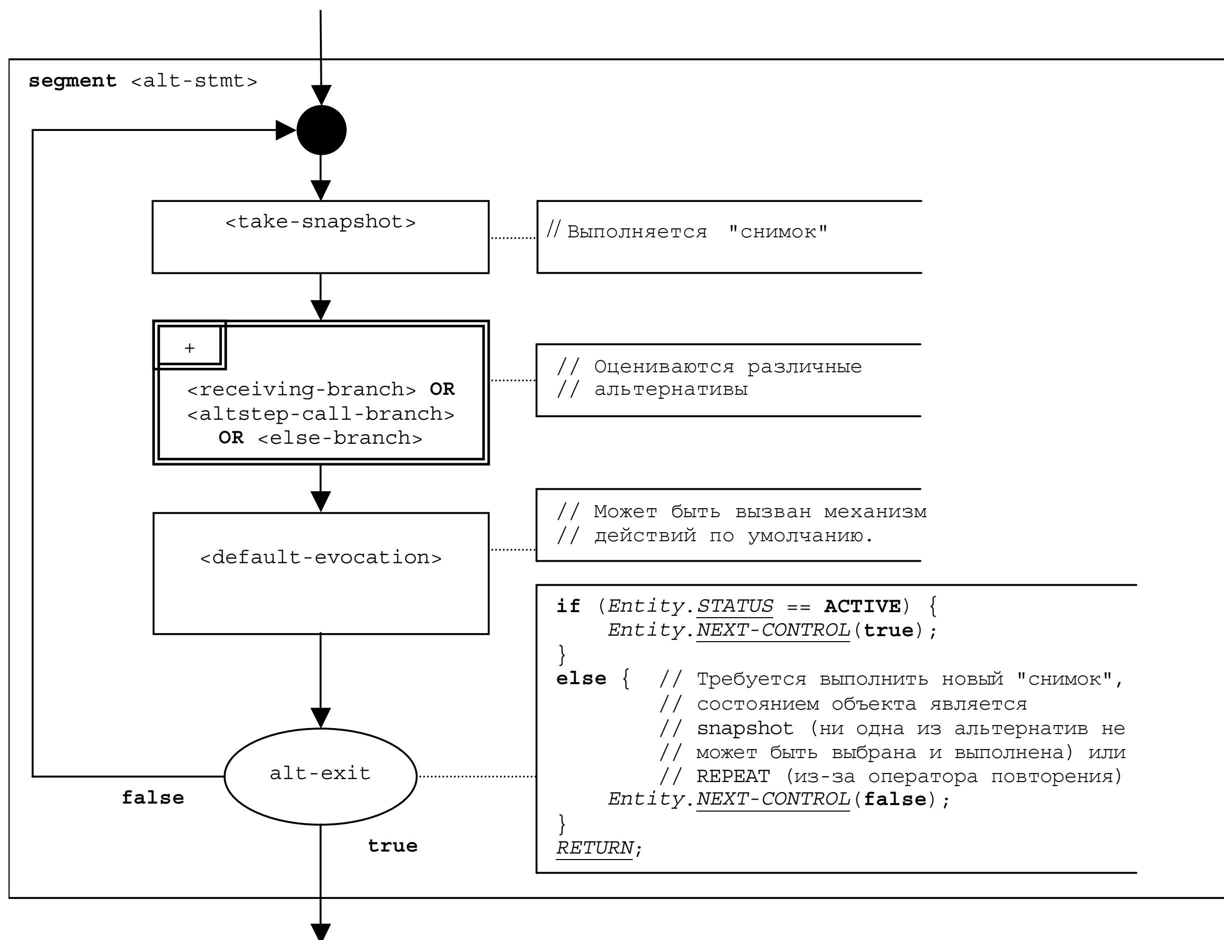


Рисунок 38/Z.143 – Сегмент <alt-stmt> потокового графа

9.3.1 Сегмент <take-snapshot> потокового графа

Сегмент `<take-snapshot>` потокового графа на рисунке 39 описывает процедуру выполнения фиксации мгновенного состояния процесса, при которой записываются значения портов, таймеров и остановленных компонентов.

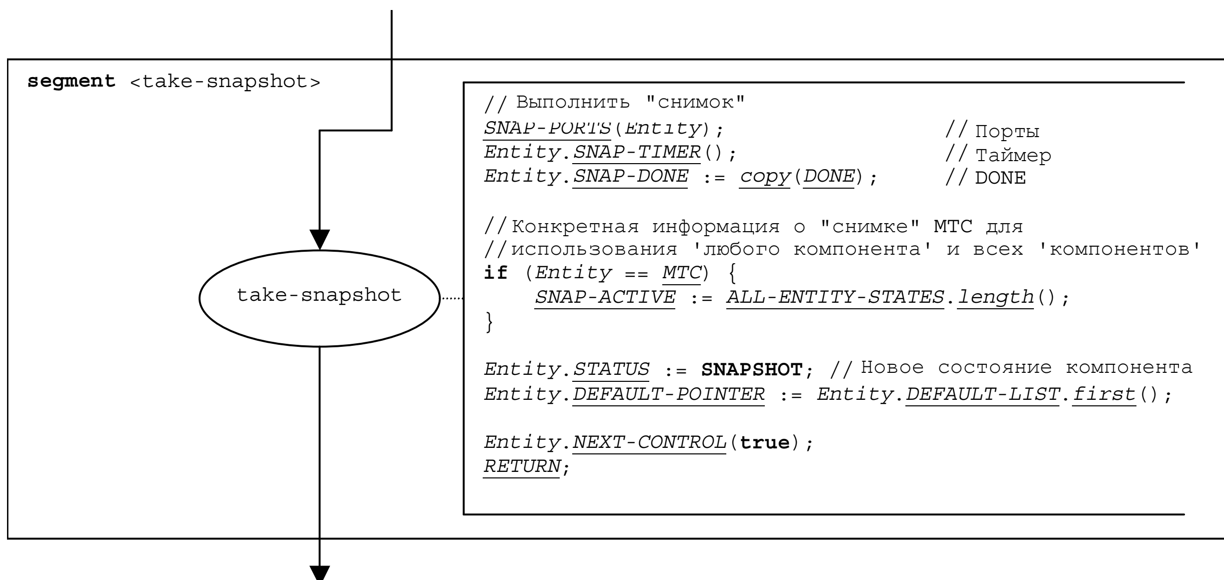


Рисунок 39/Z.143 – Сегмент <take-snapshot> потокового графа

9.3.2 Сегмент <receiving-branch> потокового графа

Выполнение сегмента `<receiving-branch>` потокового графа показано на рисунке 40.

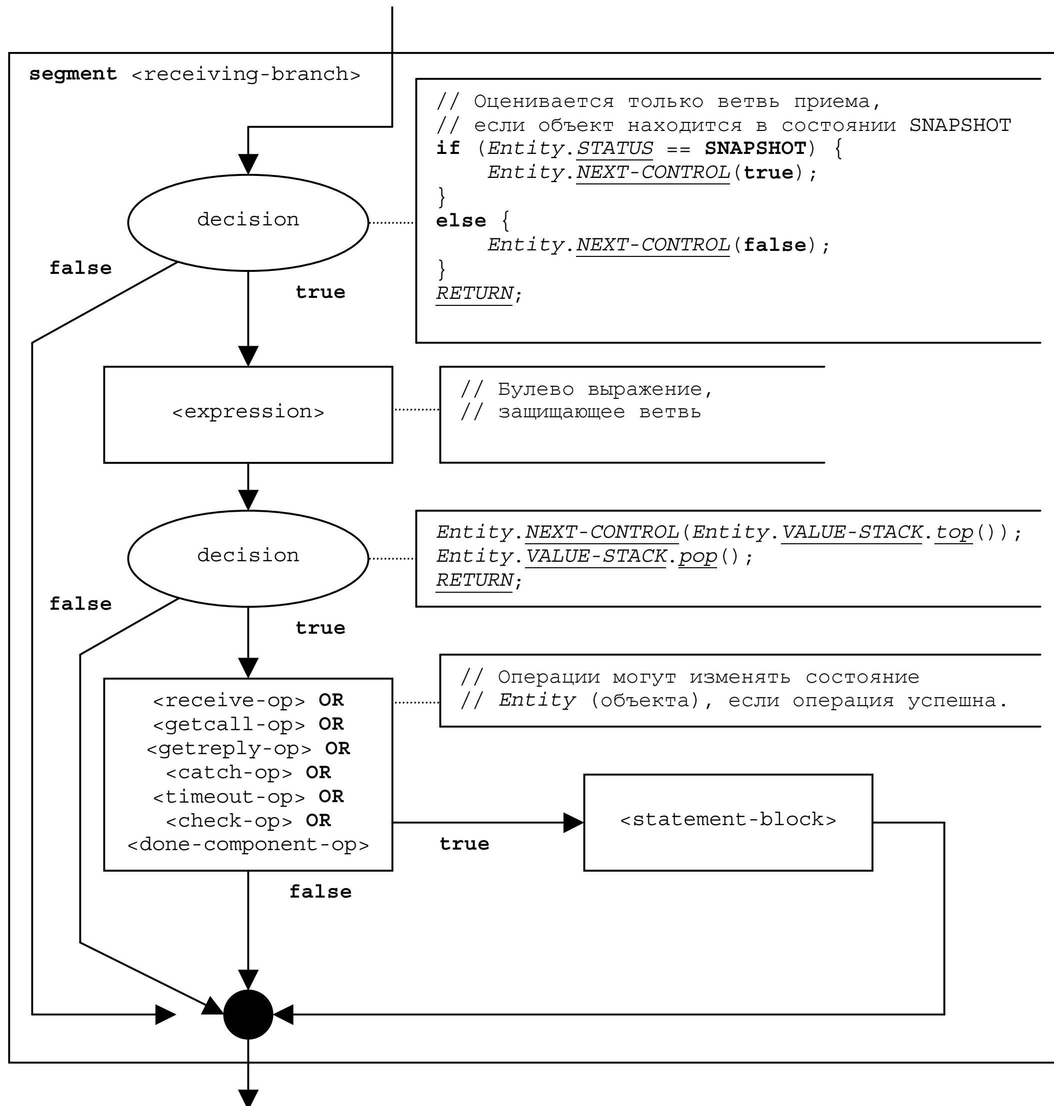


Рисунок 40/Z.143 – Сегмент <receiving-branch> потокового графа

9.3.3 Сегмент <altstep-call-branch> потокового графа

Инициирование альтернативного шага в операторе **alt** описывается сегментом <altstep-call-branch> потокового графа на рисунке 41.

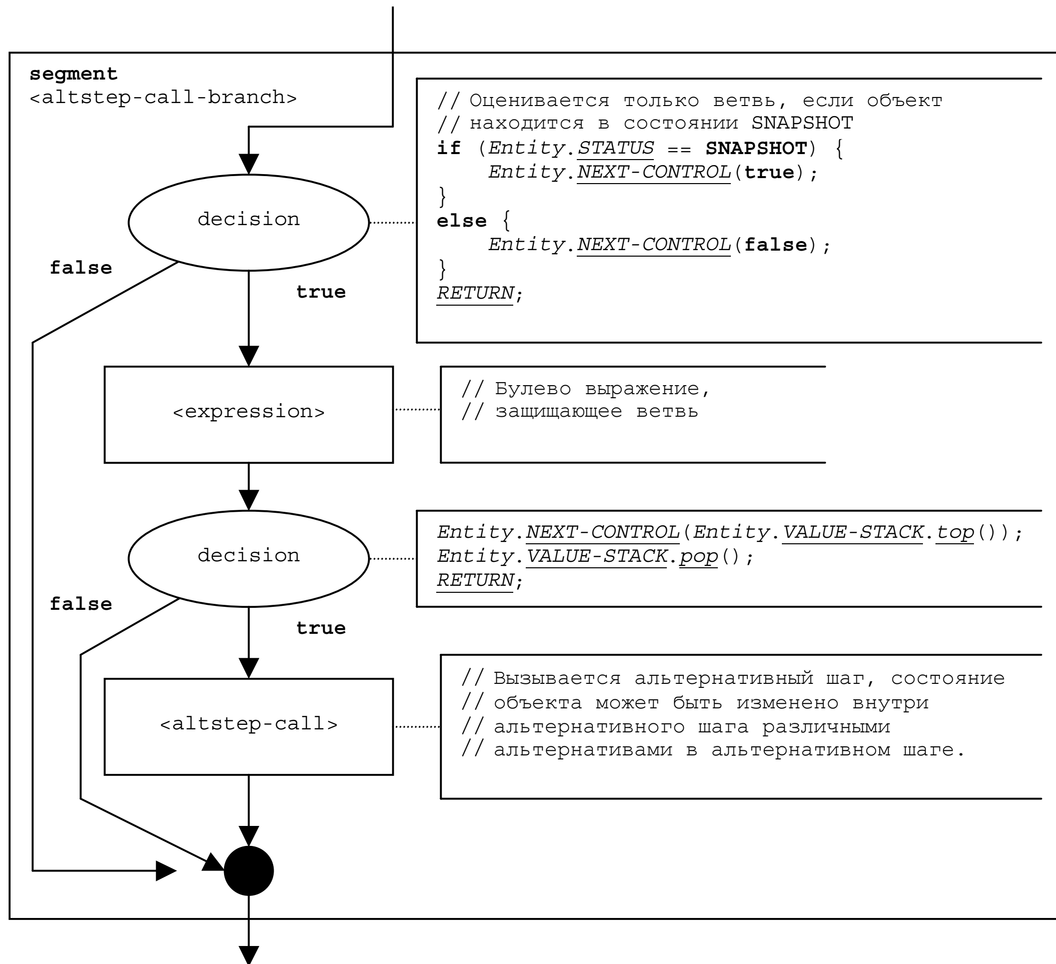


Рисунок 41/Z.143 – Сегмент <altstep-call-branch> потокового графа

9.3.4 Сегмент <else-branch> потокового графа

На рисунке 42 описано выполнение ветви **else** внутри оператора **alt** с помощью сегмента <else-branch> потокового графа.

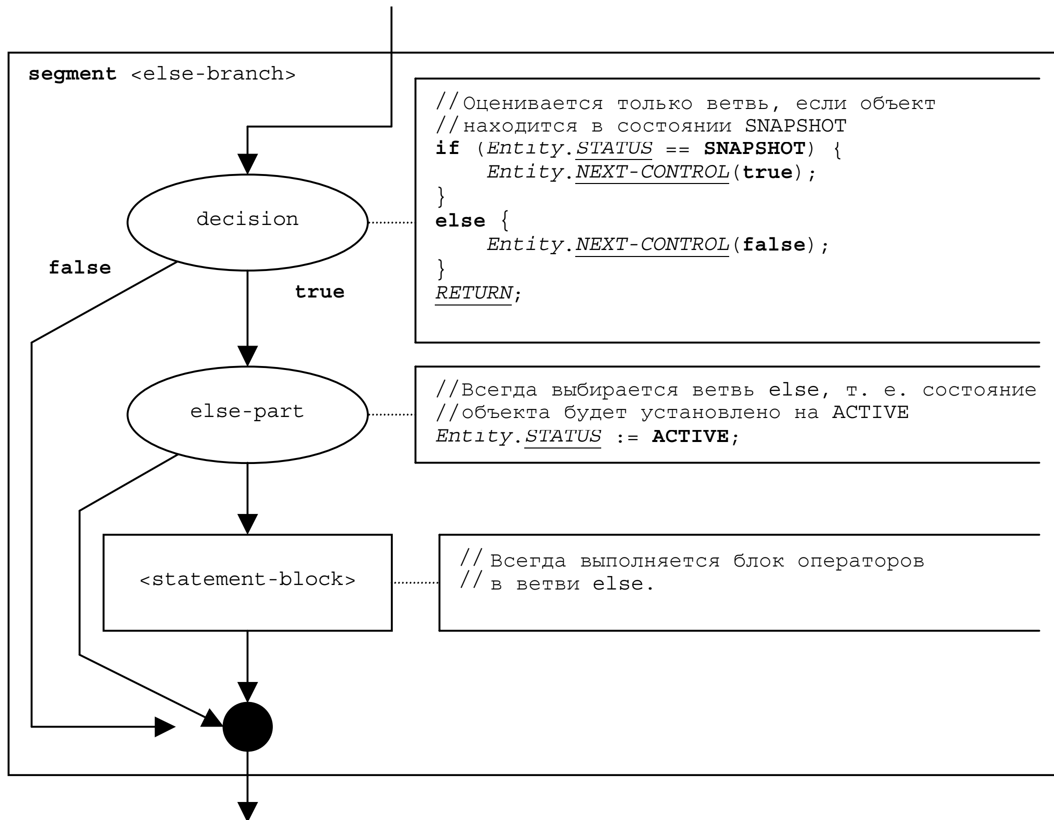


Рисунок 42/Z.143 – Сегмент <else-branch> потокового графа

9.3.5 Сегмент <default-evocation> потокового графа

Вызов поведения по умолчанию в конце операторов **alt** описывается сегментом <default-evocation> потокового графа на рисунке 43.

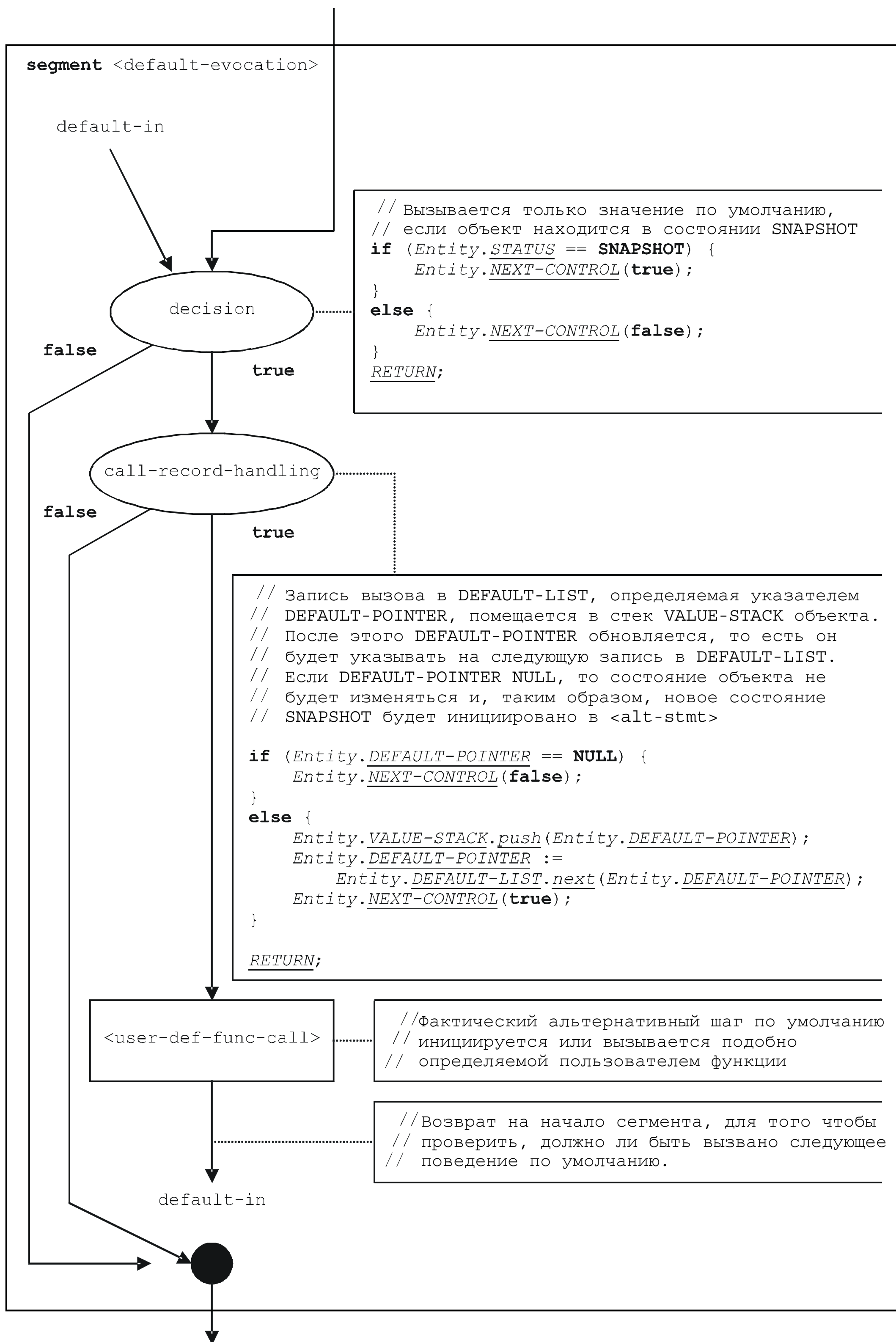


Рисунок 43/Z.143 – Сегмент <default-evocation> потокового графа

9.4 Вызов альтернативного шага

Как показано на рисунке 44, вызов альтернативного шага обрабатывается подобно вызову функции.

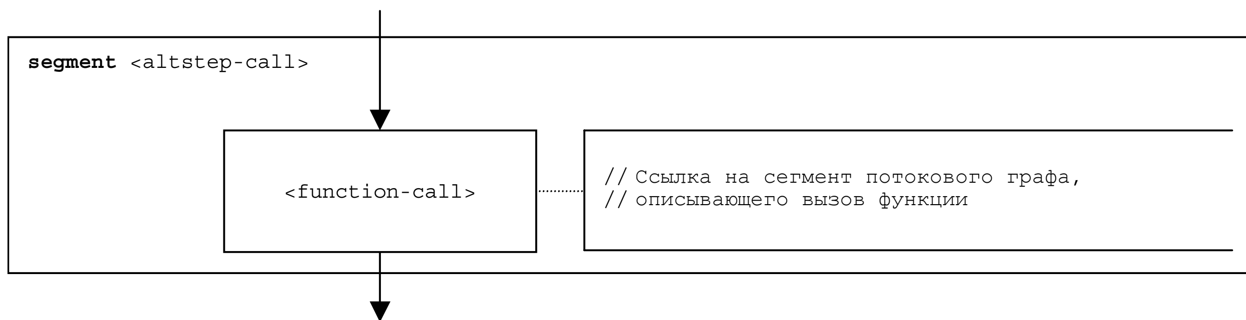


Рисунок 44/Z.143 – Сегмент <altstep-call> потокового графа

9.5 Оператор assignment

Синтаксическая структура оператора **assignment** (присвоение) имеет следующий вид:

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

Значение выражения <expression> присваивается переменной <varId>. Выполнение оператора присвоения определяется сегментом <assignment-stmt> потокового графа на рисунке 45.

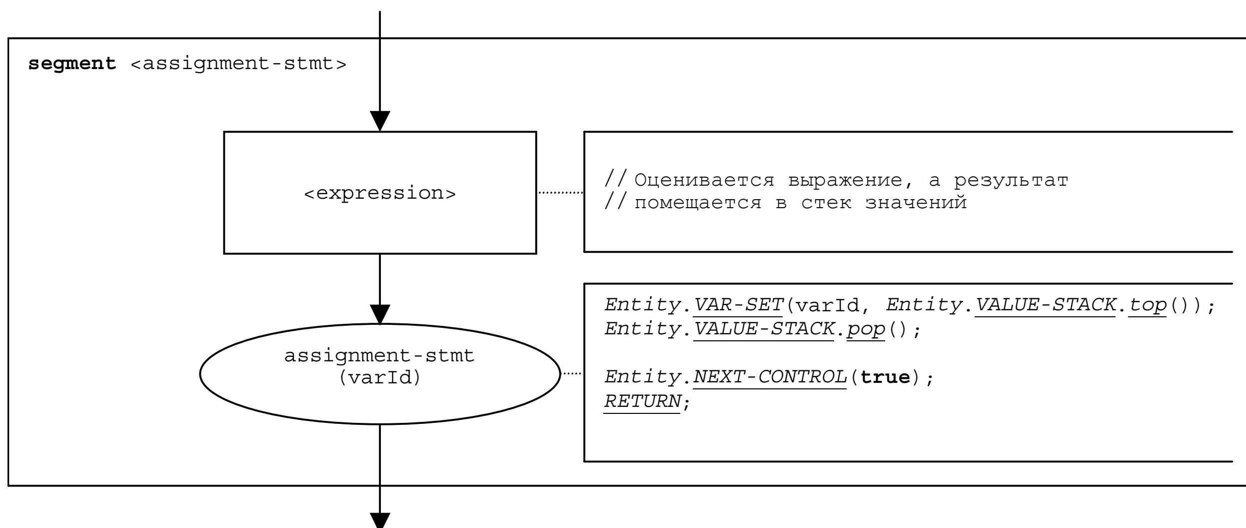


Рисунок 45/Z.143 – Сегмент <assignment-stmt> потокового графа

9.6 Операция call

Синтаксическая структура операции **call** (вызов) имеет следующий вид:

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

Факультативная часть <blocking-info> состоит либо из ключевого слова **nowait** либо из периода времени для особого состояния тайм-аута. Факультативная часть <component-expression> в разделе **to** указывает на объект получателя. Она может быть предоставлена в виде переменного значения или возвращаемого значения функции. Факультативная часть <call-reception-part> обозначает альтернативный прием в случае блокирующей операции **call**.

В операционной семантике различают *блокирующую* и *неблокирующую* операции **call**. Операция **call** является неблокирующей, если в ней используется ключевое слово **nowait**, или если вызываемая процедура является неблокирующей, т. е. она определяется путем использования ключевого слова **noblock**. Блокирующая операция **call** имеет часть <call-reception-part>.

Сегмент <call-op> потокового графа на рисунке 46 определяет выполнение операции **call**. Он отражает различие между блокирующими и неблокирующими вызовами.

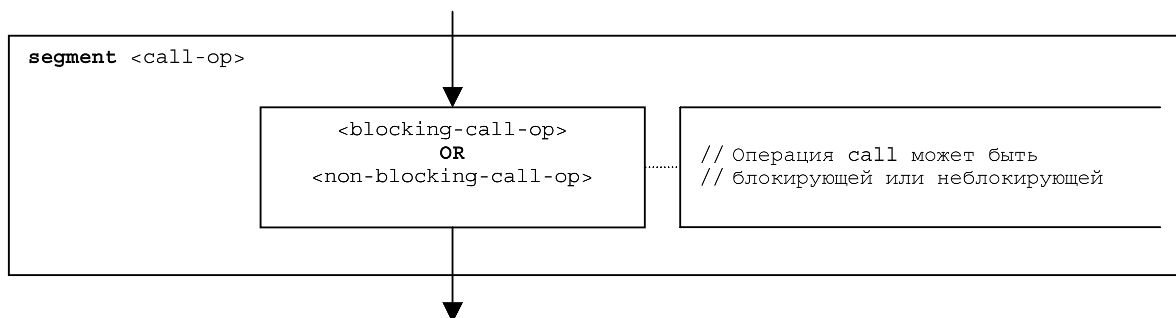


Рисунок 46/Z.143 – Сегмент <call-op> потокового графа

Для блокирующей и неблокирующей операций вызова объект получателя может быть задан в виде выражения. Эти возможности показаны на рисунках 47 и 48.

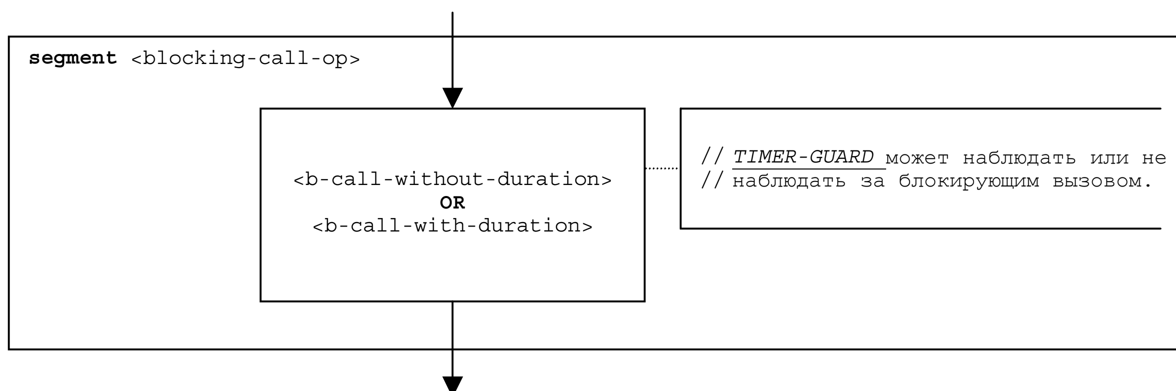


Рисунок 47/Z.143 – Сегмент <blocking-call-op> потокового графа

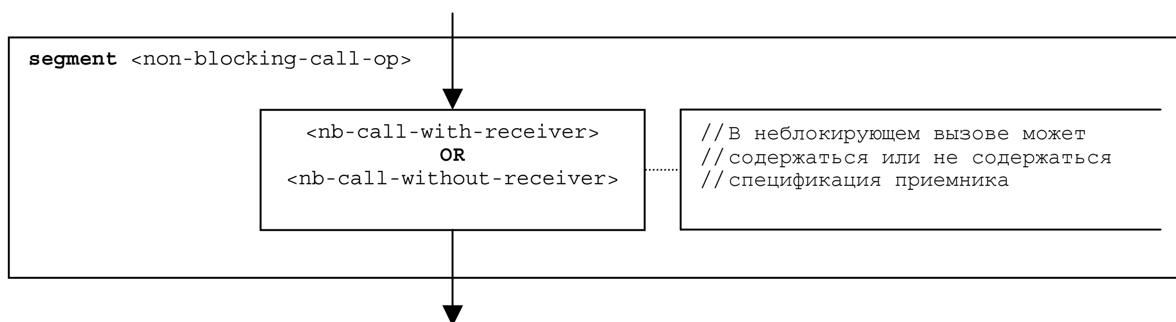


Рисунок 48/Z.143 – Сегмент <non-blocking-call-op> потокового графа

9.6.1 Сегмент <nb-call-with-receiver> потокового графа

Сегмент `<nb-call-with-receiver>` потокового графа на рисунке 49 определяет выполнение неблокирующей операции **`call,`** где получатель задан в виде выражения.

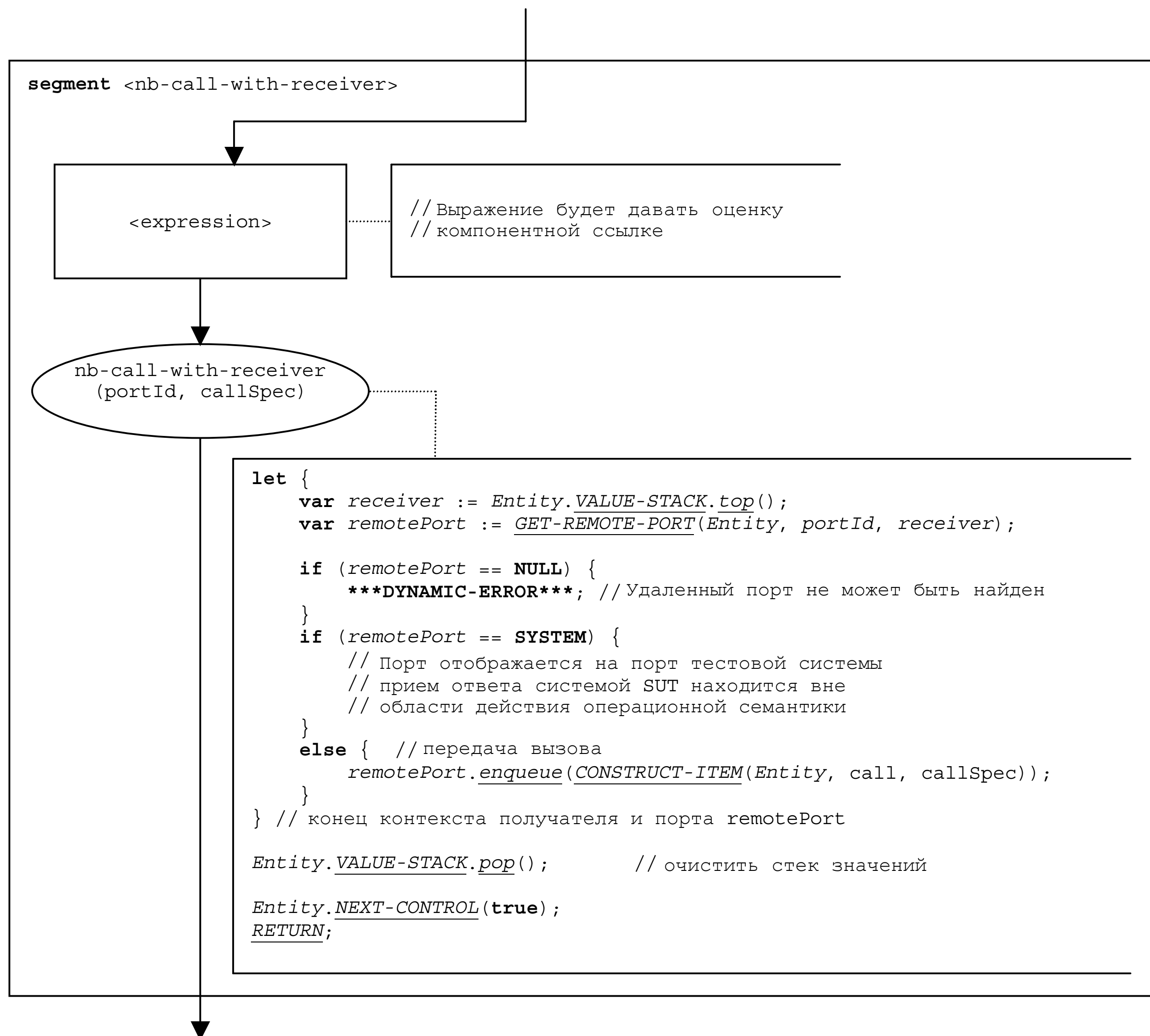


Рисунок 49/Z.143 – Сегмент <nb-call-with-receiver> потокового графа

9.6.2 Сегмент <nb-call-without-receiver> потокового графа

Сегмент `<nb-call-without-receiver>` потокового графа на рисунке 50 определяет выполнение неблокирующей операции **`call`** без **`to`**-раздела.

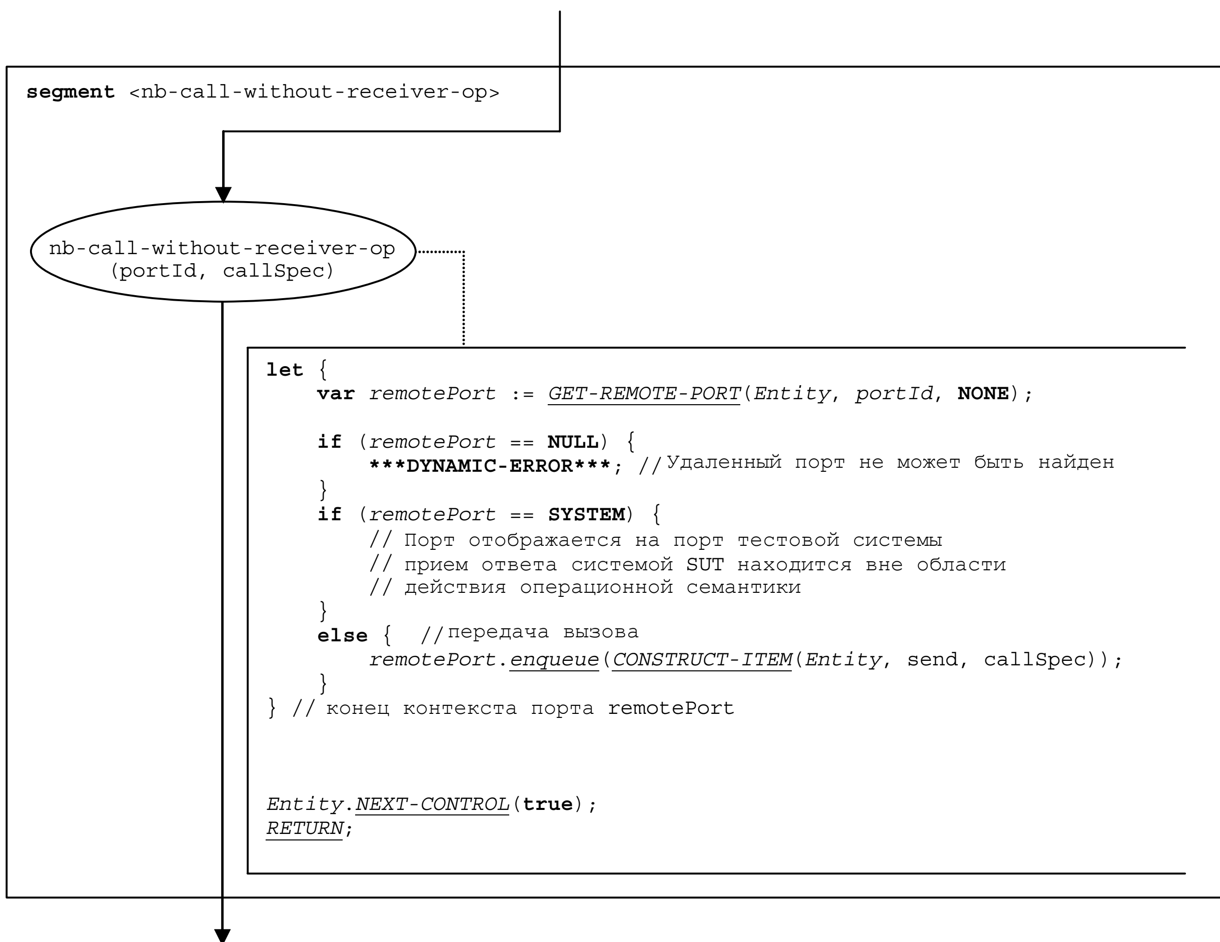


Рисунок 50/Z.143 – Сегмент <nb-call-without-receiver> потокового графа

9.6.3 Сегмент <b-call-without-duration> потокового графа

Блокирующие вызовы моделируются неблокирующим вызовом, за которым следует тело вызова, которое обрабатывает ответы и особые состояния. Сегмент `<b-call-without-duration>` потокового графа, показанный на рисунке 51, описывает выполнение блокирующего вызова без заданного периода времени в качестве защитного временного интервала.

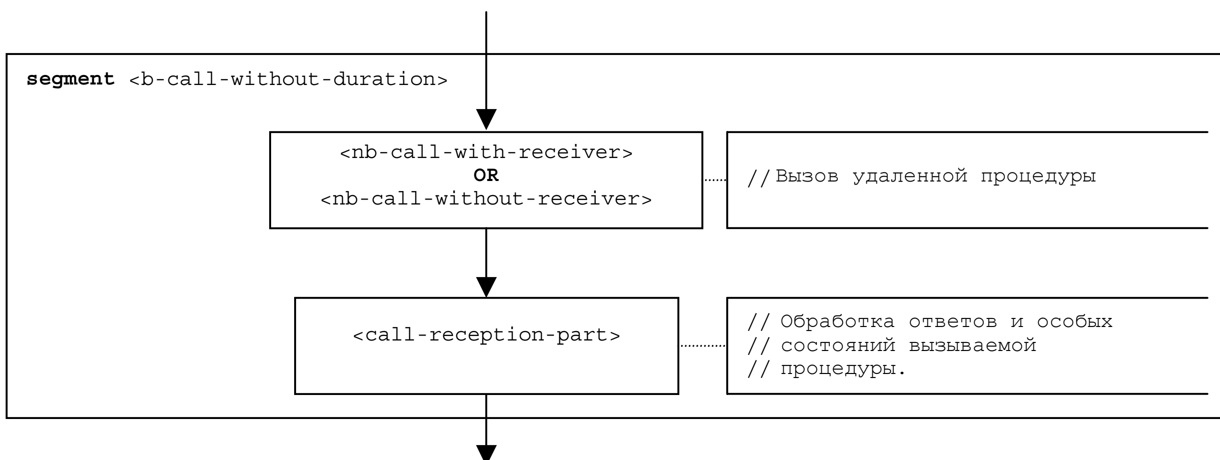


Рисунок 51/Z.143 – Сегмент <b-call-without-duration> потокового графа

9.6.4 Сегмент <b-call-with-duration> потокового графа

Сегмент <b-call-with-duration> потокового графа (см. рисунок 52) описывает выполнение блокирующего вызова с указанием того или иного периода времени в качестве защитного временного интервала.

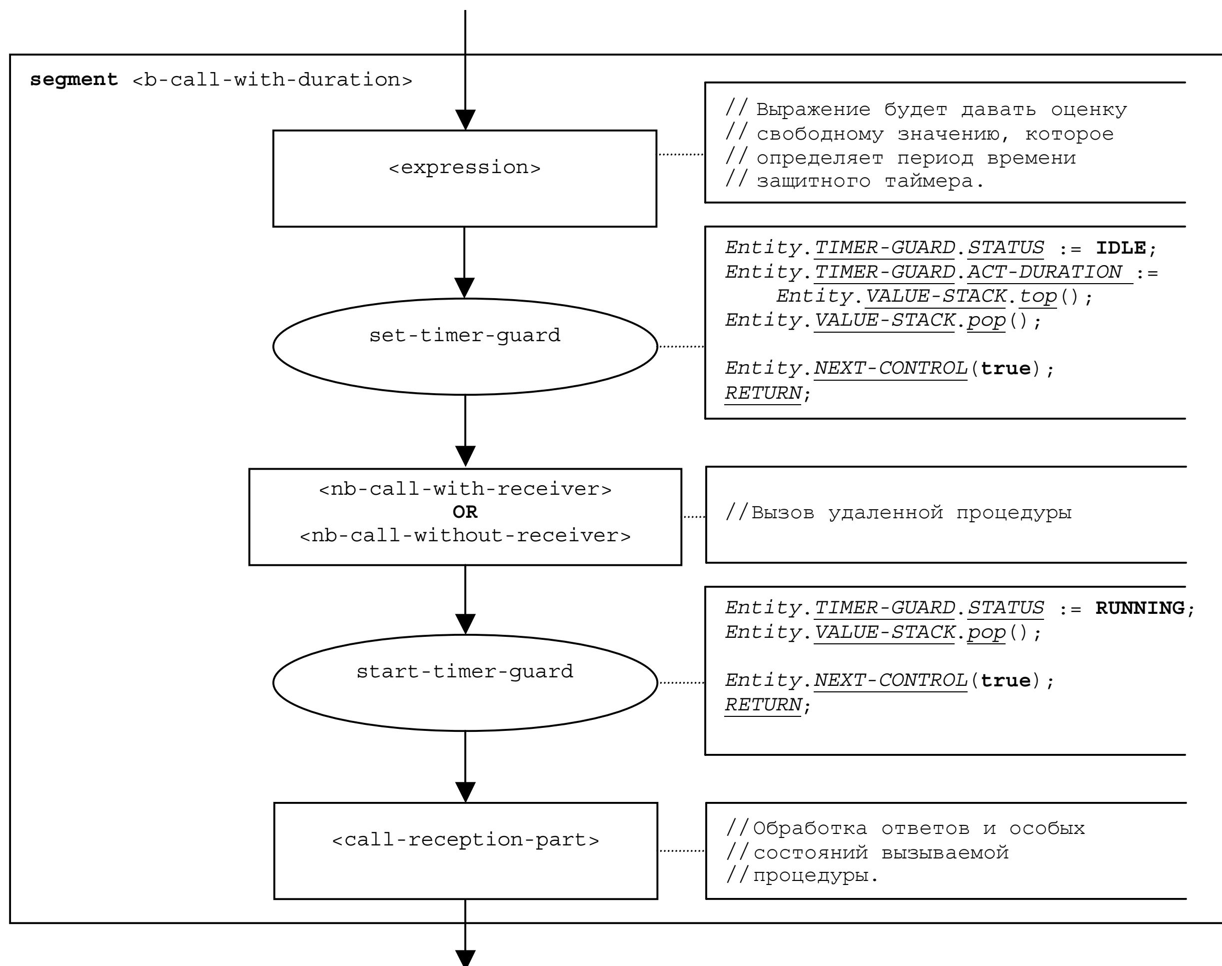


Рисунок 52/Z.143 – Сегмент <b-call-with-duration> потокового графа

9.6.5 Сегмент <call-reception-part> потокового графа

Сегмент `<call-reception-part>` потокового графа (см. рисунок 53) описывает обработку ответов, особых состояний и особого состояния тайм-аута для блокирующей операции **`call`**.

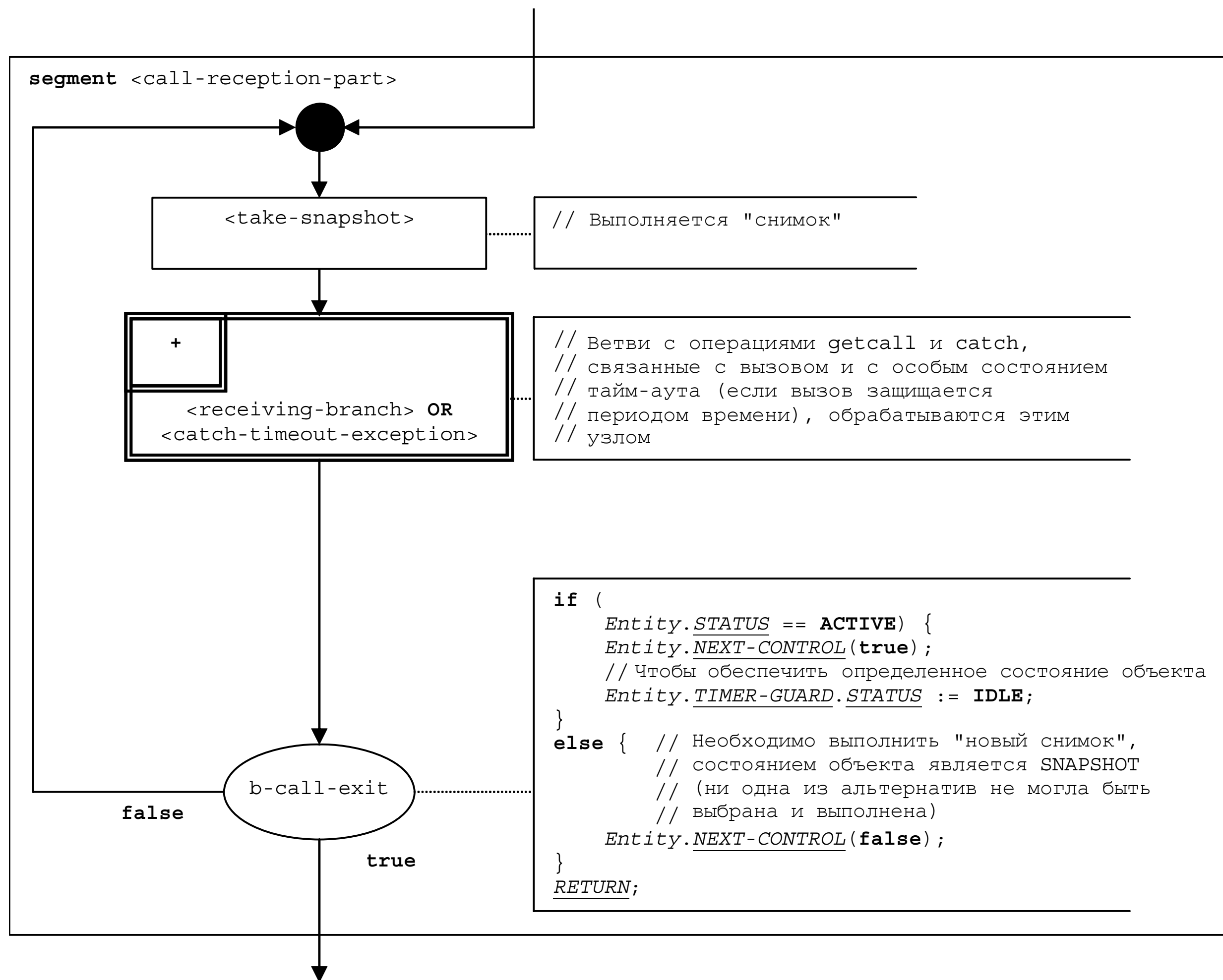


Рисунок 53/Z.143 – Сегмент <call-reception-part> потокового графа

9.6.6 Сегмент <catch-timeout-exception> потокового графа

Сегмент `<catch-timeout-exception>` потокового графа (см. рисунок 54) служит для обработки особого состояния с тайм-аутом для операции блокирующего вызова, которая защищается тем или иным периодом времени.

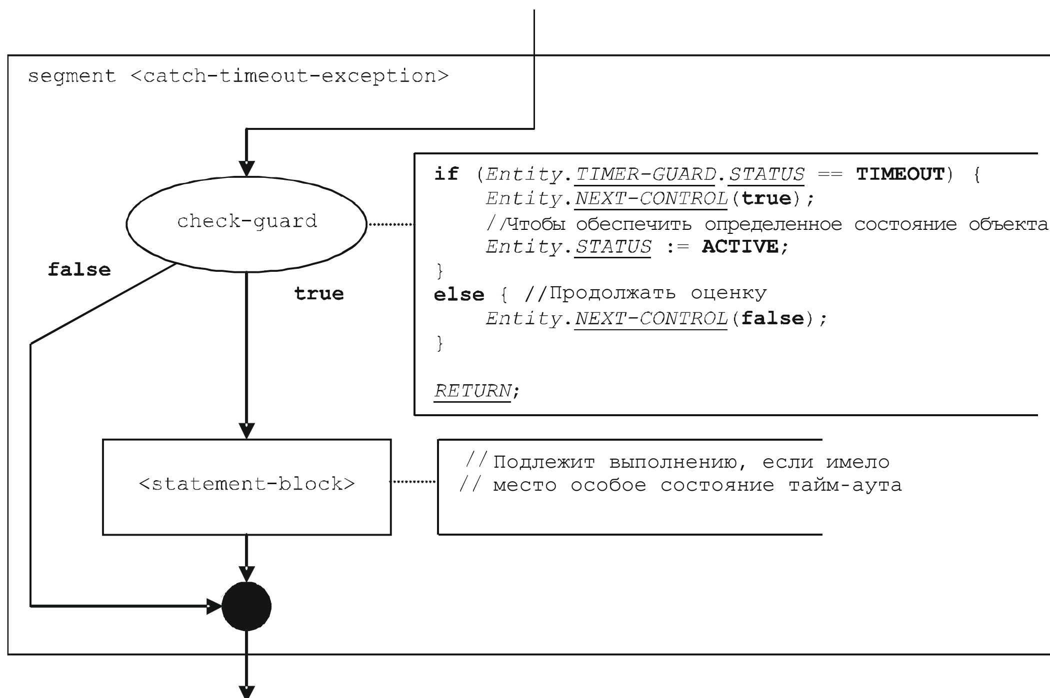


Рисунок 54/Z.143 – Сегмент <catch-timeout-exception> потокового графа

9.7 Операция catch

Синтаксическая структура операции **catch** (захват) имеет следующий вид:

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

Если не учитывать ключевое слово **`catch`**, эта синтаксическая структура идентична синтаксической структуре операции **`receive`**. Поэтому операция **`catch`** обрабатывается в операционной семантике тем же способом, что и операция **`receive`**. Это также показано в сегменте `<catch-op>` потокового графа (рисунок 55), который определяет выполнение операции **`catch`**. Данный рисунок указывает на сегменты потокового графа, связанные с операцией **`receive`** (см. п. 9.37).

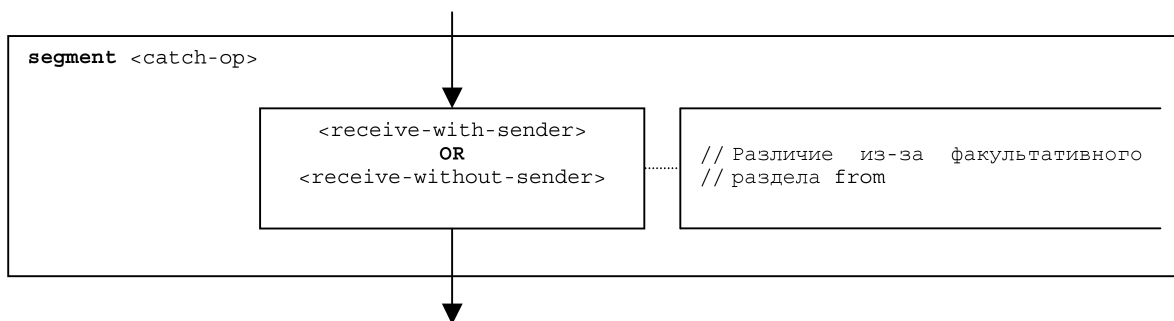


Рисунок 55/Z.143 – Сегмент <catch-op> потокового графа

9.8 Операция check

Синтаксическая структура операции **check** (проверка) имеет следующий вид:

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

Факультативное выражение <component-expression> в разделе **from** указывает на объект отправителя. Оно может быть предоставлено в виде значения переменной или возвращаемого значения функции, т. е. предполагается, что оно является выражением. Факультативная часть <assignmentPart> обозначает присвоение полученной информации, если полученная информация сравнивается со спецификацией сопоставления <matchingSpec> и с (факультативным) разделом **from**.

В операционной семантике операции **receive**, **getcall**, **catch** и **getreply** обрабатываются одинаковым способом, т. е. они описываются путем ссылки на одни и те же сегменты <receive-with-sender> и <receive-without-sender> потокового графа. В операции проверки различные операции также обрабатываются одинаковым способом. Так, сегмент <check-op> потокового графа на рисунке 56, определяющий выполнение операции **check**, также указывает только на два сегмента потокового графа. Единственное различие между сегментами <receive-with-sender> и <receive-without-sender> потокового графа состоит в том, что полученные элементы не удаляются после сопоставления.

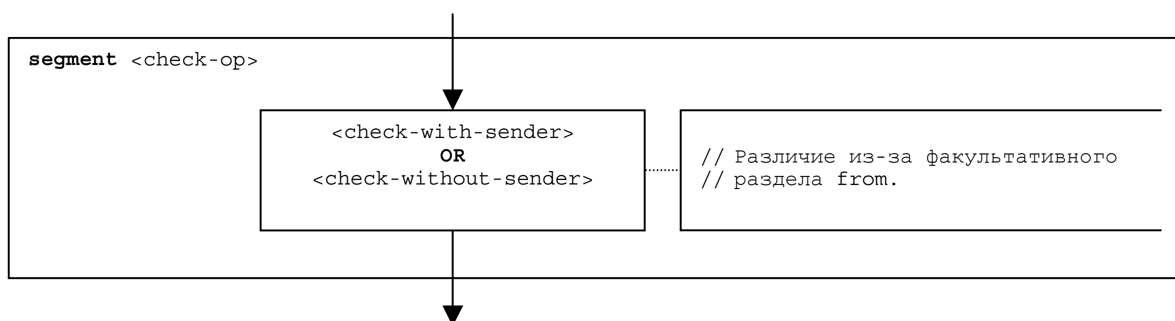


Рисунок 56/Z.143 – Сегмент <check-op> потокового графа

9.8.1 Сегмент <check-with-sender> потокового графа

Сегмент `<check-with-sender>` потокового графа на рисунке 57 определяет выполнение операции **check**, где отправитель задан в виде выражения.

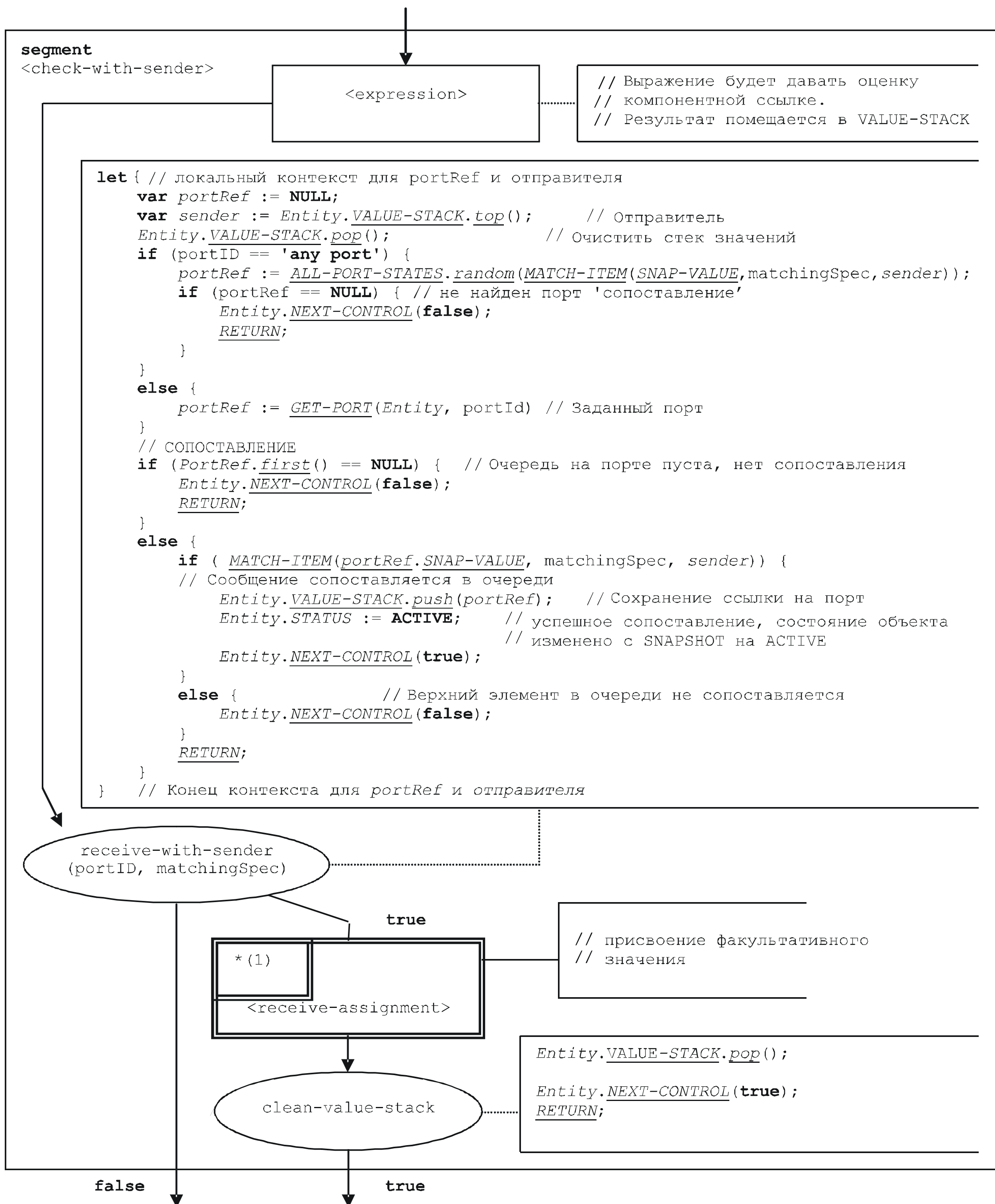


Рисунок 57/Z.143 – Сегмент <check-with-sender> потокового графа

9.8.2 Сегмент <check-without-sender> потокового графа

Сегмент `<check-without-sender>` потокового графа на рисунке 58 определяет выполнение операции **`check`** без раздела **`from`**.

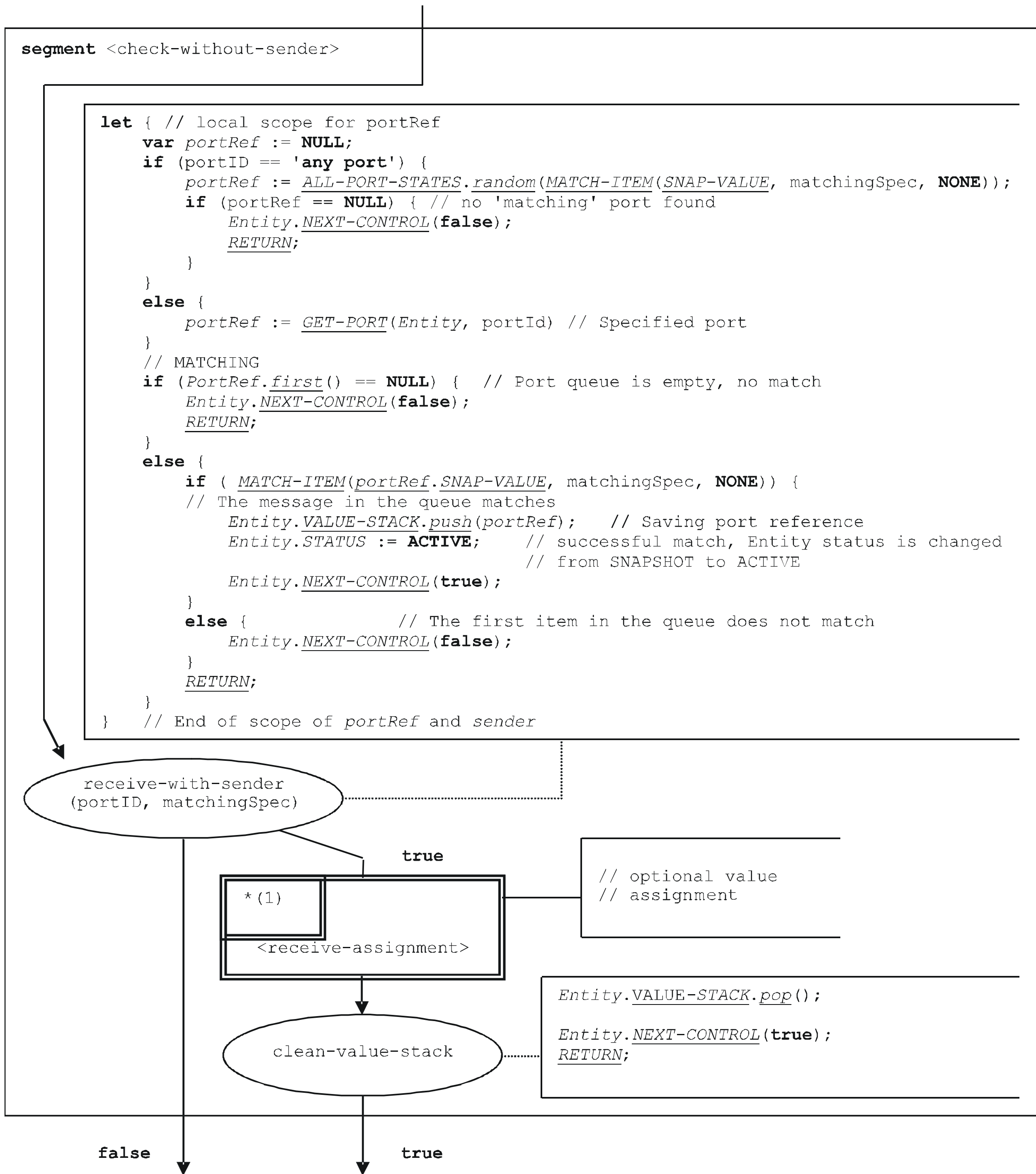


Рисунок 58/Z.143 – Сегмент <check-without-sender> потокового графа

9.9 Операция clear порта

Синтаксическая структура операции **clear** порта имеет следующий вид:

<portId>**.clear**

Сегмент <clear-port-op> потокового графа на рисунке 59 определяет выполнение операции **clear** порта.

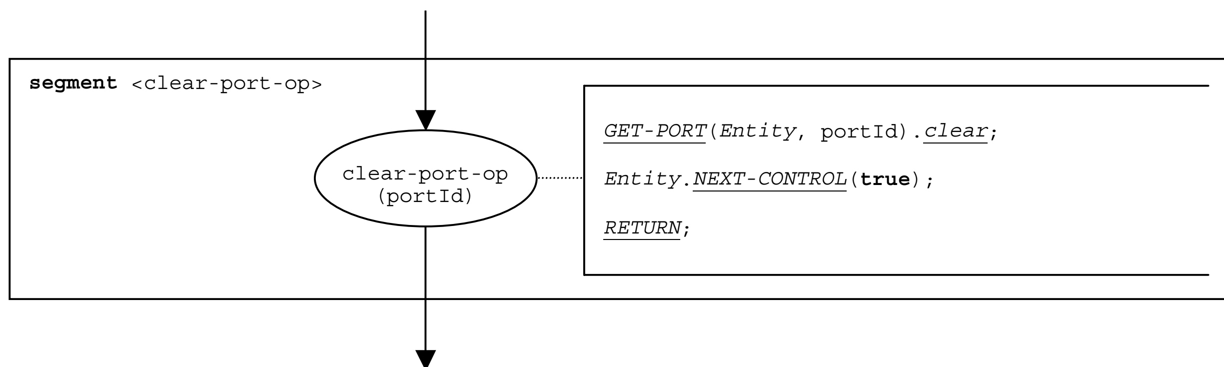


Рисунок 59/Z.143 – Сегмент <clear-port-op> потокового графа

9.10 Операция connect

Синтаксическая структура операции **connect** (соединение) имеет следующий вид:

connect(<component-expression$_1$>:<portId1>, <component-expression$_2$>:<portId2>)

Идентификаторы <portId1> и <portId2> считаются идентификаторами порта соответствующих тестовых компонентов. На компоненты, к которым принадлежат порты, ссылаются посредством компонентных ссылок <component-expression$_1$> и <component-expression$_2$>. Ссылки могут храниться в переменных или возвращаться функцией, т. е. они являются выражениями, которые дают оценку компонентным ссылкам. Для хранения компонентных ссылок используется стек значений.

Выполнение операции **connect** определяется сегментом <connect-op> потокового графа, показанным на рисунке 60. Первое выражение в описании потокового графа, которое должно быть оценено, указывает на <component-expression$_1$>, а второе выражение указывает на <component-expression$_2$>, т. е. <component-expression$_2$> находится на вершине стека значений, когда выполняется узел connect-op.

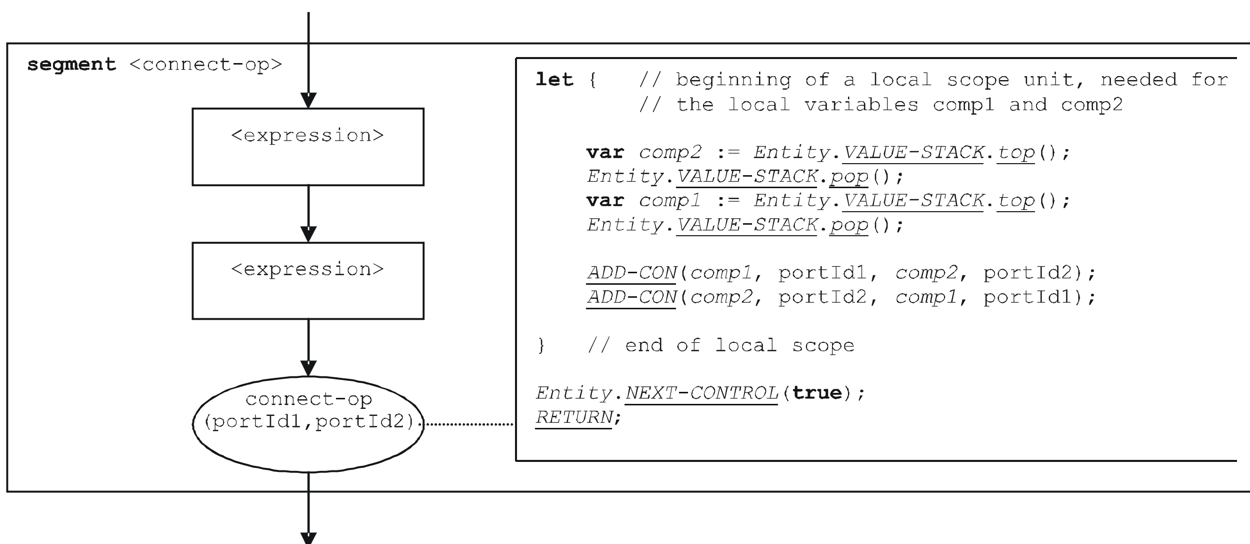


Рисунок 60/Z.143 – Сегмент <connect-op> потокового графа

9.11 Определение constant

Синтаксическая структура определения **constant** (константа) такова:

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

Значение константы считается выражением, оценивающим значение типа константы.

ПРИМЕЧАНИЕ. – Глобальные константы заменяются их значениями на шаге предобработки до того, как будет применяться данная семантика (см. п. 9.2). Локальные константы обрабатываются подобно описаниям переменных с инициализацией. Корректное использование констант, т. е. константы никогда не должны находиться на левой стороне присвоения, должно проверяться во время статического анализа семантики для модуля TTCN-3.

Сегмент <constant-definition> потокового графа на рисунке 61 определяет выполнение объявления константы, где значение константы представлено в виде выражения.

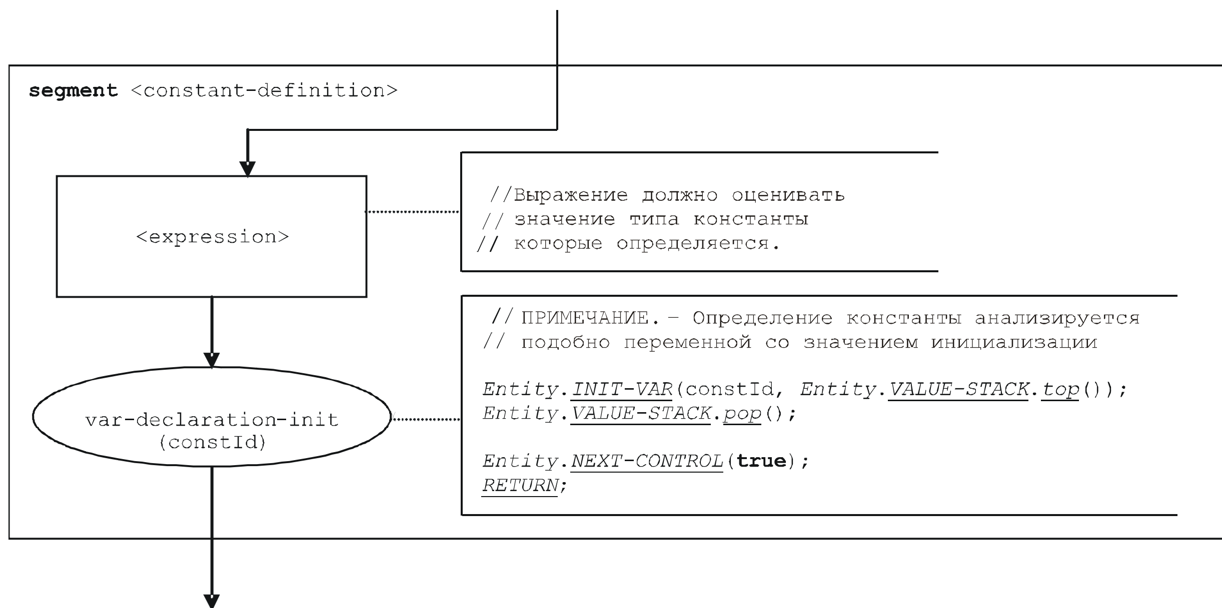


Рисунок 61/Z.143 – Сегмент <constant-definition> потокового графа

9.12 Операция create

Синтаксическая структура операции **create** (создание) имеет следующий вид:

```
<componentTypeId>.create
```

Сегмент <create-op> потокового графа на рисунке 62 определяет выполнение операции **create**.

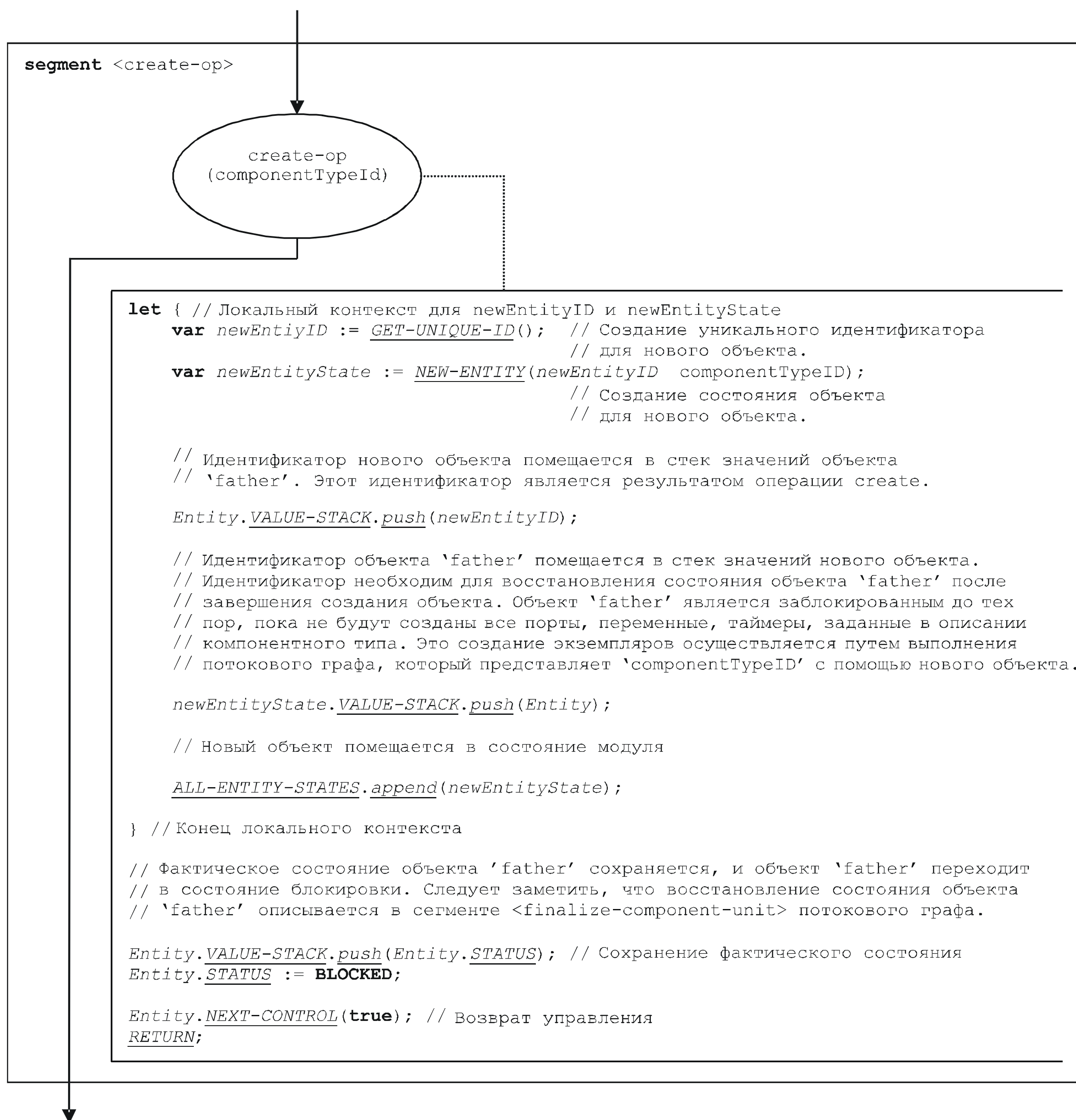


Рисунок 62/Z.143 – Сегмент <create-op> потокового графа

9.13 Оператор deactivate

Синтаксическая структура оператора **deactivate** (деактивировать) имеет следующий вид:

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

Оператор **deactivate** задает деактивацию одного или всех активных значений по умолчанию объекта, который выполняет операцию **deactivate**. Если одно значение по умолчанию деактивируется, то факультативное выражение <default-expression> будет оценивать ссылку по умолчанию, которая определяет деактивируемое значение по умолчанию. Вызов оператора **deactivate** без **<default-expression>** деактивирует все активные значения по умолчанию.

Выполнение оператора **deactivate** определяется сегментом <deactivate-stmt> потокового графа на рисунке 63-a.

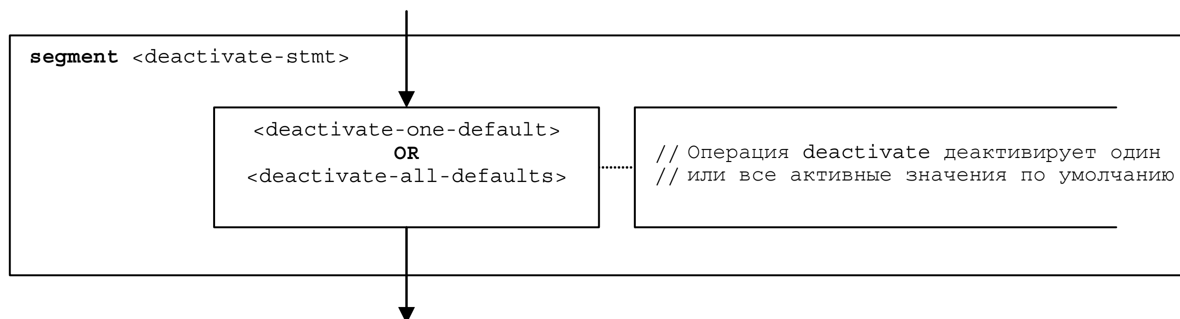


Рисунок 63-a/Z.143 – Сегмент <deactivate-stmt> потокового графа

9.13.1 Сегмент <deactivate-one-default> потокового графа

Сегмент <deactivate-one-default> потокового графа на рисунке 63-b описывает деактивацию одного активного значения по умолчанию. Значение выражения <default-expression> будет оценивать ссылку по умолчанию. Выражение может быть предоставлено в виде значения переменной или функции, возвращающей значение. Оператор **deactivate** удаляет заданное значение по умолчанию из списка *DEFAULT-LIST* объекта, выполняющего оператор **deactivate**.

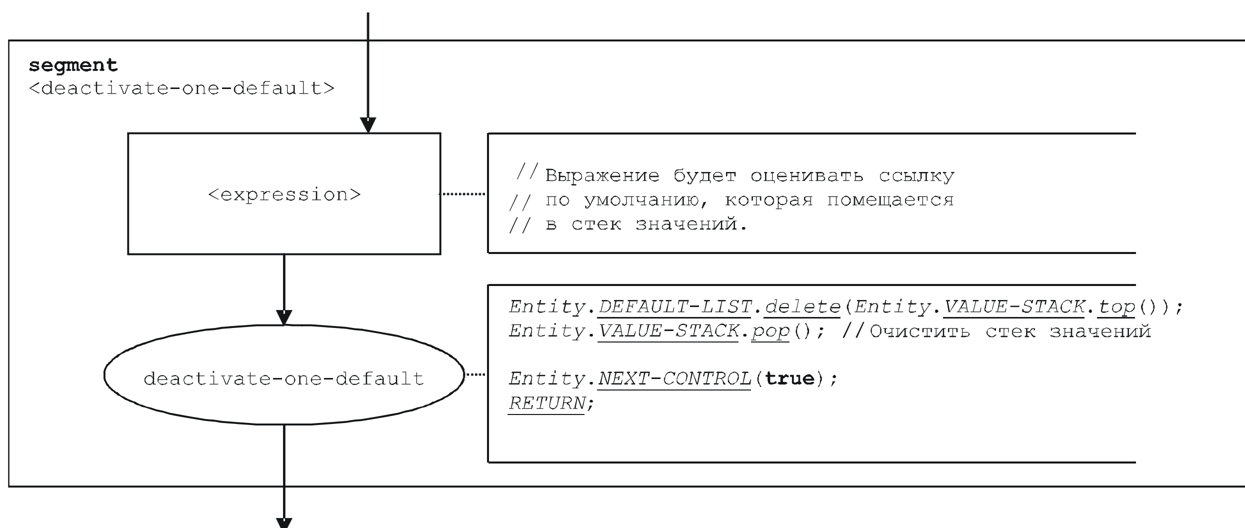


Рисунок 63-b/Z.143 – Сегмент <deactivate-one-default> потокового графа

9.13.2 Сегмент <deactivate-all-defaults> потокового графа

Сегмент <deactivate-all-defaults> потокового графа на рисунке 63-c определяет деактивацию всех активных значений по умолчанию. Оператор деактивации освобождает список *DEFAULT-LIST* объекта, который выполняет оператор **deactivate**.

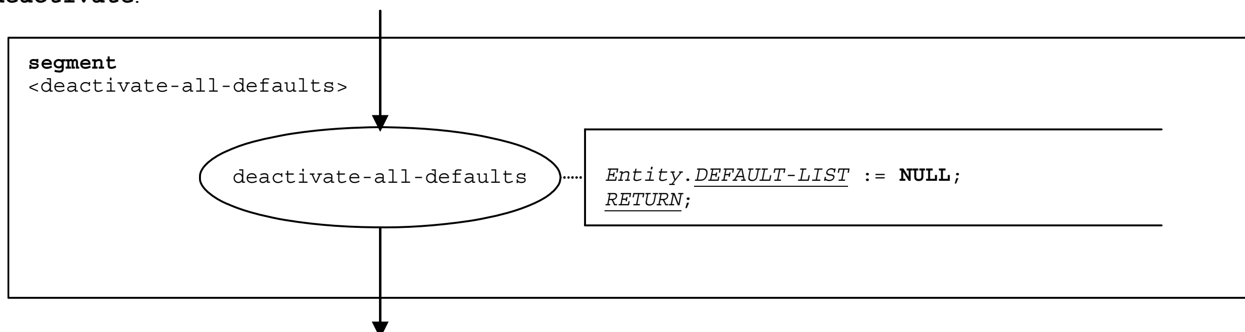


Рисунок 63-c/Z.143 – Сегмент <deactivate-all-defaults> потокового графа

9.14 Операция disconnect

Синтаксическая структура операции **disconnect** (разъединение) имеет следующий вид:

disconnect(<component-expression$_1$>:<portId1>,
<component-expression$_2$>:<portId2>)

Идентификаторы <portId1> и <portId2> считаются идентификаторами портов соответствующих тестовых компонентов. На компоненты, к которым принадлежат порты, ссылаются с помощью компонентных ссылок <component-expression$_1$> и <component-expression$_2$>. Ссылки могут храниться в перемененных или возвращаются функциями, т. е. они являются выражениями, которые дают оценку компонентным ссылкам. Для хранения компонентных ссылок используется стек значений.

Выполнение операции **disconnect** определяется сегментом <disconnect-op> потокового графа, показанным на рисунке 64. Первое выражение в сегменте потокового графа, подлежащее оценке, указывает на <component-expression$_1$>, а второе выражение – на <component-expression$_2$>, т. е. <component-expression$_2$> находится в верхней части стека значений, когда выполняется узел disconnect-op.

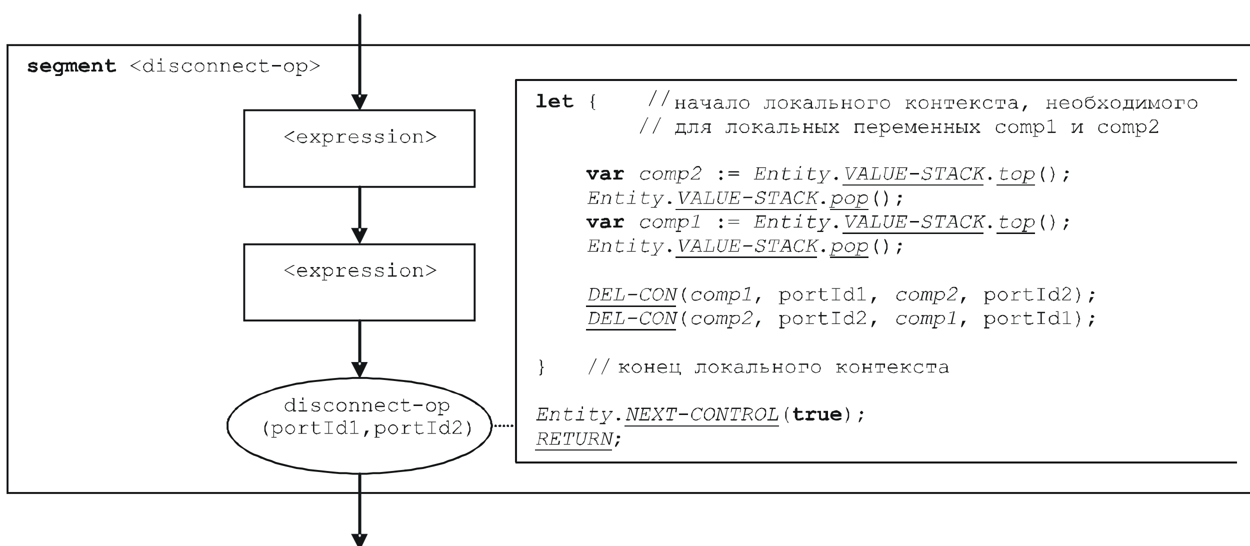


Рисунок 64/Z.143 – Сегмент <disconnect-op> потокового графа

9.15 Оператор do-while

Синтаксическая структура оператора **do-while** имеет следующий вид:

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

Выполнение оператора **do-while** определяется сегментом <do-while-stmt> потокового графа, показанным на рисунке 65.

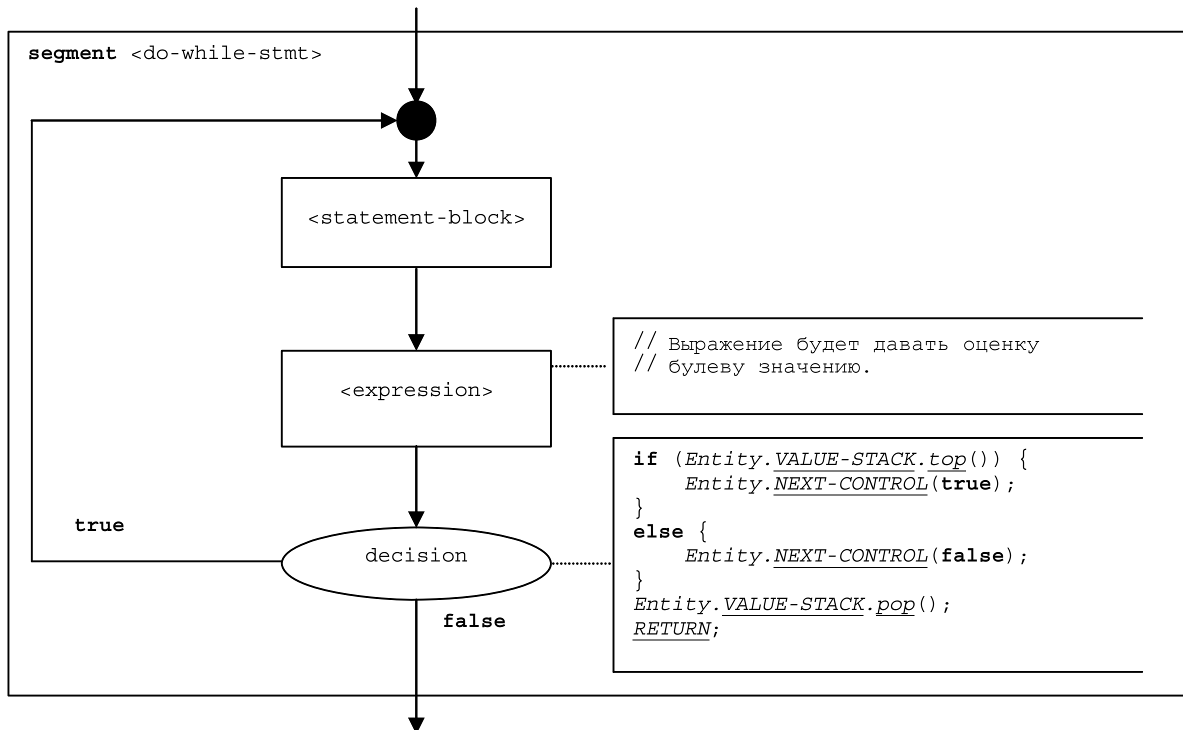


Рисунок 65/Z.143 – Сегмент <do-while-stmt> потокового графа

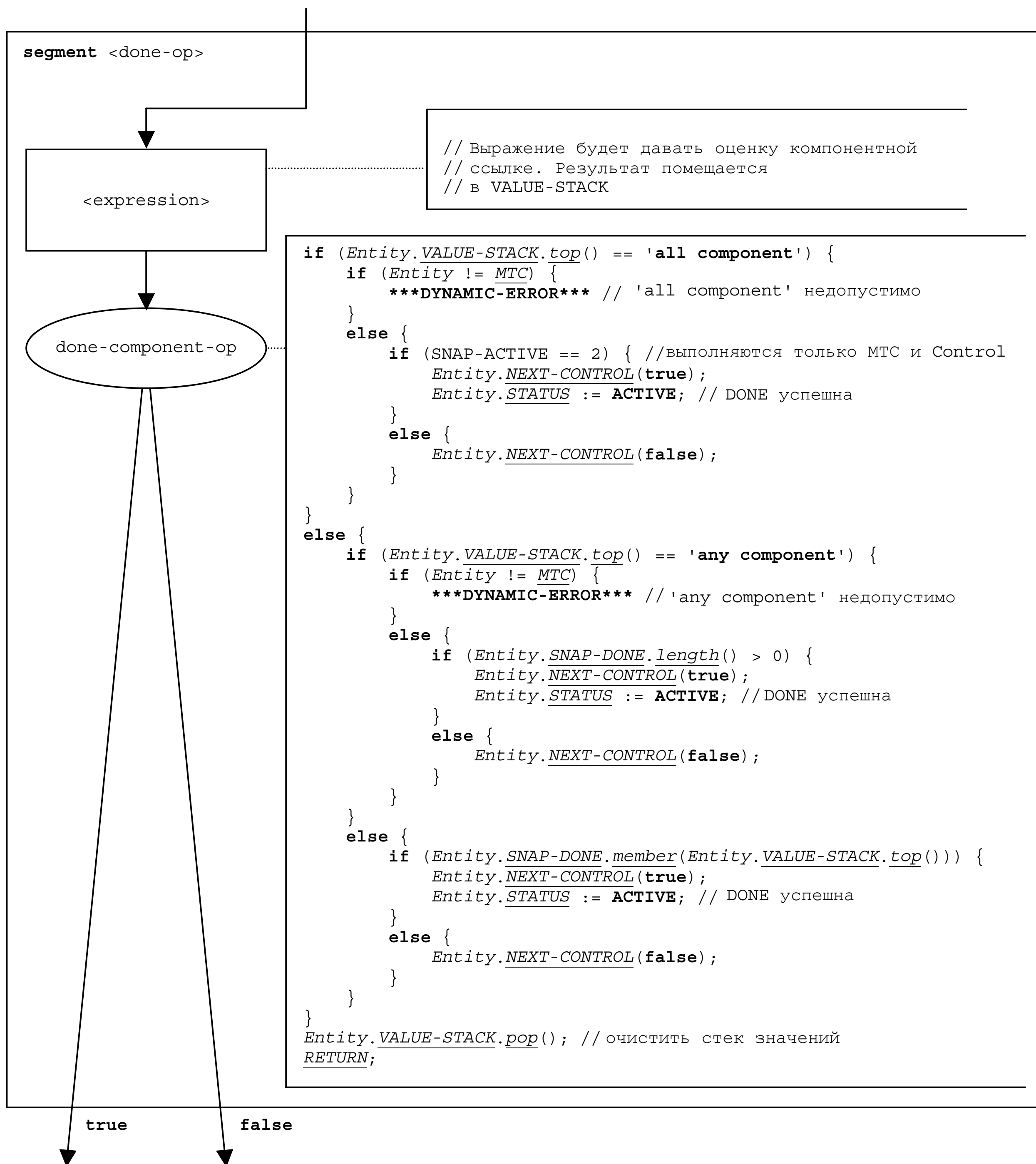
9.16 Компонентная операция done

Синтаксическая структура компонентной операции **done** имеет следующий вид:

<component-expression>.**done**

Компонентная операция **done** проверяет, выполняется ли компонент или он был остановлен. В зависимости от этого операция **done** решает, каким образом будет продолжаться поток управления. Использование компонентной ссылки определяет компонент, подлежащий проверке. Ссылка может храниться в переменной или возвращается функцией, т. е. она является выражением. В целях простоты ключевые слова '**all component**' и '**any component**' считаются специальными выражениями.

Сегмент <done-op> потокового графа на рисунке 66 определяет выполнение компонентной операции **done**.




```
if (Entity.VALUE-STACK.top() == 'all component') {
    if (Entity != MTC) {
        ***DYNAMIC-ERROR*** // 'all component' недопустимо
    }
    else {
        if (SNAP-ACTIVE == 2) { //выполняются только MTC и Control
            Entity.NEXT-CONTROL(true);
            Entity.STATUS := ACTIVE; // DONE успешна
        }
        else {
            Entity.NEXT-CONTROL(false);
        }
    }
}
else {
    if (Entity.VALUE-STACK.top() == 'any component') {
        if (Entity != MTC) {
            ***DYNAMIC-ERROR*** //'any component' недопустимо
        }
        else {
            if (Entity.SNAP-DONE.length() > 0) {
                Entity.NEXT-CONTROL(true);
                Entity.STATUS := ACTIVE; //DONE успешна
            }
            else {
                Entity.NEXT-CONTROL(false);
            }
        }
    }
    else {
        if (Entity.SNAP-DONE.member(Entity.VALUE-STACK.top())) {
            Entity.NEXT-CONTROL(true);
            Entity.STATUS := ACTIVE; // DONE успешна
        }
        else {
            Entity.NEXT-CONTROL(false);
        }
    }
}
Entity.VALUE-STACK.pop(); //очистить стек значений
RETURN;
```

Рисунок 66/Z.143 – Сегмент <done-component-op> потокового графа

9.17 Оператор execute

Синтаксическая структура оператора **execute** (выполнить) имеет следующий вид:

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

Оператор **execute** описывает выполнение тестового примера <testCaseId> с (факультативными) фактическими параметрами $<act\text{-}par_1>, \ldots, <act\text{-}par_n>$. Оператор выполнения может быть факультативно защищен каким-то периодом времени, предоставляемым в виде выражения, которое дает оценку для **float**. Если в течение заданного периода времени тестовый пример не возвратит вердикт, то имеет место особое состояние тайм-аута, тестовый пример останавливается и возвращается вердикт **error**.

ПРИМЕЧАНИЕ. – В операционной семантике останов тестового примера моделируется с помощью останова MTC. В действительности более подходящими могут оказаться другие механизмы.

Если особое состояние тайм-аута не наблюдается, то создается тестовый пример MTC, экземпляр управления (представляющий часть управления модуля TTCN-3) блокируется до тех пор, пока не закончится тестовый пример, а для дальнейшего выполнения тестового примера поток управления передается тестовому примеру MTC. Поток управления выдается обратно экземпляру управления, когда MTC завершается.

Сегмент <execute-stmt> потокового графа на рисунке 67 определяет выполнение оператора **execute**.

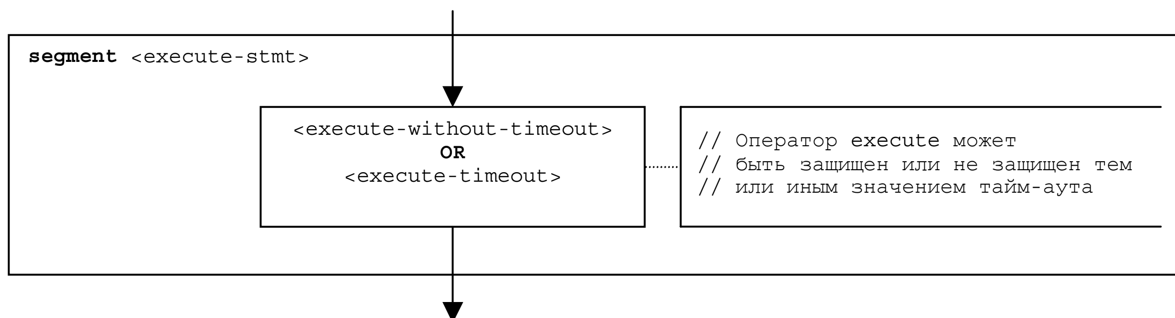


Рисунок 67/Z.143 – Сегмент <execute-stmt> потокового графа

9.17.1 Сегмент <execute-without-timeout> потокового графа

Выполнение тестового примера начинается с создания **mtc**. Затем МТС запускается поведением, определяемым в описании тестового примера. После этого управление модулем ожидает завершения тестового примера. Создание и запуск МТС можно описать, используя операторы **create** и **start**:

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

Сегмент `<execute-without-timeout>` потокового графа на рисунке 68 определяет выполнение оператора **execute** без появления особого состояния тайм-аута путем использования сегментов потокового графа для операций **create** и **start**.

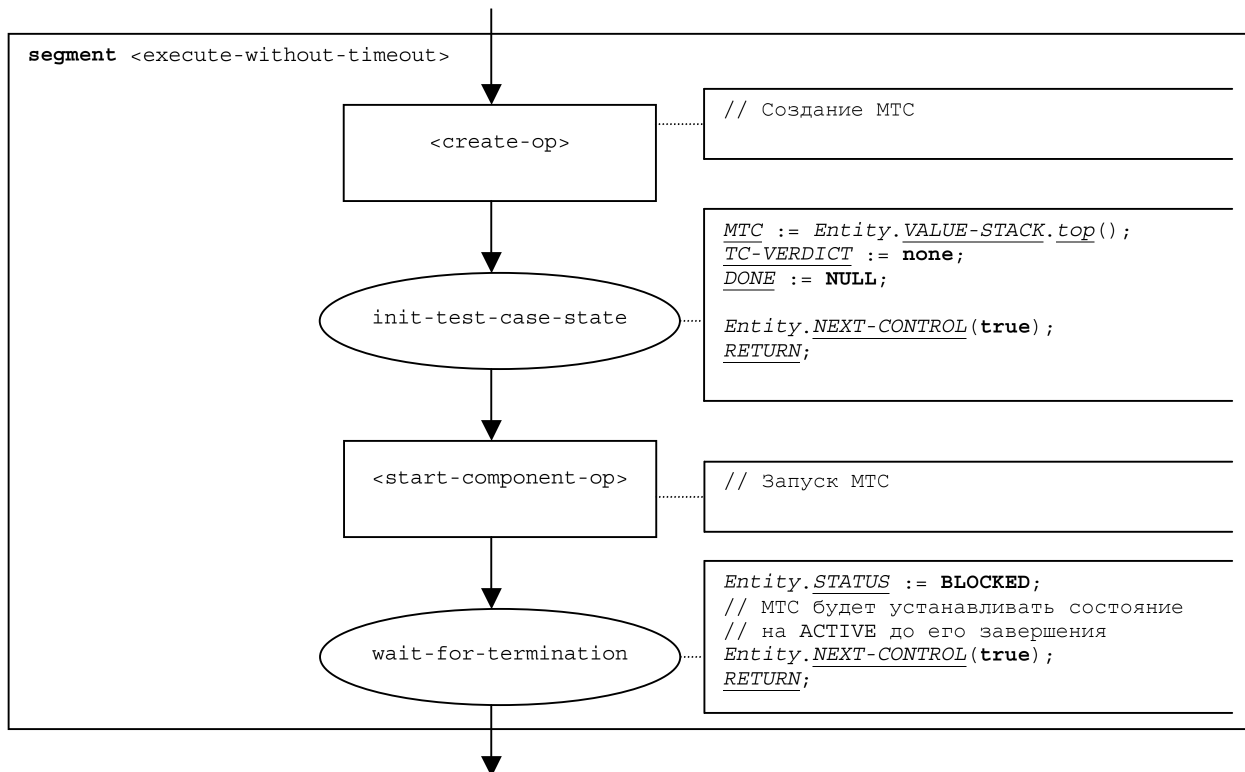


Рисунок 68/Z.143 – Сегмент <execute-without-timeout> потокового графа

9.17.2 Сегмент <execute-timeout> потокового графа

Сегмент `<execute-timeout>` потокового графа на рисунке 69 определяет выполнение оператора **`execute`**, который защищается тем или иным значением тайм-аута. Сегмент потокового графа также моделирует создание и запуск MTC с помощью операции **`create`** и **`start`**. Кроме того, *TIMER-GUARD* защищает процедуру завершения.

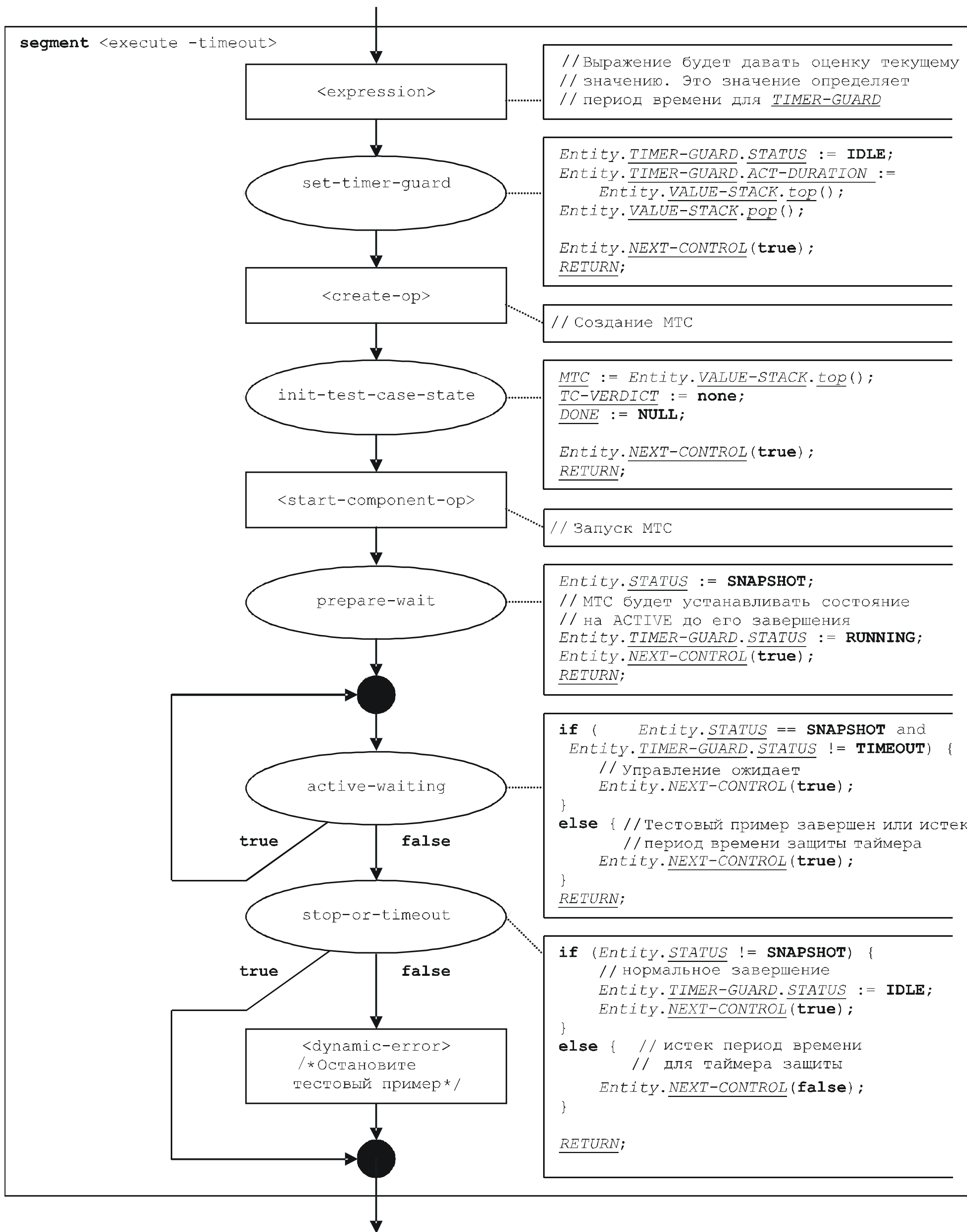


Рисунок 69/Z.143 – Сегмент <execute-timeout> потокового графа

9.18 Выражение

Для обработки выражений необходимо различать следующие четыре случая:

a) выражение является литеральным значением (или константой);

b) выражение представляет собой переменную;

c) выражение является операцией, применяемой к одному или нескольким операндам;

d) выражение является вызовом функции или операции.

Синтаксическая структура выражения такова:

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

где:

`<lit-val>` обозначает литеральное значение;

`<var-val>` обозначает значение переменной;

`<func-op-call>` обозначает вызов функции или операции;

`<operator-appl>` обозначает применение арифметических операторов, таких как +, –, **not** и т. д.

Выполнение выражения определяется сегментом `<expression>` потокового графа, показанным на рисунке 70.

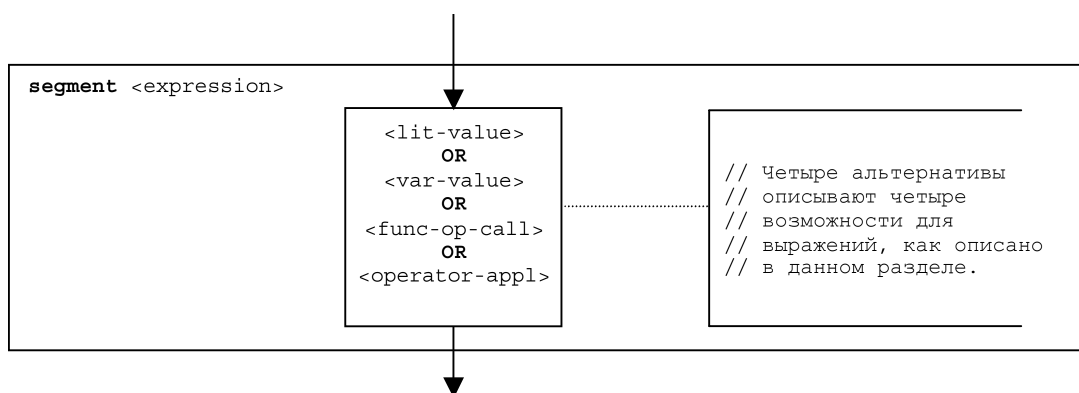


Рисунок 70/Z.143 – Сегмент <expression> потокового графа

9.18.1 Сегмент <lit-value> потокового графа

Сегмент `<lit-value>` потокового графа на рисунке 71 помещает литеральное значение в стек значений объекта.

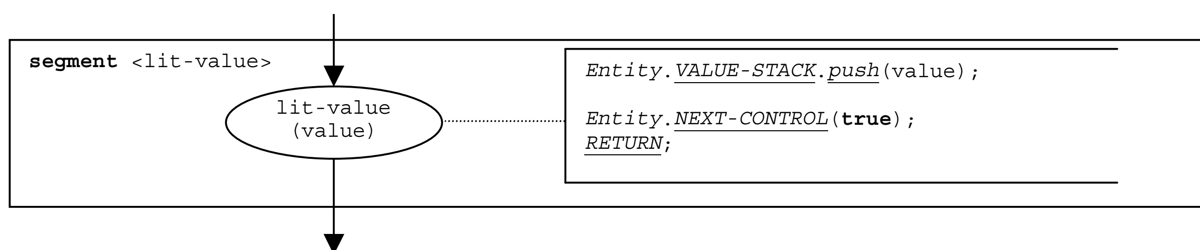


Рисунок 71/Z.143 – Сегмент <lit-value> потокового графа

9.18.2 Сегмент <var-value> потокового графа

Сегмент `<var-value>` потокового графа на рисунке 72 помещает значение переменной в стек значений объекта.

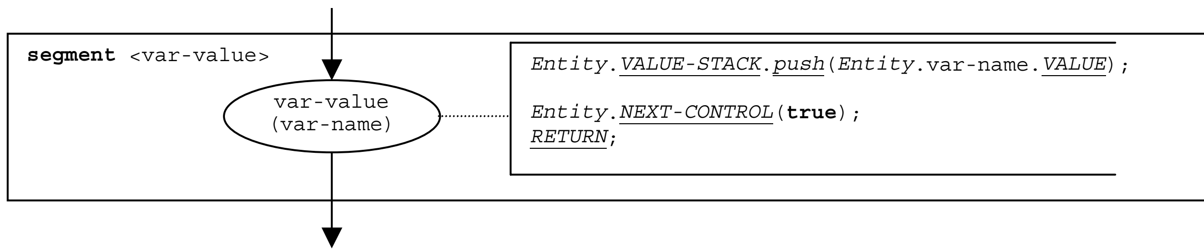


Рисунок 72/Z.143 – Сегмент <var-value> потокового графа

9.18.3 Сегмент <func-op-call> потокового графа

Сегмент `<func-op-call>` потокового графа на рисунке 73 указывает на вызовы функций и операций, которые возвращают значение, помещенное в стек значений объекта. Все эти вызовы считаются выражениями.

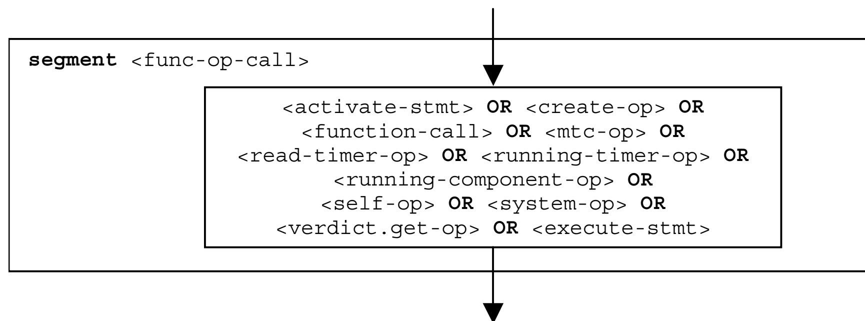


Рисунок 73/Z.143 – Сегмент <func-op-call> потокового графа

9.18.4 Сегмент <operator-appl> потокового графа

Представление потокового графа на рисунке 74 непосредственно указывает на предположение, что для оценки выражений оператора используется обратная польская нотация. Операнды оператора вычисляются и помещаются в оценочный стек. Для применения оператора операнды извлекаются из оценочного стека, и применяется оператор. Результат применения оператора в итоге заносится в оценочный стек. Как извлечение операндов, так и помещение результатов в стек рассматриваются как часть применения оператора (оператор *Entity.APPLY-OPERATOR*(operator) на рисунке 74), т. е. они не моделируются в операционной семантике.

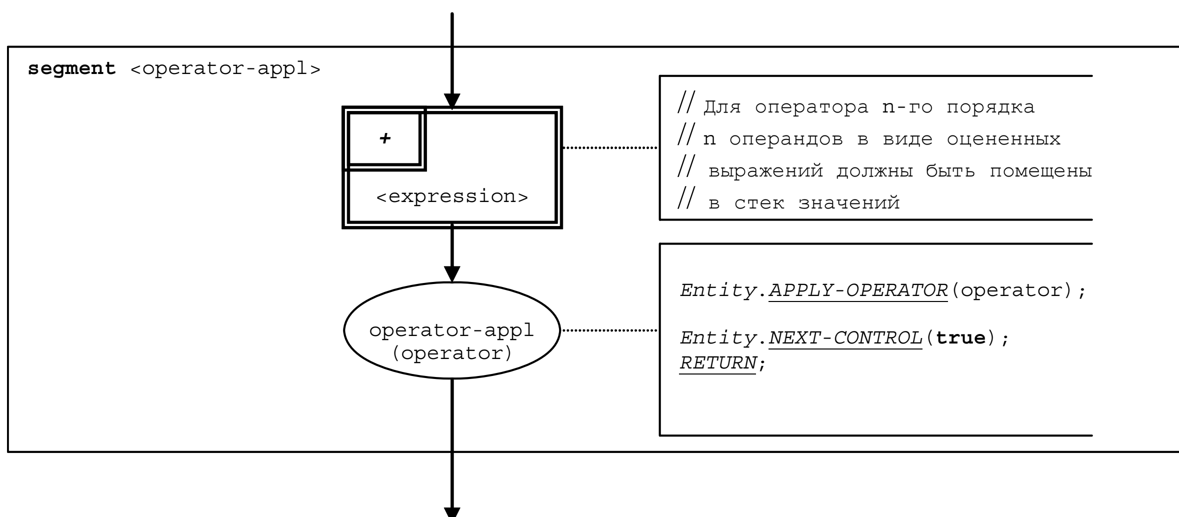


Рисунок 74/Z.143 – Сегмент <operator-appl> потокового графа

9.18b Сегмент <dynamic-error> потокового графа

В случае динамической ошибки сегмент `<dynamic-error>` потокового графа (см. рисунок 74-b) инициируется тестовой системой. Все ресурсы, выделенные для тестового примера, освобождаются, а вердикт **error** присваивается тестовому примеру. Управление передается оператору в управляющей части, следующей за оператором execute, в котором имела место ошибка.

Сегмент `<dynamic-error>` потокового графа инициируется управлением модуля в случае, когда тестовый пример не завершается в течение заданного временного интервала (см. п. 9.17.2).

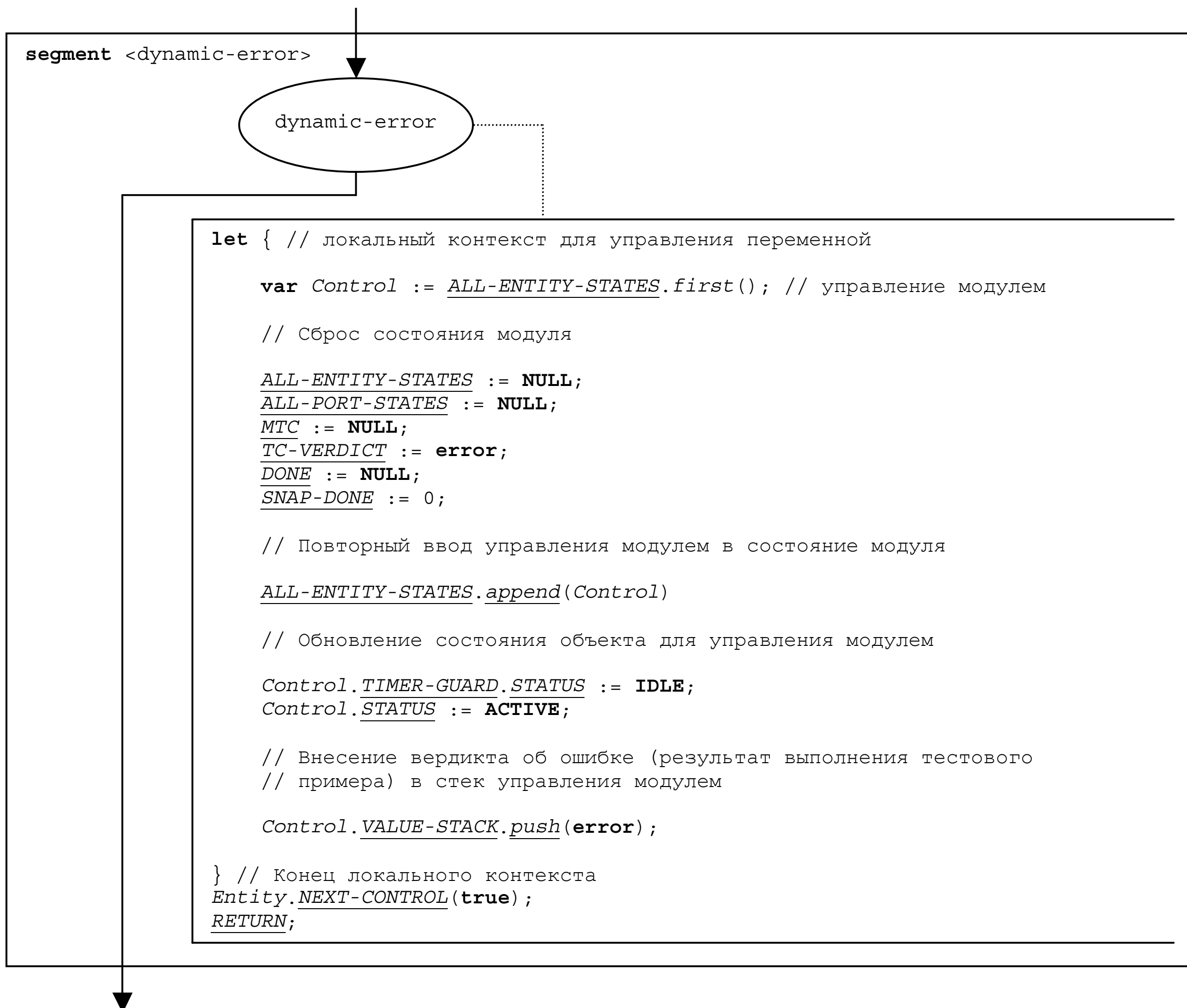


Рисунок 74-b/Z.143 – Сегмент <dynamic-error> потокового графа

9.19 Сегмент <finalize-component-init> потокового графа

Сегмент `<finalize-component-init>` потокового графа является частью потокового графа, представляющего поведение определения компонентного типа. На рисунке 75 показано его выполнение.

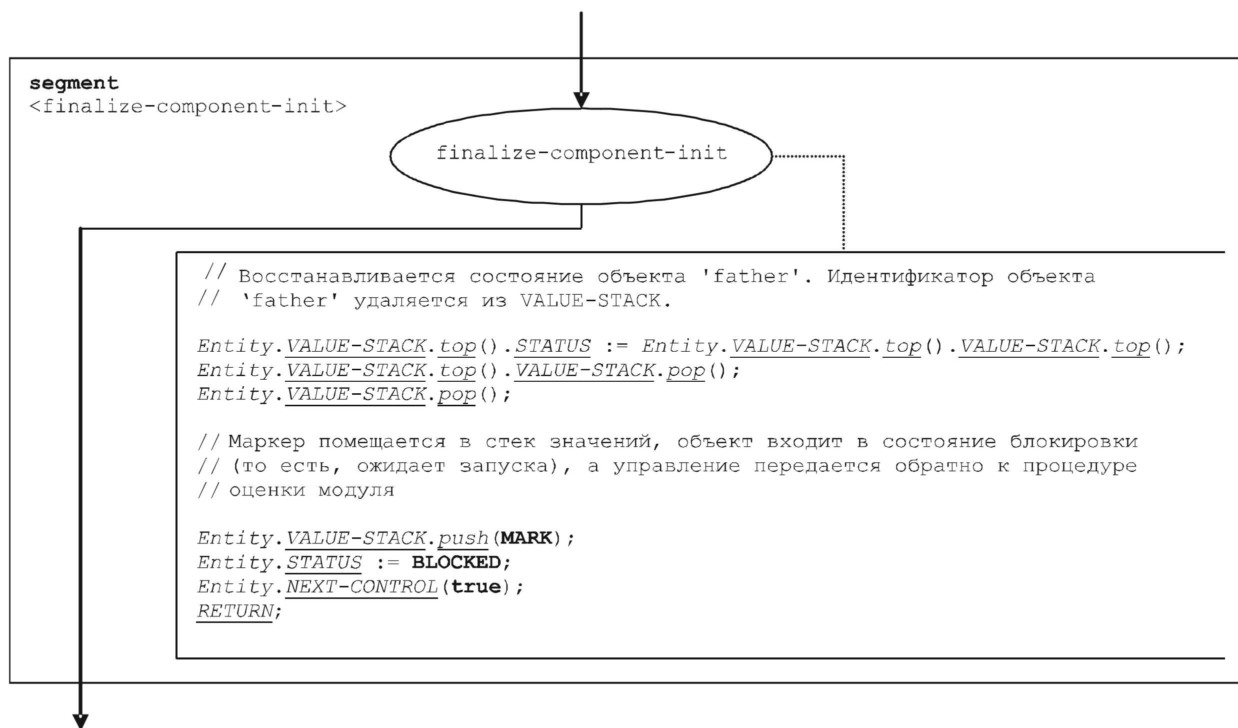


Рисунок 75/Z.143 – Сегмент <finalize-component-init> потокового графа

9.20 Сегмент <init-component-scope> потокового графа

Сегмент `<init-component-scope>` потокового графа является частью потокового графа, представляющего поведение определения компонентного типа. На рисунке 76 показано его выполнение.

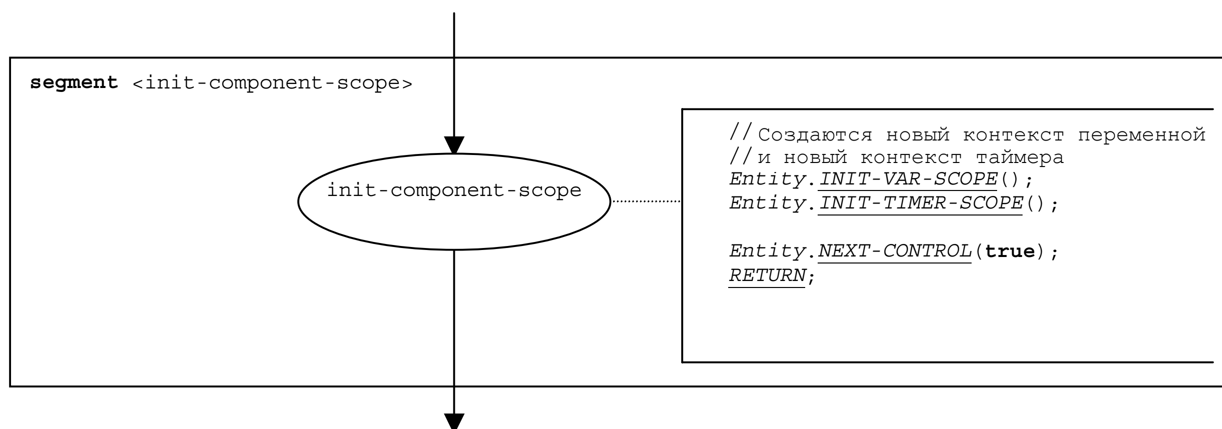


Рисунок 76/Z.143 – Сегмент <init-component-scope> потокового графа

9.21 Сегмент <parameter-handling> потокового графа

Сегмент `<parameter-handling>` потокового графа используется в начале потоковых графов, представляющих тестовые примеры, альтернативные шаги и функции. Он инициализирует новый контекст, создает переменные и таймеры для обработки параметров. Для сегмента `<parameter-handling>` потокового графа предполагается, что запись вызова вызываемого тестового примера, альтернативного шага или функции находится на вершине стека значений.

Выполнение сегмента `<parameter-handling>` потокового графа показано на рисунке 77.

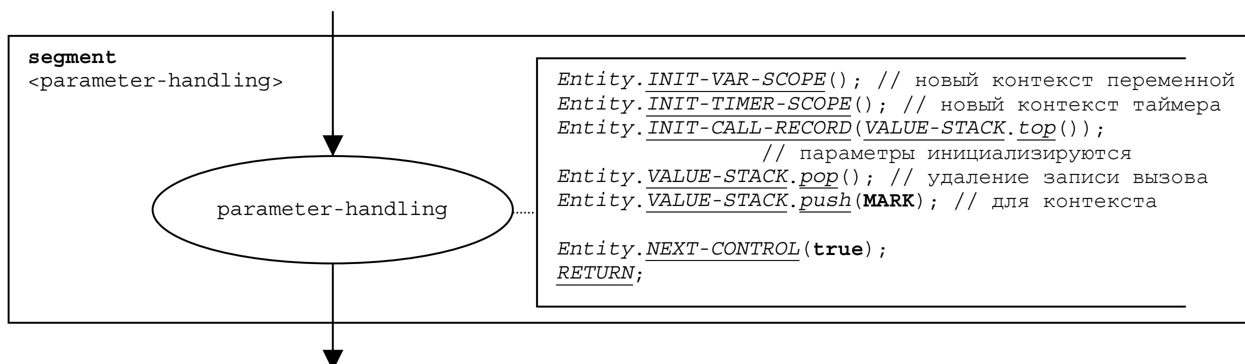


Рисунок 77/Z.143 – Сегмент <parameter-handling> потокового графа

9.22 Сегмент <statement-block> потокового графа

Синтаксическая структура блока операторов имеет следующий вид:

{ <statement$_1$>; ... ; <statement$_n$> }

Блок операторов является контекстным блоком. При входе в контекстный блок должны инициализироваться новые контексты для переменных, таймеров и стека значений. При выходе из контекстного блока должны быть уничтожены все переменные, таймеры и стековые значения данного контекста.

ПРИМЕЧАНИЕ 1. – Блок операторов не является ‘официальным’ понятием TTCN-3. Блоки операторов встречаются только в качестве тела функций, альтернативных шагов, тестовых примеров и управления модулем, а также внутри составных операторов, например, **alt**, **if-else** или **do-while**.

ПРИМЕЧАНИЕ 2. – Операции приема и вызовы альтернативного шага не могут появляться в блоках операторов, они вкладываются в операторы **alt** или операции **call**.

ПРИМЕЧАНИЕ 3. – В операционной семантике операции и объявления также обрабатываются подобно операторам, т. е. они допустимы в блоках операторов.

ПРИМЕЧАНИЕ 4. – Ряд функций TTCN-3, как например, **system** или **self**, считаются выражениями, которые непригодны в качестве автономных операторов в блоках операторов. На рисунке 78 их представления в виде потокового графа не приводятся.

Сегмент <statement-block> потокового графа на рисунке 78 определяет выполнение блока операторов.

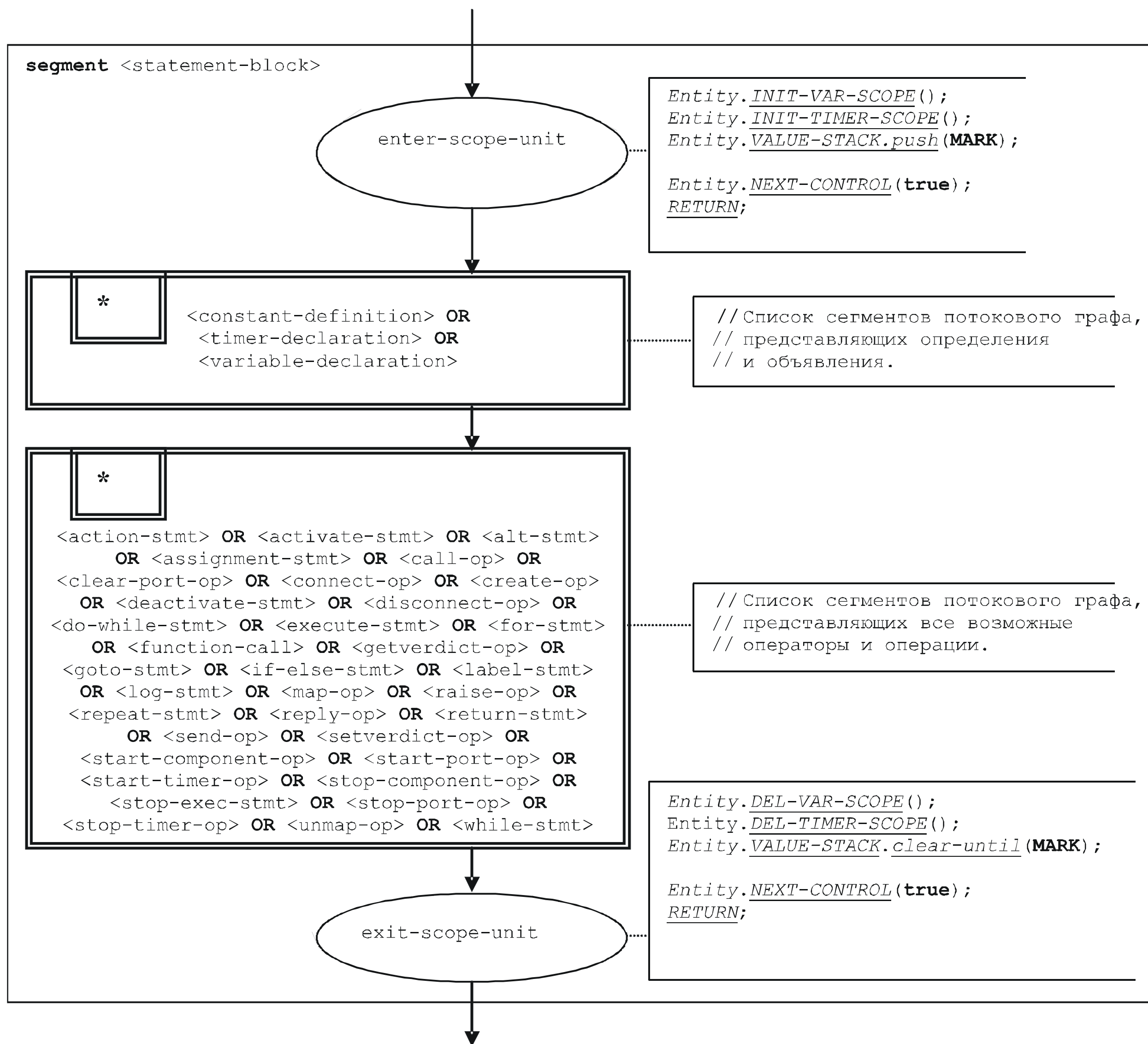


Рисунок 78/Z.143 – Сегмент <statement-block> потокового графа

9.23 Оператор for

Синтаксическая структура для **for-statement** имеет следующий вид:

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

Инициализация индексной переменной и соответствующая манипуляция с индексной переменной рассматриваются как присвоения для индексной переменной. Допустимо также объявлять и инициализировать индексную переменную непосредственно в **for-statement**. Выражение <boolean-expression> описывает критерий завершения цикла, заданного оператором **for-statement**, а блок <statement-block> описывает тело цикла.

Выполнение **for-statement** определяется сегментом <for-stmt> потокового графа, показанным на рисунке 79. Объявление начальной части <assignment> или альтернативной переменной с присвоением <var-declaration-init> (см. п. 9.57.1) описывает инициализацию индексной переменной. Часть <assignment> в ветви **true** узла decision описывает манипуляцию индексной переменной. Оператор **for-statement** является контекстным блоком для вновь объявленной индексной переменной; это моделируется с помощью узлов enter-var-scope и exit-var-scope.

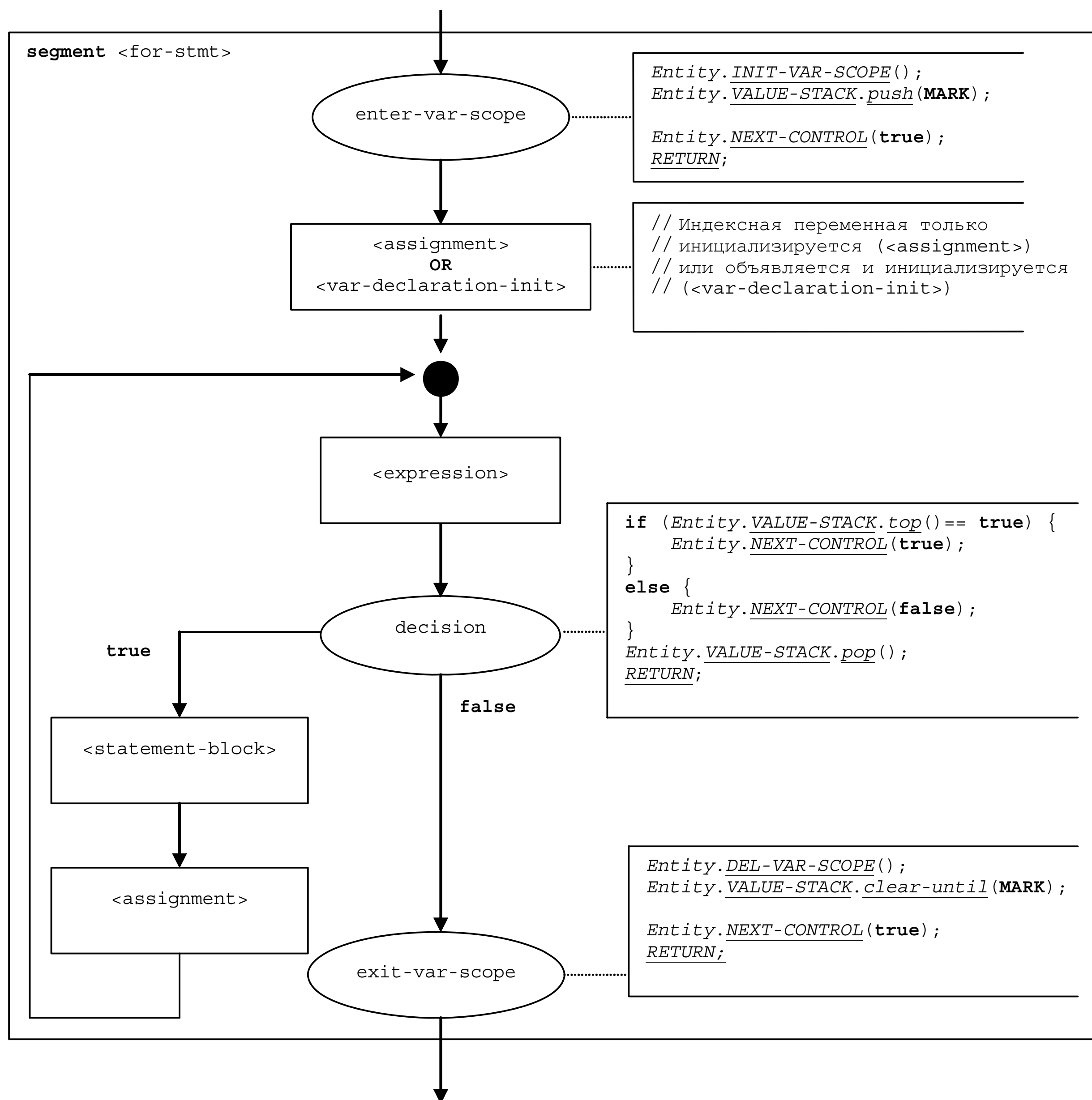


Рисунок 79/Z.143 – Сегмент <for-stmt> потокового графа

9.24 Вызов функции

Синтаксическая структура вызова функции имеет следующий вид:

<function-name>([<act-par-desc$_1$>, ... , <act-par-desc$_n$>])

Часть <function-name> обозначает имя функции, а <act-par-descr$_1$>, ... , <act-par-descr$_n$> описывают фактические значения параметра вызова функции.

ПРИМЕЧАНИЕ 1. – Вызов функции и вызов альтернативного шага обрабатываются одинаковым способом, поэтому вызов альтернативного шага (см. п. 9.4) указывает на этот раздел.

Предполагается, что для каждого <act-par-desc$_1$> известен соответствующий идентификатор формального параметра <f-par-Id$_1$>, т. е. указанную выше синтаксическую структуру можно расширить 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 рисунке 80 определяет выполнение вызова функции. Это выполнение подразделяется на три шага. На первом шаге создается запись вызова для функции <function-name>. На втором шаге вычисляются значения фактического параметра, присваиваемые соответствующему полю в записи вызова. На третьем шаге необходимо различать две ситуации: вызываемая функция является определяемой пользователем функцией (<user-def-func-call>), т. е. существует представление в виде потокового графа для функции, или же вызываемая функция является предопределенной или внешней функцией (<predef-ext-func-call>). В случае вызова функции, определяемой пользователем, управление передается вызываемой функции. В случае предопределенной или внешней функции предполагается, что запись вызова может использоваться для выполнения функции за один шаг. Вызываемая функция отвечает за правильную обработку параметров-ссылок и возвращаемого значения (должно быть помещено в стек значений), т. е. эта обработка выходит за рамки данной операционной семантики.

ПРИМЕЧАНИЕ 2. – Если вызов функции моделирует вызов альтернативного шага, то будет выбираться только ветвь <user-def-func-call>, так как существует представление вызываемого альтернативного шага в виде потокового графа.

ПРИМЕЧАНИЕ 3. – Сегмент <function call> также используется для описания запуска МТС в операторе **execute**. В этом случае конструируется запись вызова для тестового примера, и будет выбрана только ветвь <user-def-func-call>.

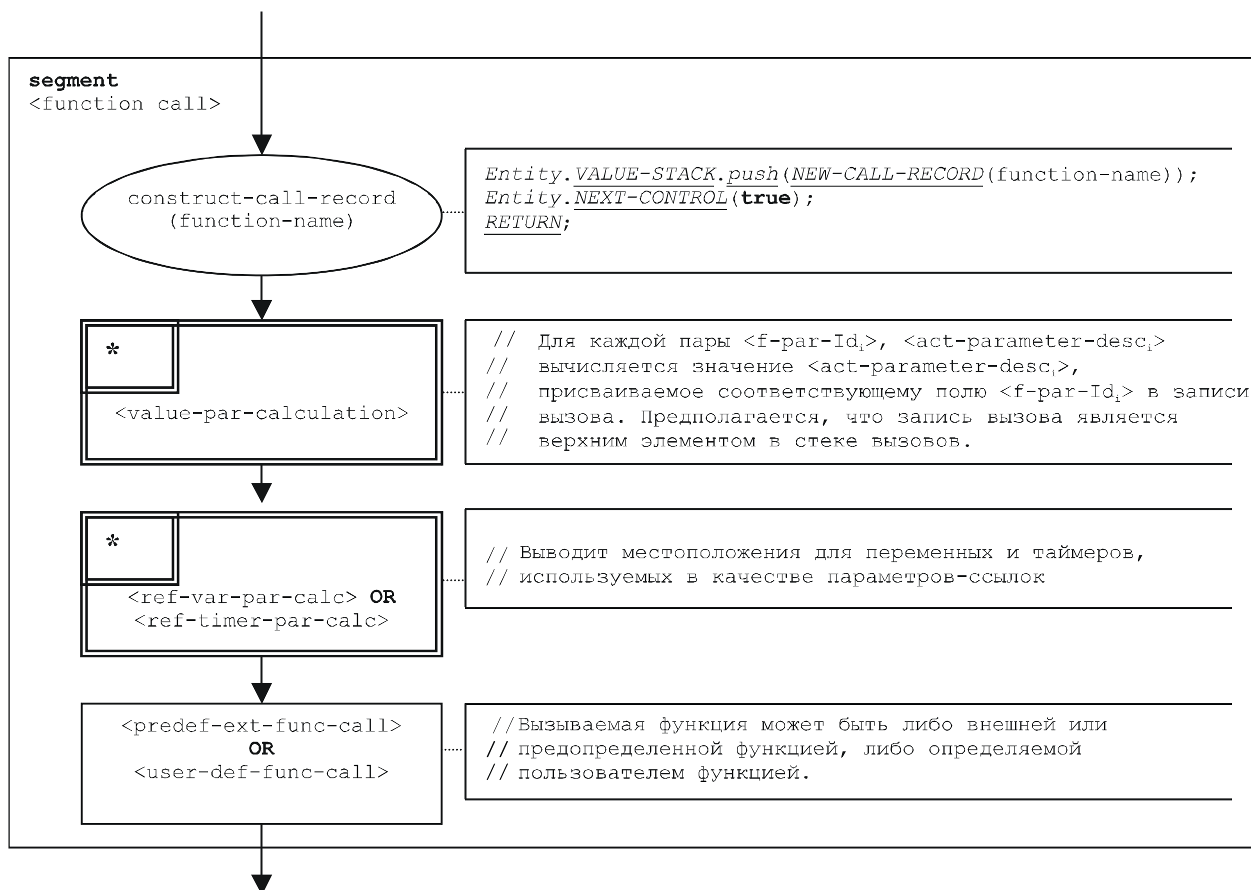


Рисунок 80/Z.143 – Сегмент <function-call> потокового графа

9.24.1 Сегмент <value-par-calculation> потокового графа

Сегмент <value-par-calculation> потокового графа используется для вычисления значений фактических параметров и для присвоения их соответствующим полям в записях вызова для функций альтернативных шагов и тестовых примеров.

Предполагается, что запись вызова является верхним элементом стека значений и что пара:

(<f-par-Id$_i$>, <act-parameter-desc$_i$>)

должна быть обработана. При этом часть <act-parameter-desc$_i$> должна быть оценена, а <f-par-Id$_i$> является идентификатором формального параметра, имеющего соответствующее поле в записи вызова в стеке значений.

Выполнение сегмента <value-par-calculation> потокового графа показано на рисунке 81.

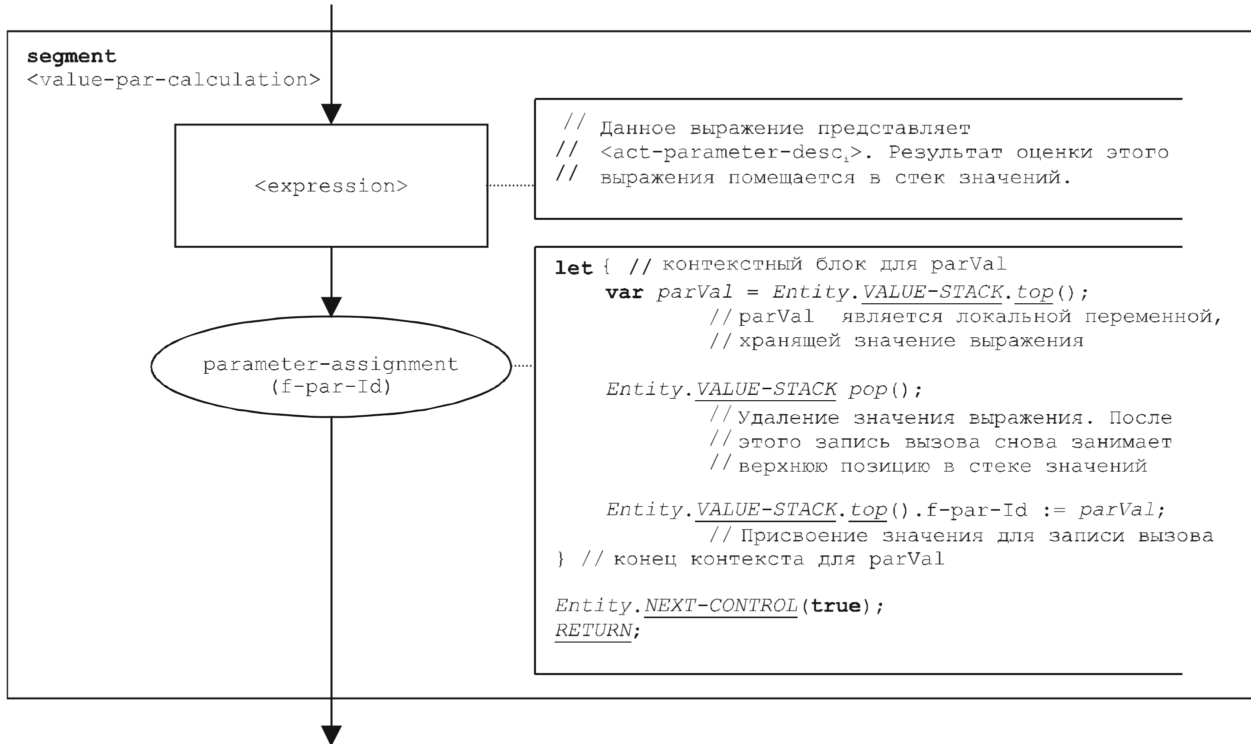


Рисунок 81/Z.143 – Сегмент <value-par-calculation> потокового графа

9.24.2 Сегмент <ref-par-var-calc> потокового графа

Сегмент <ref-par-var-calc> потокового графа используется для вывода местоположений переменных, используемых в качестве фактических параметров-ссылок, и для присвоения их соответствующим полям в записях вызова для функций, альтернативных шагов и тестовых примеров.

Предполагается, что запись вызова является верхним элементом стека значений и что пара:

(<f-par-Id$_i$>, <act-par$_i$>)

должна быть обработана. Часть <act-par$_i$> является фактическим параметром, для которого должно быть выведено местоположение, а <f-par-Id$_i$> является идентификатором формального параметра, имеющего соответствующее поле в записи вызовов в стеке значений.

Выполнение сегмента <ref-par-var-calc> потокового графа показано на рисунке 82.

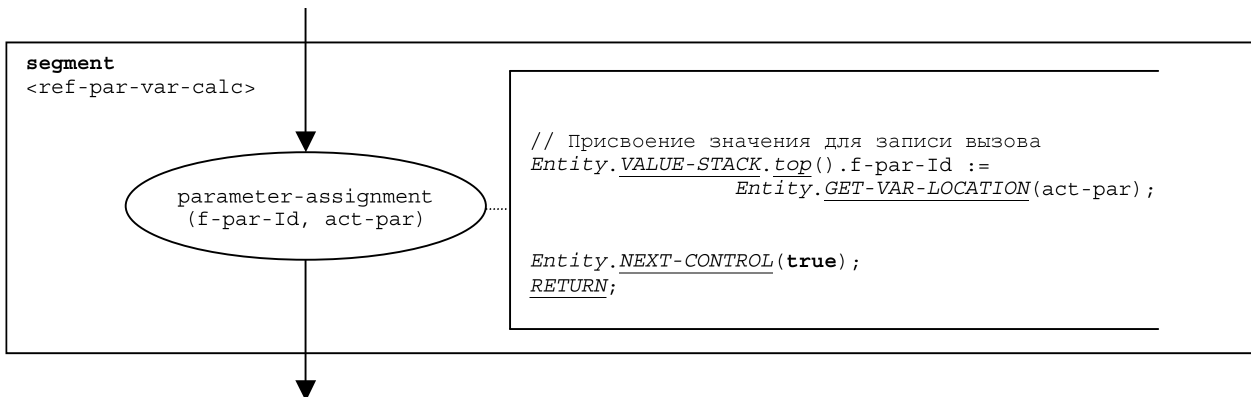


Рисунок 82/Z.143 – Сегмент <ref-par-var-calc> потокового графа

9.24.3 Сегмент <ref-par-timer-calc> потокового графа

Сегмент `<ref-par-timer-calc>` потокового графа используется для вывода местоположений таймеров, используемых в качестве фактических параметров-ссылок, и для присвоения их соответствующим полям в записях вызовов для функций, альтернативных шагов и тестовых примеров.

Предполагается, что запись вызова является верхним элементом стека значений и что пара:

$(<\text{f-par-Id}_i>, <\text{act-par}_i>)$

должна быть обработана. Часть $<\text{act-par}_i>$ является фактическим параметром, для которого должно быть выведено местоположение, а $<\text{f-par-Id}_i>$ является идентификатором формального параметра, имеющего соответствующее поле в записи вызова в стеке значений.

Выполнение сегмента `<ref-par-timer-calc>` потокового графа показано на рисунке 83.

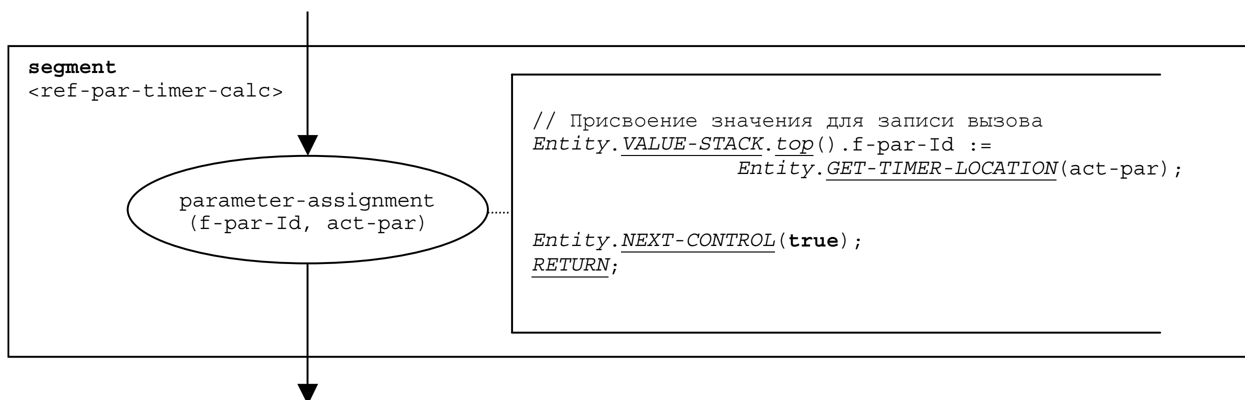


Рисунок 83/Z.143 – Сегмент <ref-par-timer-calc> потокового графа

9.24.4 Сегмент <user-def-func-call> потокового графа

Сегмент `<user-def-func-call>` потокового графа (рисунок 84) описывает передачу управления к вызываемой функции, определяемой пользователем.

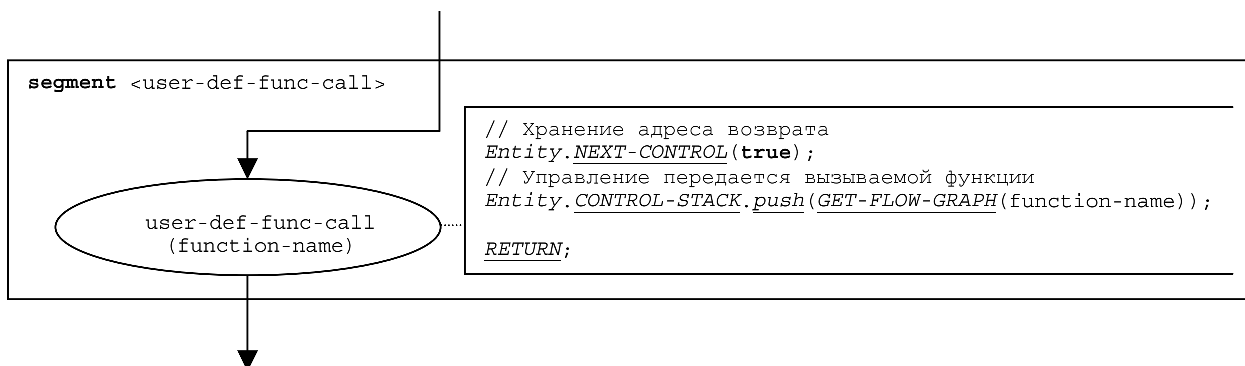


Рисунок 84/Z.143 – Сегмент <user-def-func-call> потокового графа

9.24.5 Сегмент <predef-ext-func-call> потокового графа

Сегмент <predef-ext-func-call> потокового графа (рисунок 85) описывает вызов предопределенной или внешней функции.

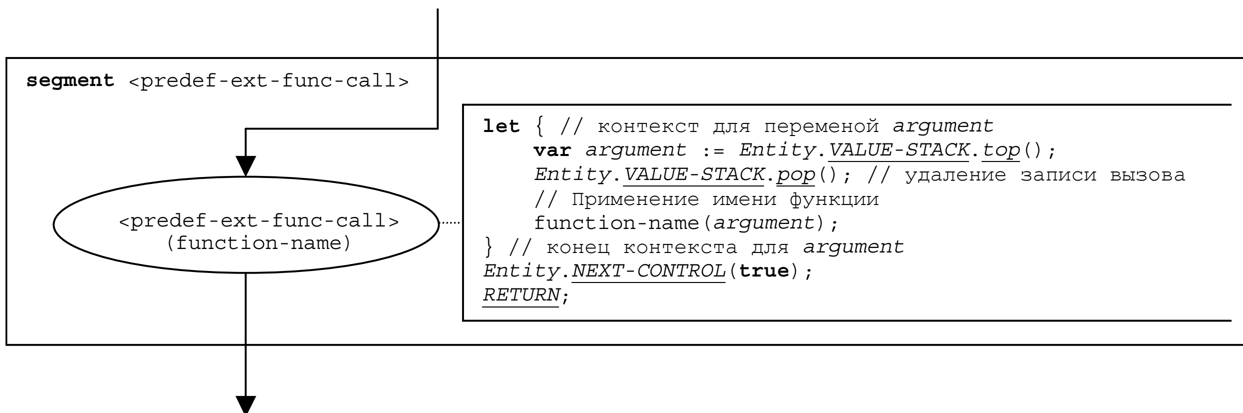


Рисунок 85/Z.143 – Сегмент <predef-ext-func-call> потокового графа

9.25 Операция getcall

Синтаксическая структура операции **getcall** имеет следующий вид:

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

Если не учитывать ключевое слово **getcall**, данная синтаксическая структура идентична синтаксической структуре операции **receive**. Поэтому в операционной семантике операция **getcall** обрабатывается тем же способом, что и операция **receive**. Это также показано в сегменте <getcall-op> потокового графа (см. рисунок 86), который определяет выполнение операции **getcall**. Этот рисунок касается сегментов потокового графа, связанных с операцией **receive** (см. п. 9.37).

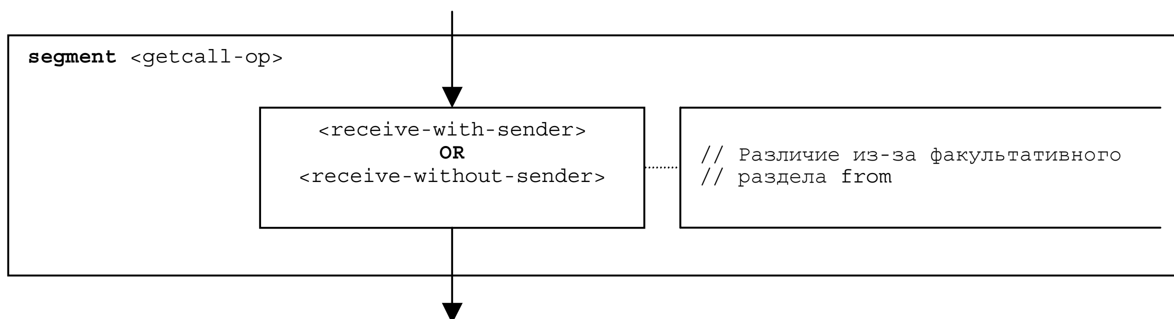


Рисунок 86/Z.143 – Сегмент <getcall-op> потокового графа

9.26 Операция getreply

Синтаксическая структура операции **getreply** имеет следующий вид:

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

Если не учитывать ключевое слово **getreply**, данная синтаксическая структура идентична синтаксической структуре операции **receive**. Поэтому в операционной семантике операция **getreply** обрабатывается тем же способом, что и операция **receive**. Это также показано в сегменте <getreply-op> потокового графа (см. рисунок 87), который определяет выполнение операции **getreply**. Этот рисунок касается сегментов потокового графа, связанных с операцией **receive** (см. п. 9.37).

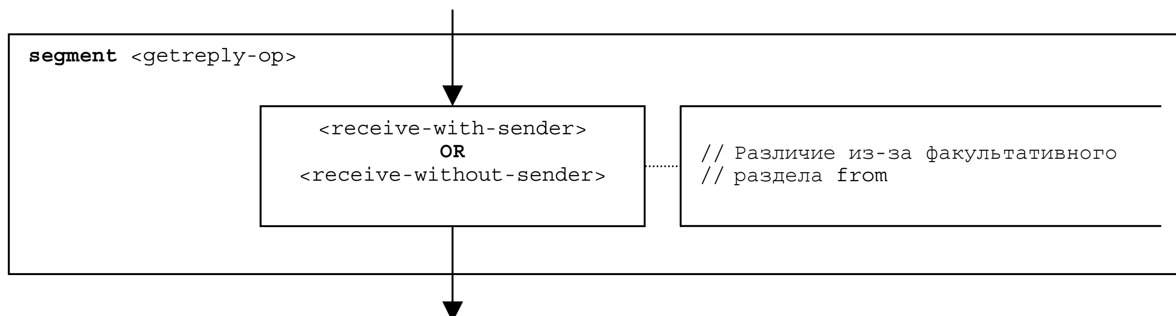


Рисунок 87/Z.143 – Сегмент <getreply-op> потокового графа

9.27 Операция getverdict

Синтаксическая структура операции **getverdict** имеет следующий вид:

```
getverdict
```

Сегмент <getverdict-op> потокового графа на рисунке 88 определяет выполнение операции **getverdict**.

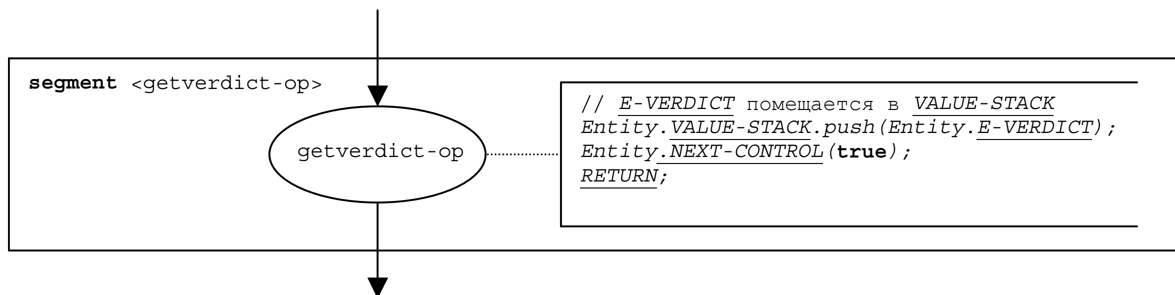


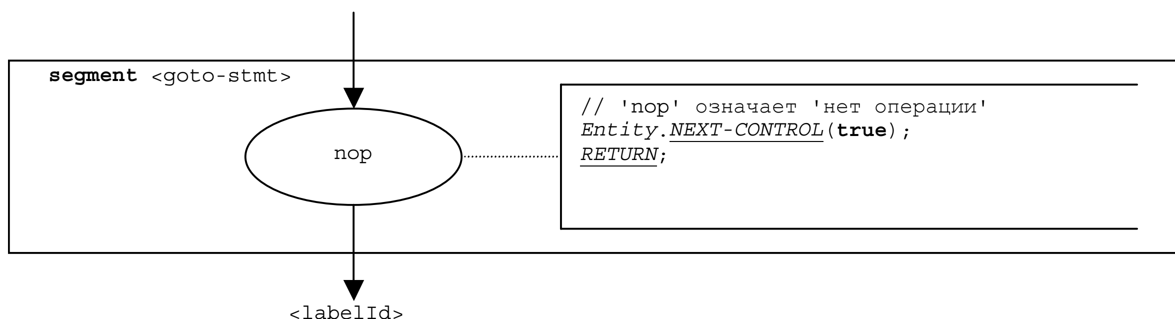
Рисунок 88/Z.143 – Сегмент <getverdict-op> потокового графа

9.28 Оператор goto

Синтаксическая структура оператора **goto** имеет следующий вид:

```
goto <labelId>
```

Сегмент <goto-stmt> потокового графа на рисунке 89 определяет выполнение оператора **goto**.



ПРИМЕЧАНИЕ. – Параметр <labelId> оператора **goto** указывает на передачу управления в то место, где определяется метка <*labelId*> (см. также п. 9.30).

Рисунок 89/Z.143 – Сегмент <goto-stmt> потокового графа

9.29 Оператор if-else

Синтаксическая структура оператора **if-else** имеет следующий вид:

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

Часть else оператора **if-else** является факультативной.

Сегмент <if-else-stmt> потокового графа на рисунке 90 определяет выполнение оператора **if-else**.

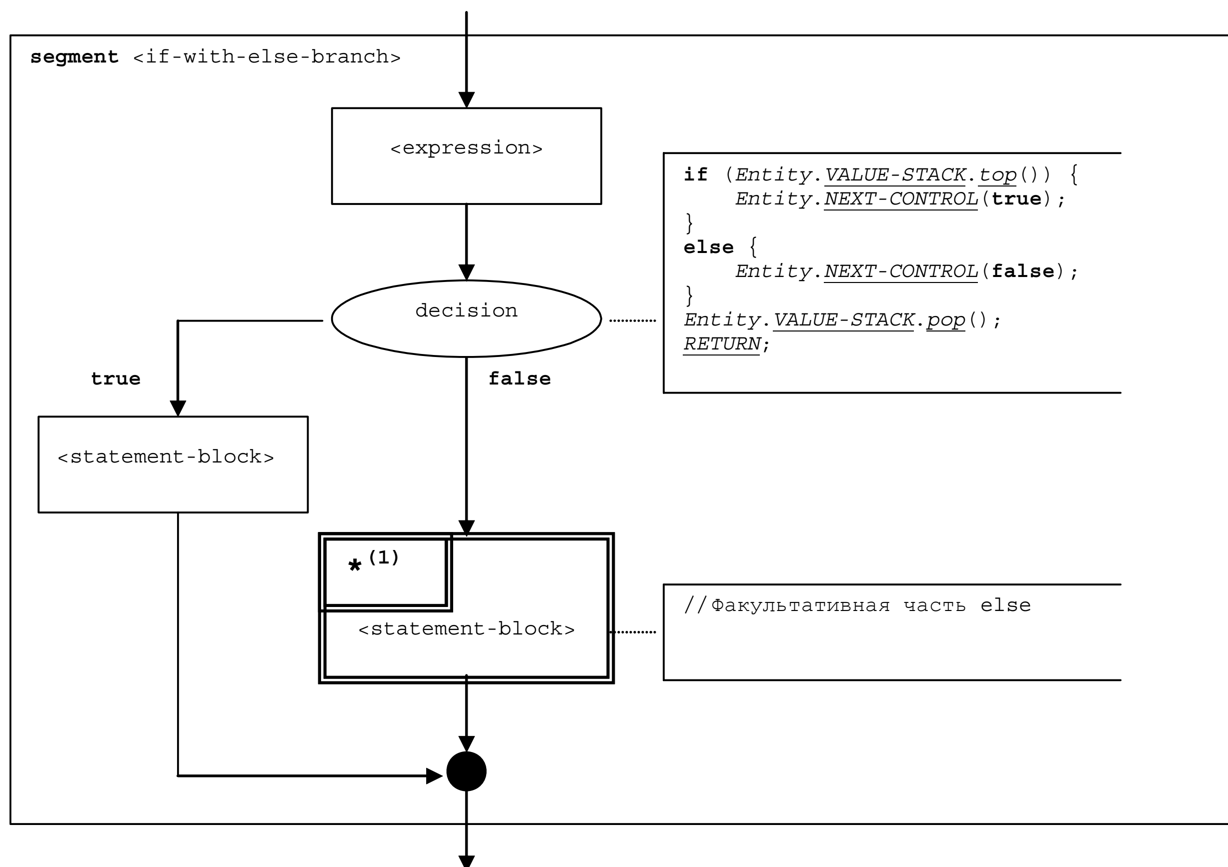


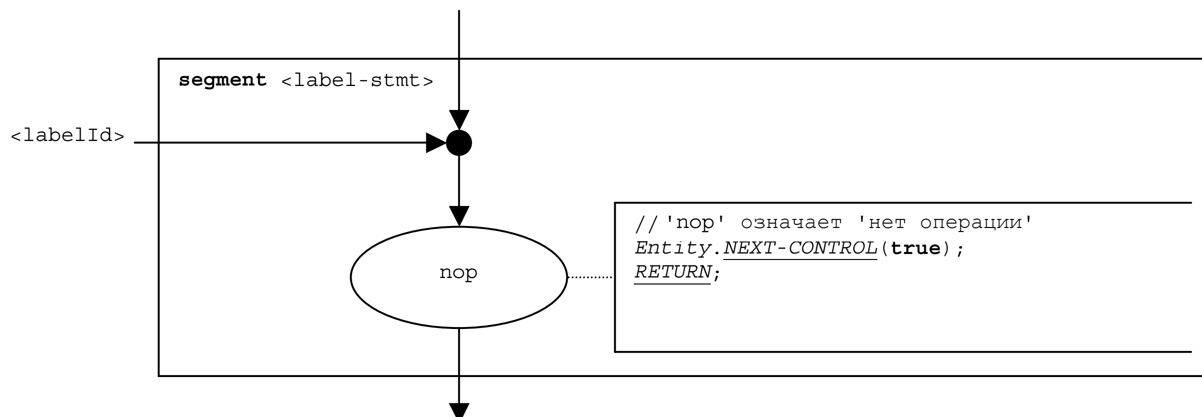
Рисунок 90/Z.143 – Сегмент <if-else-stmt> потокового графа

9.30 Оператор label

Синтаксическая структура оператора **label** (метка) имеет следующий вид:

```
label <labelId>
```

Сегмент <label-stmt> потокового графа на рисунке 91 определяет выполнение оператора **label**.



ПРИМЕЧАНИЕ. – Параметр <labelId> оператора label указывает на возможность того, что метка может быть адресатом для выполнения перехода с помощью оператора **goto** (см. также п. 9.28).

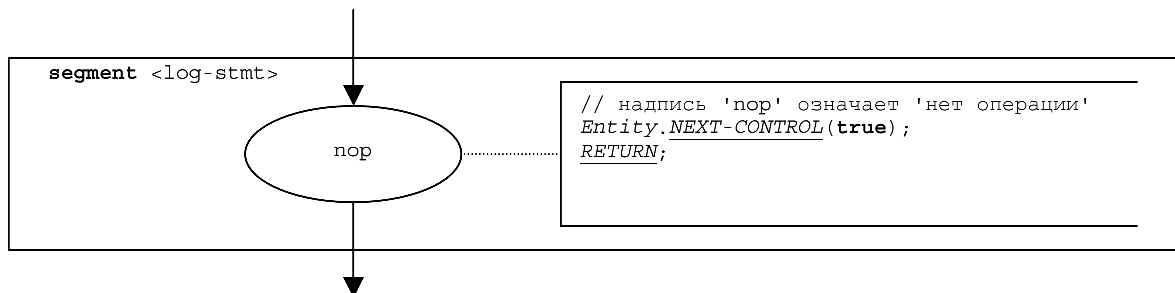
Рисунок 91/Z.143 – Сегмент <label-stmt> потокового графа

9.31 Оператор log

Синтаксическая структура оператора **log** имеет следующий вид:

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

Сегмент <log-stmt> потокового графа на рисунке 92 определяет выполнение оператора **log**.



ПРИМЕЧАНИЕ. – Параметр <informal description> оператора **log** не имеет смысла для операционной семантики и поэтому не представлен в сегменте потокового графа.

Рисунок 92/Z.143 – Сегмент <log-stmt> потокового графа

9.32 Операция map

Синтаксическая структура операции **map** (отображение) имеет следующий вид:

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

Идентификаторы <portId1> и <portId2> считаются идентификаторами порта соответствующего интерфейса тестового компонента и тестовой системы. На компоненты, к которым принадлежит <portId1>, ссылаются с помощью компонентной ссылки <component-expression>. Эта ссылка может храниться в переменных или она возвращается функцией, т. е. она является выражением, которое дает оценку компонентной ссылке. Для хранения компонентной ссылки используется стек значений.

ПРИМЕЧАНИЕ. – Для операции **map** не важно, появляется ли оператор **system**:<portId> в качестве первого или второго параметра. В целях простоты предполагается, что это всегда второй параметр.

Выполнение операции **map** определяется сегментом <map-op> потокового графа, показанного на рисунке 93.

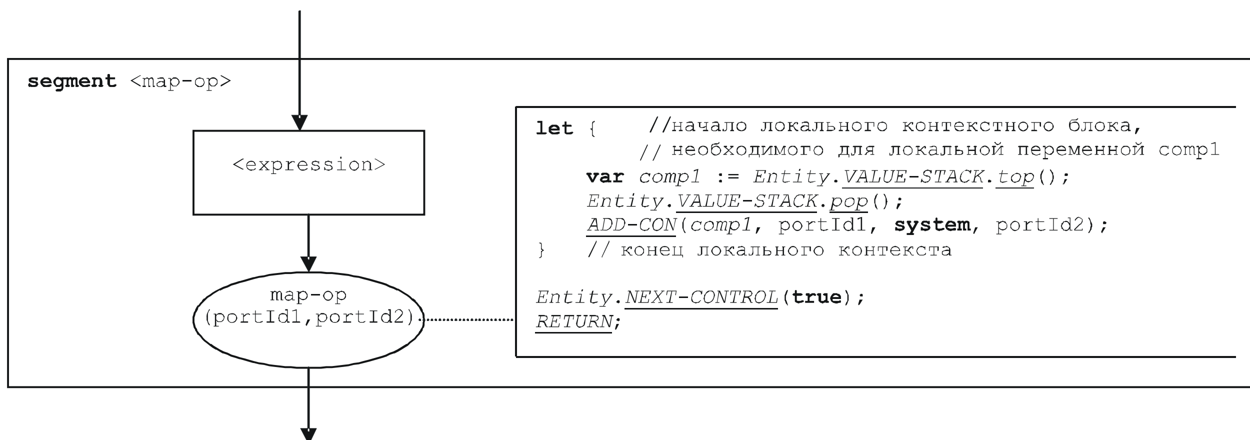


Рисунок 93/Z.143 – Сегмент <map-op> потокового графа

9.33 Операция mtc

Синтаксическая структура операции **mtc** имеет следующий вид:

```
mtc
```

Сегмент <mtc-op> потокового графа на рисунке 94 определяет выполнение операции **mtc**.

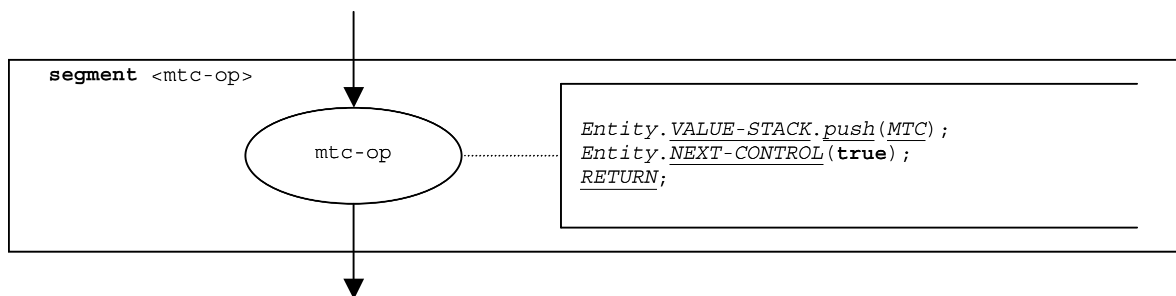


Рисунок 94/Z.143 – Сегмент <mtc-op> потокового графа

9.34 Объявление port

Синтаксическая структура объявления **port** имеет следующий вид:

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

Объявление порта можно найти в определениях компонентного типа. Результатом объявления порта является создание нового порта, когда создается новый компонент соответствующего типа. Сегмент <port-declaration> потокового графа на рисунке 95 определяет выполнение объявления порта.

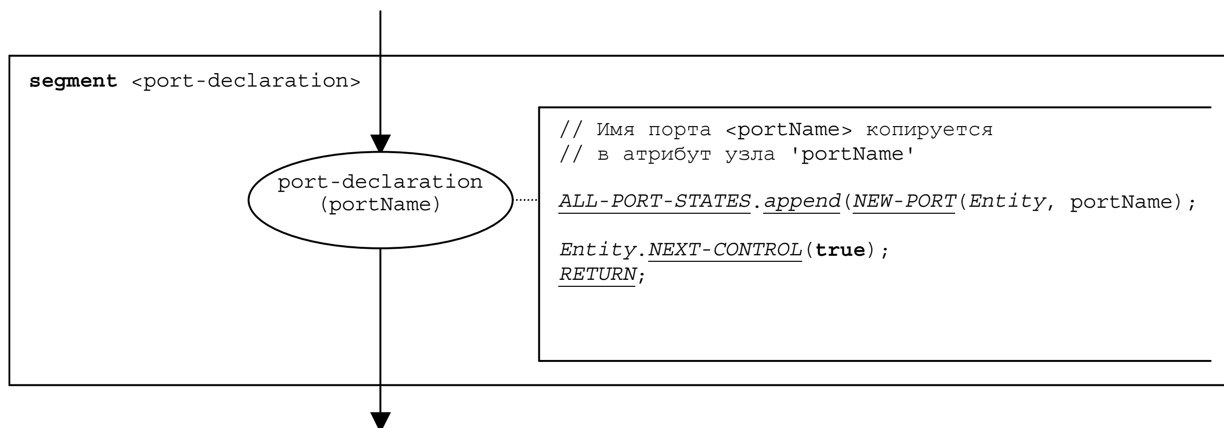


Рисунок 95/Z.143 – Сегмент <port-declaration> потокового графа

9.35 Операция raise

Синтаксическая структура операции **raise** имеет следующий вид:

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

Факультативная часть <component-expression> в разделе **to** указывает на объект получателя. Она может быть предоставлена в виде значения переменной или возвращаемого значения функции.

Сегмент потокового графа <raise-op> на рисунке 96 определяет выполнение операции **raise**.

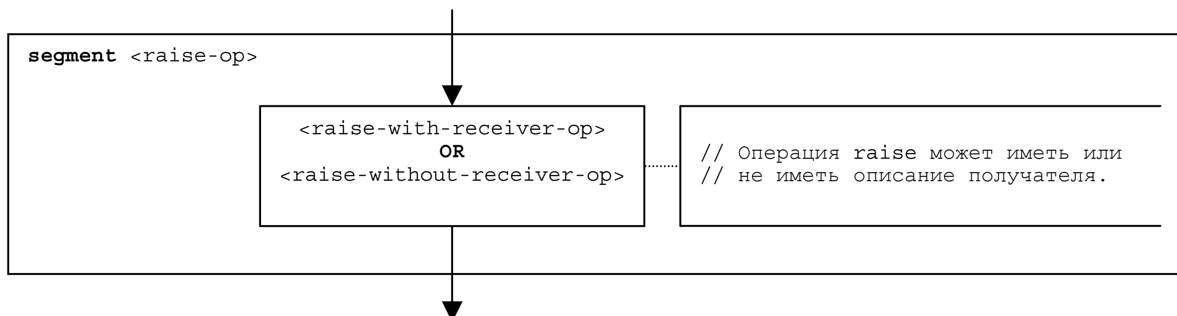


Рисунок 96/Z.143 – Сегмент <raise-op> потокового графа

9.35.1 Сегмент <raise-with-receiver-op> потокового графа

Сегмент `<raise-with-receiver-op>` потокового графа на рисунке 97 определяет операцию **`raise`**, где получатель задан в виде выражения.

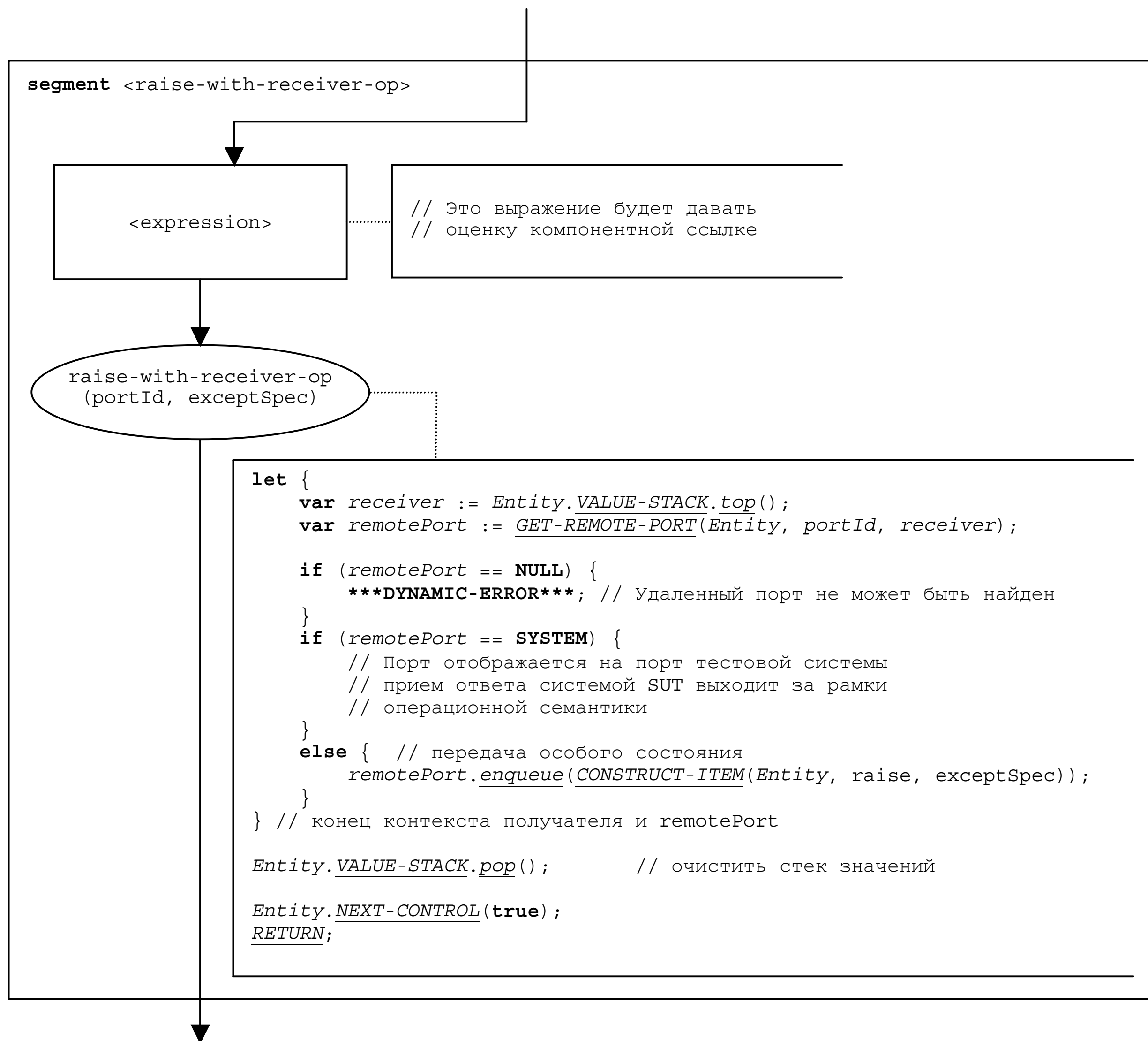


Рисунок 97/Z.143 – Сегмент <raise-with-receiver-op> потокового графа

9.35.2 Сегмент <raise-without-receiver-op> потокового графа

Сегмент `<raise-without-receiver-op>` потокового графа на рисунке 98 определяет выполнение операции raise без раздела **to**.

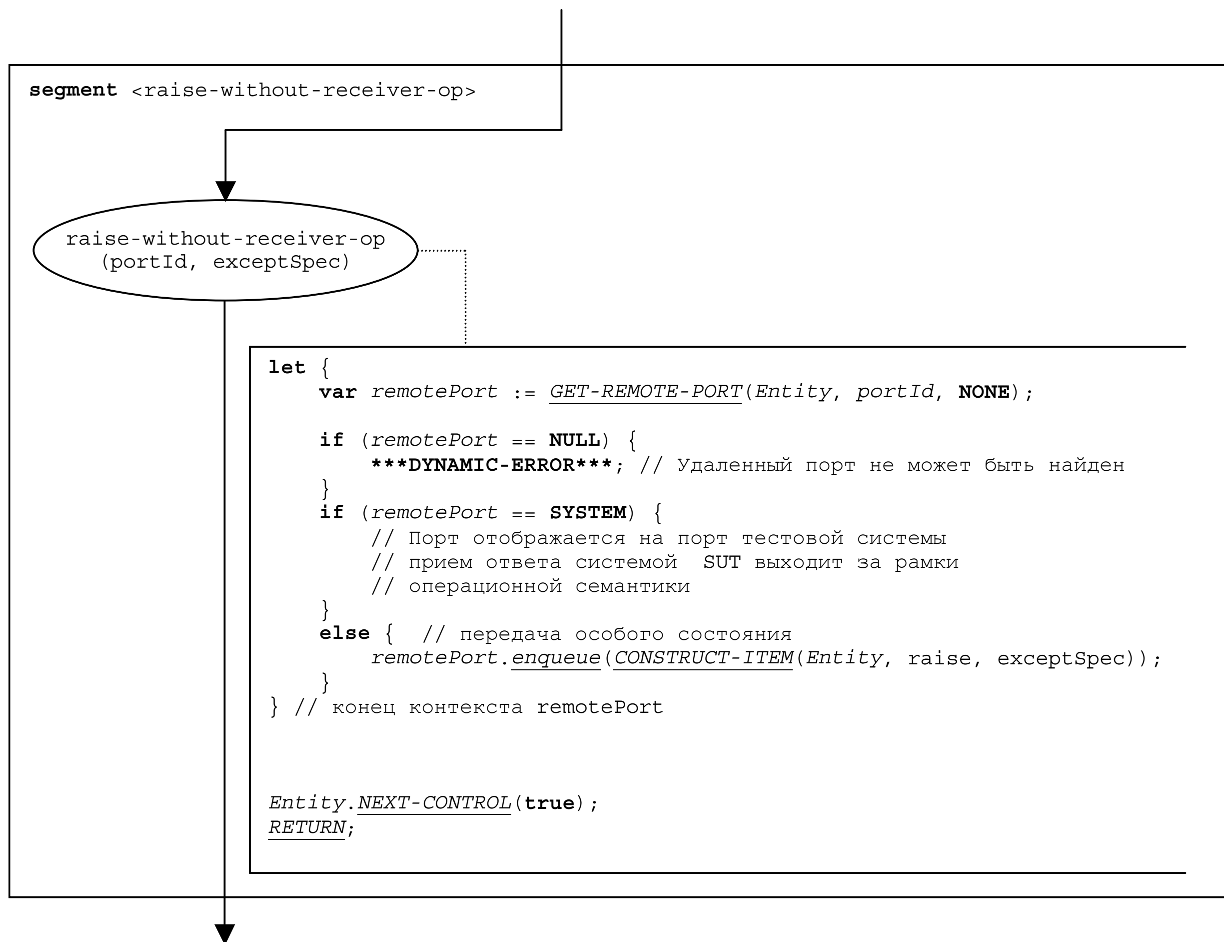


Рисунок 98/Z.143 – Сегмент <raise-without-receiver-op> потокового графа

9.36 Операция read таймера

Синтаксическая структура операции **read** (чтение) таймера имеет следующий вид:

```
<timerId>.read
```

Сегмент `<read-timer-op>` потокового графа на рисунке 99 определяет выполнение операции **read** таймера**.**

Использование операции **read** таймера в булевой защите оператора **alt** или блокирующей операции **call** отличается от ее использования во всех других случаях. При использовании в булевой защите результат операции **read** таймера базируется на фактической фиксации мгновенного состояния процесса, т. е. результат операции определяют компоненты *SNAP-STATUS* и *SNAP-VALUE* связывания таймера, а во всех других случаях – компоненты *STATUS*, *ACT-DURATION* и *TIME-LEFT* связывания таймера.

```
segment <read-timer-op>

                    read-timer-op
                      (timerId)

   let { // локальный контекст для переменной myValue

       var float myValue;

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

           }
           else {
               myValue := 0.0;
           }
       }

       Entity.VALUE-STACK.push(myValue);

   } // конец локального контекста

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

Рисунок 99/Z.143 – Сегмент <read-timer-op> потокового графа

9.37 Операция receive

Синтаксическая структура операции **receive** (получение) имеет следующий вид:

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

Факультативная часть <component-expression> в разделе **from** относится к объекту отправителя. Она может быть предоставлена в виде значения переменной или возвращаемого значения функции, т. е. предполагается, что это – выражение. Факультативная часть <assignmentPart> обозначает присвоение полученной информации, если полученное сообщение сопоставляется со спецификацией <matchingSpec> сопоставления и с (факультативным) разделом **from**.

Сегмент <receive-op> потокового графа на рисунке 100 определяет выполнение операции **receive**.

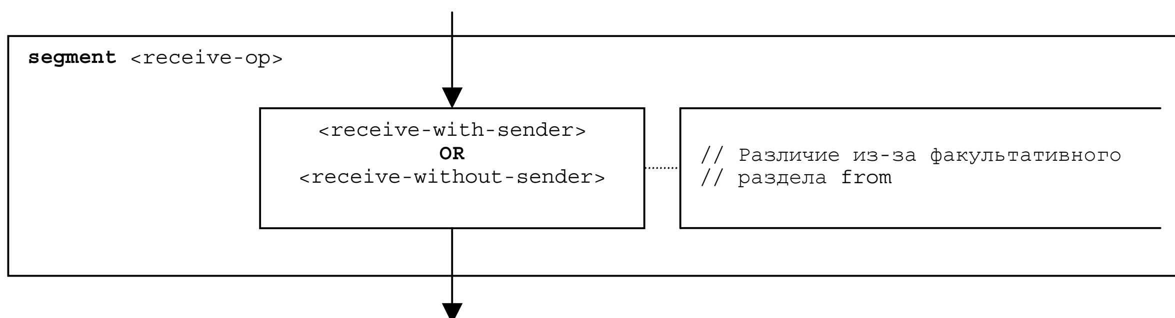


Рисунок 100/Z.143 – Сегмент <receive-op> потокового графа

9.37.1 Сегмент <receive-with-sender> потокового графа

Сегмент `<receive-with-sender>` потокового графа на рисунке 101 определяет выполнение операции **`receive`**, где отправитель задан в виде выражения.

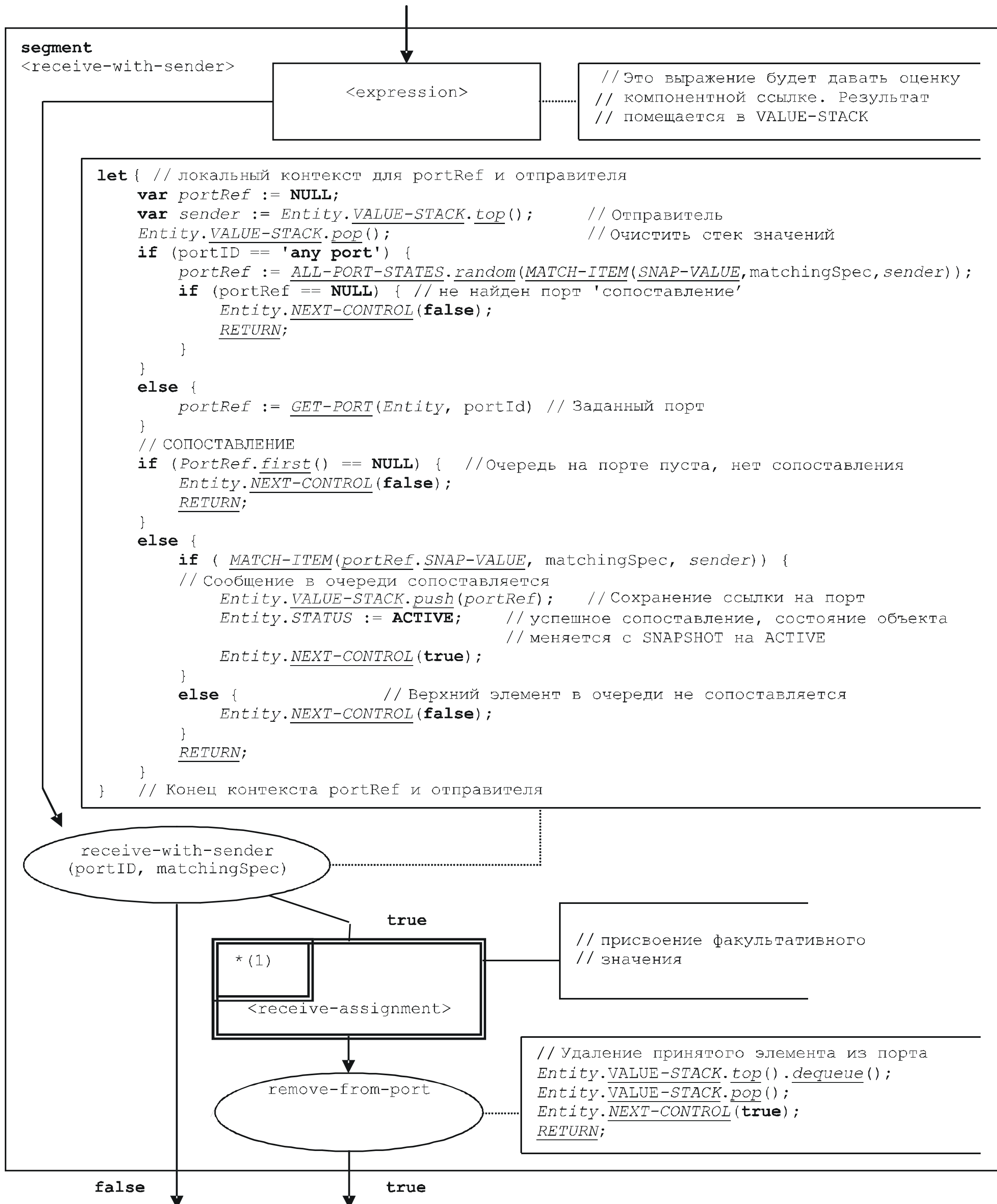


Рисунок 101/Z.143 – Сегмент <receive-with-sender> потокового графа

9.37.2 Сегмент <receive-without-sender> потокового графа

Сегмент `<receive-without-sender>` потокового графа на рисунке 102 определяет выполнение операции **`receive`** без раздела **`from`**.

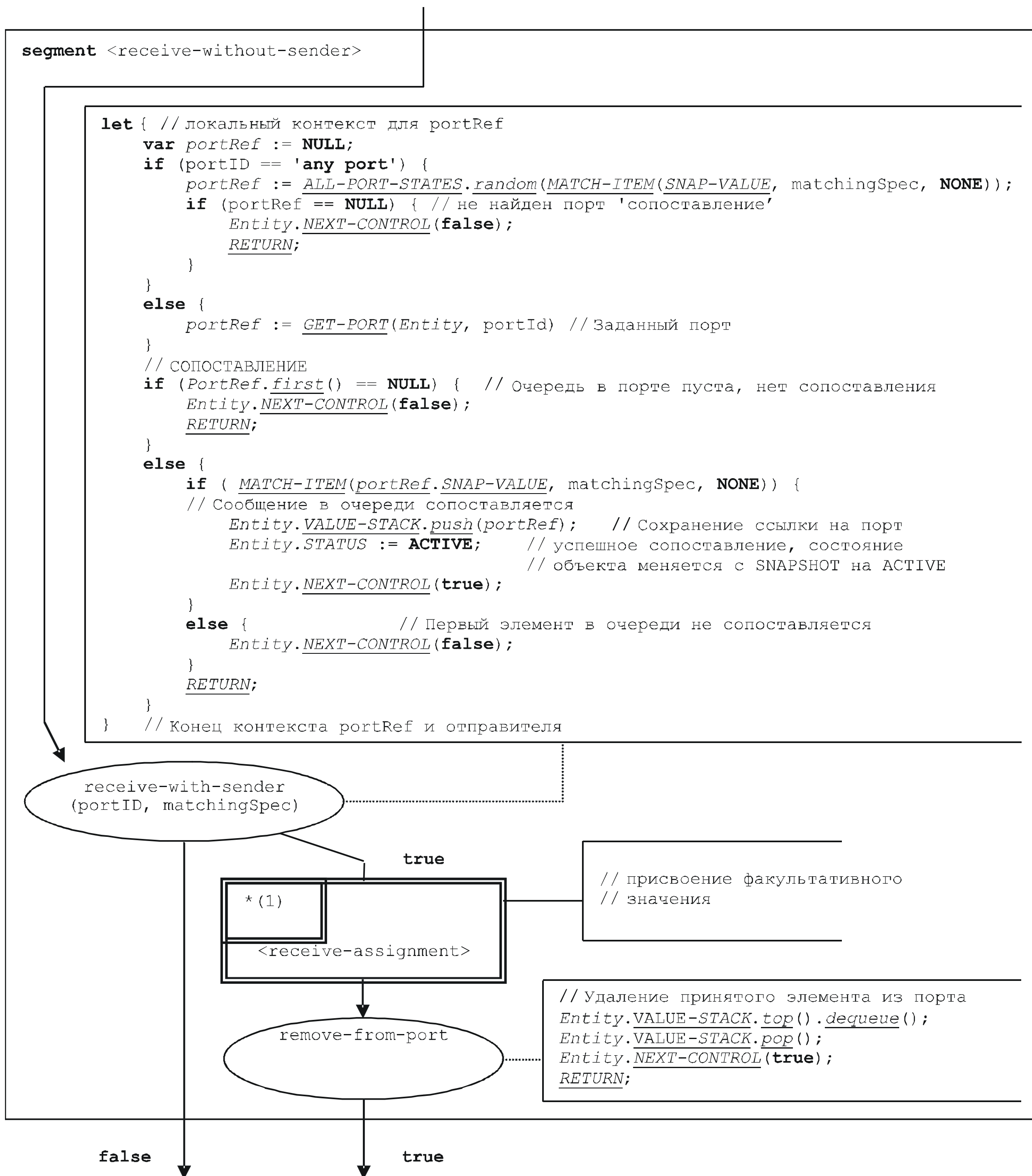


Рисунок 102/Z.143 – Сегмент <receive-without-sender> потокового графа

9.37.3 Сегмент <receive-assignment> потокового графа

Сегмент `<receive-assignment>` потокового графа на рисунке 103 определяет вывод информации из принятых сообщений и их присвоение переменным.

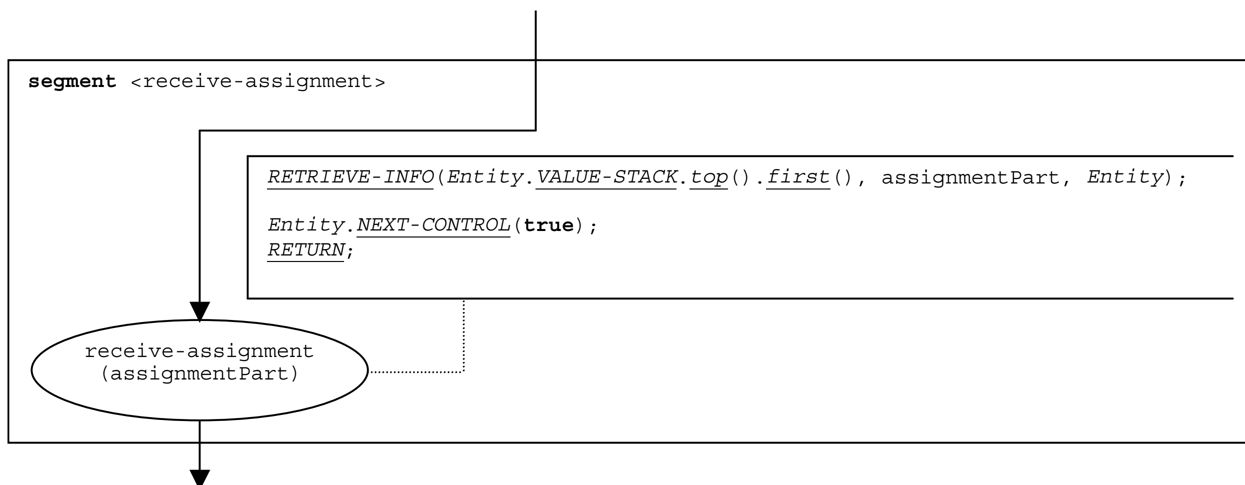


Рисунок 103/Z.143 – Сегмент <receive-assignment> потокового графа

9.38 Оператор repeat

Синтаксическая структура оператора **`repeat`** (повторение) имеет следующий вид:

```
repeat
```

В основном оператор **`repeat`** является оператором **`return`** без возвращаемого значения, который также меняет состояние объекта на **`REPEAT`**. Состояние **`REPEAT`** будет вызывать повторную оценку оператора **`alt`**, в котором был выполнен оператор повторения. Сегмент `<repeat-stmt>` потокового графа, показанный на рисунке 104, определяет выполнение оператора **`repeat`**.

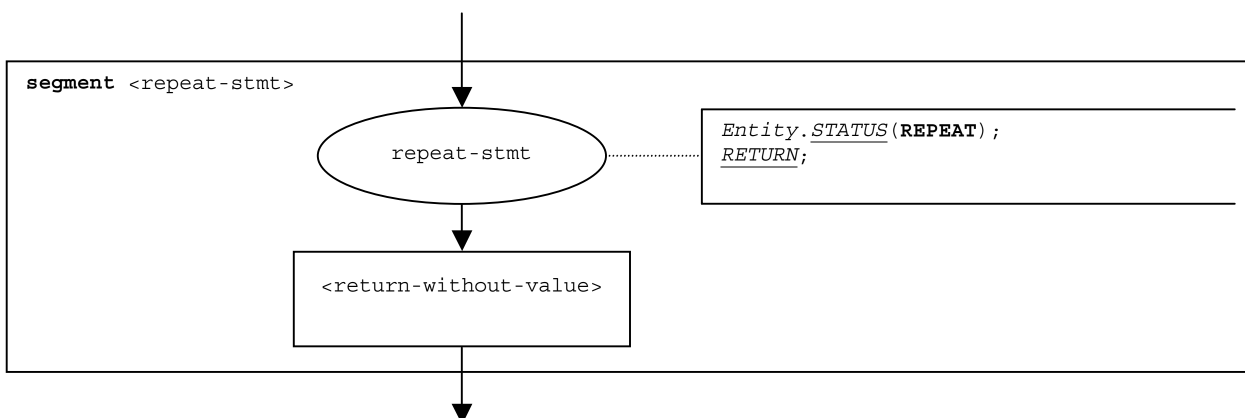


Рисунок 104/Z.143 – Сегмент <repeat-stmt> потокового графа

9.39 Операция reply

Синтаксическая структура операции **`reply`** (ответ) имеет следующий вид:

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

Факультативная часть `<component expression>` в разделе **`to`** относится к объекту получателя. Она может быть предоставлена в виде значения переменной или в виде возвращаемого значения функции.

Сегмент <reply-op> потокового графа на рисунке 105 определяет выполнение операции **reply**.

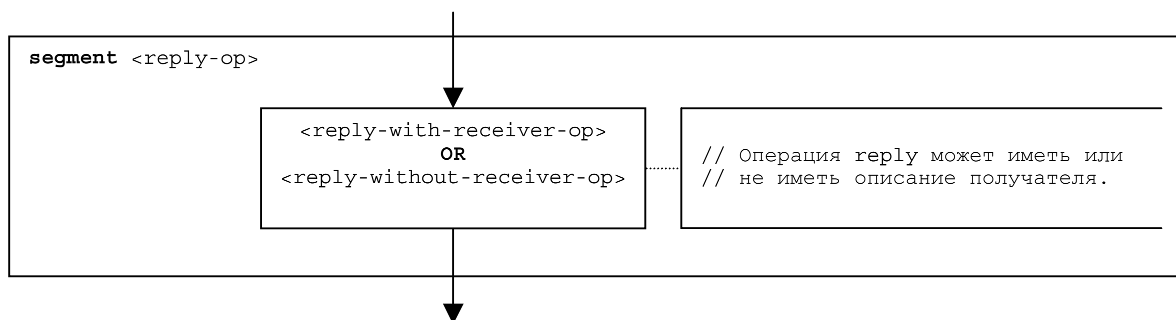


Рисунок 105/Z.143 – Сегмент <reply-op> потокового графа

9.39.1 Сегмент <reply-with-receiver-op> потокового графа

Сегмент <reply-with-receiver-op> потокового графа на рисунке 106 определяет выполнение операции **reply**, где получатель задан в виде выражения.

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

<expression>        // Это выражение будет давать оценку
                    // компонентной ссылке

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

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

    if (remotePort == NULL) {
        ***DYNAMIC-ERROR***; // Удаленный порт не может быть найден
    }
    if (remotePort == SYSTEM) {
        // Порт отображается на порт тестовой системы
        // прием ответа системой SUT выходит за рамки операционной
        // семантики
    }
    else {  // передача ответа
        remotePort.enqueue(CONSTRUCT-ITEM(Entity, reply, replySpec));
    }
} // конец контекста получателя и remotePort

Entity.VALUE-STACK.pop();       // очистить стек значений

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

Рисунок 106/Z.143 – Сегмент <reply-with-receiver-op> потокового графа

9.39.2 Сегмент <reply-without-receiver-op> потокового графа

Сегмент `<reply-without-receiver-op>` потокового графа на рисунке 107 определяет выполнение операции reply без раздела **to**.

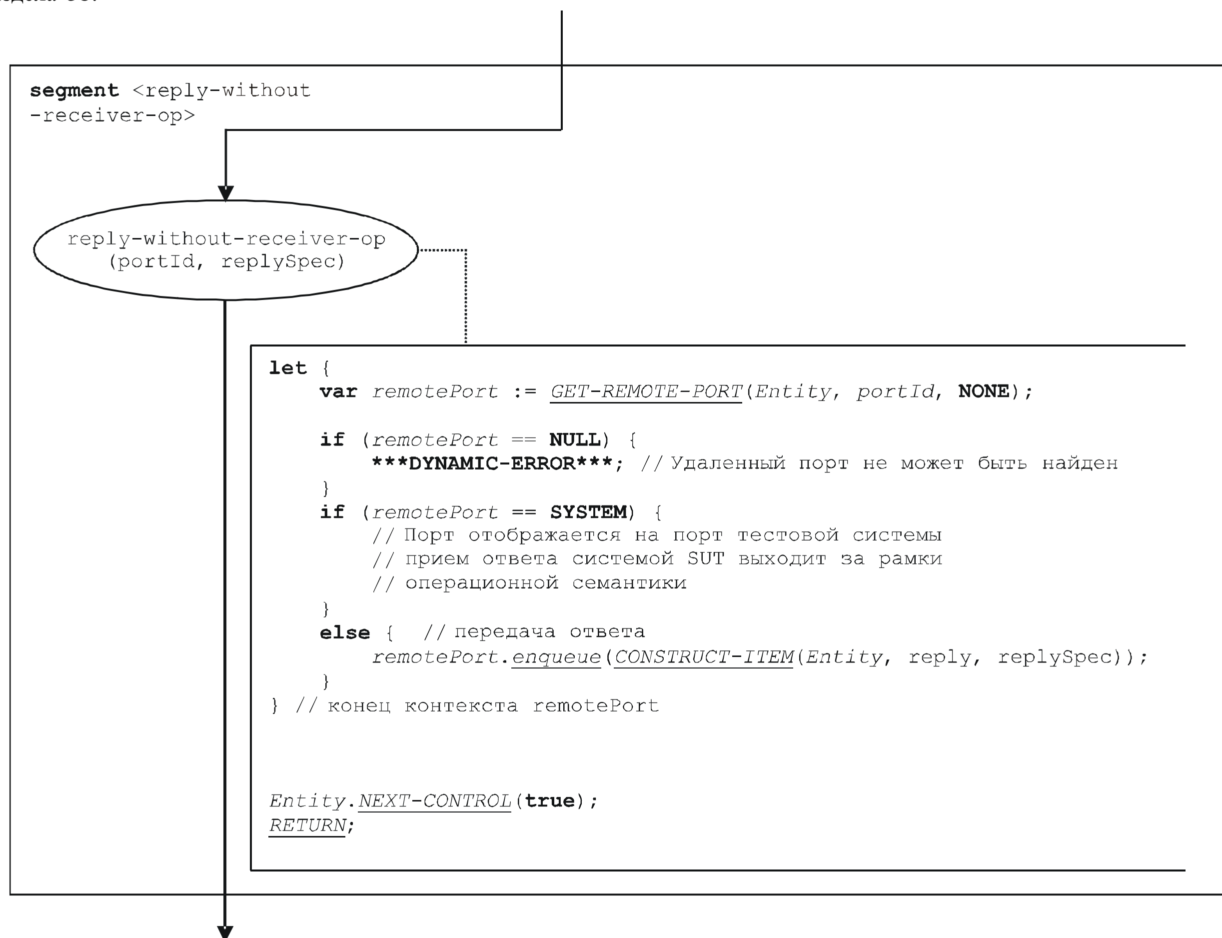


Рисунок 107/Z.143 – Сегмент <reply-without-receiver-op> потокового графа

9.40 Оператор return

Синтаксическая структура оператора **return** (возврат) имеет следующий вид:

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

Факультативная часть `<expression>` описывает возможное возвращаемое значение функции. Выполнение оператора возврата означает, что управление в фактическом контекстном блоке прекращается, т. е. переменные и таймеры, известные только в этом контексте, должны быть удалены, а стек значений должен быть обновлен. Оператор **return** по своему действию схож с компонентной операцией **stop**, если он является последним оператором в описании поведения.

ПРИМЕЧАНИЕ. – Тестовые примеры и управление модулем будут всегда заканчиваться компонентной операцией **stop**. Это вызвано их представлением в виде потокового графа (см. п. 8.2). Только другие тестовые компоненты могут заканчиваться оператором **return**.

Сегмент `<return-stmt>` потокового графа на рисунке 108 определяет выполнение оператора **return**.

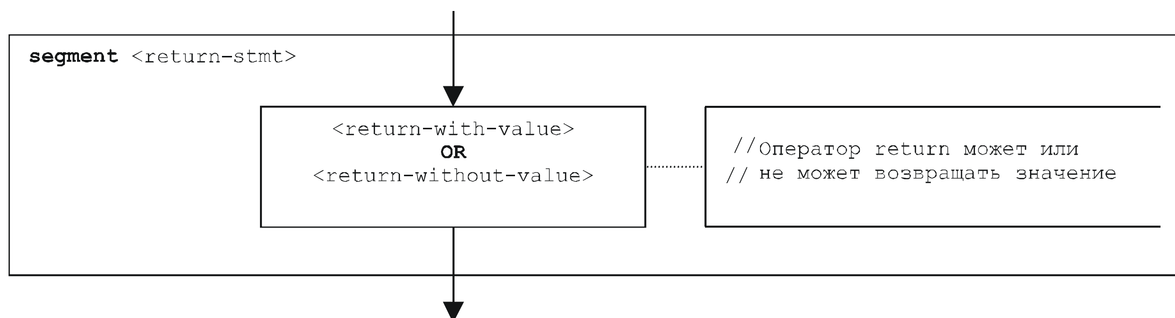


Рисунок 108/Z.143 – Сегмент <return-stmt> потокового графа

9.40.1 Сегмент <return-with-value> потокового графа

Сегмент `<return-with-value>` потокового графа на рисунке 109 определяет выполнение оператора **`return`**, который возвращает значение, заданное в виде выражения.

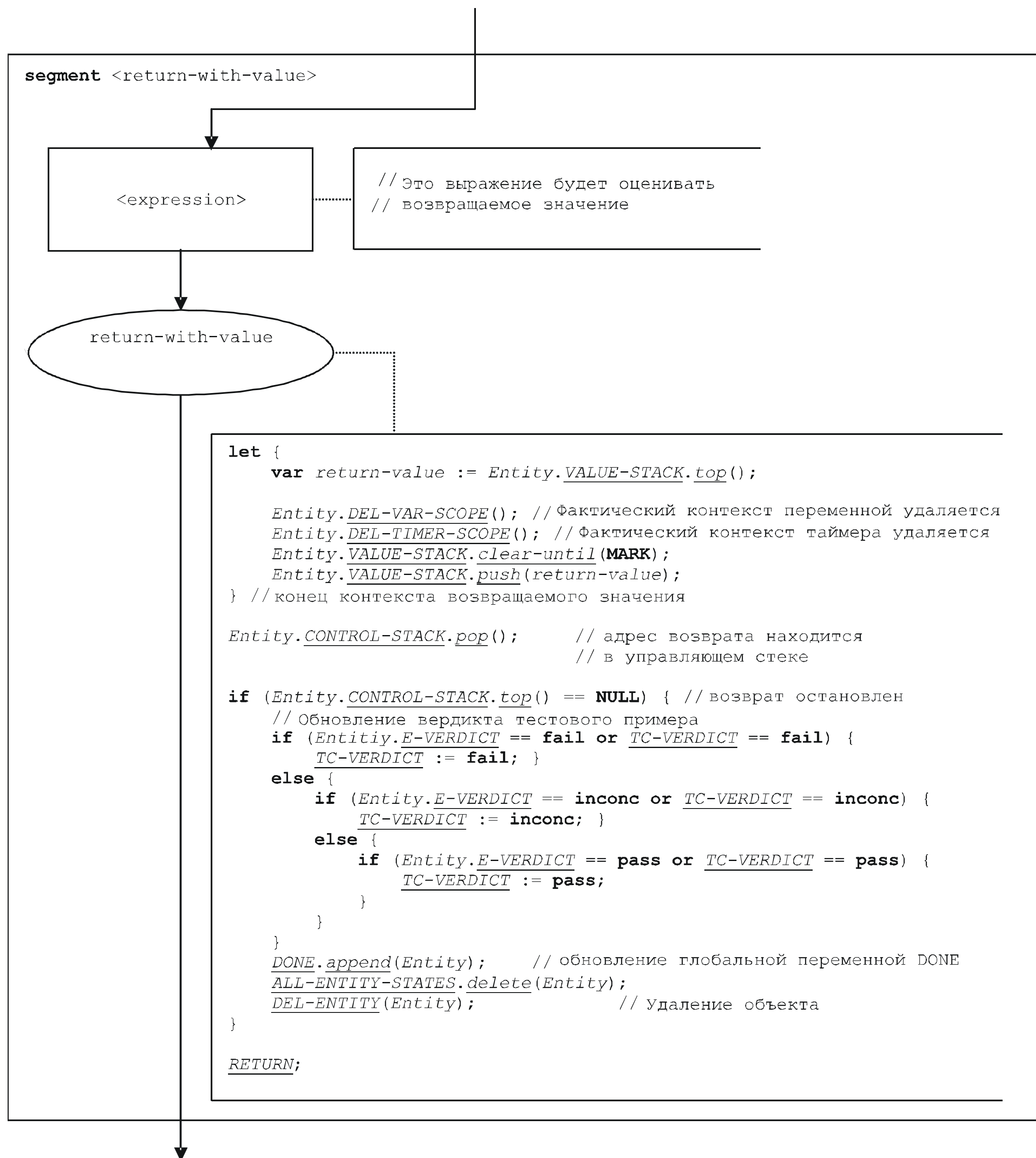


Рисунок 109/Z.143 – Сегмент <return-with-value> потокового графа

9.40.2 Сегмент <return-without-value> потокового графа

Сегмент `<return-without-value>` потокового графа на рисунке 110 определяет выполнение оператора **`return`**, который не возвращает значение.

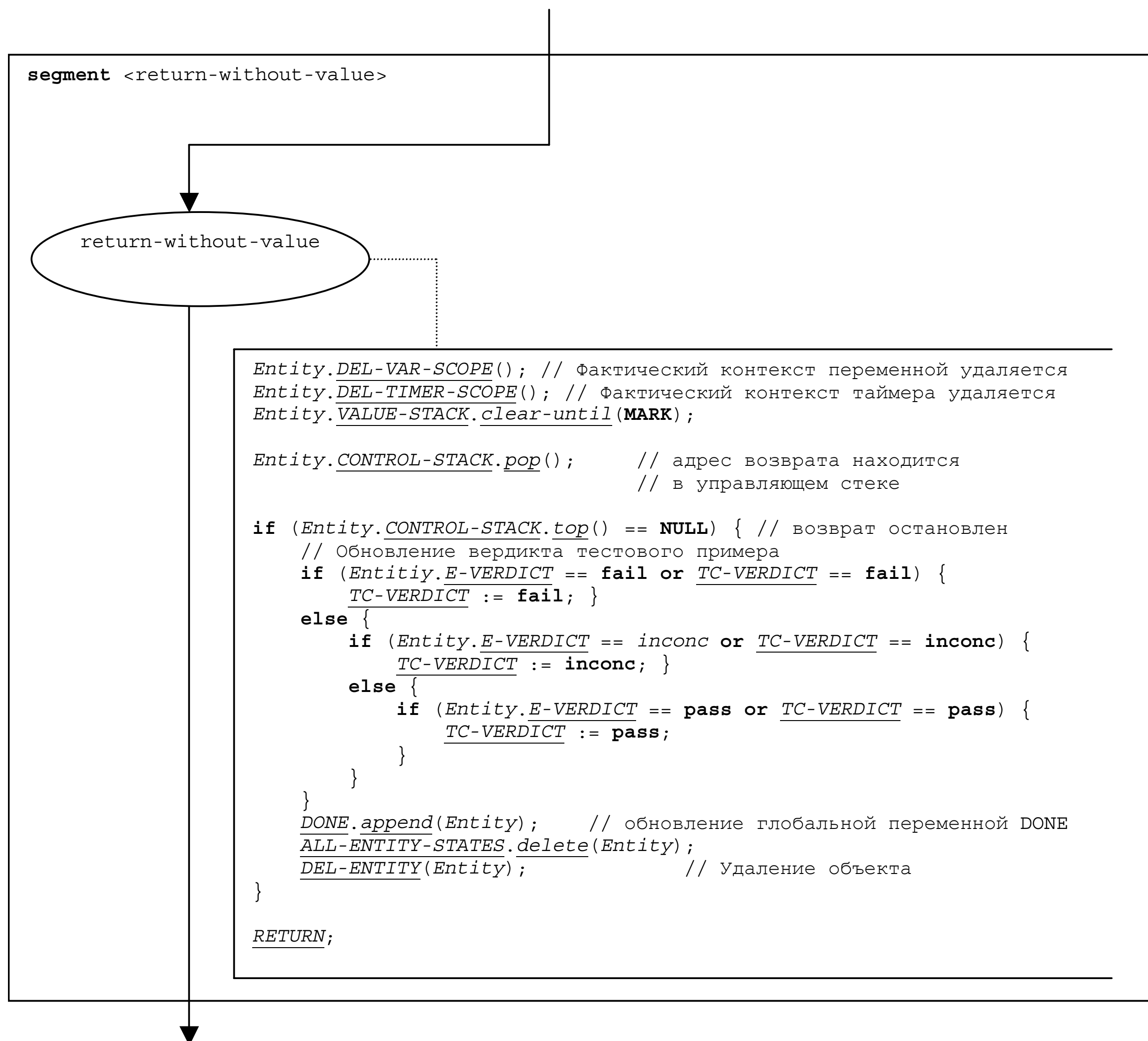


Рисунок 110/Z.143 – Сегмент <return-without-value> потокового графа

9.41 Компонентная операция running

Синтаксическая структура компонентной операции **running** имеет следующий вид:

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

Компонентная операция **running** проверяет, является ли компонент активным или был остановлен. Компонент, подлежащий проверке, идентифицируется компонентной ссылкой, которая может быть представлена в виде переменной или функции, возвращающей значение, т. е. она является выражением. В целях простоты ключевые слова '**all component**' и '**any component**' рассматриваются как специальные выражения.

Компонентная операция **running** отличается по ее использованию в булевой защите оператора **alt** или блокирующей операции **call** и использованию во всех других случаях. При использовании в булевой защите результат компонентной операции **running** базируется на фактической фиксации мгновенного состояния процесса. Во всех других случаях непосредственно оценивается информация о состоянии.

Результат компонентной операции **running** помещается в стек значений объекта, называемого операцией.

Сегмент <running-component-op> потокового графа на рисунке 111 определяет выполнение компонентной операции **running**.

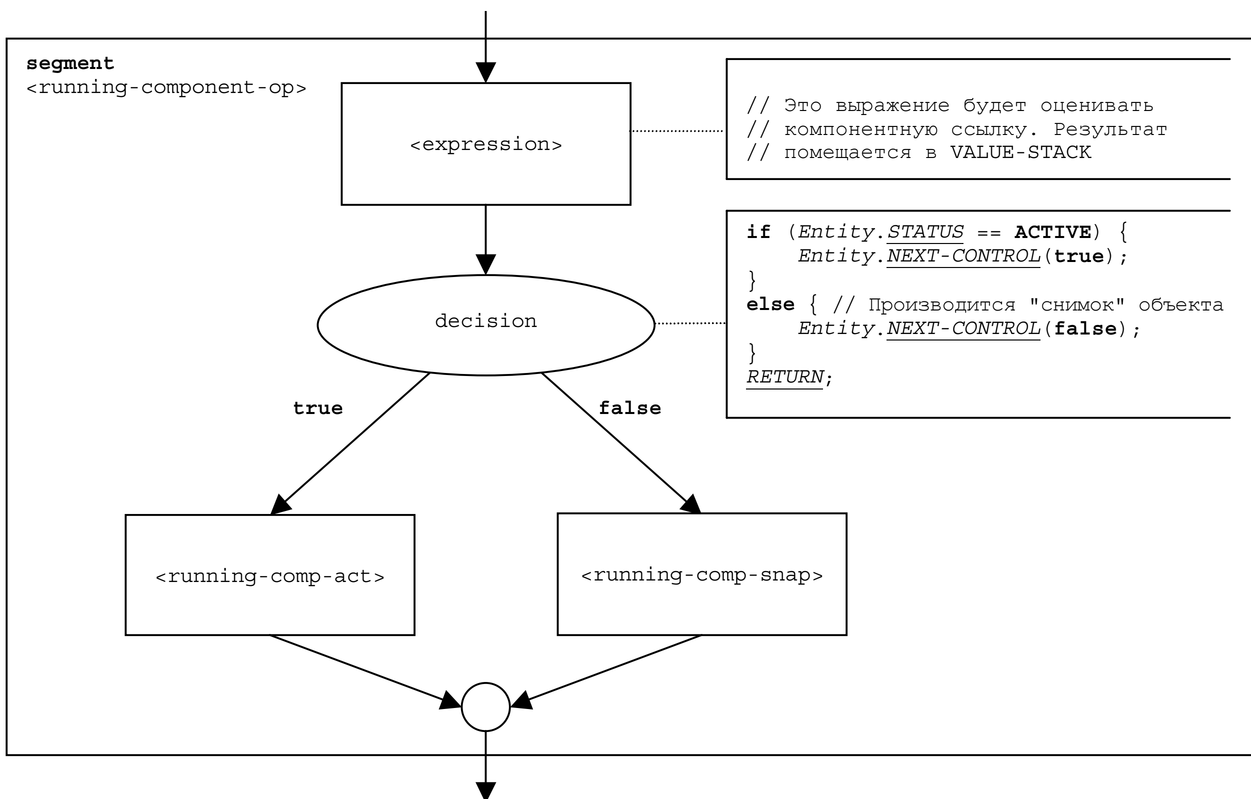


Рисунок 111/Z.143 – Сегмент <running-component-op> потокового графа

9.41.1 Сегмент <running-comp-act> потокового графа

Сегмент `<running-comp-act>` потокового графа на рисунке 112 описывает выполнение компонентной операции **`running`** вне процедуры фиксации мгновенного состояния процесса, т. е. объект находится в состоянии **`ACTIVE`**.

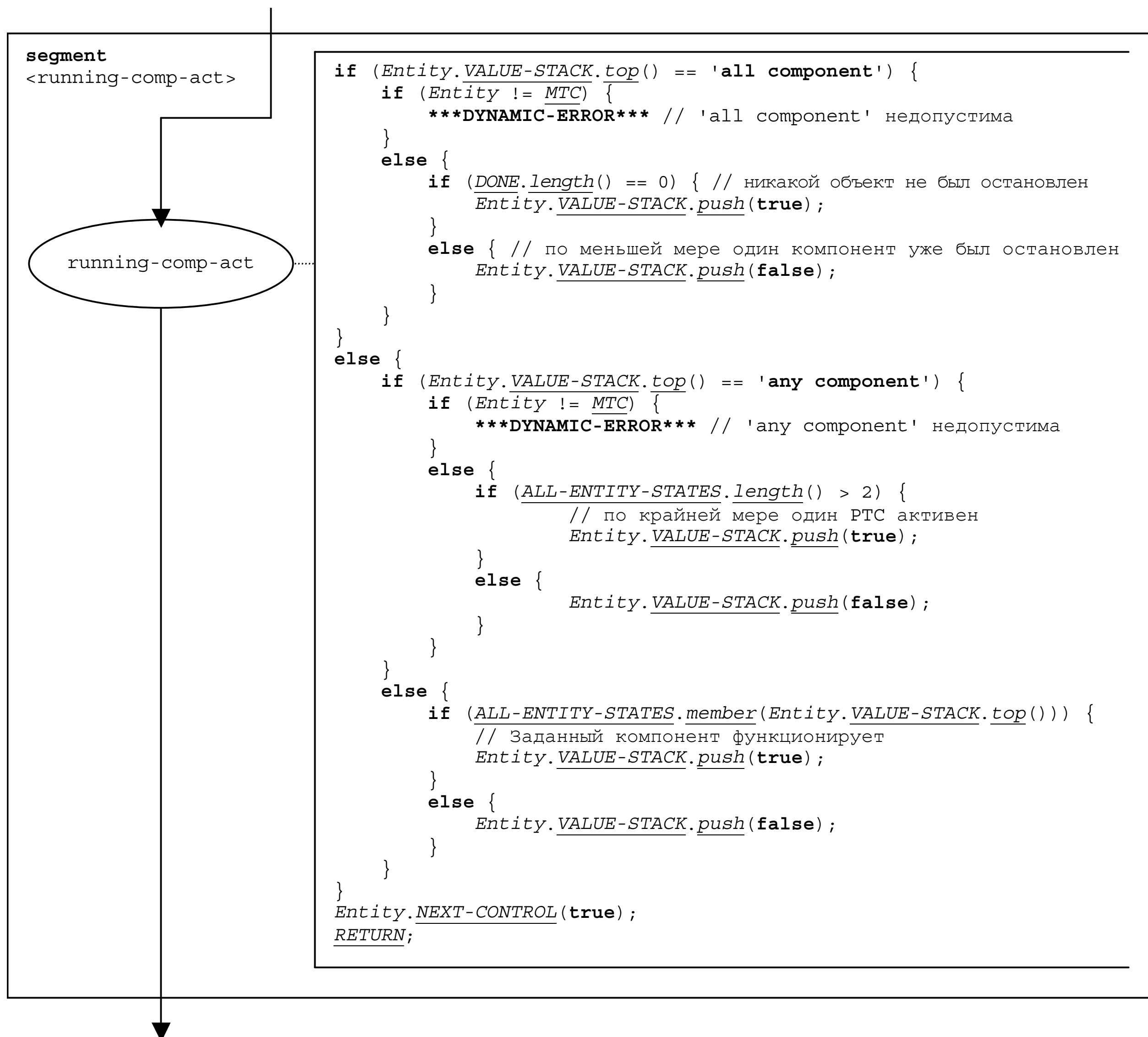


Рисунок 112/Z.143 – Сегмент <running-comp-act> потокового графа

9.41.2 Сегмент <running-comp-snap> потокового графа

Сегмент `<running-comp-snap>` потокового графа на рисунке 113 описывает выполнение компонентной операции **`running`** во время оценки фиксации мгновенного состояния процесса, т. е. объект находится в состоянии **`SNAPSHOT`**.

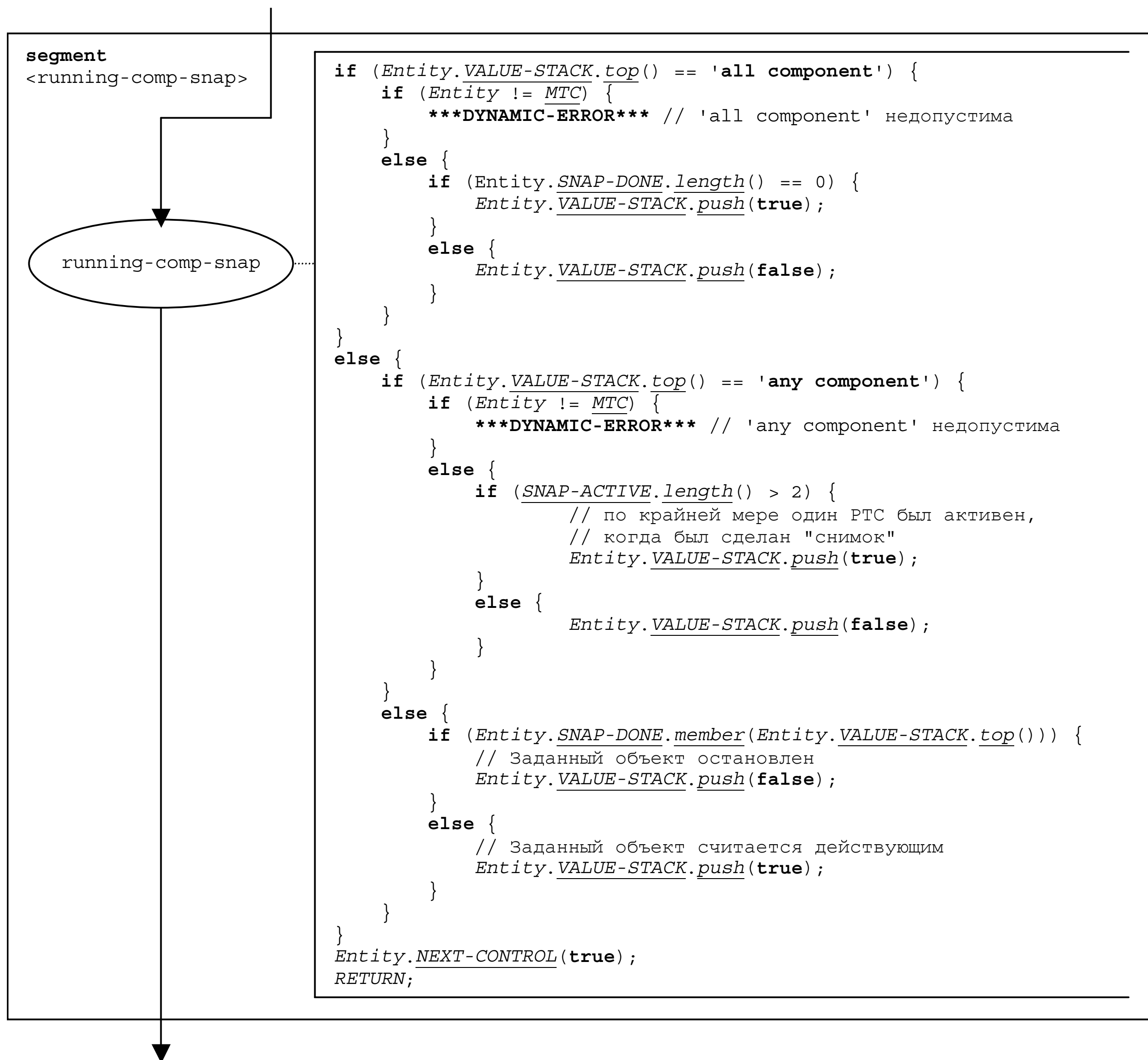


Рисунок 113/Z.143 – Сегмент <running-comp-snap> потокового графа

9.42 Операция running таймера

Синтаксическая структура операции **running** таймера имеет следующий вид:

```
<timerId>.running
```

Сегмент <running-timer-op> потокового графа на рисунке 114 определяет выполнение операции **running** таймера.

Использование операции **running** таймера в булевой защите оператора **alt** или блокирующей операции **call** отличается от ее использования во всех других случаях. При использовании в булевой защите результат операции **running** таймера базируется на фактической фиксации мгновенного состояния процесса, т. е. результат операции определяет компонент *SNAP-STATUS* связывания таймера, а во всех других случаях – компонент *STATUS* связывания таймера.

Ключевое слово **any** обрабатывается как специальное значение идентификатора timerId.

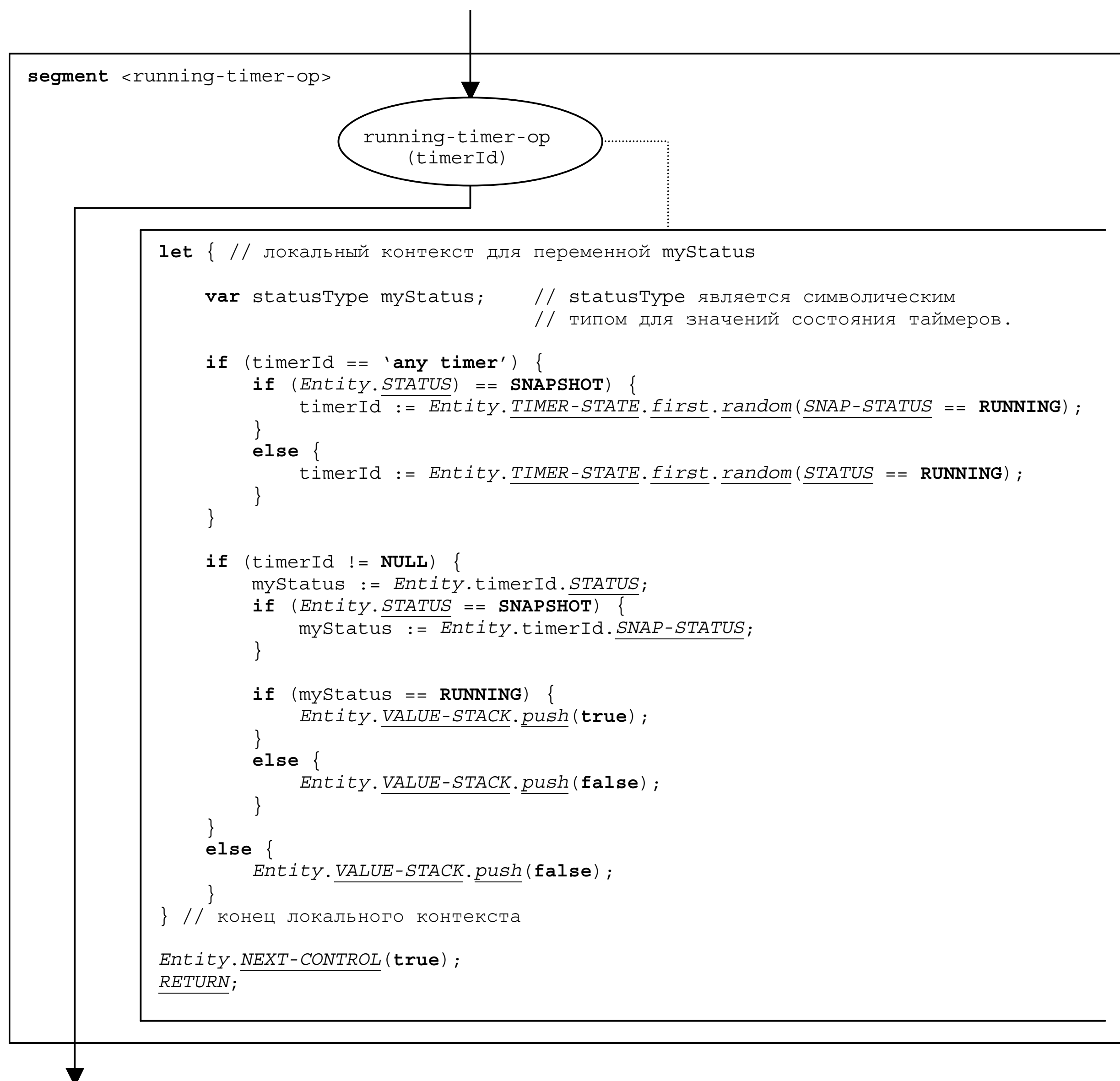


Рисунок 114/Z.143 – Сегмент <running-timer-op> потокового графа

9.43 Операция self

Синтаксическая структура операции **self** имеет следующий вид:

self

Сегмент <self-op> потокового графа на рисунке 115 определяет выполнение операции **self**.

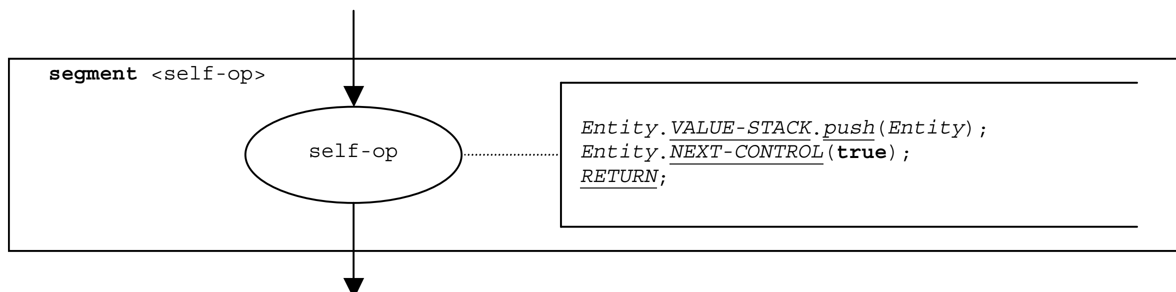


Рисунок 115/Z.143 – Сегмент <self-op> потокового графа

9.44 Операция send

Синтаксическая структура операции **send** (отправить) имеет следующий вид:

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

Факультативная часть <component-expression> в разделе **to** указывает на объект получателя. Она может быть представлена в виде значения переменной или возвращаемого значения функции.

Сегмент <send-op> потокового графа на рисунке 116 определяет выполнение операции **send**.

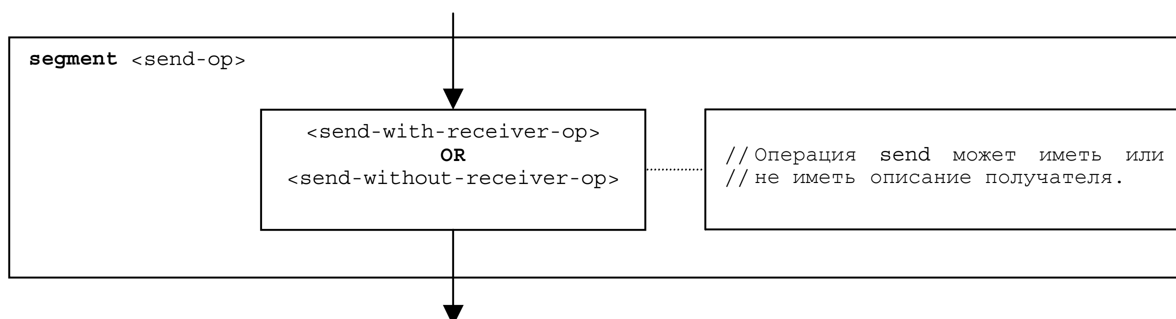


Рисунок 116/Z.143 – Сегмент <send-op> потокового графа

9.44.1 Сегмент <send-with-receiver-op> потокового графа

Сегмент <send-with-receiver-op> потокового графа на рисунке 117 определяет выполнение операции **send**, где получатель задан в виде выражения.

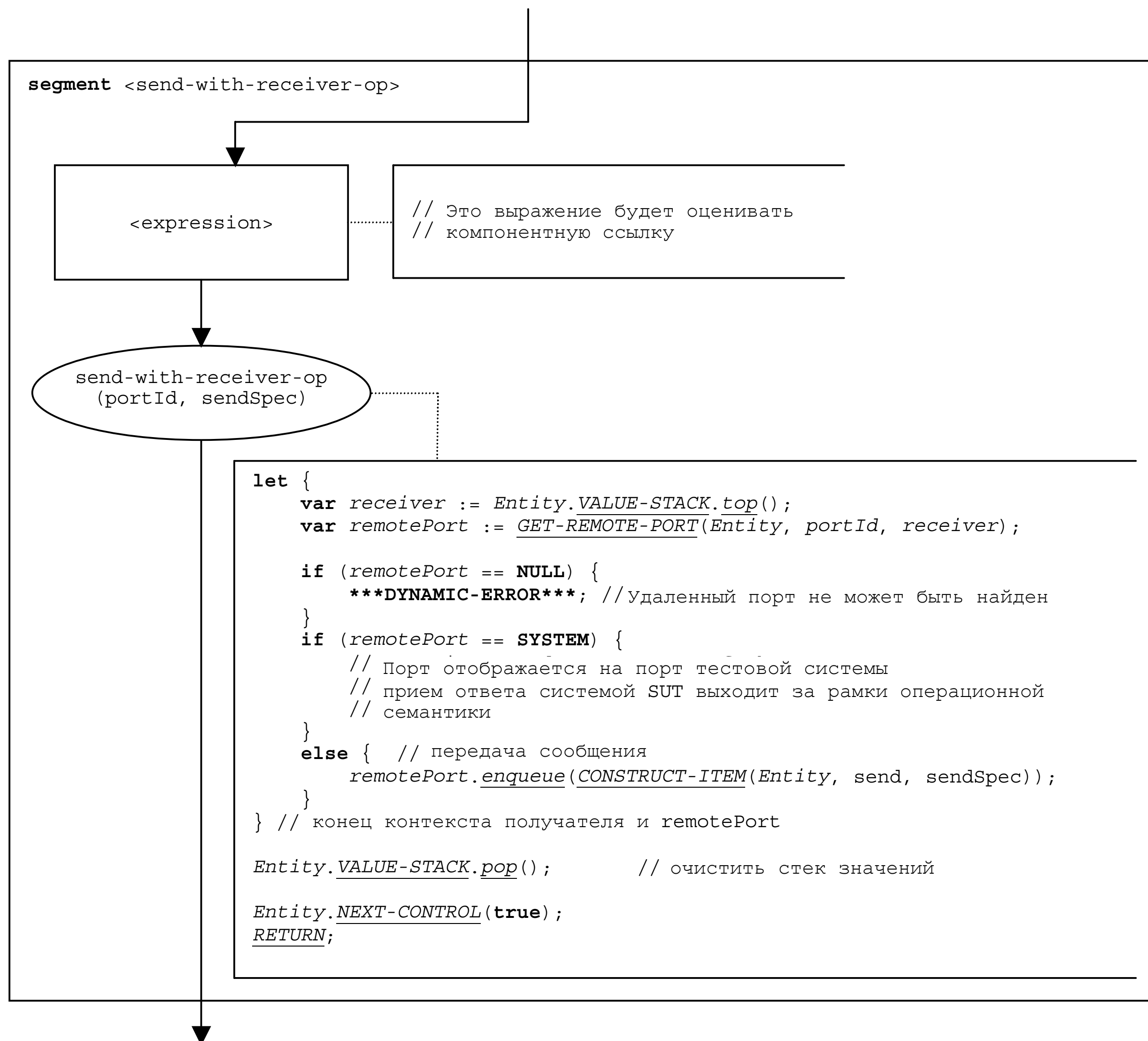


Рисунок 117/Z.143 – Сегмент <send-with-receiver-op> потокового графа

9.44.2 Сегмент <send-without-receiver-op> потокового графа

Сегмент `<send-without-receiver-op>` потокового графа на рисунке 118 определяет выполнение операции **send** в разделе **to**.

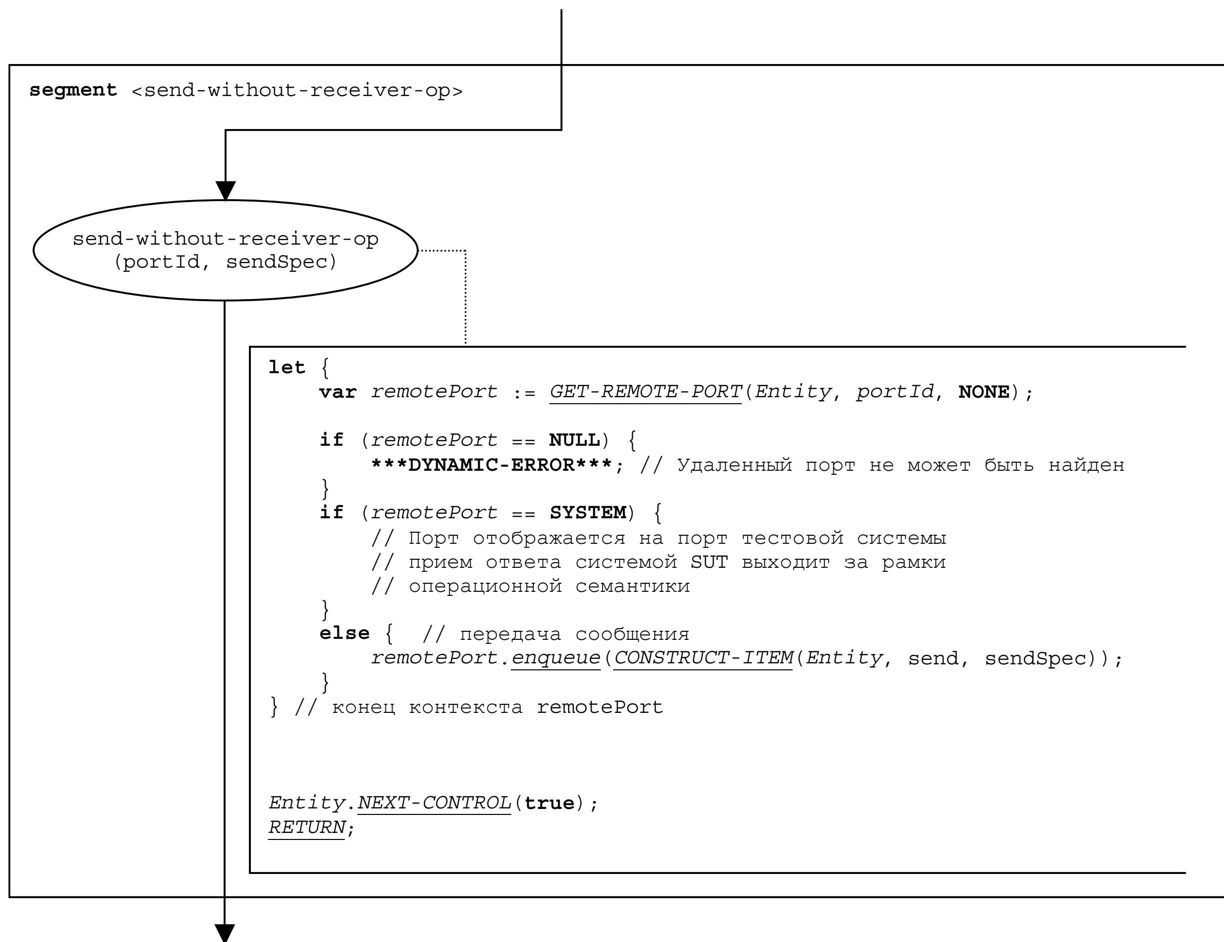


Рисунок 118/Z.143 – Сегмент <send-without-receiver-op> потокового графа

9.45 Операция setverdict

Синтаксическая структура операции **setverdict** имеет следующий вид:

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

Параметр <verdicttype-expression> операции **setverdict** является выражением, которое будет оценивать значение типа **verdicttype**, т. е. **none**, **pass**, **inconc** или **fail**. Выражение оценивается до применения операции **setverdict**.

Сегмент <setverdict-op> потокового графа на рисунке 119 определяет выполнение операции **setverdict**.

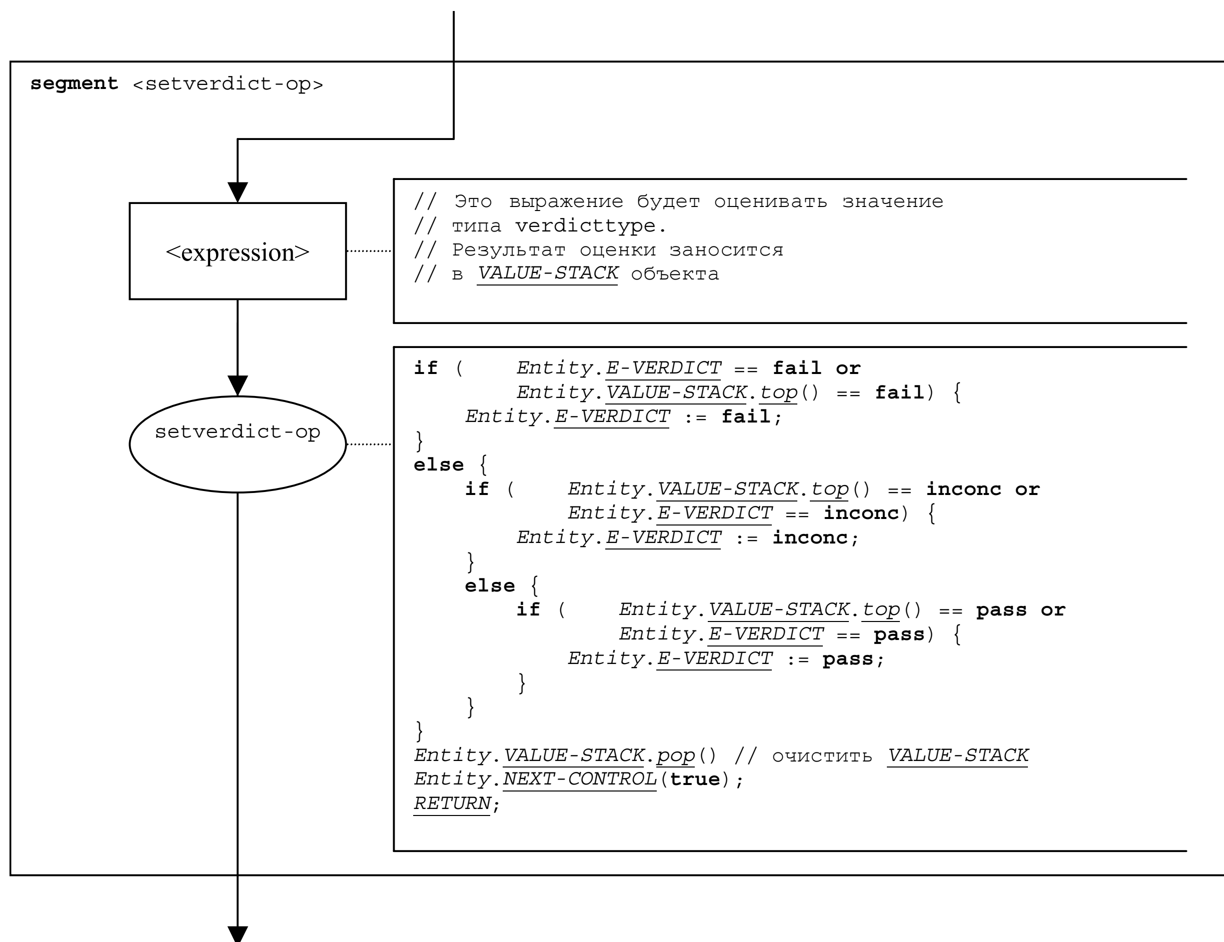


Рисунок 119/Z.143 – Сегмент <setverdict-op> потокового графа

9.46 Компонентная операция start

Синтаксическая структура компонентной операции **start** (запуск) имеет следующий вид:

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

Компонентная операция **start** запускает вновь созданный компонент. Подлежащий запуску компонент идентифицируется путем использования компонентной ссылки. Эта ссылка может храниться в переменной или возвращаться функцией, т. е. она является выражением, оценивающим компонентную ссылку.

Часть <function-name> обозначает имя функции, которая определяет поведение нового компонента, а части <act-par-descr$_1$>, ..., <act-par-descr$_n$> обеспечивают описание значений фактических параметров части <function-name>. В функциях, на которые ссылаются в компонентных операциях **start**, допустимы только параметры-значения. Описания фактических параметров представляются в виде выражений, которые должны оцениваться перед тем, как может быть выполнен вызов. Обработка формальных и фактических параметров-значений подобна их обработке в вызовах функций (см. п. 9.24).

Сегмент <start-component-op> потокового графа на рисунке 120 определяет выполнение компонентной операции **start**. Компонентная операция start выполняется за четыре шага. На первом шаге создается запись вызова. На втором шаге вычисляются значения фактических параметров. На третьем шаге выводится ссылка на подлежащий запуску компонент, а на четвертом шаге новому компоненту передается управление и запись вызова.

ПРИМЕЧАНИЕ. – Сегмент потокового графа на рисунке 120 включает обработку параметров-ссылок (`<ref-var-par-calc>`). Параметры-ссылки необходимы для объяснения параметров-ссылок в тестовых примерах. В операционной семантике предполагается, что эти параметры обрабатываются компонентом MTC.

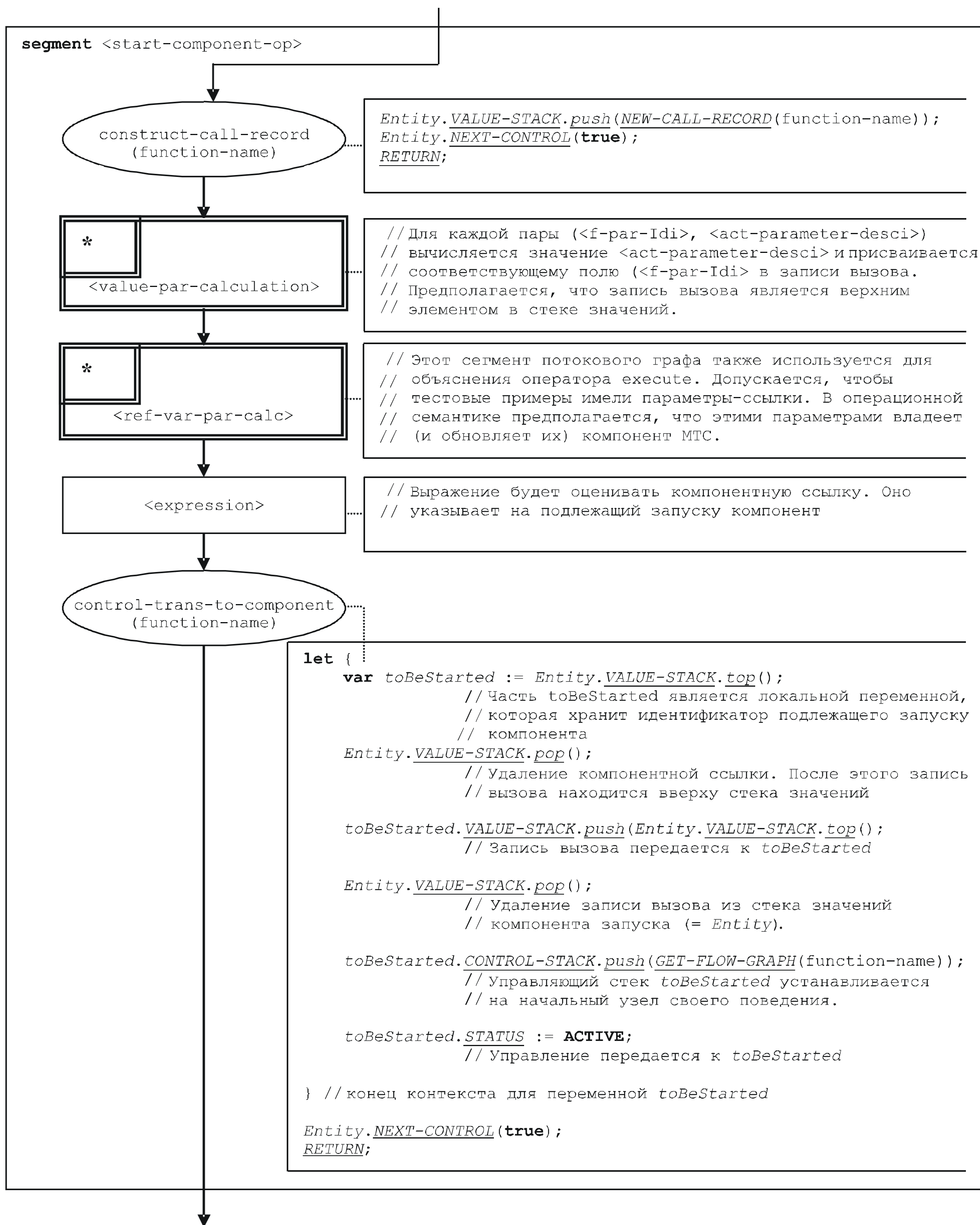


Рисунок 120/Z.143 – Сегмент <start-component-op> потокового графа

9.47 Операция start порта

Синтаксическая структура операции **start** порта (запуск) имеет следующий вид:

```
<portId>.start
```

Сегмент <start-port-op> потокового графа на рисунке 121 определяет выполнение операции **start** порта.

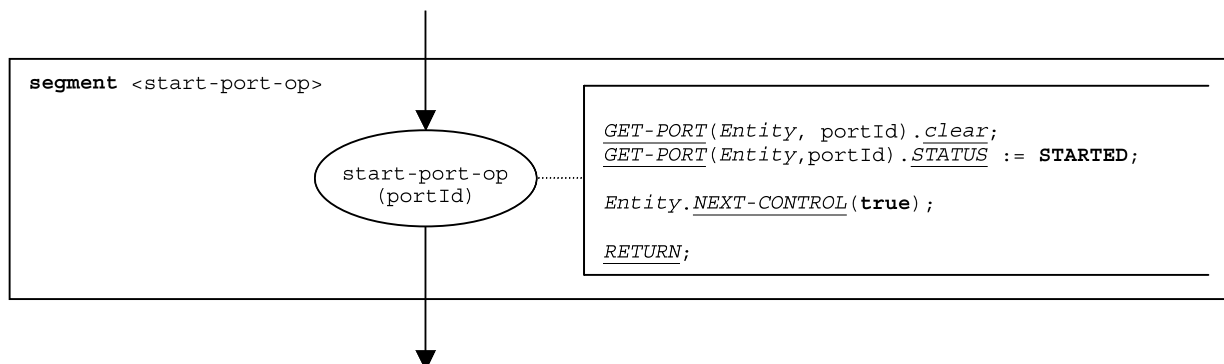


Рисунок 121/Z.143 – Сегмент <start-port-op> потокового графа

9.48 Операция start таймера

Синтаксическая структура операции **start** (запуск) таймера имеет следующий вид:

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

Факультативный параметр <float-expression> операции **start** таймера обозначает фактический период времени таймера. Если он не представлен, то в операции **start** будет использоваться период времени по умолчанию. Выражение будет оценивать значение типа **float**. Если такое выражение имеется, оно будет оцениваться до применения операции **start**. Результат оценки помещается в *VALUE-STACK* объекта (*Entity)*.

Сегмент <start-timer-op> потокового графа на рисунке 122 определяет выполнение операции **start** таймера.

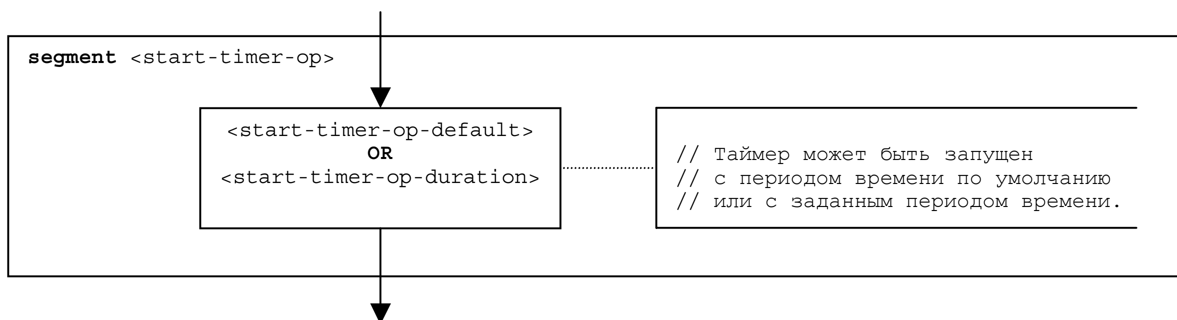


Рисунок 122/Z.143 – Сегмент <start-timer-op> потокового графа

9.48.1 Сегмент <start-timer-op-default> потокового графа

Сегмент `<start-timer-op-default>` потокового графа на рисунке 123 определяет выполнение операции **`start`** таймера с указанием значения по умолчанию.

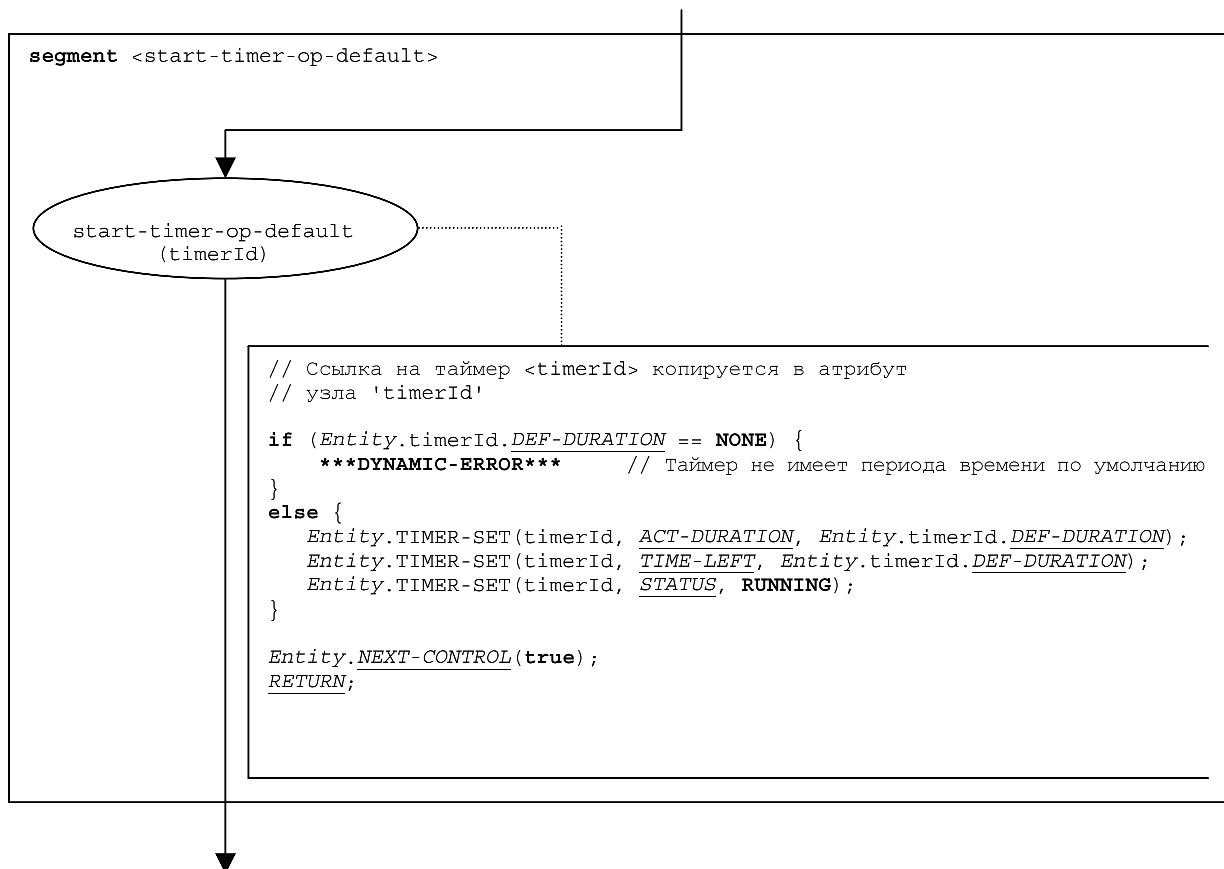


Рисунок 123/Z.143 – Сегмент <start-timer-op-default> потокового графа

9.48.2 Сегмент <start-timer-op-duration> потокового графа

Сегмент `<start-timer-op-duration>` потокового графа на рисунке 124 определяет выполнение операции **`start`** таймера с указанием заданного периода времени.

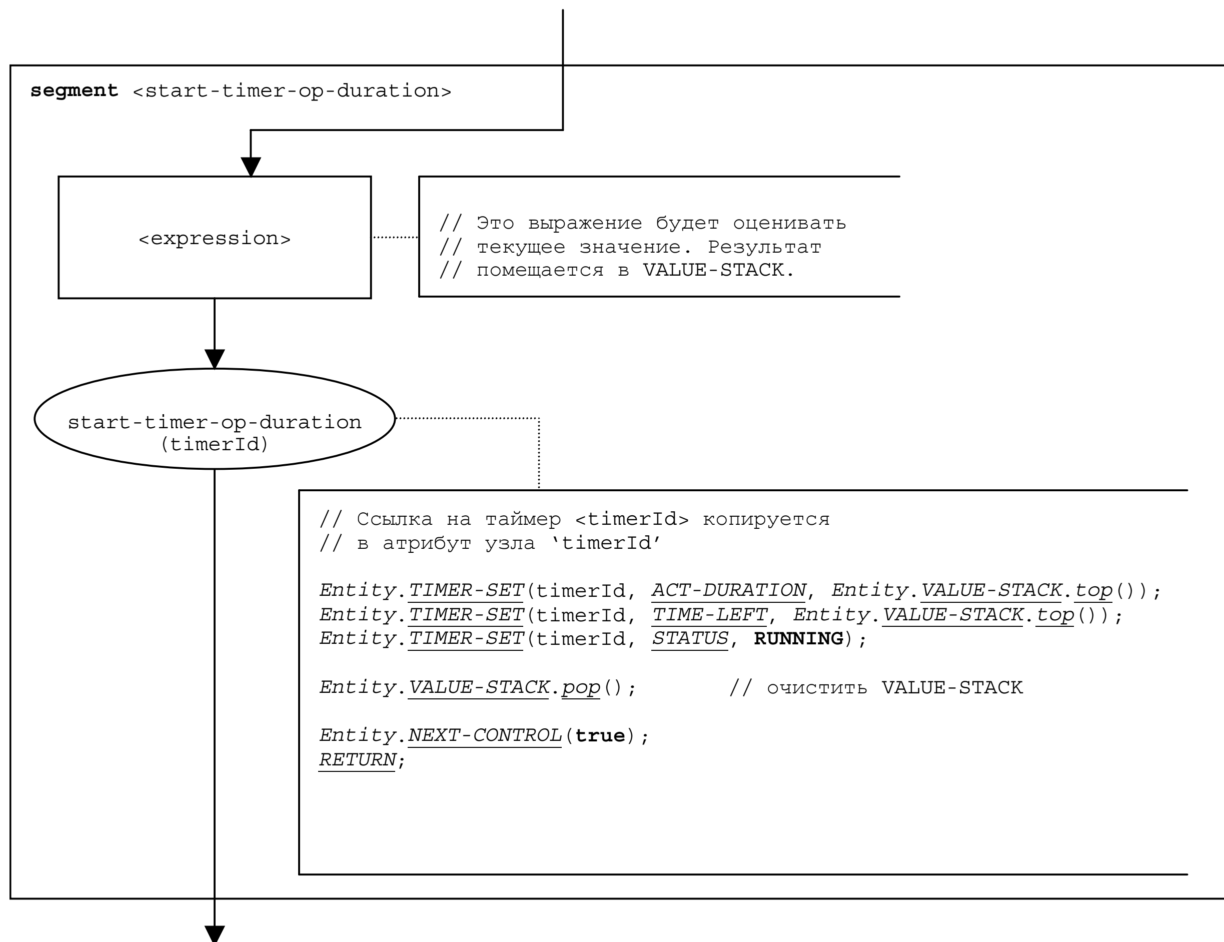


Рисунок 124/Z.143 – Сегмент <start-timer-op-duration> потокового графа

9.49 Компонентная операция stop

Синтаксическая структура компонентной операции **`stop`** (останов) имеет следующий вид:

```
<component-expression>.stop
```

Компонентная операция **`stop`** останавливает заданный компонент. Все тестовые компоненты будут остановлены, т. е. тестовый пример завершается, если компонент MTC остановлен (например, **`mtc.stop`**) или останавливается сам (например, **`self.stop`**). Компонент MTC может остановить все параллельные тестовые компоненты, используя ключевое слово **`all`**, т. е. **`all component.stop`**.

Подлежащий останову компонент идентифицируется компонентной ссылкой, представленной в виде выражения, например, значения или функции, возвращающей значение. В целях простоты ключевое слово '**`all component`**' считается специальным значением части `<component-expression>`. Операции **`mtc`** и **`self`** оцениваются согласно разделам 9.33 и 9.43.

Сегмент <stop-component-op> потокового графа на рисунке 125 определяет выполнение компонентной операции **stop**.

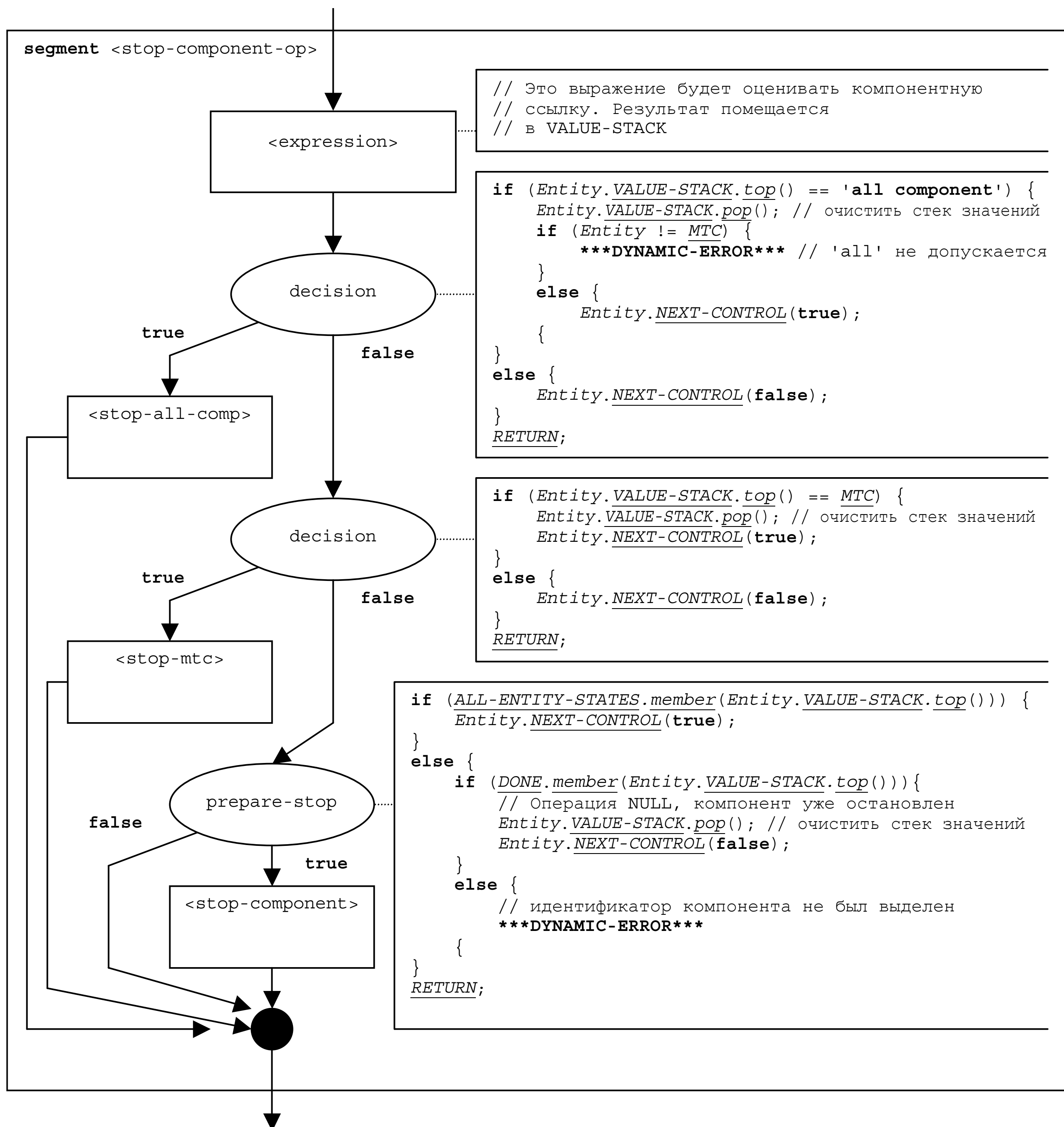


Рисунок 125/Z.143 – Сегмент <stop-component-op> потокового графа

9.49.1 Сегмент <stop-mtc> потокового графа

Сегмент <stop-mtc> потокового графа на рисунке 126 описывает процесс останова компонента MTC тестового примера. Результатом является то, что тестовый пример завершается, т. е. вычисляется конечный вердикт и помещается в стек значений управления модулем, освобождаются все ресурсы, список DONE состояния модуля очищается, а все тестовые компоненты, включая MTC, завершаются.

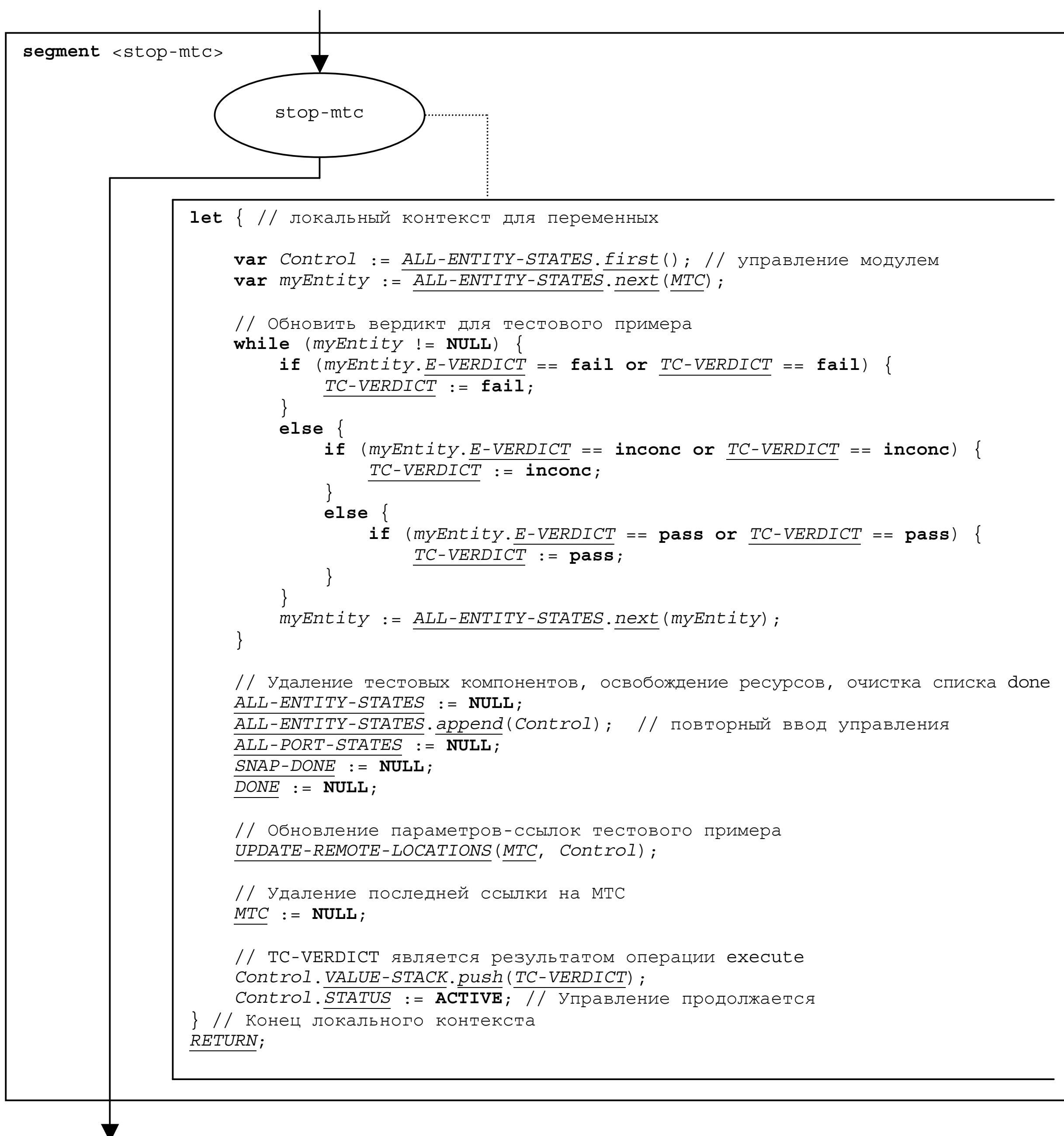


Рисунок 126/Z.143 – Сегмент <stop-mtc-op> потокового графа

9.49.2 Сегмент <stop-component> потокового графа

Сегмент <stop-component> потокового графа на рисунке 127 описывает процесс останова параллельного тестового компонента, т. е. не являющегося компонентом МТС или управления модулем. Результатом является то, что обновляются вердикт *TC-VERDICT* тестового примера и список завершенных тестовых компонентов (*DONE*), и то, что компонент удаляется из состояния модуля. В потоковом графе <stop-component> предполагается, что идентификатор компонента, подлежащего останову, находится на вершине стека значений компонента, выполняющего данный сегмент.

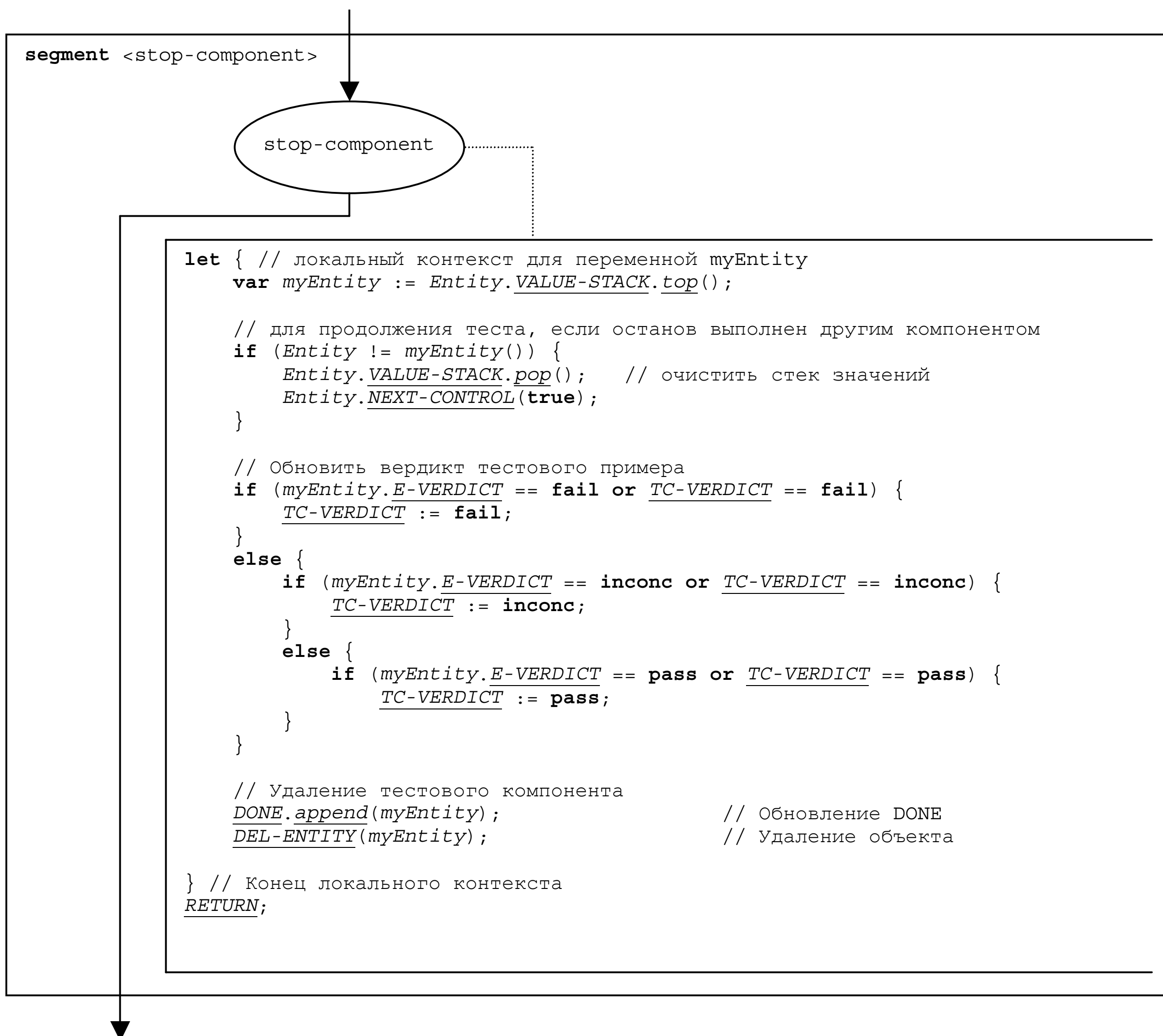


Рисунок 127/Z.143 – Сегмент <stop-component> потокового графа

9.49.3 Сегмент <stop-all-comp> потокового графа

Сегмент `<stop-all-comp>` потокового графа на рисунке 128 описывает процесс останова всех параллельных тестовых компонентов тестового примера.

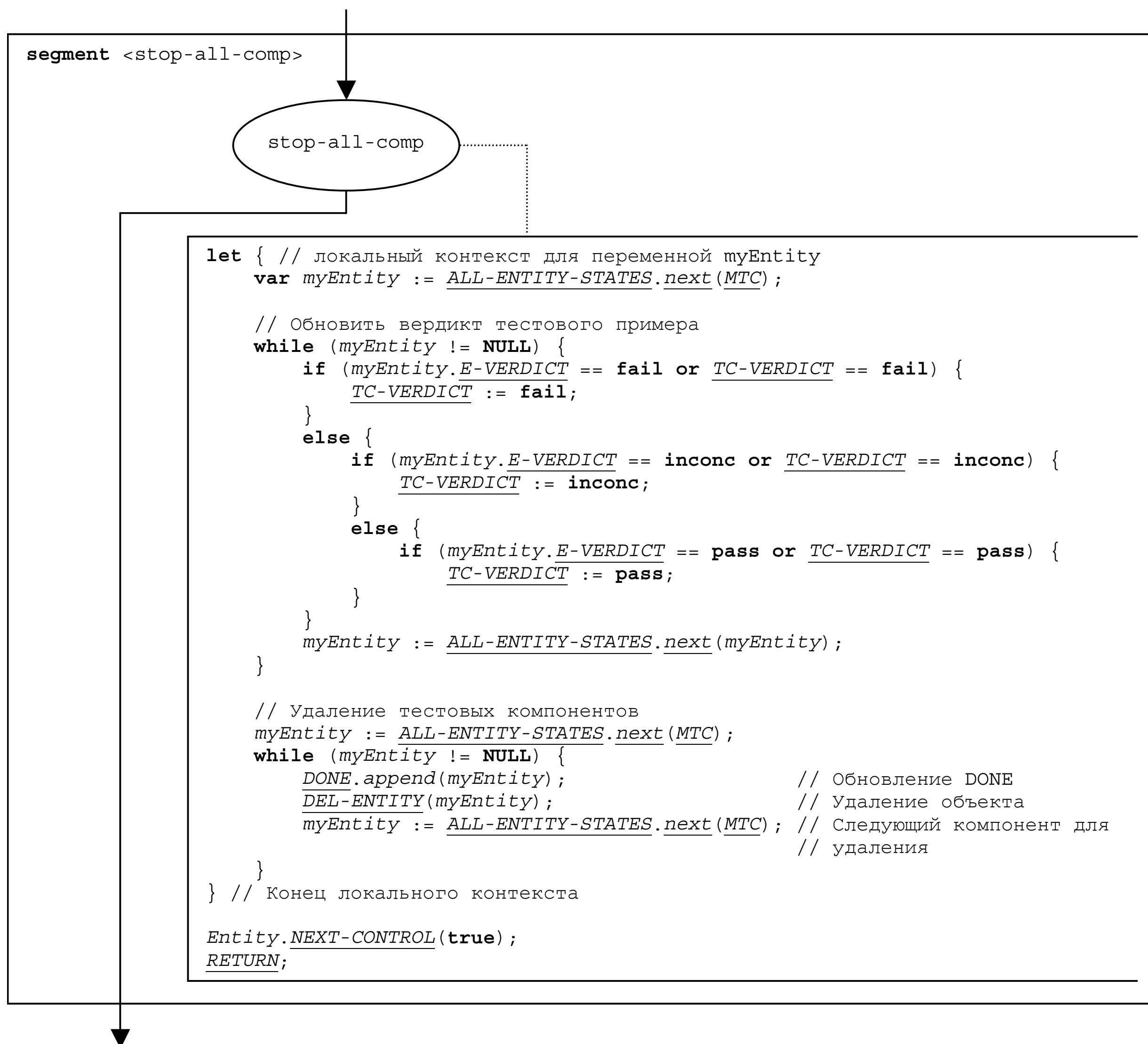


Рисунок 128/Z.143 – Сегмент <stop-all-comp> потокового графа

9.50 Оператор stop выполнения

Семантическая структура оператора **`stop`** (останов) выполнения имеет следующий вид:

```
stop
```

Результат оператора **`stop`** выполнения зависит от объекта, который осуществляет выполнение оператора **`stop`** выполнения:

a) Если **`stop`** выполняется управлением модуля, то выполнение тестирования заканчивается, т. е. все тестовые компоненты и управление модулем удаляются из состояния модуля.

b) Если **`stop`** выполняется компонентом МТС, то все параллельные тестовые компоненты и МТС останавливают выполнение. Глобальный вердикт тестового примера обновляется и заносится в стек значений управления модулем. В итоге управление передается обратно к управлению модулем, а компонент МТС завершается.

c) Если **`stop`** выполняется тестовым компонентом, то обновляются глобальный вердикт *TC-VERDICT* тестового примера и глобальный список *DONE*. Затем этот компонент полностью удаляется из модуля.

Сегмент <stop-exec-stmt> потокового графа на рисунке 129 описывает выполнение оператора останова.

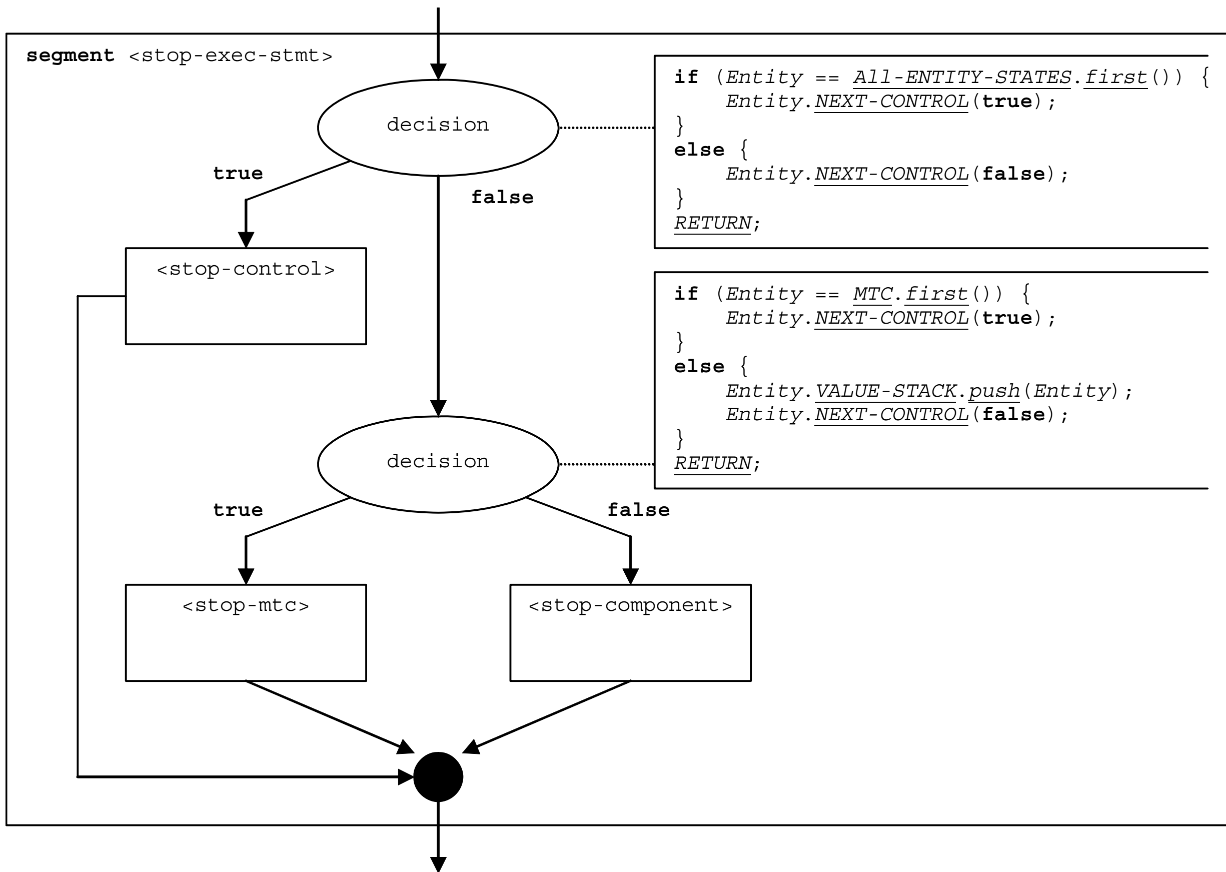


Рисунок 129/Z.143 – Сегмент <stop-exec-stmt> потокового графа

9.50.1 Сегмент <stop-control> потокового графа

Сегмент <stop-control> потокового графа на рисунке 130 описывает процесс останова выполнения модуля. Результатом является то, что *ALL-ENTITY-STATES* устанавливается на **NULL**, т. е. выполняется условие завершения процедуры оценки модуля (см. п. 8.6).

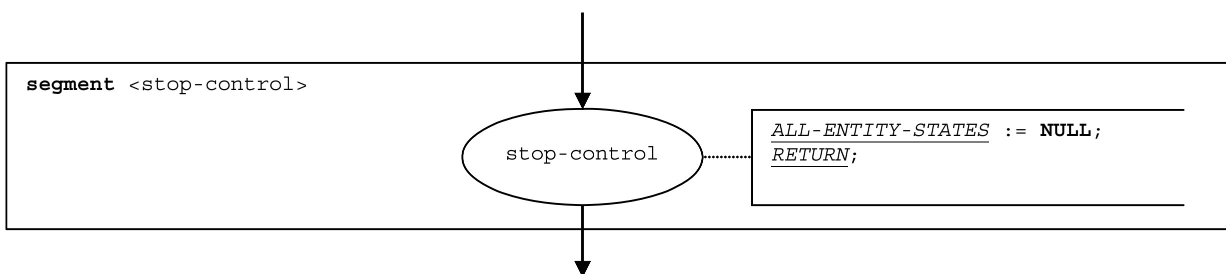


Рисунок 130/Z.143 – Сегмент <stop-control> потокового графа

9.51 Операция stop порта

Семантическая структура операции **stop** (останов) порта имеет следующий вид:

<portId>**.stop**

Сегмент <stop-port-op> потокового графа определяет выполнение операции **stop** порта.

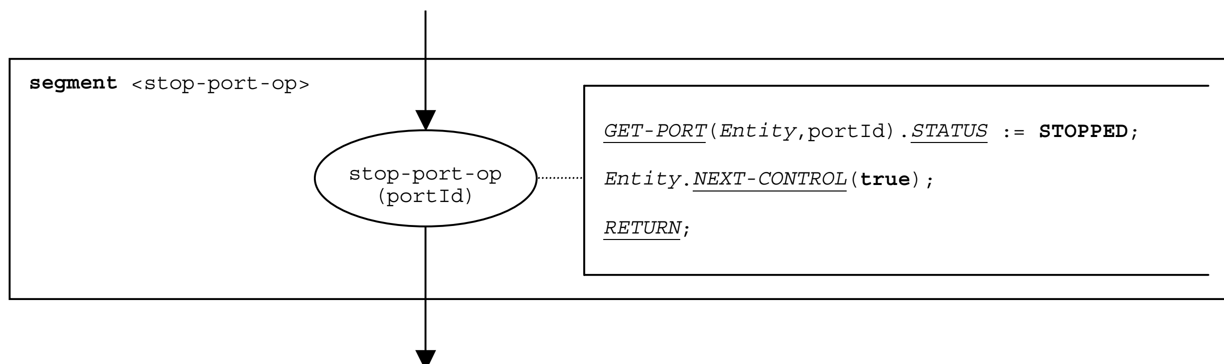


Рисунок 131/Z.143 – Сегмент <stop-port-op> потокового графа

9.52 Операция stop таймера

Семантическая структура операции **stop** (останов) таймера имеет следующий вид:

<timerId>**.stop**

Сегмент <stop-timer-op> потокового графа на рисунке 132 определяет выполнение операции **stop** таймера.

Ключевое слово **all** обрабатывается в качестве специального значения для timerId.

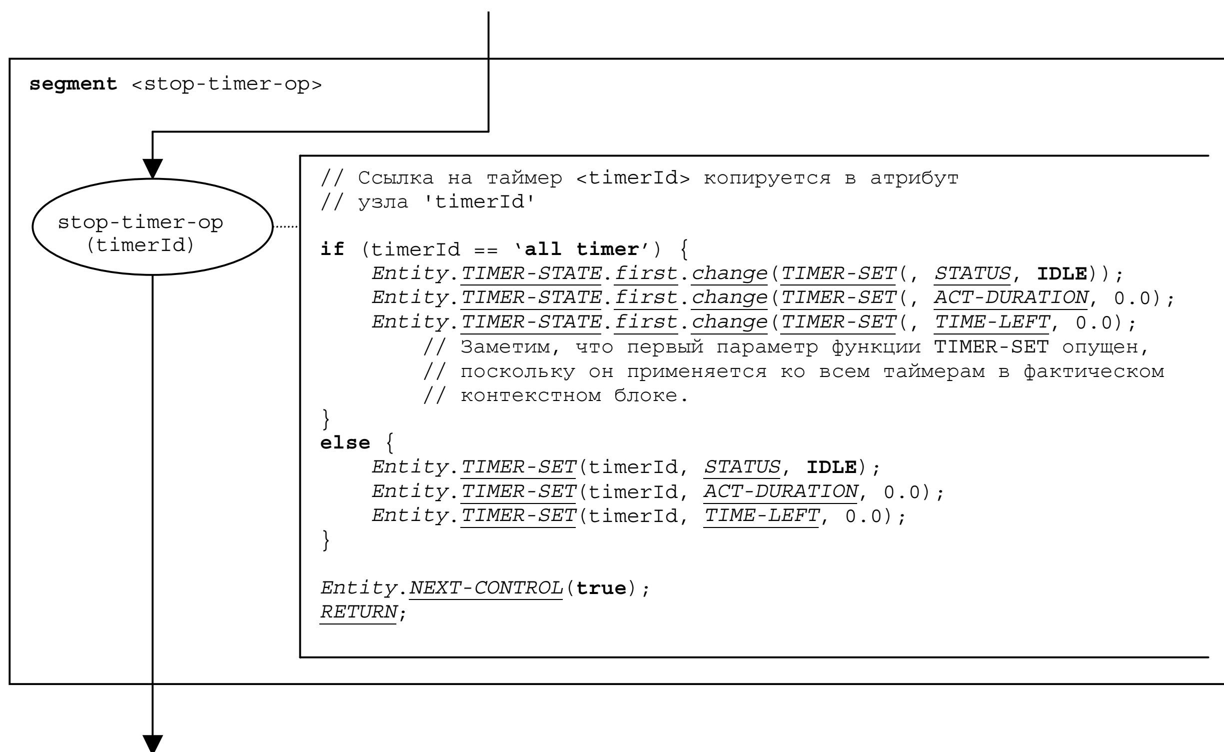


Рисунок 132/Z.143 – Сегмент <stop-timer-op> потокового графа

9.53 Операция system

Семантическая структура операции **system** (система) имеет следующий вид:

system

Сегмент <system-op> потокового графа на рисунке 133 определяет выполнение операции **system**.

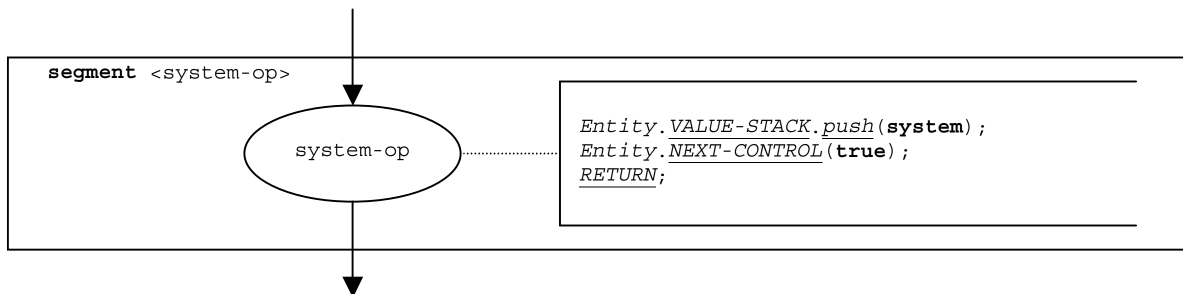


Рисунок 133/Z.143 – Сегмент <system-op> потокового графа

9.54 Объявление timer

Семантическая структура объявления **timer** (таймер) имеет следующий вид:

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

Результатом объявления таймера является создание нового связывания таймера. Объявление периода времени по умолчанию является факультативным. Значение по умолчанию рассматриваются в качестве выражения, оценивающего значение типа **float**.

Сегмент <timer-declaration> потокового графа на рисунке 134 определяет выполнение объявления таймера.

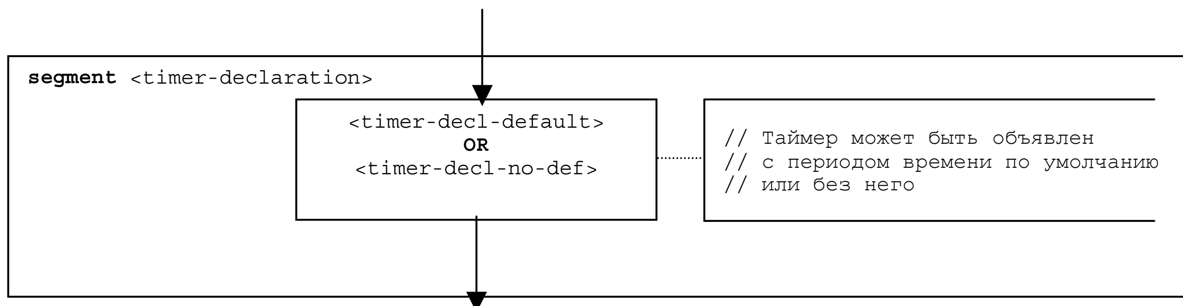


Рисунок 134/Z.143 – Сегмент <timer-declaration> потокового графа

9.54.1 Сегмент <timer-decl-default> потокового графа

Сегмент `<timer-decl-default>` потокового графа на рисунке 135 определяет выполнение объявления таймера, где период времени по умолчанию дается в виде выражения.

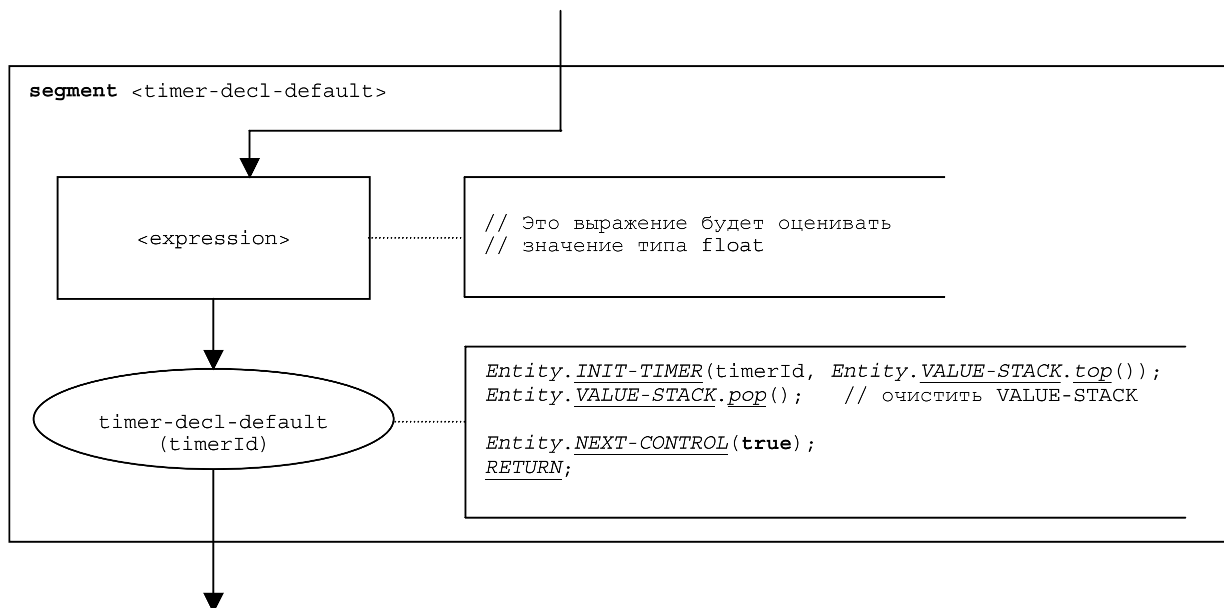


Рисунок 135/Z.143 – Сегмент <timer-decl-default> потокового графа

9.54.2 Сегмент <timer-decl-no-def> потокового графа

Сегмент `<timer-decl-no-def>` потокового графа на рисунке 136 определяет выполнение объявления таймера, когда не указывается период времени по умолчанию, т. е. период времени по умолчанию для таймера является неопределенным.

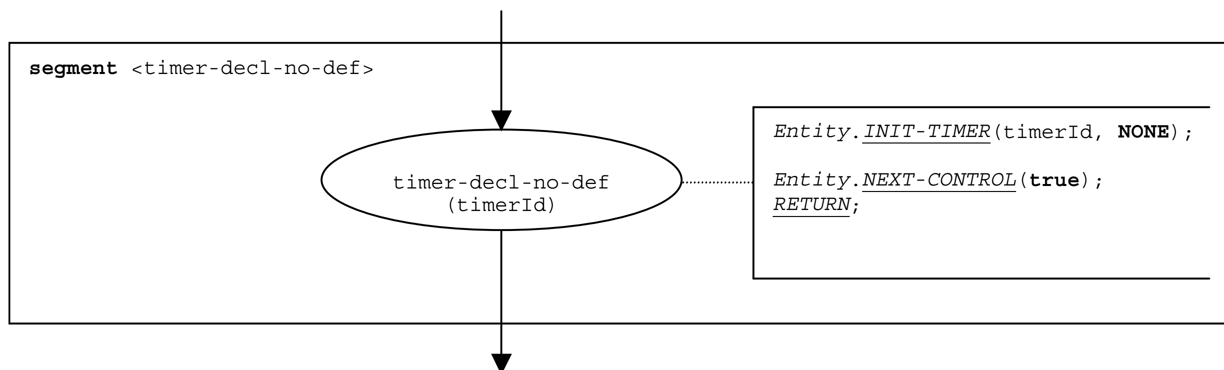


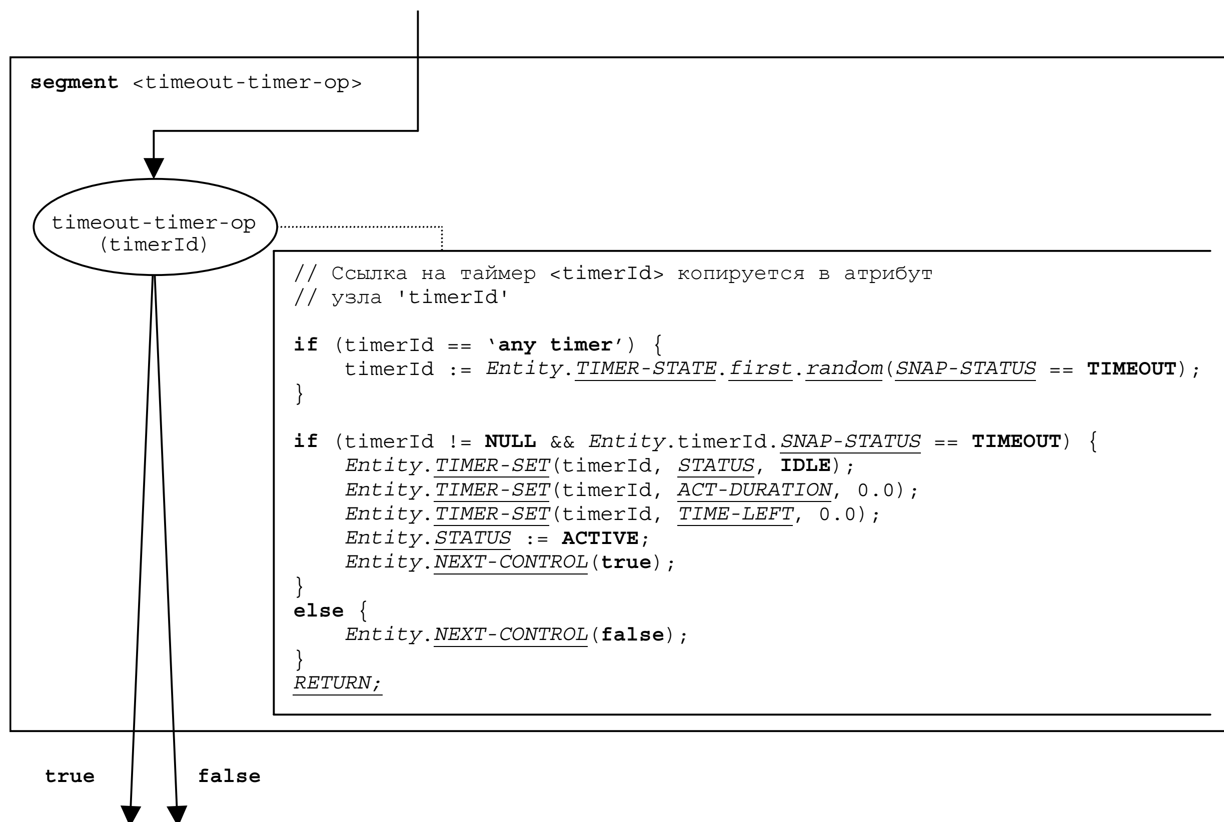
Рисунок 136/Z.143 – Сегмент <timer-decl-no-def> потокового графа

9.55 Операция timeout таймера

Семантическая структура операции **timeout** (тайм-аут) таймера имеет следующий вид:

```
<timerId>.timeout
```

Сегмент <timeout-timer-op> потокового графа на рисунке 137 определяет выполнение операции **timeout** таймера.



ПРИМЕЧАНИЕ 1. – Операция **timeout** вложена в оператор **alt**. Ее оценка базируется на фактической фиксации мгновенного состояния процесса, т. е. решение базируется на компоненте *SNAP-STATUS* в связывании таймера. Если операция тайм-аута успешна, т. е. *SNAP-STATUS* == TIMEOUT, то таймер устанавливается в состояние IDLE, а состояние компонента изменяется с SNAPSHOT на ACTIVE.

ПРИМЕЧАНИЕ 2. – Когда **timeout** производит оценку в отношении **true** или **false**, то выполнение либо продолжается с оператором, следующим за операцией **timeout** (ветвь **true**), либо в операторе **alt** должна быть проверена следующая альтернатива (ветвь **false**).

ПРИМЕЧАНИЕ 3. – Ключевое слово **any** интерпретируется подобно специальному значению для timerId.

Рисунок 137/Z.143 – Сегмент <timeout-timer-op> потокового графа

9.56 Операция unmap

Синтаксическая структура операции **unmap** (не отображать) имеет следующий вид:

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

Идентификаторы <portId1> и <portId2> считаются идентификаторами порта соответствующего интерфейса тестового компонента и тестовой системы. На компоненты, к которым принадлежит порт <portId1>, ссылаются с помощью компонентной ссылки <component-expression>. Ссылка может храниться в переменных, либо возвращается функцией, т. е. она является выражением, которое оценивает компонентную ссылку. Для хранения компонентной ссылки используется стек значений.

ПРИМЕЧАНИЕ. – Для операции **unmap** не имеет значения, появляется ли оператор **system**:<portId> в качестве первого или в качестве второго параметра. В целях простоты предполагается, что это всегда второй параметр.

Выполнение операции **unmap** определяется сегментом <unmap-op> потокового графа, показанным на рисунке 138.

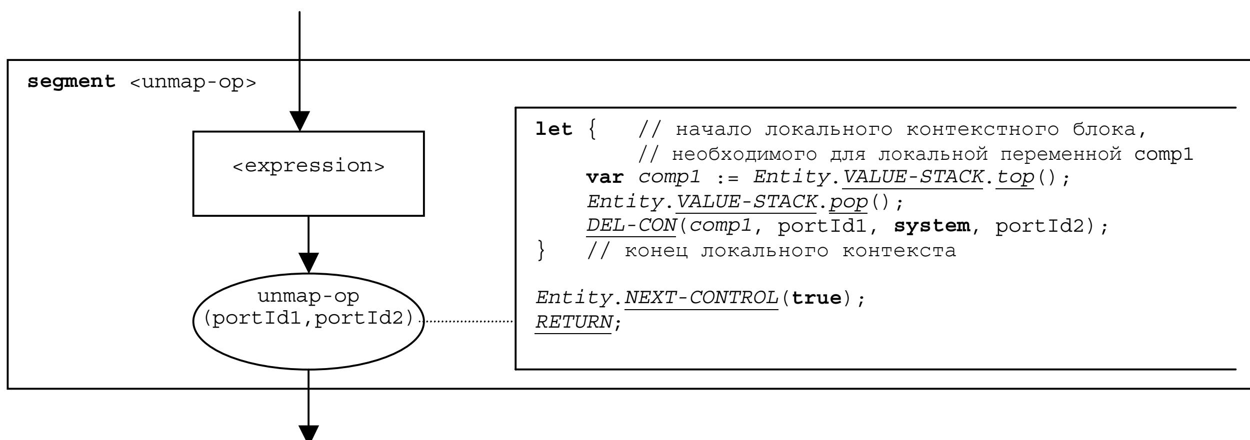


Рисунок 138/Z.143 – Сегмент <unmap-op> потокового графа

9.57 Объявление variable

Синтаксическая структура объявления **variable** (переменная) имеет следующий вид:

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

Инициализация переменной путем предоставления начального значения (в виде выражения) является факультативной. Считается, что начальное значение – это выражение, оценивающее значение типа переменной.

Сегмент <variable-declaration> потокового графа на рисунке 139 определяет выполнение объявления переменной.

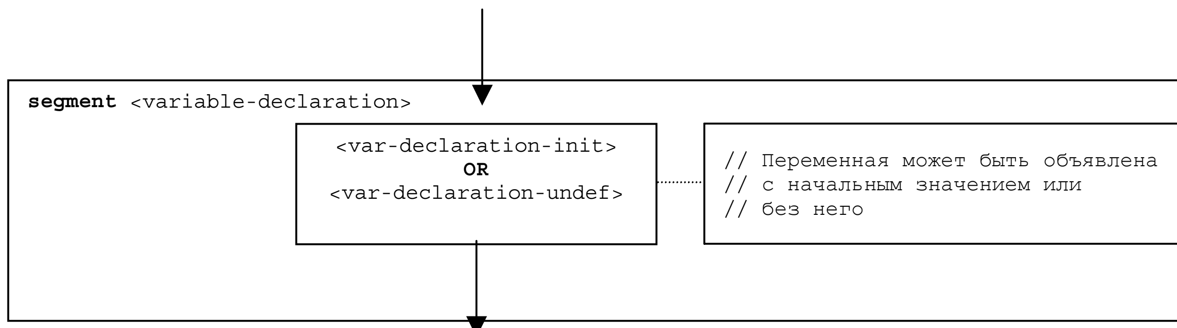


Рисунок 139/Z.143 – Сегмент <variable-declaration> потокового графа

9.57.1 Сегмент <var-declaration-init> потокового графа

Сегмент <var-declaration-init> потокового графа на рисунке 140 определяет выполнение объявления переменной, когда начальное значение предоставляется в виде выражения.

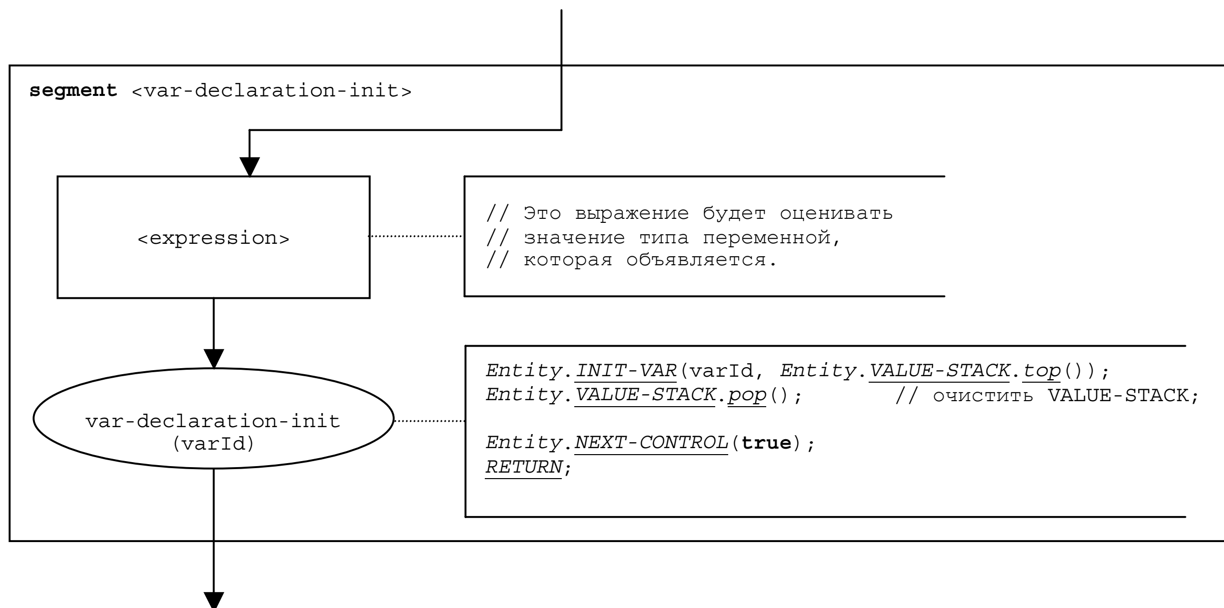


Рисунок 140/Z.143 – Сегмент <var-declaration-init> потокового графа

9.57.2 Сегмент <var-declaration-undef> потокового графа

Сегмент <var-declaration-undef> потокового графа на рисунке 141 определяет выполнение объявления переменной, когда начальное значение не предоставлено, т. е. значение переменной не определено.

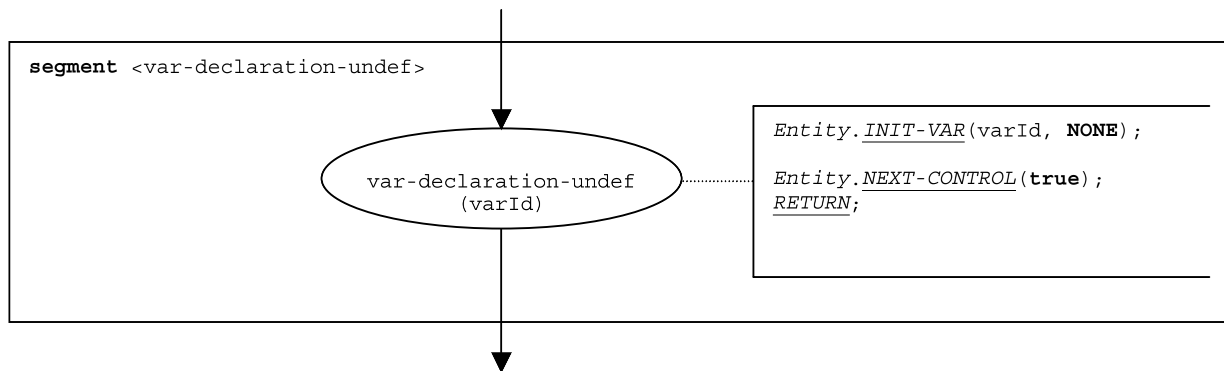


Рисунок 141/Z.143 – Сегмент <var-declaration-undef> потокового графа

9.58 Оператор while

Синтаксическая структура оператора **while** имеет следующий вид:

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

Выполнение оператора **while** определяется сегментом <while-stmt> потокового графа, показанным на рисунке 142.

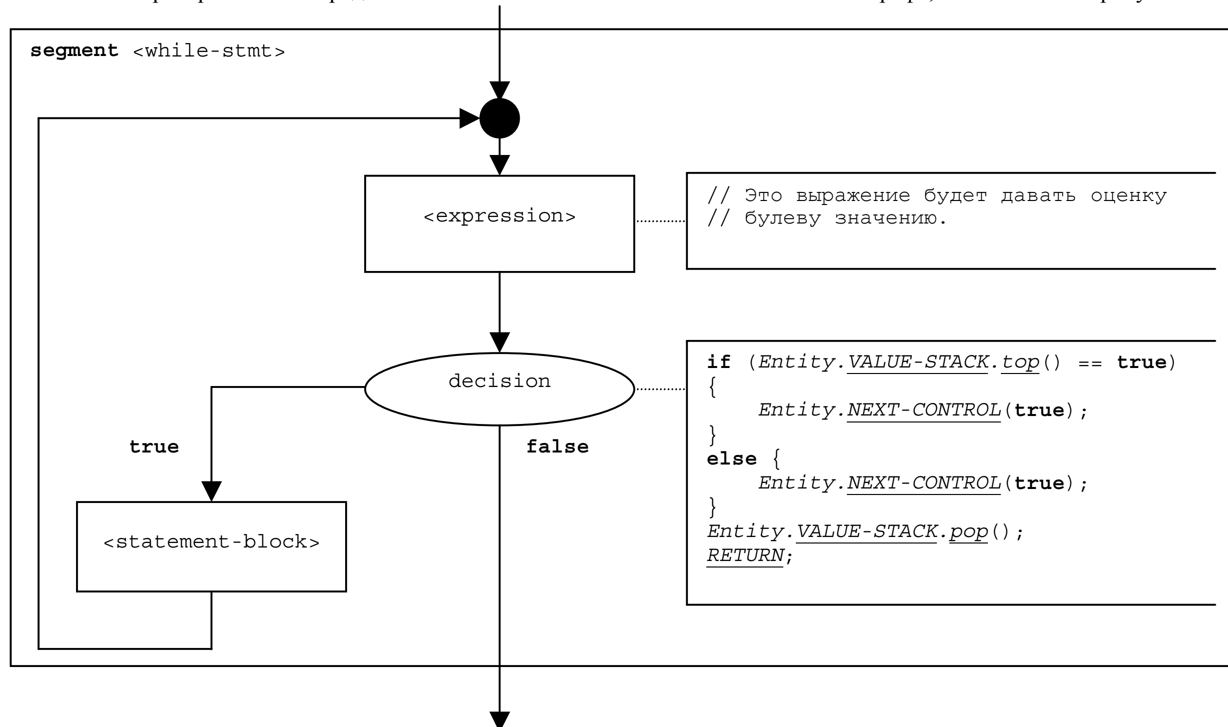


Рисунок 142/Z.143 – Сегмент <while-stmt> потокового графа

10 Списки компонентов операционной семантики

10.1 Функции и состояния

Имя	Описание	Раздел
ACT-DURATION	Период времени, с которым был запущен активный таймер	8.3.2.4
add	Списочная операция: прибавляет элемент в список в качестве первого элемента	8.3.1.1
ADD-CON	Прибавляет соединение к состоянию порта	8.3.3.2
ALL-ENTITY-STATES	Компонентное состояние в состоянии модуля	8.3.1
ALL-PORT-STATES	Состояния порта в состоянии модуля	8.3.1
append	Списочная операция: добавляет элемент к списку в качестве последнего элемента	8.3.1.1
APPLY-OPERATOR	Применение операций, подобных +, – или /	8.6.2
change	Списочная операция: изменяет все элементы списка	8.3.1.1
clear	Стековая операция 'clear': очищает стек	8.3.2.1
clear	Операция 'clear' очереди: удаляет все элементы из очереди	8.3.3.2
clear-until	Стековая операция 'clear-until': выбирает элементы до тех пор, пока конкретный элемент не станет верхним элементом в стеке.	8.3.2.1
CONNECTIONS-LIST	Список соединений порта	8.3.3
CONSTRUCT-ITEM	Конструирует элемент, подлежащий передаче	8.4.4
CONTINUE-COMPONENT	Фактический компонент продолжает свое выполнение	8.6.2
CONTROL-STACK	Стек узлов потокового графа, обозначающий фактическое состояние управления объекта	8.3.2
DATA-STATE	Состояние данных в состоянии объекта	8.3.2

Имя	Описание	Раздел
DEF-DURATION	Период времени по умолчанию для таймера	8.3.2.4
DEFAULT-LIST	Список активных значений по умолчанию в состоянии объекта	8.3.2
DEFAULT-POINTER	Указывает на фактическое значение по умолчанию во время оценки по умолчанию	8.3.2
DEL-CON	Удаляет соединение из состояния порта	8.3.3.2
DEL-ENTITY	Удаляет объект из состояния модуля	8.3.4
DEL-TIMER-SCOPE	Удаляет контекст таймера	8.3.2.5
DEL-VAR-SCOPE	Удаляет контекст переменной	8.3.2.3
delete	Списочная операция: удаляет элемент из списка	8.3.1.1
dequeue	Операция очереди: удаляет первый элемент из очереди	8.3.3.2
DONE	Идентификаторы завершенных тестовых компонентов (часть состояния модуля)	8.3.1
E-VERDICT	Локальный тестовый вердикт тестового компонента	8.3.2
enqueue	Операция очереди: помещает элемент в очередь в качестве последнего элемента	8.3.3.2
first	Операция 'first' очереди: возвращает первый элемент очереди	8.3.3.2
first	Списочная операция: возвращает первый элемент списка	8.3.1.1
GET-FLOW-GRAPH	Выводит начальный узел потокового графа	8.2.7
GET-PORT	Выводит ссылку на порт	8.3.3.2
GET-REMOTE-PORT	Выводит ссылку на удаленный порт	8.3.3.2
GET-TIMER-LOC	Выводит местоположение таймера	8.3.2.5
GET-UNIQUE-ID	Возвращает новый уникальный идентификатор, когда он вызывается	8.6.2
GET-VAR-LOC	Выводит местоположение переменной	8.3.2.3
INIT-CALL-RECORD	Инициализирует переменные для параметров связи, базирующейся на процедурах, в фактическом контекстном блоке тестового компонента	8.5.1
INIT-FLOW-GRAPHS	Инициализирует обработку потокового графа	8.6.2
INIT-TIMER	Создает новое связывание таймера	8.3.2.5
INIT-TIMER-LOC	Создает новое связывание таймера с существующим местоположением	8.3.2.5
INIT-TIMER-SCOPE	Инициализирует новый контекст таймера	8.3.2.5
INIT-VAR	Создает новое связывание переменной	8.3.2.3
INIT-VAR-LOC	Создает новое связывание переменной с существующим местоположением	8.3.2.3
INIT-VAR-SCOPE	Инициализирует новый контекст переменной	8.3.2.3
length	Списочная операция: возвращает длину списка	8.3.1.1
M-CONTROL	Идентификатор управления модулем в состоянии модуля	8.3.1
MATCH-ITEM	Проверяет, сопоставляется ли принимаемое сообщение, вызов, ответ или особое состояние с операцией приема	8.4.5
member	Списочная операция: проверяет, является ли элемент элементом списка	8.3.1.1
MTC	Ссылка на MTC в состоянии модуля	8.3.1
NEW-CALL-RECORD	Создает запись вызова для вызова функции	8.5.1
NEW-ENTITY	Создает новое состояние вызова	8.3.2.1
NEW-PORT	Создает новый порт	8.3.3.2
NEXT	Выводит последующий узел для данного узла в потоковом графе	8.1.6
next	Списочная операция: возвращает следующий элемент в списке	8.3.1.1
NEXT-CONTROL	Извлекает верхний узел потокового графа из управляющего стека и помещает следующий узел потокового графа в управляющий стек	8.3.2.1
OWNER	Владелец порта	8.3.3
pop	Стековая операция 'pop': извлекает элемент из стека	8.3.2.1
PORT-NAME	Имя порта	8.3.3
push	Стековая операция 'push': помещает элемент в стек	8.3.2.1
random	Списочная операция: возвращает случайным образом элемент списка	8.3.1.1
REMOTE-ENTITY	Удаленный объект в соединении в состоянии порта	8.3.3.1

Имя	Описание	Раздел
REMOTE-PORT-NAME	Имя порта в соединении в состоянии порта	8.3.3.1
RETRIEVE-INFO	Выводит информацию из принятого сообщения, вызова, ответа или особого состояния	8.4.6
RETURN	Возвращает управление процедуре оценки модуля	8.6.2
SNAP-ACTIVE	Число активных тестовых компонентов, когда MTC выполняет фиксацию мгновенного состояния процесса (часть состояния модуля)	8.3.1
SNAP-DONE	Список завершенных тестовых компонентов во время, когда проводится фиксация мгновенного состояния процесса	8.3.2
SNAP-PORTS	Предоставляет функциональные возможности фиксации мгновенного состояния процесса, т. е. обновляет SNAP-VALUE	8.3.3.2
SNAP-STATUS	Состояние Snapshot таймера	8.3.2.4
SNAP-TIMER	Предоставляет функциональные возможности фиксации мгновенного состояния процесса и обновляет SNAP-VALUE и SNAP-STATUS	8.3.2.5
SNAP-VALUE	Оценка проведения фиксации мгновенного состояния процесса для таймера	8.3.2.4
SNAP-VALUE	Для семантики фиксации мгновенного состояния процесса, обновляемой при выполнении "снимка"	8.3.3
STATUS	Состояние (**ACTIVE**, **SNAPSHOT**, **REPEAT** или **BLOCKED**) управления модулем или тестового компонента	8.3.2
STATUS	Состояние (**IDLE**, **RUNNING** или **TIMEOUT**) таймера	8.3.2.4
STATUS	Состояние (**STARTED** или **STOPPED**) порта	8.3.3
TC-VERDICT	Вердикт тестового примера в состоянии модуля	8.3.1
TIME-LEFT	Время, оставшееся у работающего таймера до его выхода в состояние тайм-аута	8.3.2.4
TIMER-GUARD	Таймер, защищающий операторы `execute` и операции `call`	8.3.2
TIMER-NAME	Имя таймера	8.3.2.4
TIMER-SET	Установка значений таймера	8.3.2.5
TIMER-STATE	Состояние таймера в состоянии объекта	8.3.2
top	Стековая операция 'top': возвращает верхний элемент из стека	8.3.2.1
UPDATE-REMOTE-REFERENCES	Обновляет таймеры и переменные с одним и тем же местоположением в различных объектах на одно и то же значение	8.3.4
VALUE	Значение переменной	8.3.2.2
VALUE-QUEUE	Очередь порта	8.3.3
VALUE-STACK	Стек значений для хранения результатов выражений, операндов, операций и функций	8.3.2
VAR-NAME	Имя переменной	8.3.2.2
VAR-SET	Установка значения переменной	8.3.2.3
*****DYNAMIC-ERROR*****	Описывает появление динамической ошибки	8.6.2
<identifier>	Уникальный идентификатор тестового компонента	8.3.2
<location>	Поддерживает контекстные блоки, параметры-ссылки и параметры таймера. Представляет местоположение хранения для таймеров и переменных	8.3.2.2, 8.3.2.4

10.2 Специальные ключевые слова

Ключевое слово	Описание	Раздел
ACTIVE	*STATUS* состояния объекта	8.3.2
BLOCKED	*STATUS* состояния объекта	8.3.2
IDLE	*STATUS* состояния таймера	8.3.2.4
MARK	Используется как маркер для *VALUE-STACK*	8.3.2
NONE	Используется для описания неопределенного значения	8.3.2.3, 8.3.2.5, 8.3.3.2

Ключевое слово	Описание	Раздел
NULL	Символическое значение для ссылочных и ссылочно-подобных типов в целях указания на то, что нет объектов для адресации	8.3.1.1, 8.3.2.1, 8.3.3, 8.3.3.2, 8.6.1.1
REPEAT	*STATUS* состояния объекта	8.3.2
RUNNING	*STATUS* состояния таймера	8.3.2.4
SNAPSHOT	*STATUS* состояния объекта	8.3.2
STARTED	*STATUS* порта	8.3.3
STOPPED	*STATUS* порта	8.3.3
TIMEOUT	*STATUS* состояния таймера	8.3.2.4

10.3 Потоковые графы описаний поведения TTCN-3

	Ссылка	
	Рисунок	Раздел
Управление модулем	18	8.2.2
Тестовые примеры	19	8.2.3
Функции	20	8.2.4
Альтернативные шаги	21	8.2.5
Определения компонентного типа	22	8.2.6

10.4 Сегменты потокового графа

Идентификатор	Соответствующая конструкция TTCN-3	Ссылка	
		Рисунок	Раздел
<action-stmt>	оператор **action**	36	9.1
<activate-stmt>	оператор **activate**	37	9.2
<alt-stmt>	оператор **alt**	38	9.3
<altstep-call>	инициирование альтернативного шага	44	9.4
<altstep-call-branch>	оператор **alt**	41	9.3.3
<assignment-stmt>	присвоение	45	9.5
<b-call-with-duration>	операция **call**	52	9.6.4
<b-call-without-duration>	операция **call**	51	9.6.3
<blocking-call-op>	операция **call**	47	9.6
<call-op>	операция **call**	46	9.6
<call-reception-part>	операция **call**	53	9.6.5
<catch-op>	операция **catch**	55	9.7
<catch-timeout-exception>	операция **call**	54	9.6.6
<check-op>	операция **check**	56	9.8
<check-with-sender>	операция **check**	57	9.8.1
<check-without-sender>	операция **check**	58	9.8.2
<clear-port-op>	операция **clear** порта	59	9.9
<connect-op>	операция **connect**	60	9.10
<constant-definition>	определение **constant**	61	9.11
<create-op>	операция **create**	62	9.12
<deactivate-stmt>	оператор **deactivate**	63	9.13
<default-evocation>	оператор **alt**	43	9.3.5
<disconnect-op>	операция **disconnect**	64	9.14
<do-while-stmt>	оператор **do-while**	65	9.15
<done-component-op>	компонентная операция **done**	66	9.16

Идентификатор	Соответствующая конструкция TTCN-3	Ссылка	
		Рисунок	Раздел
<else-branch>	оператор **alt**	42	9.3.4
<execute-stmt>	оператор **execute**	67	9.17
<execute-timeout>	оператор **execute**	69	9.17.2
<execute-without-timeout>	оператор **execute**	68	9.17.1
<expression>	выражение	70	9.18
<finalize-component-init>	используется в определениях компонентного типа	75	9.19
<for-stmt>	оператор **for**	79	9.23
<func-op-call>	выражение	73	9.18.3
<function-call>	вызов функции	80	9.24
<getcall-op>	операция **getcall**	86	9.25
<getreply-op>	операция **getreply**	87	9.26
<getverdict-op>	операция **getverdict**	88	9.27
<goto-stmt>	оператор **goto**	89	9.28
<if-else-stmt>	оператор **if-else**	90	9.29
<init-component-scope>	используется в определениях компонентного типа	76	9.20
<label-stmt>	оператор **label**	91	9.30
<lit-value>	выражение	71	9.18.1
<log-stmt>	оператор **log**	92	9.31
<map-op>	операция **map**	93	9.32
<mtc-op>	операция **mtc**	94	9.33
<nb-call-without-receiver>	операция **call**	50	9.6.2
<nb-call-with-receiver>	операция **call**	49	9.6.1
<non-blocking-call-op>	операция **call**	48	9.6
<operator-appl>	выражение	74	9.18.4
<parameter-handling>	обработка параметров функций, альтернативных шагов и тестовых примеров	77	9.21
<port-declaration>	объявление порта	95	9.34
<predef-ext-func-call>	вызов функции (вызов предопределенной или внешней функции)	85	9.24.5
<raise-op>	операция **raise**	96	9.35
<raise-with-receiver-op>	операция **raise**	97	9.35.1
<raise-without-receiver-op>	операция **raise**	98	9.35.2
<read-timer-op>	операция **read** таймера	99	9.36
<receive-assignment>	операция **receive**	103	9.37.3
<receive-op>	операция **receive**	100	9.37
<receive-with-sender>	операция **receive**	101	9.37.1
<receive-without-sender>	операция **receive**	102	9.37.2
<receiving-branch>	оператор **alt**	40	9.3.2
<ref-par-var-calc>	вызов функции (обработка параметров-ссылок)	82	9.24.2
<ref-par-timer-calc>	вызов функции (обработка параметров таймера)	83	9.24.3
<repeat-stmt>	оператор **repeat**	104	9.38
<reply-op>	операция **reply**	105	9.39
<reply-with-receiver-op>	операция **reply**	106	9.39.1
<reply-without-receiver-op>	операция **reply**	107	9.39.2
<return-stmt>	оператор **return**	108	9.40

Идентификатор	Соответствующая конструкция TTCN-3	Ссылка	
		Рисунок	Раздел
<return-with-value>	оператор **return**	109	9.40.1
<return-without-value>	оператор **return**	110	9.40.2
<running-component-op>	компонентная операция **running**	111	9.41
<running-comp-act>	компонентная операция **running**	112	9.41.1
<running-comp-snap>	компонентная операция **running**	113	9.41.2
<running-timer-op>	операция **running** таймера	114	9.42
<self-op>	операция **self**	115	9.43
<send-op>	операция **send**	116	9.44
<send-with-receiver-op>	операция **send**	117	9.44.1
<send-without-receiver-op>	операция **send**	118	9.44.2
<setverdict-op>	операция **setverdict**	119	9.45
<start-component-op>	компонентная операция **start**	120	9.46
<start-port-op>	операция **start** порта	121	9.47
<start-timer-op>	операция **start** таймера	122	9.48
<start-timer-op-default>	операция **start** таймера	123	9.48.1
<start-timer-op-duration>	операция **start** таймера	124	9.48.2
<statement-block>	блок операторов в составных операторах	78	9.22
<stop-component-op>	компонентная операция **stop**	125	9.49
<stop-mtc>	компонентная операция **stop** (останов MTC)	126	9.49.1
<stop-component>	компонентная операция **stop** (останов одиночного тестового компонента)	127	9.49.2
<stop-all-comp>	компонентная операция **stop** (останов всех компонентов)	128	9.49.3
<stop-exec-stmt>	оператор **stop** выполнения	129	9.50
<stop-control>	оператор **stop** выполнения (останов управления модулем)	130	9.50.1
<stop-port-op>	операция **stop** порта	131	9.51
<stop-timer-op>	операция **stop** таймера	132	9.52
<system-op>	операция **system**	133	9.53
<take-snapshot>	оператор **alt**	39	9.3.1
<timeout-timer-op>	операция **timeout**	137	9.55
<timer-declaration>	объявление таймера	134	9.54
<timer-decl-default>	объявление таймера	135	9.54.1
<timer-decl-no-def>	объявление таймера	136	9.54.2
<timeout-timer-op>	операция **timeout**	137	9.55
<unmap-op>	операция **unmap**	138	9.56
<user-def-func-call>	вызов функции (вызов функции, определяемой пользователем)	84	9.24.4
<value-par-calculation>	вызов функции (обработка параметров-значений)	81	9.24.1
<var-declaration-init>	объявление переменной	140	9.57.1
<var-declaration-undef>	объявление переменной	141	9.57.2
<var-value>	выражение	72	9.18.2
<variable-declaration>	объявление переменной	139	9.57
<while-stmt>	оператор **while**	140	9.58

СЕРИИ РЕКОМЕНДАЦИЙ МСЭ-Т

Серия A	Организация работы МСЭ-Т
Серия D	Общие принципы тарификации
Серия E	Общая эксплуатация сети, телефонная служба, функционирование служб и человеческие факторы
Серия F	Нетелефонные службы электросвязи
Серия G	Системы и среда передачи, цифровые системы и сети
Серия H	Аудиовизуальные и мультимедийные системы
Серия I	Цифровая сеть с интеграцией служб
Серия J	Кабельные сети и передача сигналов телевизионных и звуковых программ и других мультимедийных сигналов
Серия K	Защита от помех
Серия L	Конструкция, прокладка и защита кабелей и других элементов линейно-кабельных сооружений
Серия M	Управление электросвязью, включая СУЭ и техническое обслуживание сетей
Серия N	Техническое обслуживание: международные каналы передачи звуковых и телевизионных программ
Серия O	Требования к измерительной аппаратуре
Серия P	Качество телефонной передачи, телефонные установки, сети местных линий
Серия Q	Коммутация и сигнализация
Серия R	Телеграфная передача
Серия S	Оконечное оборудование для телеграфных служб
Серия T	Оконечное оборудование для телематических служб
Серия U	Телеграфная коммутация
Серия V	Передача данных по телефонной сети
Серия X	Сети передачи данных, взаимосвязь открытых систем и безопасность
Серия Y	Глобальная информационная инфраструктура, аспекты межсетевого протокола и сети последующих поколений
Серия Z	**Языки и общие аспекты программного обеспечения для систем электросвязи**

Отпечатано в Швейцарии
Женева, 2006 г.