МЕЖДУНАРОДНЫЙ СОЮЗ ЭЛЕКТРОСВЯЗИ

МСЭ-Т Z.130

СЕКТОР СТАНДАРТИЗАЦИИ
ЭЛЕКТРОСВЯЗИ МСЭ

(07/2003)

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

Методы формального описания (FDT) – Расширенный язык описания объектов (eODL)

Расширенный язык описания объектов (eODL): Методика разработки распределенных программных компонентов – концептуальная основа, нотации и техника отображений

Рекомендация МСЭ-Т Z.130

РЕКОМЕНДАЦИИ МСЭ-Т СЕРИИ Z

ЯЗЫКИ И ОБЩИЕ АСПЕКТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ СИСТЕМ ЭЛЕКТРОСВЯЗИ

МЕТОДЫ ФОРМАЛЬНОГО ОПИСАНИЯ (FDT)	
Язык спецификации и описания (ЯСО)	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-компьютер" для руководства сетями электросвязи	Z.360–Z.369
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	
Окружающая среда распределенной обработки	Z.600–Z.609

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

Рекомендация МСЭ-Т Z.130

Расширенный язык определения объектов (eODL): Методика разработки распределенных программных компонентов – концептуальная основа, нотации и техника отображения

Резюме

Настоящая Рекомендация предназначена для тех, кто проектирует, обеспечивает реализацию и управление распределенными системами, а также для разработчиков инструментальных средств поддержки распределенных систем.

В данной Рекомендации подробно описывается Расширенный язык определения объектов МСЭ (eODL-МСЭ). Язык eODL-МСЭ применяется для компонентно-ориентированной разработки распределенных систем с четырех различных, но связанных между собой представлений: вычисления, реализации, внедрения и целевой окружающей среды. Каждое представление связано с определенной целью моделирования, выраженной посредством специализированных абстрактных концепций. Типы вычислительных объектов с (операционными, потоковыми, сигнальными) интерфейсами и портами являются основными концепциями представления вычисления, абстрактно описывающими распределенные программные компоненты в терминах своих потенциальных интерфейсов. Артефакты как абстракции контекстов конкретных языков программирования и их отношений к интерфейсам формируют представление реализации. Представление внедрения описывает программные объекты (программные компоненты) в двоичном виде и реализованные ими сущности вычисления. Представление целевой окружающей среды обеспечивает концепции моделирования физической сети, в которой следует осуществлять внедрение программных компонентов. Все концепции этих представлений связаны друг с другом. Эти связи составляют неотъемлемую основу методики и инструментальных средств, поддерживающих процесс разработки программного обеспечения, начиная от проектирования и до внедрения, через реализацию и интеграцию. Фаза тестирования данной Рекомендацией пока еще не рассматривается.

Язык eODL-МСЭ является расширением языка определения объектов МСЭ ODL-МСЭ [1] и заменяет последний. Первоначально язык ODL-МСЭ был разработан как расширение языка ODP-IDL [9] и определял концепции вычисления, основанные на терминологии ODP [2], [3]. Язык eODL следует этому же принципу. Однако в основу определений положен метамодельный подход, а не подход традиционного абстрактного синтаксиса. Одним из преимуществ метамодельного подхода является возможность использования инструментальных средств, имеющих отношение к MOF [4], для поддержки автоматизации переходов модели между различными фазами разработки программного обеспечения. Другим преимуществом является возможность создания из метамодели экземпляра конкретной модели, которая может представляться существующими языками таким образом, что обеспечивается интеграция различных подходов к проектированию.

Предполагается, что читатели данной Рекомендации знакомы с IDL [5], UML [11], MOF.

Определение языка eODL поддерживается следующими приложениями и дополнениями:

- Приложение A вводит текстовый синтаксис для eODL, предназначенный для применения при представлении спецификаций eODL. При определении синтаксиса используется стиль EBNF.
- Приложение B определяет отображение между метамоделью eODL и текстовым синтаксисом, определенным в Приложении A.
- Приложение C обеспечивает отображение из eODL в ЯСО-2000 МСЭ, что позволяет обеспечить автоматическое преобразование модели eODL в модель ЯСО-2000.
- Приложение D содержит программную ссылку на представление [12] XML метамодели eODL в соответствии с метаформатом обмена (XMI) [6] XML. Она дается в отдельном файле для того, чтобы позволить импортирование и обработку метамодели eODL средствами UML.
- В Параграфе 1 приводится обзор того, как eODL используется теми, кто проектирует, обеспечивает реализацию и управление распределенной системой. Конкретный пример применения приводится в Дополнении I.
- В Дополнении II описывается весь процесс разработки при использовании eODL и возможная инструментальная поддержка.

Источник

Рекомендация МСЭ-Т Z.130 одобрена 22 июля 2003 г. Исследовательской комиссией 17 МСЭ-Т (2001-2004) в соответствии с процедурой 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, а также за выпуск Рекомендаций по ним с целью стандартизации электросвязи на всемирной основе.

Всемирная ассамблея по стандартизации электросвязи (ВАСЭ), которая проводится каждые четыре года, определяет темы для изучения Исследовательскими комиссиями МСЭ-Т, которые, в свою очередь, 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 данной Рекомендации является добровольным. Однако Рекомендация может содержать определенные обязательные положения (например, для обеспечения возможности взаимодействия или применимости) и соответствие этой Рекомендации достигается лишь при условии, когда все эти обязательные положения учтены. Слова "следовать" или некоторые другие обязывающие слова, как "должен" и их негативные эквиваленты используются для выражения требований. Применение таких слов не подразумевает, что от любой стороны требуется соответствие с данной Рекомендацией.

ПРАВА ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

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

СОДЕРЖАНИЕ

Рекомендация МСЭ-Т Z.130

Расширенный язык описания объектов (eODL): Методика разработки распределенных программных компонентов – концептуальная основа, нотации и техника отображения

1 Область определения

Обеспечение эффективной методикой и инструментальной поддержкой при разработке и конструировании распределенных систем является ключевым фактором, дающим возможность дальнейшего развития информационной технологии. Системы электросвязи являются особыми распределенными системами, состоящими из компонентов, которые распределены по сетям и должны иметь дело с аспектами параллелизма, автономности, синхронности и связи. Разработка высокоэффективных и масштабируемых систем является комплексной и трудной задачей, где инструментальные средства должны обеспечить поддержку всех фаз процесса разработки – от сбора требований до интеграции, тестирования и внедрения через проектирование и реализацию.

Генерация машинного кода объектно-ориентированных моделей проекта приводит к многократно-используемым, исполняемым компонентам. Такие компоненты интегрируют аспекты, зависимые от рабочего цикла окружающей среды и технологии платформы промежуточного программного обеспечения, с объектно-ориентированной моделью проекта, характерной для предприятия. Каждый программный компонент имеет физическое представление (например, двоичный файл), которое должно быть доступно для исполнения на специальном узле распределенной системы. Основное направление данной Рекомендации – проектирование таких компонентов.

Методика разработки распределенных систем в значительной степени влияет на сокращение времени до выхода на рынок распределенных приложений и услуг электросвязи. Подходящая трактовка всех видов коммуникационных аспектов находится в самой природе целевой прикладной области. Эти аспекты охватывают от транзакционных требований к объекту взаимодействия до стратегий безопасности, включая вопросы качества обслуживания. Принимая во внимание широкое признание техники объектного промежуточного программного обеспечения, его платформы обеспечивают идеальную окружающую среду для реализации таких проектов. Среди них: обычная архитектура CORBA [5], компоненты CORBA [7] и другие распределенные платформы обработки.

Данная Рекомендация направлена на все процессы разработки программного обеспечения, рассматривая следующие фазы жизненного цикла программного обеспечения:

- фаза проектирования;
- фаза реализации;
- фаза интеграции;
- операционная фаза.

В данной Рекомендации не рассматриваются требования фазы сбора требований.

Особый акцент данной Рекомендации делается на технической поддержке переходов между фазами для достижения их автоматизации. Определяющей техникой является подход, управляемый моделями, основанный на вполне определенной метамодели (см. [11]). Эта метамодель позволяет интеграцию нескольких существующих языков проектирования, таких как ЯСО [8], UML и CORBA-IDL. Метамодель представляет собой определение концепций для рассматриваемых фаз жизненного цикла программного обеспечения. Модели, конкретизируемые на основе метамодели, могут быть представлены с использованием существующих языков. Так как некоторые концепции не охватываются каким-либо существующим языком, в данной Рекомендации также определяется конкретный синтаксис: eODL (Расширенный ODL). При определении языков подход, основанный на метамодели, заменяет хорошо известный подход абстрактного синтаксиса. Язык eODL-МСЭ – это пересмотренный язык ODL-МСЭ. Определение синтаксиса приводится в Приложении А.

Поэтому метамодель является независимой от характерных нотаций проекта. Модели проекта могут разрабатываться с применением различных нотаций, но в их основе лежат те же самые концепции. Проектная информация может обмениваться на базе метамодели. Как нотация, так и метамодель являются независимыми от индивидуального рабочего цикла окружающей среды. Один и тот же

проект может отображаться на различные целевые окружающие среды. Это дает высокую гибкость и также является важным для аспекта переиспользования компонентных моделей проекта.

Интеграция данной Рекомендации в процессы разработки программного обеспечения показывается на рис. 1. Начиная с четкого определения требований, устанавливается модель проекта. Данная Рекомендация определяет концепции, которые активизируют различные представления модели проекта. Каждое представление охватывает различные аспекты разрабатываемой системы. Концепции метамодели позволяют разработку моделей, достаточно мощных, чтобы автоматически получить скелетные схемы программных компонентов. Эти скелетные схемы формируют начальную точку фазы реализации, где они должны заканчиваться бизнес-логикой. В фазе интеграции законченные программные компоненты должны интегрироваться в целевую окружающую среду.

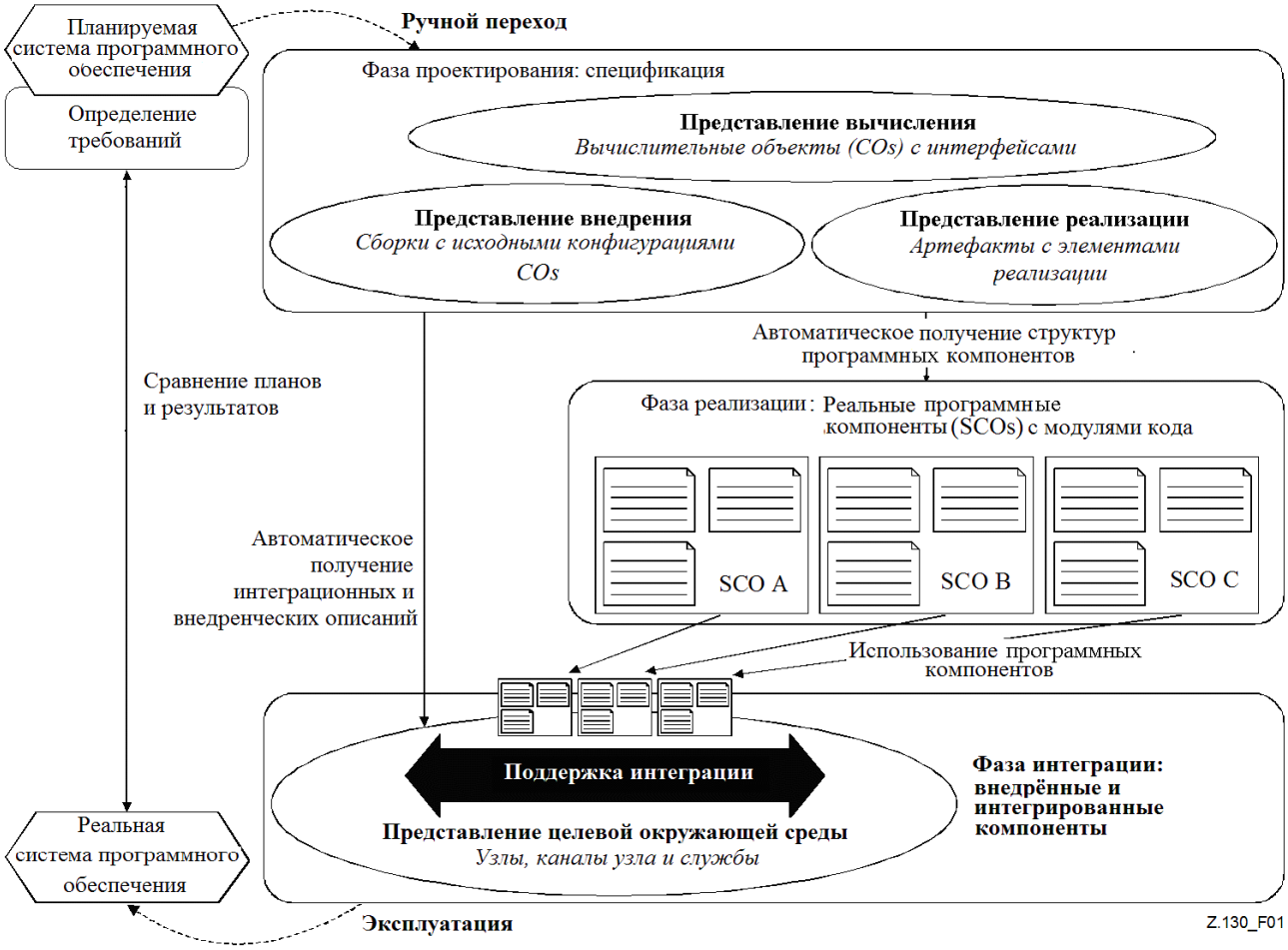


Рисунок 1/Z.130 – Разработка программного обеспечения

Данная Рекомендация также содержит концепции, позволяющие описать топологию и свойства целевой окружающей среды. Описание целевой окружающей среды совместно с автоматически генерируемым внедрением и описаниями интеграции, разработанными на основе фазы проектирования, способствует автоматизации внедрения. После фазы интеграции разработанная система программного обеспечения может быть введена в эксплуатацию.

2 Библиографические ссылки

Следующие Рекомендации МСЭ-Т и другие ссылки содержат положения, которые посредством ссылки в этом тексте составляют положения данной Рекомендации. На время публикации указанные издания являлись действующими. Все рекомендации и другие ссылки подлежат пересмотру; поэтому пользователям данной Рекомендации предлагается изучить возможность применения самого последнего издания этих рекомендаций и других ссылок, перечисленных ниже. Список Рекомендаций МСЭ-Т, действующих в настоящее время, публикуется регулярно. Ссылка на какой-либо документ в пределах данной Рекомендации не дает ему, как автономному документу, статус рекомендации.

[1] Рекомендация МСЭ-Т Z.130 (1999 г.): Язык МСЭ определения объекта.

[2] Рекомендация МСЭ-Т X.902 (1995 г.) | МОС/МЭК 10746-2 (1996 г.): Информационные технологии – Открытая распределенная обработка – Эталонная модель: основы.

[3] Рекомендация МСЭ-Т X.903 (1995 г.) | МОС/МЭК 10746-3 (1996 г.): Информационные технологии – Открытая распределенная обработка – Эталонная модель: архитектура.

[4] Формальный документ/00-04-03 OMG: Спецификация средства метаобъекта (MOF), Версия 1.3.

[5] Формальный документ/01-02-33 OMG: Общая архитектура брокера объектных заявок и спецификация, Редакция 2.4.2.

[6] Формальный документ/00-11-02 OMG: Спецификация обмена метаданными XML, Версия 1.1.

[7] Формальный документ/02-06-65 OMG: Компонент CORBA, Версия 3.0.

[8] Рекомендация МСЭ-Т Z.100 (2002 г.): Язык спецификации и описания (ЯСО).

[9] Рекомендация МСЭ-Т X.920 (1997 г.) | МОС/МЭК 14750 (1999 г.): Информационные технологии – Открытая распределенная обработка – Язык определения интерфейса.

[10] Рекомендация МСЭ-Т Z.600 (2000 г.): Архитектура окружающей среды распределенной обработки.

3 Сокращения

В данной Рекомендации используются следующие сокращения:

API	Интерфейс прикладного программирования
ASN.1	Система абстрактных синтаксических нотаций Один
CCM	Компонентная модель CORBA
CO	Вычислительный объект
COM	Модель компонентных объектов
CORBA	Общая архитектура брокера объектных заявок
CWM	Общая метамодель хранилища данных
DTD	Определение типа документа в языке XML
EBNF	Расширенная нормальная форма Бэкуса-Наура
EJB	Предприятие JavaBeans™
IDL	Язык определения интерфейса
MDA	Архитектура, управляемая моделями
MOF	Средство метаобъекта
OCL	Язык ограничения объектов

ODL	Язык описания объектов
OMG	Группа управления объектами
OSD	Описание открытого программного обеспечения
RM-ODP	Эталонная модель открытой распределенной обработки
ЯСО	Язык спецификации и описания
TINA	Информационная сетевая архитектура электросвязи
UML	Унифицированный язык моделирования
XMI	Обмен метаданными XML
XML	Язык XML, расширяемый язык разметки

4 Определения

В данной Рекомендации даются определения следующих терминов:

4.1 артефакт: абстракция конкретных конструкций языка программирования, например, класс в объектно-ориентированных языках (представленный в виде кодовых модулей), вложенный программным компонентом. Экземпляр **артефакта представляет собой** (в смысле модели) состояние, поведение и идентичность **СО**.

4.2 сборка: описание системы распределенного программного обеспечения множеством всех принимающих участие **типов СО** и **исходной конфигурации** (используется CCM).

4.3 исключение: специальная разновидность операции **окончания** в случае возникновения ошибок (определяется RM-ODP).

4.4 класс: концепция классификации объектов. На основе описания класса объекты могут быть конкретизированы (определяется MOF.)

4.5 соединение: концепция обмена **ссылками интерфейса**, которые относятся к определениям *порта* в соответствии с их специальным видом определения (отношение *использование* или *предоставление*). (Определяется RM-ODP в качестве вычислительного связывания; используется CCM).

4.6 архитектура компонентов: **окружающая среда распределенной обработки**, поддерживающая взаимодействие распределенных **программных компонентов**.

4.7 целевая платформа: **архитектура компонентов** с поддержкой **внедрения** и распределенного исполнения, где компоненты предполагаются для внедрения.

4.8 вычислительный объект (СО): функциональная единица, которая является результатом функциональной декомпозиции моделируемой системы программного обеспечения (определяется RM-ODP).

4.9 тип вычислительного объекта (тип СО): шаблон для конкретизации **вычислительных объектов** (определяется RM-ODP; идентифицируется CCM в качестве компонента CORBA).

4.10 поглощать: концепция моделирования потенциального получения сигнала (используется CCM).

4.11 взаимодействие типа непрерывного носителя, **сигнальное** и **операционное**: **взаимодействие** между объектами **СО** посредством **элементов взаимодействия** операционного, сигнального или *типа непрерывного носителя*: **операция**, **атрибут**, **поглощать**, **порождать**, **сток**, **исток** (определяется RM-ODP и TINA (взаимодействие операционное и *типа непрерывных носителей*).)

4.12 тип данных: предписание допустимой структуры, содержимого и поведения данных; это является элементом модели типа данных (т. е. CORBA-IDL), которая представляет собой базу информационных моделей (определяется CORBA).

4.13 окружающая среда распределенной обработки: техническая база, которая поддерживает взаимодействия между объектами распределенной системы (определяется TINA).

4.14 внедрение: процесс создания физических представлений **программных компонентов**, доступных в **узлах**, их установка таким образом, что они готовы для исполнения и установки **исходной конфигурации**.

4.15 элемент реализации: отношение между **элементом взаимодействия** и **артефактом**, где экземпляр **артефакта** отвечает за поведение **элемента взаимодействия**.

4.16 реализовать: отношение между **артефактами** и **типами CO**, где экземпляры **артефакта представляют собой** поведение **типа CO**.

4.17 исходный CO: объект **CO**, который создается в начале **рабочего цикла** системы распределенного программного обеспечения.

4.18 исходная конфигурация: множество **исходных объектов CO** и **исходных соединений** (используется моделью CCM).

4.19 исходное соединение: связывание, которое устанавливается исходно в начале **рабочего цикла** системы распределенного программного обеспечения (используется моделью CCM).

4.20 стратегия конкретизации: основанное на стратегии описание конкретизации различных концепций реализации, моделируемых **артефактами**.

4.21 взаимодействие: действие, в котором затрагивается окружающая среда объекта (определяется моделью RM-ODP).

4.22 элемент взаимодействия: обобщение концепций **операция**, **атрибут**, **сток**, **исток**, **поглощать** и **порождать**.

4.23 интерфейс: ссылочное агрегирование возможных взаимодействий **CO** (определяется моделью RM-ODP).

4.24 атрибут интерфейса: специальная разновидность операций, например, стенографические знаки для операций *получить* и *установить* для заданного **типа данных** (используется архитектурой CORBA).

4.25 ссылка интерфейса: ссылка на **интерфейс** (соответствует ссылке на объект CORBA).

4.26 тип интерфейса: описание множества **элементов взаимодействия**, например, поименованных и идентифицируемых конечных точек возможной связи (определяется моделью RM-ODP; соответствует интерфейсу OMG IDL).

4.27 множество носителей: агрегирование носителей.

4.28 тип носителя: объявление, используемое при кодировании, передаче и декодировании данных на **носителе**.

4.29 носитель: элементарный однонаправленный непрерывный поток данных (заменяет нотацию потока в Рекомендации МСЭ-Т Z.600 [10].)

4.30 метамодель: определение концепций моделирования для конструкций моделей определенного домена (определяется MOF).

4.31 мета-метамодель: определение концепций моделирования для конструкций **метамоделей** (определяется MOF).

4.32 порт множественный: **порт**, который динамически поддерживает регистрацию и извлечение множественных **ссылок интерфейса** (используется CCM).

4.33 пространство имен: концепция структурирования идентификаторов элементов модели (определяется RM-ODP).

4.34 узел: устройство, которое используется для интерпретации модулей кода программного компонента (определяется RM-ODP).

4.35 объект: модель сущности, где сущность - это любое явление в рассматриваемой области; объект обладает идентичностью, состоянием и поведением (определяется RM-ODP).

4.36 операция: элемент операционного взаимодействия, описанный множеством параметров и возможных окончаний (определяется моделью RM-ODP, архитектурой CORBA.)

4.37 **параметр**: элемент вызова операции, описанный направлением обмена информацией и **типом данных** (определяется CORBA.)

4.38 **порт**: сущность, предназначенная для регистрации и извлечения **ссылок интерфейса CO** (используется CCM).

4.39 **порождать**: **элемент взаимодействия** для отправления **сигналов** (используется CCM).

4.40 **порт предоставленный**: **порт**, где могут извлекаться **ссылки интерфейса** соответствующего **CO** (используется CCM).

4.41 **представлять**: соответствие **программных компонентов типам CO** (определяется UML).

4.42 **требовать**: отношение **типа интерфейса** и **типа CO**, при котором объекты **COs** этого **типа CO** требуют **ссылки интерфейса типа интерфейса** из окружающей среды **CO** (определяется TINA.)

4.43 **рабочий цикл**: время, в течение которого исполняется **программный компонент**.

4.44 **сигнал**: **элемент взаимодействия** для асинхронного обмена элементарными сообщениями (определяется RM-ODP).

4.45 **порт простой: порт**, где может регистрироваться и извлекаться простая **ссылка интерфейса** (используется CCM).

4.46 **сток**: **элемент взаимодействия** для получения **множества носителей** (определяется TINA).

4.47 **программный компонент**: сущность, состоящая из последовательности инструкций (модулей кода), представленных физически (в виде специального формата данных), которая может быть собрана в структурированные программные компоненты или в систему программного обеспечения; для обеспечения композиции **программных компонентов**, их функциональность обеспечивается посредством вполне определенных **типов интерфейсов**.

В процессе исполнения **программных компонентов** объекты претворяются в виде экземпляров классов (представленных в виде модулей кода); (определяется в [16]).

4.48 **программный пакет:** пакет **программных компонентов** (используется CCM).

4.49 **исток: элемент взаимодействия** для отправки **множества носителей** (определяется TINA).

4.50 **поддерживать:** отношение **типа интерфейса** и **типа CO,** при котором **COs** этого **типа CO** поддерживают **ссылки интерфейса типа интерфейса** из окружающей среды **CO** (определяется TINA).

4.51 **окончание:** конец вызванной **операции** (определяется RM-ODP).

4.52 **параметр сигнала:** составная часть **сигнала** для переноса информации; относится к **типу данных** (определяется ЯСО).

4.53 **порт используемый: порт**, где могут регистрироваться **ссылки интерфейса** (используется CCM).

5 Метамодель

Метамодель определяет концепции моделирования для построения моделей какой-либо определенной области. В данной Рекомендации **метамодель** представляет собой средство метаобъекта (MOF), удовлетворяющее **метамодели**. **Метамодель** описывается посредством диаграмм класса UML. Семантика представлена естественным языком. При необходимости учитываются правила формальной корректности. Средство MOF является стандартом, принятым в OMG (группа управления объектами) при метамоделировании. Средство MOF определяет общую структуру для описания и представления метаинформации.

Средство MOF определяет четырехуровневую архитектуру, изображенную на рис. 2:

- На уровне M3 мы находим простую **мета-метамодель** (модель MOF), которая определяет основные концепции, необходимые для описания любой метамодели объектно-ориентированным способом. Основными конструкциями являются: класс, ассоциация, тип данных, атрибут класса и наследование класса.

- На уровне M2 мы находим **метамодели** (языки). **Метамодель** обеспечивает абстракции, которые необходимы для построения моделей. Она описывается в терминах базовых конструкций M3 (на практике абстрактный синтаксис **метамодели** обеспечивается совокупностью диаграмм класса). Примерами **метамоделей** являются UML, CWM (хранилище данных) и **метамодель** CCM.
- На уровне M1 мы находим *модели*. Они описываются в терминах одной из **метамоделей**, определенных на уровне M2. Примером модели является модель сетевого уровня на UML, которая определяет след.
- На уровне M0 мы находим данные, которые представляют собой экземпляры модели на уровне M1. Примером данных является перечень записей, представляющих следы.

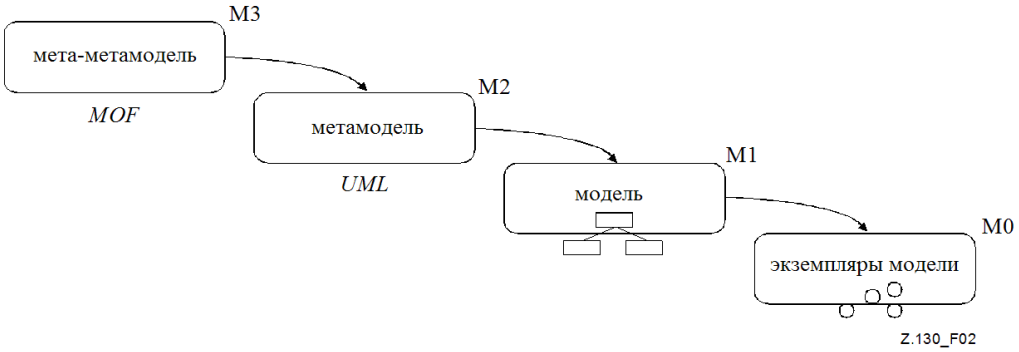


Рисунок 2/Z.130 – Уровни MOF

В дополнение к конструкциям базового языка для объектно-ориентированного описания **метамоделей** MOF стандартизирует **типы интерфейсов** (интерфейсы OMG IDL), которые могут использоваться для работы на объектах модели. Кроме того, соответствующий стандарт XMI [6] стандартизирует способ, которым модель может представляться в формате потока XML [12]. Словарь XML, используемый для представления, зависит только от объектов **метамодели**.

5.1 Определения и соглашения

В данном параграфе определяются концепции модели MOF (**мета-метамодели**), которые используются для определения концепций eODL (**метамодели**). На рис. 3 приведена нотация, которая используется для обеспечения обозримости атрибутов и операций. Она используется единообразно на всех рисунках UML. Дополнительная информация приведена [4].

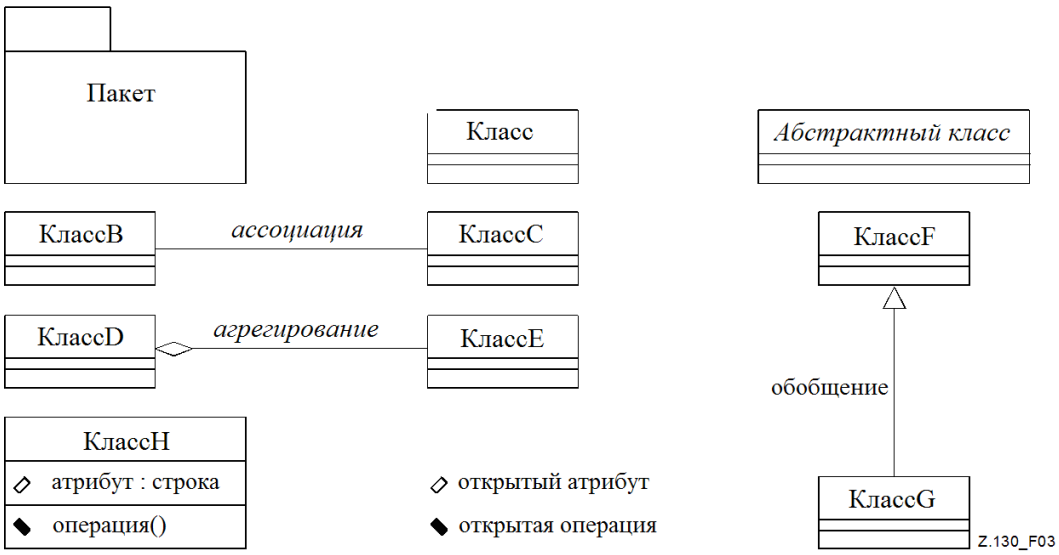


Рисунок 3/Z.130 – Нотация UML концепций MOF

5.1.1 ***класс и объект***: класс – это описание множества объектов, имеющих те же самые индивидуальные характеристики класса. Класс используется при классификации и служит базовой концепцией для конструкции. Объект – это экземпляр некого класса.

5.1.2 ***обобщение***: обобщение является однонаправленной связью между двумя классами. Обобщение объединяет специальный и общий класс. Специальный класс наследует все характеристики общего класса.

5.1.3 ***ассоциация***: ассоциация – это отношение между двумя классами. В случае наличия экземпляров двух классов, между ними может или не может конкретизироваться объединение. Объединение может перемещаться от одного участвующего объекта к другому.

5.1.4 ***агрегирование***: Агрегирование представляет собой направленную взаимосвязь между классами, при которой полагается, что экземпляры агрегированных классов (части) содержатся в экземплярах класса-контейнера (т. е. агрегирующего класса). Ее семантикой является отношение 'has-a'. Существует разница между сильным и слабым агрегированием относительно жизненного цикла частей и их контейнера. В данной Рекомендации всегда применяется сильное агрегирование. Оно используется для симметрии композиции по отношению к декомпозиции.

Метамодель eODL определяется путем использования нотации UML для MOF (см. рис. 4). Ограничения и правила формальной корректности, которые составляют часть этой метамодели и являются существенными для ее семантики, даются в виде английского текста. Диаграммы UML, которые появляются на рисунках в последующих параграфах, показывают только части законченной **метамодели** eODL. Для понимания семантики необходимо прочитать поясняющий текст и, в особенности, ограничения, содержащиеся в тексте. Те ограничения, которые уже описаны как часть **метамодели** IDL, в данной Рекомендации не повторяются. Вместо этого см. [7].

В Приложении D дается ссылка на законченную **метамодель**, включающую все ограничения, как на поток XMI.

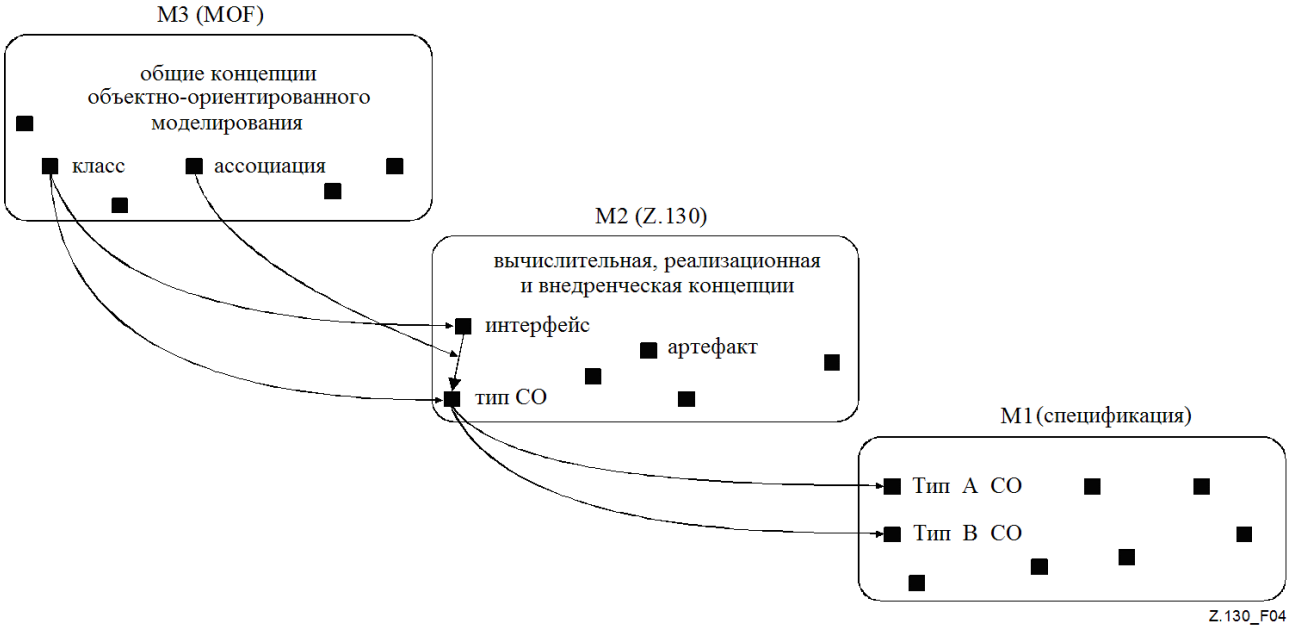


Рисунок 4/Z.130 – Объектно-ориентированные концепции

5.2 Присваивание имен и определение областей определения

Правила присваивания имен и определения областей видимости даются для обеспечения возможности однозначной идентификации элементов модели.

Взятая в целом модель формирует область определения имен. Каждая сущность, которая формирует новую область определения, является экземпляром абстрактного метакласса *Container*. Содержащиеся элементы области определения являются экземплярами абстрактного метакласса *Contained*. Определение метаклассов *Container* и *Contained* как абстрактных подразумевает, что все

экземпляры являются экземплярами производных не абстрактных метаклассов. Взаимосвязь *Container*-*Contained* часто используется в **метамодели**.

Каждая поименованная сущность (экземпляр метакласса *Contained*) имеет идентификатор для указания имени. Идентификатор является атрибутом метакласса *Contained*. Идентификаторы двух различных поименованных сущностей, принадлежащих одному и тому же *Container* (точки definedIn одного и того же элемента модели), должны быть различными.

Чтобы допустить в модели строгие области определения, вводится метакласс *ModuleDef*. Метакласс *ModuleDef* является частью **метамодели** CORBA-IDL, которая находится в основе **метамодели,** используемой в данной Рекомендации. Он является конкретным метаклассом и может быть конкретизирован. Он не имеет дополнительных свойств.

Каждый экземпляр *Container* формирует namespace. Обобщение от *Contained* до *Container* выражает способность вложенности областей определения имен (см. рис. 5).

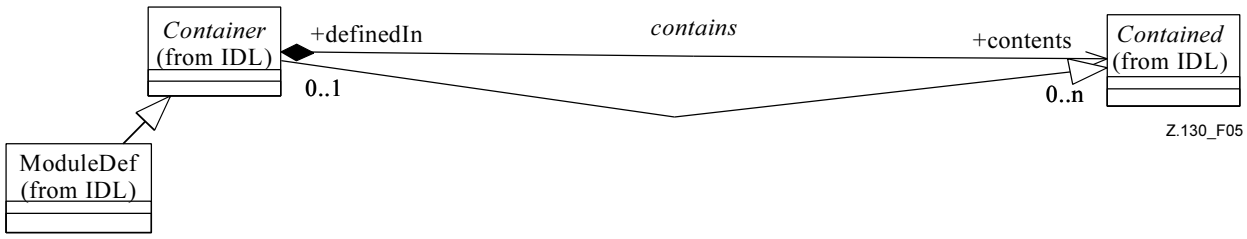


Рисунок 5/Z.130 – Присваивание имен и определение областей видимости

5.3 Концепции вычисления

5.3.1 Использованные концепции CORBA-IDL

Чтобы ввести **типы данных**, **операции**, **атрибуты**, **исключения** и **типы интерфейсов**, как концепции моделирования, в основу **метамодели** eODL положена **метамодель** CORBA-IDL.

Все эти концепции моделирования позволяют определение основных стандартных блоков вычислительных спецификаций. Одной из целей вычислительной спецификации является определение сигнатур **вычислительных объектов** (**CO**) на своих **портах**. Так как данная Рекомендация вводит моделирование, основанное на типе, для описания подобных сигнатур являются важными **типы данных**.

5.3.1.1 Типы данных, типы интерфейсов, операции, атрибуты, исключения

Типами данных в моделях являются экземпляры метаклассов, полученные из абстрактного метакласса *IDLType*. Это подразумевает включение целой системы типа данных CORBA-IDL. Для обеспечения открытости данной Рекомендации система типа данных может обмениваться путем использования абстрактного метакласса *IDLType*. На рис. 6 показывается подмножество **метамодели** CORBA-IDL для **типов данных**. **Метамодель** и ее описание могут быть найдена в [7].

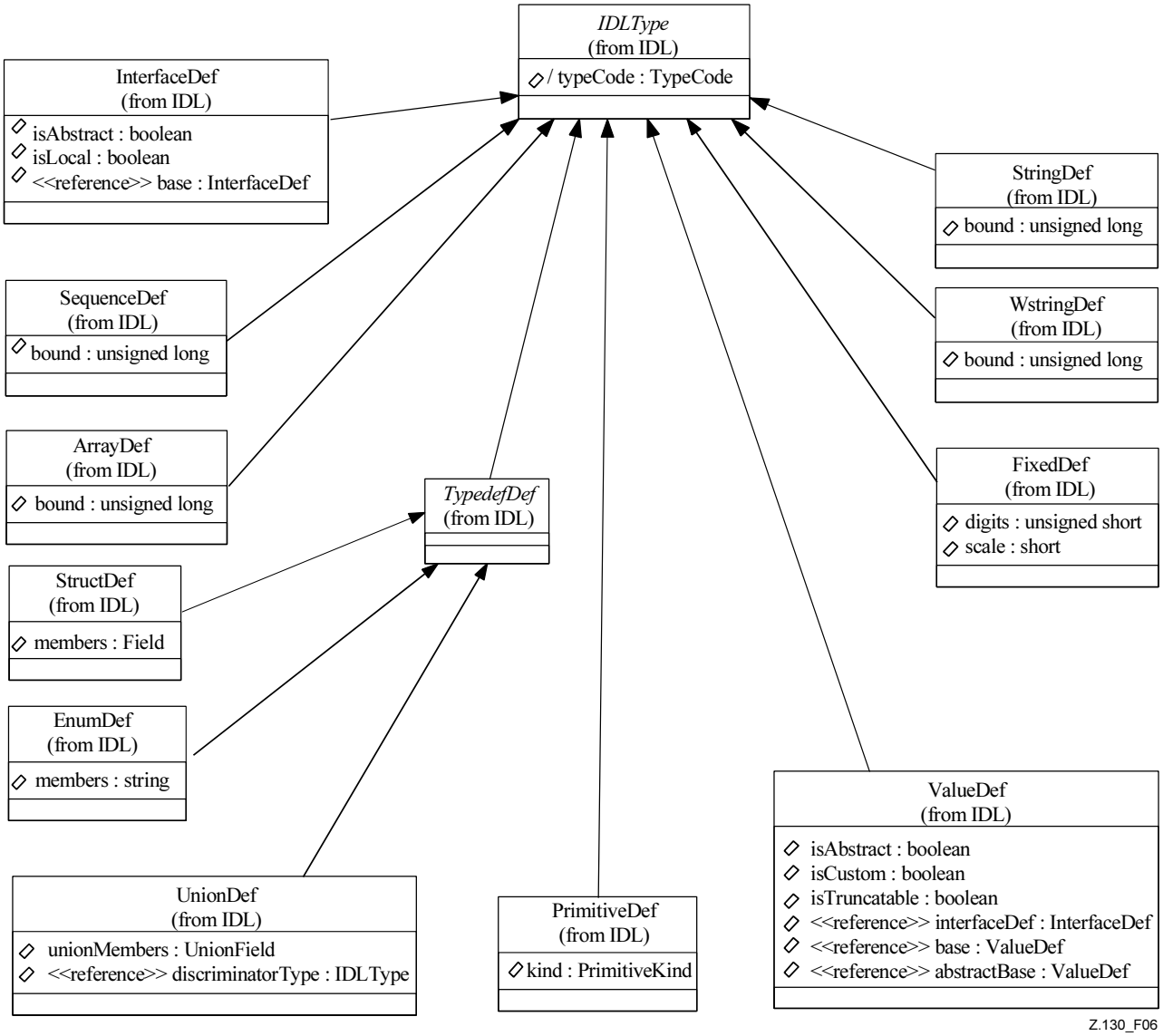


Рисунок 6/Z.130 – Типы данных

Система типа данных CORBA содержит все общеиспользуемые примитивы **типов данных**, например, *long*, *float*, *char*, *string* и т. д. Кроме того, имеются концепции для описания структурированных **типов данных**, например, *array*, *struct*, *sequence*, *union* и т. д. **Тип данных** *value* является дополнительным типом, который используется для прохода объектов, использующих объект семантикой значения. Подобно классам в языках программирования экземпляр типа *value* может агрегировать **атрибуты** и **операции**.

Для использования других определений типа данных, отличающихся от определений, включенных в архитектуру CORBA, вводится элемент ExternType. Он является конкретизацией концепции TypedefDef архитектуры CORBA. Здесь идентификатор атрибута относится к определению типа данных, предоставляемого внешне (см. рис. 7).

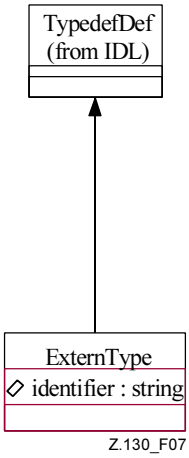


Рисунок 7/Z.130 – Тип внешний

Атрибуты, **операции**, **исключения** и **параметры** являются концепциями, необходимыми для определения операционных взаимодействий. **Операция**, определенная в модели, содержит список параметров, тип для возможного значения возврата и список неудачных окончаний, моделируемых **исключениями**. **Исключения** переносят информацию и определяются таким же образом, как и **тип данных** *StructDef*. Они содержат список элементов этой информации. **Атрибут** является сокращенной нотацией при моделировании операций, используемых для получения и установки поименованной переменной определенного типа. Кроме того, **исключения** могут складываться с **атрибутами**. В этом случае делается различие между теми **исключениями**, которые возможно установились в течение **операции** установки для этого **атрибута** и теми, которые устанавливаются в течение **операции** получения. **Метамодель** для **исключений** показана на рис. 8.

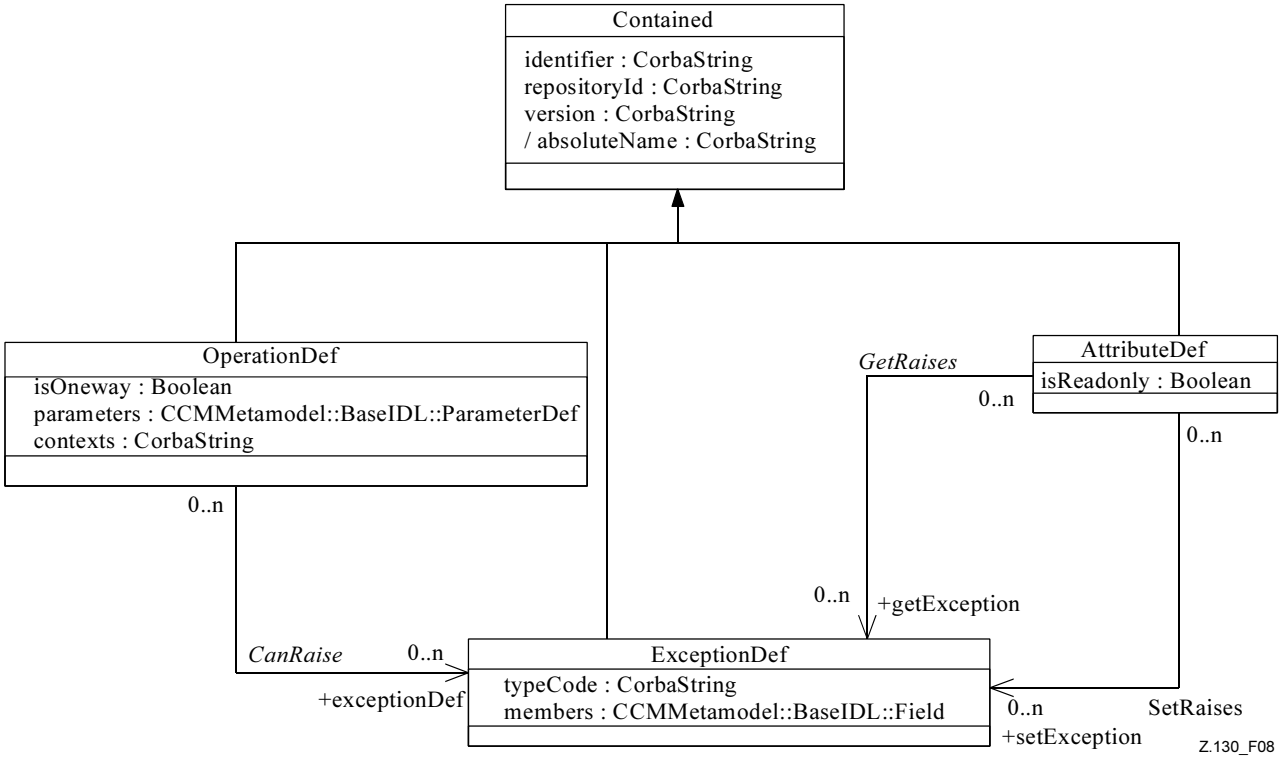


Рисунок 8/Z.130 – Исключения

Ниже описываются метаклассы *OperationDef*, *AttributeDef*, *ParameterDef* и *ExceptionDef*, полученные из **метамодели** CORBA-IDL. **Метамодель** для **атрибутов** и **операций** показана ниже (см. рис. 9 и 10).

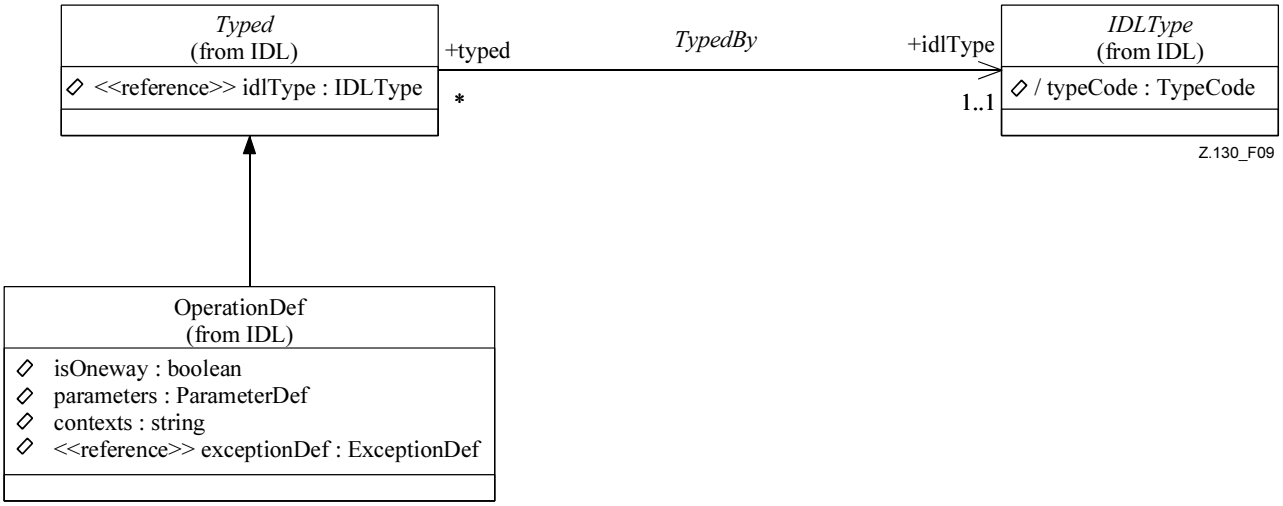


Рисунок 9/Z.130 – Операции

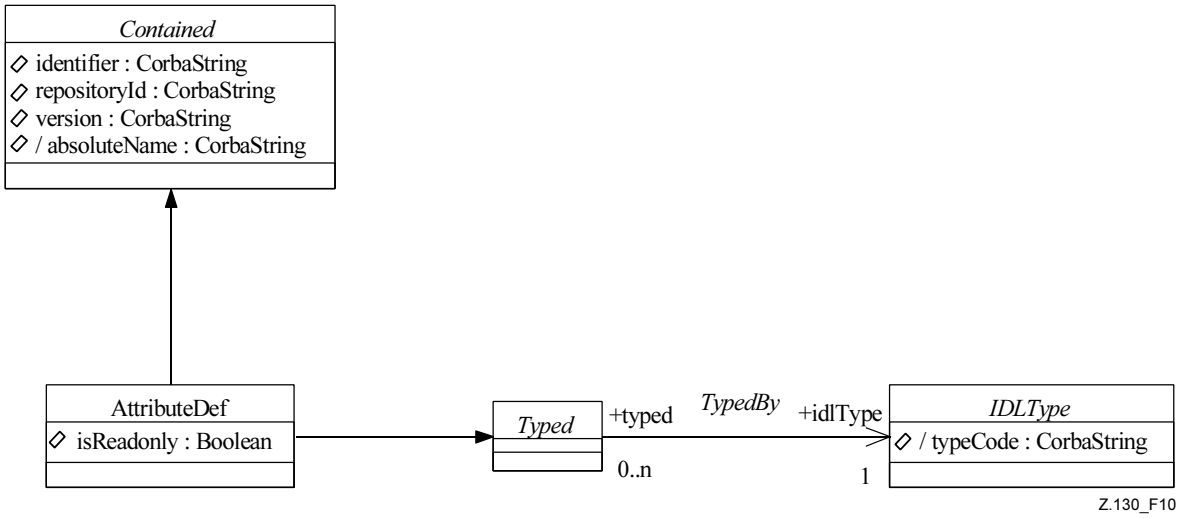


Рисунок 10/Z.130 – Атрибуты

Типы интерфейсов используются при определении сигнатур для возможных взаимодействий в системе. Концепция **типа интерфейса** уже известна из языка OMG IDL, где она названа интерфейсом. В языке OMG IDL интерфейсы лишь агрегируют **элементы** операционного **взаимодействия**. Это означает, что они являются контейнерами для **атрибутов** и **операций**. Это показано на рис. 11.

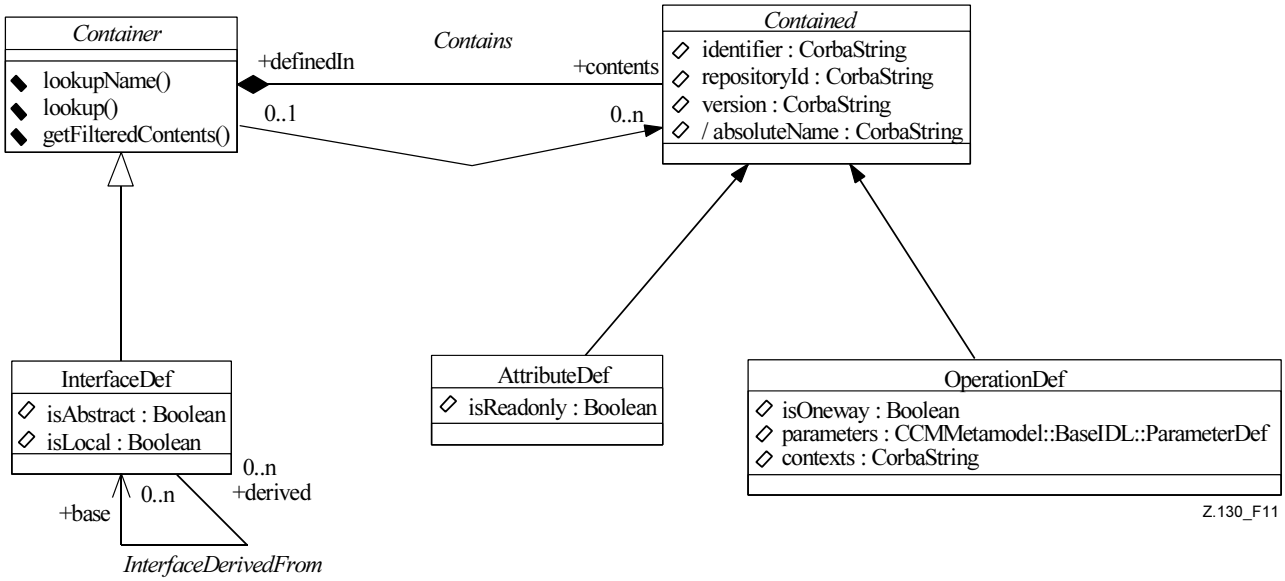


Рисунок 11/Z.130 – Операционные интерфейсы

5.3.2 Сигналы и параметры сигналов

Метамодель

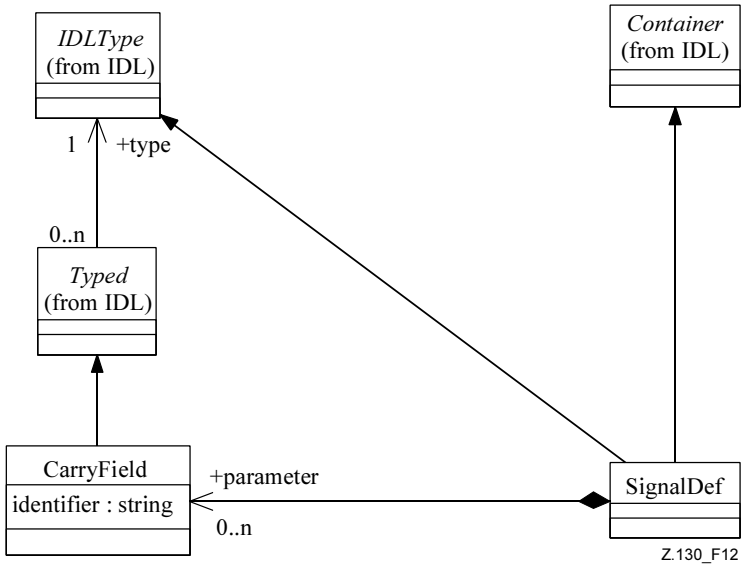


Рисунок 12/Z.130 – Сигнал и параметр сигнала

Семантика

Для расширения концепций моделирования, предложенных языком CORBA-IDL, а также для включения других видов взаимодействий в моделирование распределенных систем необходимы дополнительные концепции моделирования (см. рис. 12).

Сигналы используются для моделирования взаимодействий, опирающихся на сигналы, что означает асинхронный разделенный обмен сообщениями между системными сущностями. **Сигналы** переносят информацию. **Сигналы** моделируются экземплярами метакласса *SignalDef*. Переносимая информация (называемая **параметрами сигнала**) моделируется как экземпляры *CarryField*, причем каждый ссылается на экземпляр *ValueDef*, который является специальным типом данных языка CORBA-IDL. Каждый **параметр сигнала** является идентифицируемым, путем использования имени в контексте определения **сигнала**. Имена должны быть однозначно определяемыми.

Определение **сигнала** устанавливает структуру и свойства информации, которая переносится при конкретном сигнальном взаимодействии между системными сущностями. Оно еще не ассоциируется с **типом интерфейса**. Определения сигналов могут повторно использоваться в различных определениях типа интерфейса. Они играют ту же самую роль, как и **типы данных** в определении **операций**. Они являются стандартными блоками **сигнального взаимодействия**.

Определения **сигнала** в моделях могут происходить только в **пространствах имен**, являющихся либо модулями, либо глобальным **пространством имен**, сформированным самой спецификацией.

Типы IDL, которые используются для спецификации параметров **сигналов**, должны быть экземплярами типа *value*.

5.3.3 Тип носителя, носитель, множество носителей

Метамодель

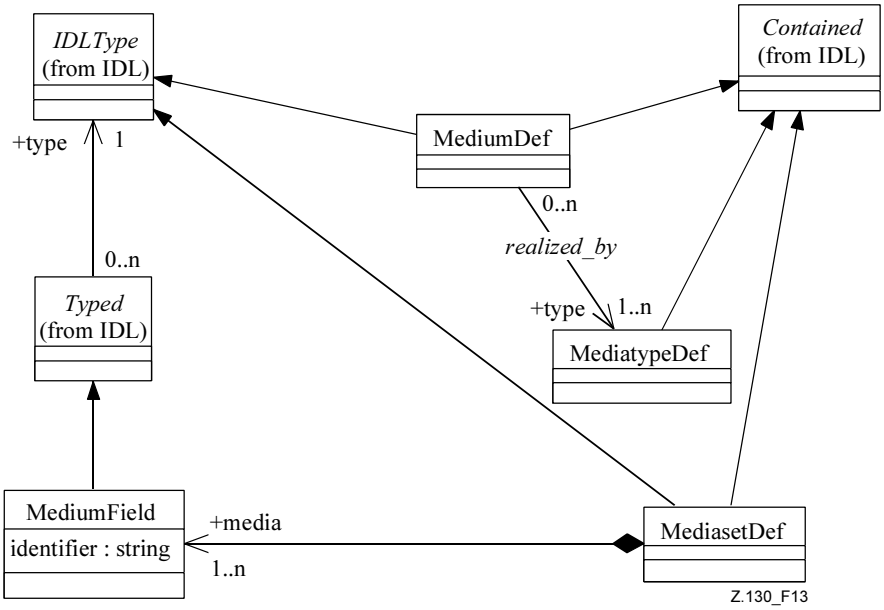


Рисунок 13/Z.130 – Носитель, тип носителя, множество носителей

Семантика

В дополнение к **операционному** и **сигнальному** взаимодействию обмен **непрерывными носителями** в распределенных системах также является важным видом взаимодействия. Для этого вида взаимодействия должны определяться концепции моделирования. Это сделано в данной Рекомендации точно так, как и для **операционного** и **сигнального** взаимодействий. Во-первых, должны быть определены основные стандартные блоки для **элементов взаимодействия**. Ими являются **носитель**, **множество носителей** и **тип носителя** (см. рис. 13).

Концепция **множества носителей** используется для моделирования **взаимодействия** типа **непрерывных** носителей. В **метамодели** это обеспечивается определением класса *MediasetDef*. Экземпляры этого класса агрегируют экземпляры класса *MediumDef* в поименованном списке, где каждый элемент имеет *MediumField* тип. Концепция **носителя** используется для моделирования одного элементарного потока данных между двумя сущностями. **Носитель** имеет смысл мультимедийной информации, подобной фильмам или аудио последовательностям. Обмен требует наличия кодирования, декодирования и форматов передачи, которые моделируются экземплярами класса *MediatypeDef*. **Носитель** может быть представлен одним или более **типов носителей**. **Множества носителей** необходимы для моделирования обмена между двумя различными носителями, которые связаны между собой (как видео- и аудиоданные фильма) и должны иметь одинаковые свойства при взаимодействиях.

Определения **носителя**, **множества носителей** и **типов носителей** в моделях может происходить только в пространствах имен, являющихся либо модулями, или глобальным пространством имен, формируемым самой спецификацией.

5.3.4 Поглощать и порождать

Метамодель

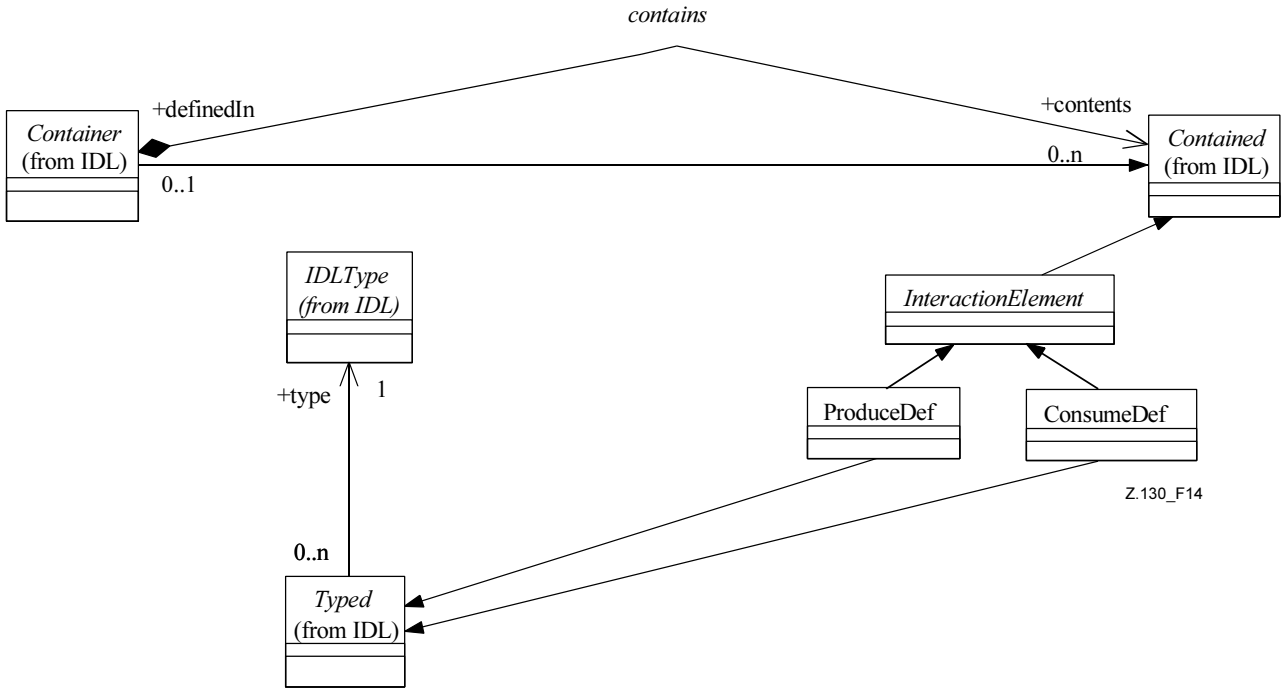


Рисунок 14/Z.130 – Потреблять и производить

Семантика

В **рабочем цикле сигналы** будут обмениваться между функциональными сущностями посредством своих **интерфейсов**. Поэтому **типы интерфейсов** предлагают необходимые сигнатуры. В случае **сигнальной** коммуникации сигнатура содержит тип (определение **сигнала**), имя **элемента взаимодействия** и признак того, порождается ли **сигнал** или поглощается посредством этого интерфейса. Таким образом, концепции **поглощать** и **порождать** являются **элементами взаимодействия** и используются для моделирования поглощения или порождения **сигналов** (см. рис. 14). Они являются элементами **сигнального** взаимодействия в том же смысле, как **операции** и **атрибуты** являются элементами **операционного** взаимодействия. Как и всякий **элемент взаимодействия**, **поглощать** и **порождать** являются идентифицируемыми элементами. Они включаются в **метамодель** с определением метаклассов *ConsumeDef* и *ProduceDef*, которые наследуют абстрактный метакласс *InteractionElement*, который сам наследует *Contained*. *ConsumeDef* и *ProduceDef* являются подклассами Typed. Такое наследование используется для установления отношения к метаклассу *SignalDef*, который адресуется посредством **порождать** или **поглощать**.

Заметим, что метакласс *SignalDef* является подклассом *IDLType*. Требуется, чтобы экземпляр *IDLType*, ассоциированный с определением **порождать** или **поглощать**, являлся бы **сигналом**. С этой целью концепция **сигнала** определяется как конкретизация *IDLType*.

5.3.5 Сток и исток

Метамодель

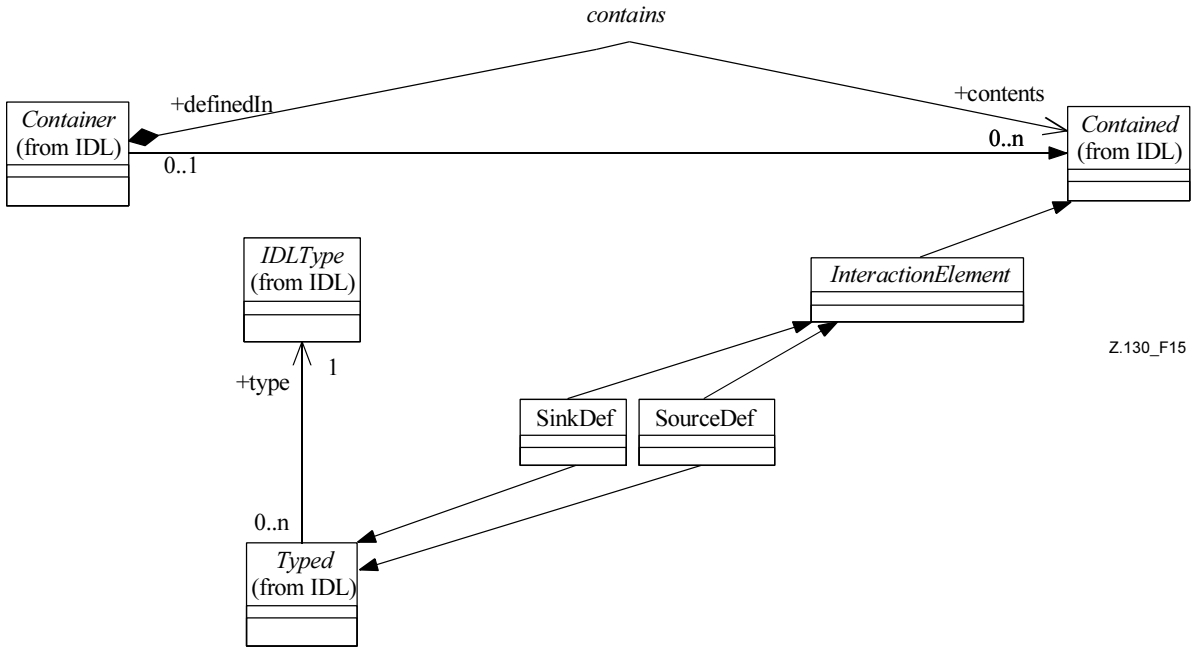


Рисунок 15/Z.130 – Сток и исток

Семантика

Для завершения перечня возможных **элементов взаимодействия**, применяемых для указания сигнатур **типа интерфейса**, должны быть определены **элементы взаимодействия** типа **непрерывных носителей**. Подобно **сигнальным** взаимодействиям, они характеризуются информацией, которой обмениваются при взаимодействии в **рабочем цикле** (**множество носителей**), идентификатором и направлением коммуникации, т. е. является ли интерфейс **стоком** или **истоком** по отношению к взаимодействию. Следовательно, концепции **стока** и **истока** являются элементами взаимодействия при моделировании поглощения или порождения **множеств носителей** (см. рис. 15). Они являются элементами взаимодействия типа **непрерывных носителей** в том же смысле, как **операции** и **атрибуты** являются элементами **операционного** взаимодействия. Как всякий **элемент взаимодействия**, **сток** и **исток** являются идентифицируемыми элементами. Они включаются в **метамодель** с определением метаклассов *SinkDef* и *SourceDef*, которые наследуют метакласс *InteractionElement*, который сам наследует *Contained*. *SinkDef* и *SourceDef* являются подклассами *Typed*. Эта наследственность используется для установления отношения к *MediasetDef*, который адресуется посредством концепций **сток** или **исток**.

Заметим, что *MediasetDef* является подклассом *IDLType*. Единственным конкретным *IDLType*, разрешенным быть ассоциированным со **стоком** и **истоком**, является тип носителя.

5.3.6 Тип интерфейса

Метамодель

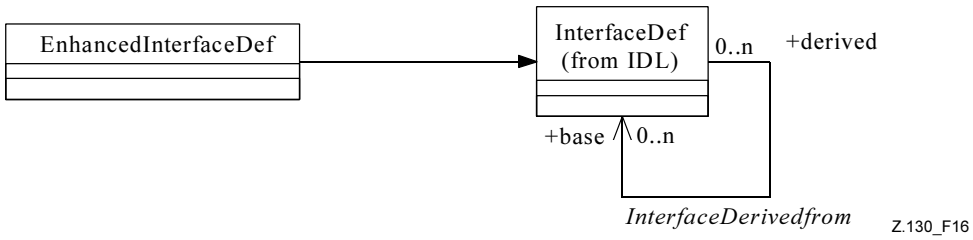


Рисунок 16/Z.130 – Тип интерфейса

Семантика

Концепция **тип интерфейса** используется для указания подмножества потенциальных **взаимодействий** объектов **СО типов СО**. **Типы интерфейса** агрегируют **элементы взаимодействия** взаимодействующих **операционных**, **сигнальных** и типа **непрерывных носителей** видов. Учитывая это, семантика **типа интерфейса** расширяется по сравнению с RM-ODP: **типы интерфейсов** обеспечивают общий контекст для **элементов взаимодействия** *различных* видов взаимодействия. Для клиентов, имеющих ссылку к такому **интерфейсу**, возможно использование всех **элементов взаимодействия** независимо от используемого вида взаимодействия. Этот аспект является предметом заботы окружающей среды рабочего цикла.

Тип интерфейса в **метамодели** является экземпляром класса, названным *EnhancedInterfaceDef* (см. рис. 16). Так как **метамодель** IDL уже содержит значение агрегирования **элементов операционного взаимодействия**, класс *EnhancedInterfaceDef* наследует *InterfaceDef*. Из этого следует, что правила наследования и ограничения *InterfaceDef* также применимы и к *EnhancedInterfaceDef*.

Кроме того, **типы интерфейсов** являются контейнерами для элементов взаимодействия **порождать**, **поглощать**, **сток** и **исток**.

5.3.7 Типы СО, отношения поддерживать и требовать

Метамодель

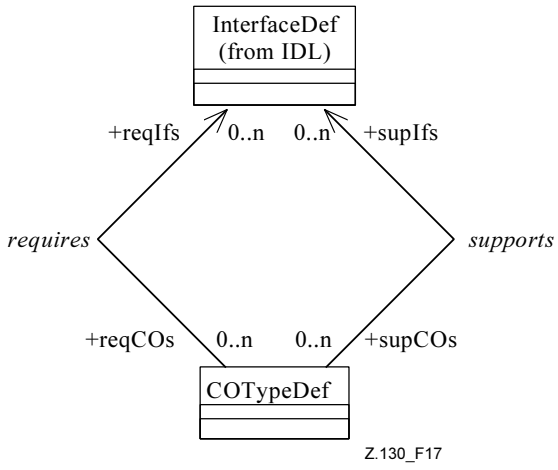


Рисунок 17/Z.130 – Типы СО, поддерживать и требовать

Семантика

Концепция **типа CO** используется для указания функциональной декомпозиции системы. Экземпляры **типа CO** (**COs**) являются автономными интерактивными сущностями, которые инкапсулируют состояние и поведение. **COs** взаимодействуют со своей окружающей средой посредством вполне определенных **интерфейсов**. Эти **интерфейсы** определяются с использованием концепции **тип интерфейса**, описанной выше.

Тип CO может поддерживать или требовать **тип интерфейса**. Поддерживать **тип интерфейса** означает, что **COs** данного **типа CO** предоставляют интерфейсы данного **типа интерфейса**. Требовать **тип интерфейса** означает, что **COs** данного **типа CO** используют **интерфейсы** данного типа **интерфейса**. **Тип CO** является экземпляром класса *COTypeDef* в **метамодели**. Этикетки *supports* (поддерживать) и *requires* (требовать) идентифицируют ассоциации между *COTypeDef* и *InterfaceDef* (см. рис. 17).

Для того чтобы получить доступ к **COs** в **рабочем цикле**, *COTypeDef* получается из *InterfaceDef* (как показано на рис. 17). При этом экземпляры могут конфигурироваться с использованием атрибутов, которые определены данным **типом CO**. Важно отметить, что только для **типа CO** разрешается содержать **элементы взаимодействия** атрибутивного вида. Никаких других **элементов взаимодействия** не разрешается. Более того, наследование отношений между **типами CO** и **типами интерфейсов** не может смешиваться, т. е. **типы CO** могут наследовать только **типы CO**, а **типы интерфейсов** - только **типы интерфейсов**.

5.3.8 Определение порта предоставленного и порта используемого

Метамодель

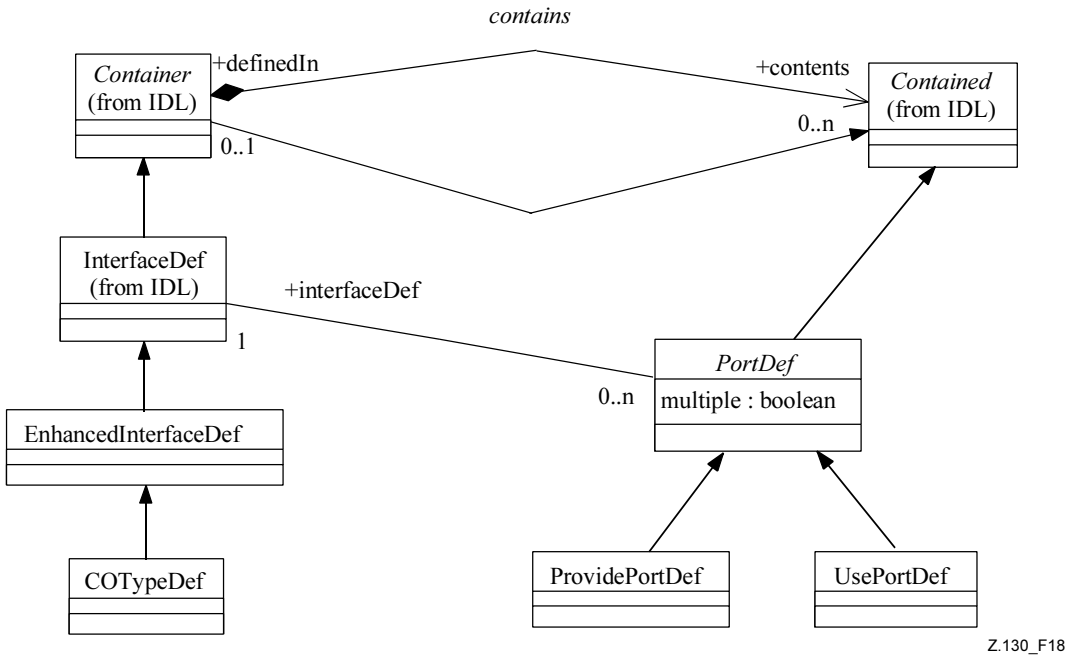


Рисунок 18/Z.130 – Порт предоставленный и порт используемый

Семантика

В распределенной системе, которая определяется путем использования данной Рекомендации, **COs** являются функциональными сущностями. Они обмениваются информацией через свои поддерживаемые и требуемые **интерфейсы**. Однако конфигурация распределенной системы всегда является проблемой, в особенности в отношении получения и обмена **ссылками интерфейса**, что является необходимым условием при **взаимодействии**. Поэтому данная Рекомендация вводит концепцию **порта**, как поименованную точку взаимодействия, в которой в **рабочем цикле** может быть либо получена ссылка поддерживаемого **интерфейса CO**, либо зарегистрирована ссылка используемого **интерфейса**.

Концепции **порта предоставленного** и **порта используемого** применяются для моделирования **портов типа CO**, которые используются окружающей средой на основе имени либо для получения ссылки к **интерфейсу** (**порт предоставленный**), либо для хранения ссылки к **интерфейсу** (**порт используемый**). Применение концепций **поддерживать** и **требовать** позволяет выразить лишь потенциальное предоставление или использование **типов интерфейсов** в контексте **типа CO**, а не конкретных механизмов того, как окружающая среда **CO** получает доступ к этим контекстам взаимодействия. Концепции **порта предоставленного** и **порта используемого** определяются как экземпляры класса *ProvidePortDef* и *UsePortDef*. Оба класса наследуют абстрактный класс *PortDef*. Класс *PortDef* наследует *Contained*, что означает, что экземпляр *COTypeDef* может содержать определения **порта предоставленного** и **порта используемого**. Определения **порта предоставленного** и **порта используемого** всегда ассоциируются с определением **интерфейса** (см. рис. 18).

Определения **порта** разрешаются только в пределах определений **типа CO**. **Интерфейс**, для которого определен **порт предоставленный**, автоматически является поддерживаемым **интерфейсом**. **Интерфейс**, для которого определен **порт используемый**, автоматически является требуемым **интерфейсом**.

5.4 Концепции реализации

5.4.1 Артефакт и схема конкретизации

Метамодель

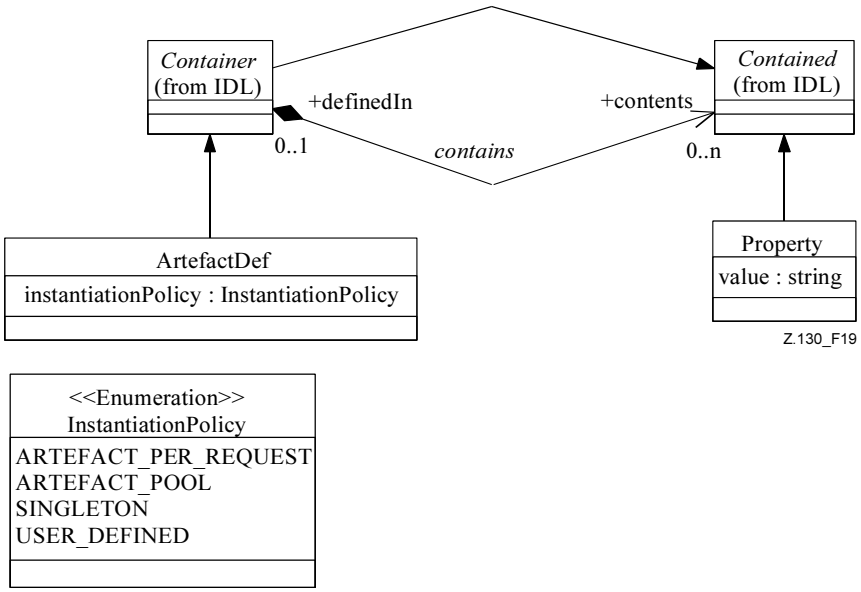


Рисунок 19/Z.130 – Артефакт и схема конкретизации

Семантика

Концепция **артефакта** применяется при описании контекста языка программирования (например, класса объектно-ориентированного языка программирования) в модели. Экземпляры концепции **артефакта** представляют поведение **COs**. Поэтому они обеспечивают бизнес-логику **типов CO**. Отношения между **артефактами** и поведенческими частями **COs** определяются ассоциациями между **артефактами** и **элементами взаимодействия типов интерфейса**. Контексты языков программирования, которые моделируются экземплярами концепции **артефакта**, будут конкретизированы для обработки в **рабочем цикле**, например, вызовы **операций**, входы **сигналов** или данные **непрерывных носителей**. Стратегии, используемые при конкретизации, указываются экземплярами схемы конкретизации концепции. Допустимые схемы можно видеть на рис. 19. При необходимости могут быть добавлены дополнительные схемы. Разделение концепций **артефакта** и **типа CO** обеспечивает полную гибкость при проектировании распределенного приложения:

- Представление внешнее (как окружающая среда может взаимодействовать с **CO**) отделяется от представления внутреннего (как обеспечивается поведение **CO**).
- Для представлений внешнего и внутреннего могут использоваться разные деревья наследования.
- Переиспользование существующего поведения и существующих определений **интерфейса** возможно, и не зависит одно от другого.

Концепция **артефакта** в **метамодели** выражается экземпляром класса, называемого *ArtefactDef*. Как показано на рис. 19, схема конкретизации концепции моделируется как атрибут этого класса типа перечисления с перечислителями.

5.4.2 Отношение реализации

Метамодель

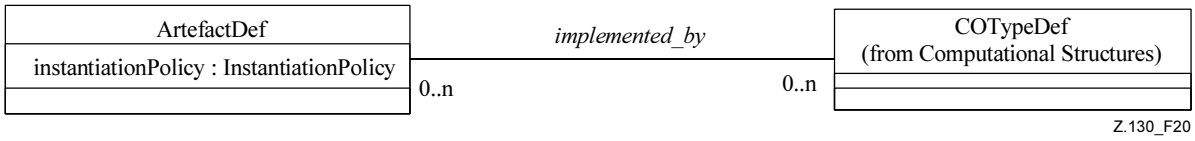


Рисунок 20/Z.130 – Отношение реализации

Семантика

Экземпляры концепции **артефакта** описывают реализацию ожидаемого поведения **элементов взаимодействия типов интерфейсов**, которые поддерживаются или требуются **типами CO**. Как описывалось выше, представления внешнее и внутреннее **CO** являются полностью автономными друг от друга. Тем не менее, отношения между ними должны описываться для того, чтобы выбрать правильное поведение, когда **CO** оказывается вовлеченным в определенное взаимодействие. Поэтому модель описывает, какое множество **артефактов** обеспечивает поведение **COs** этого типа **CO**. Это отношение в **метамодели** определяется ассоциацией *implemented_by* между *COTypeDef* и *ArtefactDef* (см. рис. 20).

5.4.3 Элемент реализации

Метамодель

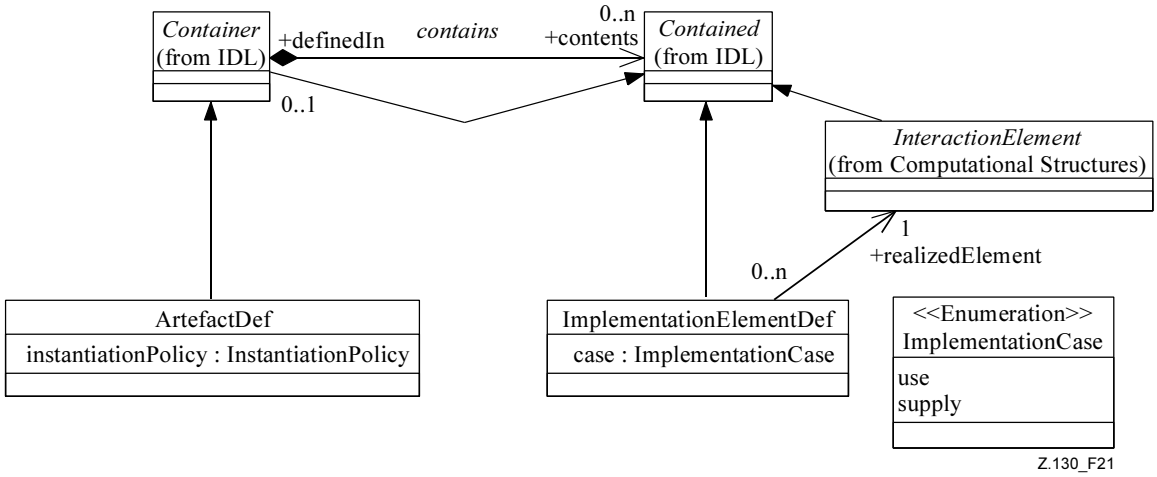


Рисунок 21/Z.130 – Элемент реализации

Семантика

В контексте экземпляра **артефакта** концепция **элемента реализации** используется для обозначения того, что поведенческая часть артефакта (например, метод класса, который моделируется экземпляром концепции **артефакта**) **представляет** отдельный **элемент взаимодействия типа интерфейса**. Как объяснялось выше, данная концепция является необходимой для обеспечения дополнительных деталей отношению *implemented_by*. Отношение *implemented_by* говорит, что **артефакт** делает вклад в поведение **CO**, без указания на то, какая часть **артефакта** является ответственной, за какую часть поведения **CO**. Эти детали обеспечиваются **элементами реализации артефакта**. Эта информация является необходимой для ассоциирования поведения с взаимодействием в рабочем цикле.

Концепция **элемента реализации** указывается как экземпляр класса с именем *ImplementationElementDef*. Этот класс наследует *Contained*; экземпляры *ImplementationElementDef* могут содержаться экземплярами **артефактов** (см. рис. 21). *ImplementationCase* определяет направление реализации **элемента реализации**: либо использование, или предоставление поведения может представляться **элементом реализации** относительно определенного **элемента взаимодействия**.

5.5 Концепции внедрения

5.5.1 Программные компоненты и зависимость компонентов

Метамодель

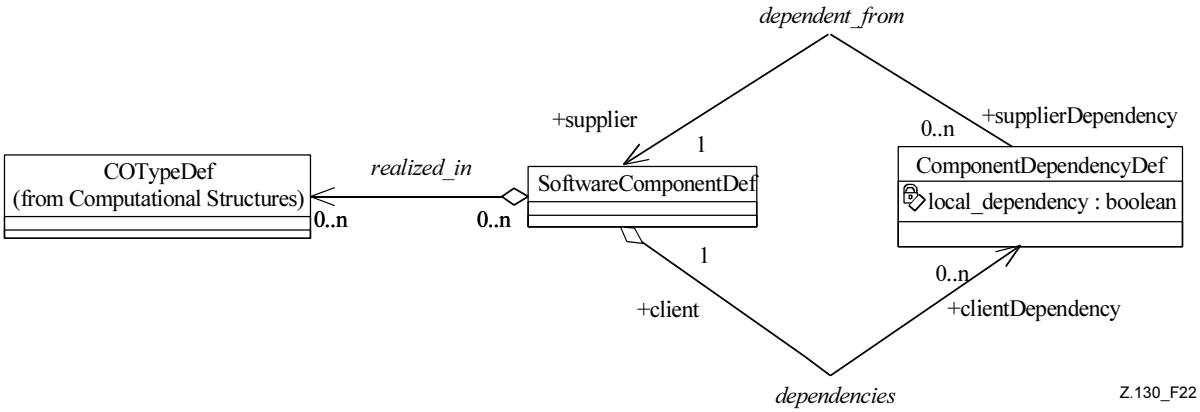


Рисунок 22/Z.130 – Программные компоненты и зависимость компонентов

Семантика

Концепция программных компонентов отображает фактическое программное обеспечение в проектной модели. Она идентифицирует сущность **внедрения** и позволяет дополнительное описание посредством использования свойств. **Программный компонент** может, но не обязан, **представлять** произвольное число **типов CO**. Поэтому он содержит последовательности инструкций, которые при исполнении в узле, претворяют **COs**, т. е. они обеспечивают поведение, состояние и идентичность **COs**. В **метамодели** данная концепция вводится метаклассом *SoftwareComponentDef*. Чтобы показать, какой тип **CO** представлен определенным **программным компонентом**, существует концепция отношения **представлять**, которая вводится в **метамодель** ассоциацией между метаклассом *COTypeDef* и *SoftwareComponentDef* (см. рис. 22).

Для своего правильного исполнения **программные компоненты** могут потребовать дополнительные **программные компоненты**. Для этого в модели определяется метакласс *SoftwareDependencyDef*. Он содержит атрибут *local_dependency*, который устанавливает, должен ли быть локально доступен требуемый **программный компонент**. Метакласс *SoftwareComponentDef* может содержать произвольное число *SoftwareDependencyDefs,* причем каждый *SoftwareDependencyDef* ассоциативно связан с другим *SoftwareComponentDef*, указывающим требуемое программное обеспечение.

5.5.2 Сборка и исходная конфигурация

Метамодель

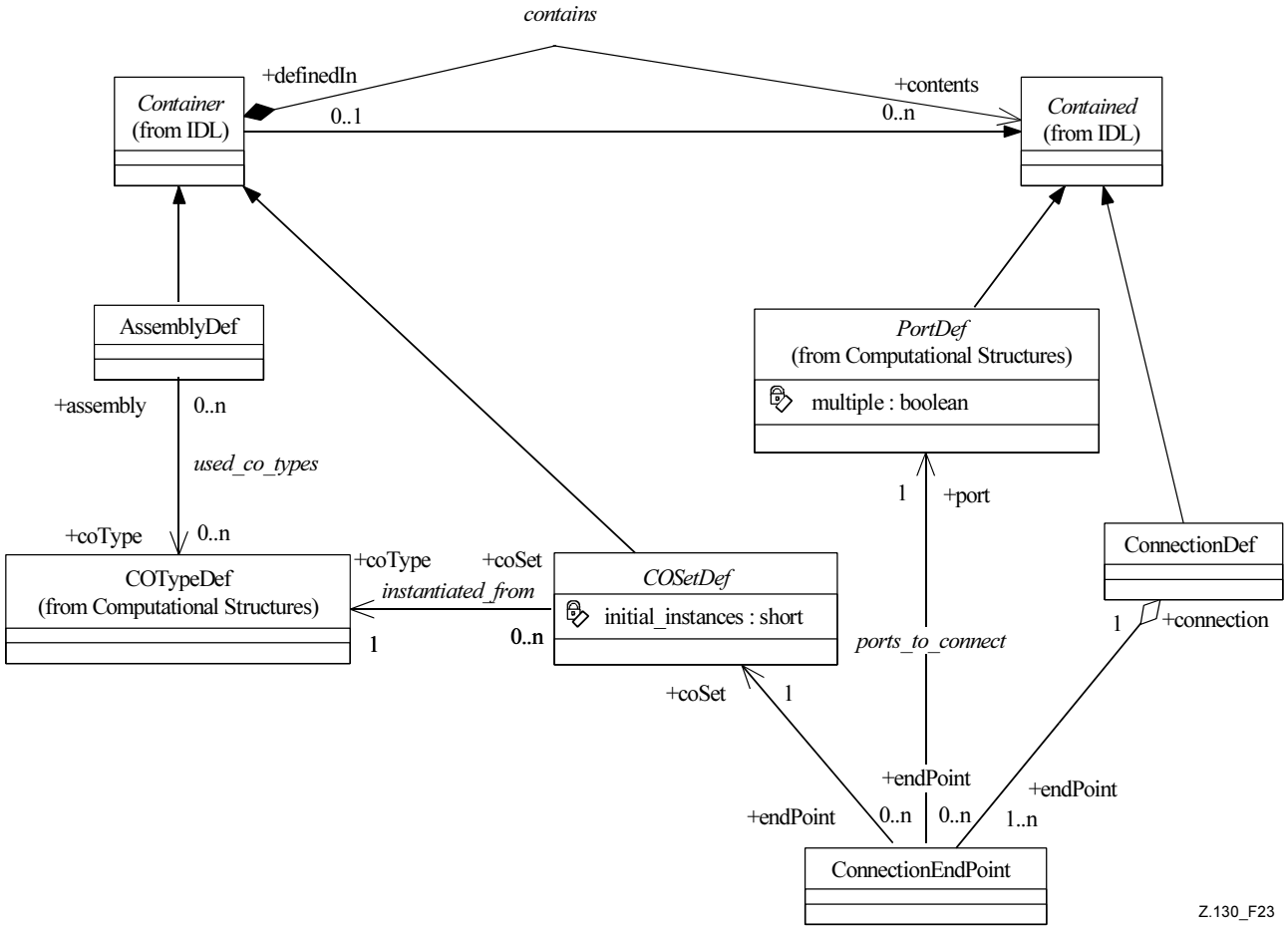


Рисунок 23/Z.130 – Сборка и исходная конфигурация

Семантика

Концепция **сборки** используется для моделирования систем программного обеспечения посредством указания типов CO, вовлеченных в систему, и для моделирования **исходной конфигурации** системы. **Исходной конфигурацией** является конфигурация, которая устанавливается в начале времени исполнения системы программного обеспечения и состоит из **исходных COs** и их **исходных соединений**. Концепция сборки в **метамодели** моделируется метаклассом *AssemblyDef*. **Типы CO** ассоциируются введением ассоциации между метаклассами *AssemblyDef* и *COTypeDef* (см. рис. 23).

Для моделирования исходных **COs метамодель** содержит метакласс *COSetDef*. Метакласс *COSetDef* определяет создание произвольного числа экземпляров ассоциированного **типа CO**. Число экземпляров определяется атрибутом *initial_instances*. Метакласс *COSetDef* содержится в метаклассе *AssemblyDef*.

Для моделирования **исходных соединений метамодель** содержит метакласс *ConnectionDef*. **Соединение** устанавливается между портами участвующих **COs** посредством обмена **ссылками интерфейса COs**. Эти ссылки получаются от **CO**, где **тип CO** имеет определение **порта предоставленного** и пересылается к **CO**, чей тип имеет определение **порта используемого**. Метакласс *ConnectionDef* в **метамодели** состоит из множества *ConnectionEndPoints*. Множество *ConnectionEndPoint* ассоциируется с *PortDef* метаклассов *COTypeDef* и *COSetDef*. Каждый **CO** метакласса *COSetDef*, ассоциированного с *ConnectionEndPoint*, соединяется с каждым **CO** каждого метакласса COSetDef, ассоциированного с другой *ConnectionEndPoints*, агрегированной с тем же самым *ConnectionDef*.

5.5.3 Свойства и ограничения

Метамодель

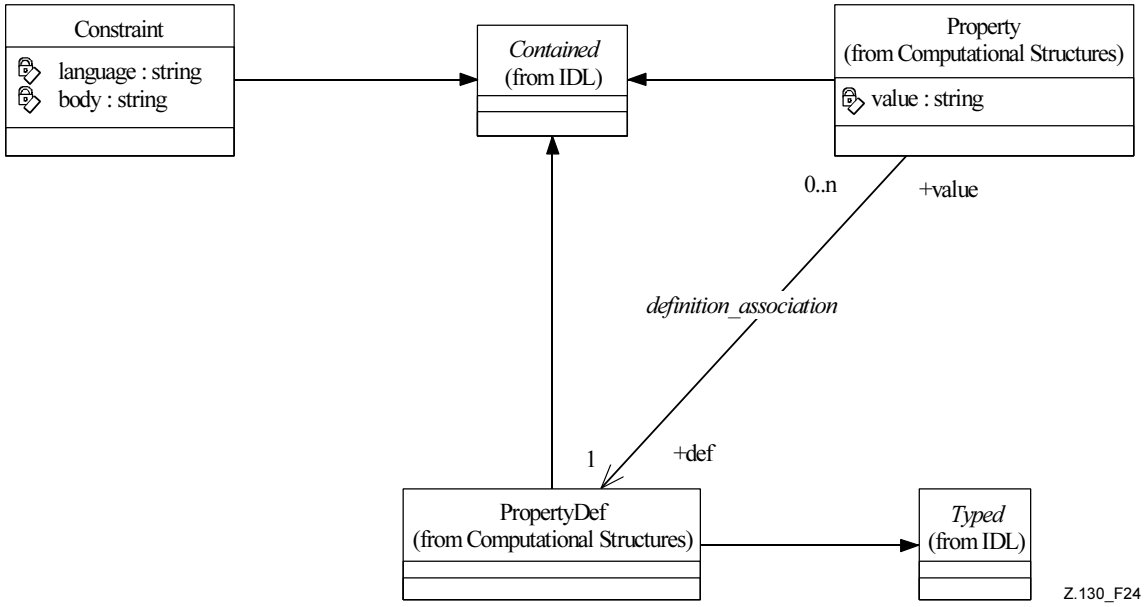


Figure 24/Z.130 – Свойства и ограничения

Семантика

Концепция **свойств**, которые могут присоединяться к элементам модели, отражается в **метамодели** метаклассами *PropertyDef* и *Property* (см. рис. 24). **Метамодель** проводит различие между определением свойства и значением свойства. Метакласс *Property* содержит значение, представленное значением атрибута строки, а метакласс *PropertyDef* - спецификацию **типа данных** для значения, т. к. он наследует метакласс *Typed*. Например, **тип CO** может определять свойства, необходимые для его конфигурации, в то время как **CO** может определять соответствующие значения свойства.

Концепция **ограничения** отражается в **метамодели** метаклассом *Constraint*. Он имеет два атрибута. Атрибут *language* определяет язык, на котором написано ограничение и который должен использоваться для оценки, и атрибута *body*, содержащего фактическое представление строки ограничения. Выбор языка ограничения остается за пользователем и не устанавливается данной Рекомендацией. Таким образом, может выбираться любой подходящий язык, а семантики ограничений здесь не определяются, но остаются для инструментов обработки. Объект **CO** может определять множество ограничений для выражения разрешенных комбинаций значений свойства. **Сборка** может определять взаимное размещение ограничений на текущих компонентах. Определения свойств ссылок или атрибутов квалифицируются своими именами.

5.6 Концепции целевой окружающей среды

Спецификация распределения и **внедрения** достигается моделированием фактических целевых окружающих сред, логических целевых окружающих сред и отображением логических сущностей на фактические узлы окружающих сред. В некоторых случаях при помощи инструментального средства может автоматически извлекаться модель фактической целевой окружающей среды.

5.6.1 Целевая окружающая среда, Node и NodeLink

Метамодель

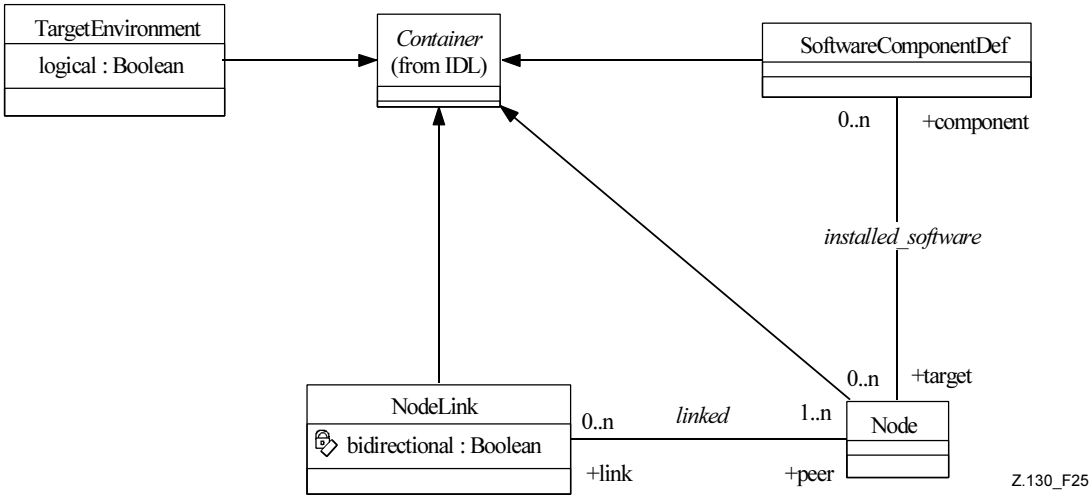


Рисунок 25/Z.130 – Целевая окружающая среда, Node и NodeLink

Семантика

Целевая окружающая среда моделирует физически распределенную окружающую среду в рабочем цикле, например, сеть электросвязи, состоящую из узлов и линий связи между узлами. В **метамодели** метакласс *TargetEnvironment* является контейнером для *Node* и *NodeLink* (см. рис. 25). Атрибут *logical* в *TargetEnvironment* используется для обозначения того, отражает ли модель текущую окружающую среду или потенциальную.

Метакласс *Node* представляет концепцию узла, что означает элемент целевой окружающей среды, отличающийся, по крайней мере, одним или несколькими процессорами, блоком памяти и операционной системой. Заметим, что данная физическая машина может исполнять роль более чем только одного логического *Node*. Узел обращается к установленному программному обеспечению (**программным компонентам**, компиляторам, интерпретаторам и т. п.) и установленному аппаратному обеспечению (представляемому драйвером программного обеспечения). Такие свойства, как операционная система или процессор, описываются как предопределенные свойства, перечисленные ниже.

Концепция линий между узлами представляется метаклассом *NodeLink*, как физическая линия между двумя или более *Nodes* (например, совместно используемая шина). *Реверсивный* булевый атрибут в *NodeLink* применяется только, когда *NodeLink* ассоциируется точно с двумя *Nodes*. Когда *реверсивный* является ложью, очередность двух *Nodes*, ассоциированных с *NodeLink*, интерпретируется следующим образом: первый экземпляр *Node* соответствует узлу истока, а второй экземпляр *Node* представляет собой узел назначения. Поэтому *взаимосвязанная* ассоциация является упорядоченной.

Метаклассы *Node* и *NodeLink* являются контейнерами для *PropertyDef* и *Property*. Это означает, что предопределенные свойства и свойства, определенные пользователем, могут присоединяться непосредственно к экземплярам узла и экземплярам узловых линий.

5.6.1.1 Предопределенные свойства Node и NodeLinks

Имеются некоторые предопределенные свойства *Node* и *NodeLink* (см. табл. 1). С этой целью, используя IDL, вводятся следующие неявные **типы данных**:

```
struct ProcessorType {
      string family;
      string type;
      integer frequency;
}
```

```
struct OSType {
      string name;
      string version;
}
```

Таблица 1/Z.130 – Предопределенные свойства

Сущность	Предопределенное название свойства	Тип	Описание	Обязательный
Node	Процессор	ProcessorType	процессор в составе узла	Да
	Память	Целое число	максимальная память узла (Кбайт)	Нет
	Операционная система	OSType	идентификация операционной системы узла	Да
NodeLink	Полоса частот	Целое число	максимальная скорость передачи (Кбайт/с)	Нет

5.6.2 InstallationMap

Метамодель

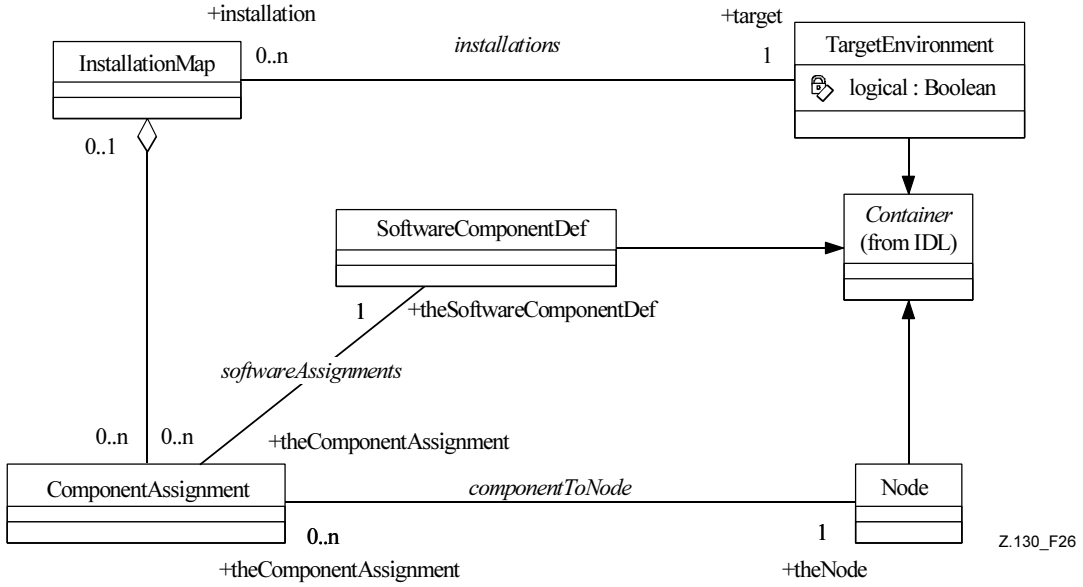


Рисунок 26/Z.130 – Отображение инсталляции

Семантика

При моделировании *TargetEnvironment* ее узлам могут присваиваться соответствующие сущности. Есть два вида сущностей: модули программного обеспечения, представляющие программное обеспечение в узле, и множества **CO**, представляющие конкретные экземпляры **типов CO**.

Отображение инсталляции представляет собой способ, которым реализации распределяются в целевой окружающей среде. Он состоит из множества инсталляционных присвоений, каждое из которых ассоциирует один **программный компонент** с одним узлом. Отображение инсталляции относится к узлам целевой окружающей среды. Представление отображения инсталляции является метаклассом *InstallationMap*. Представление присвоения инсталляции является метаклассом *ComponentAssignment* (см. рис. 26).

5.6.3 InstantiationMap

Метамодель

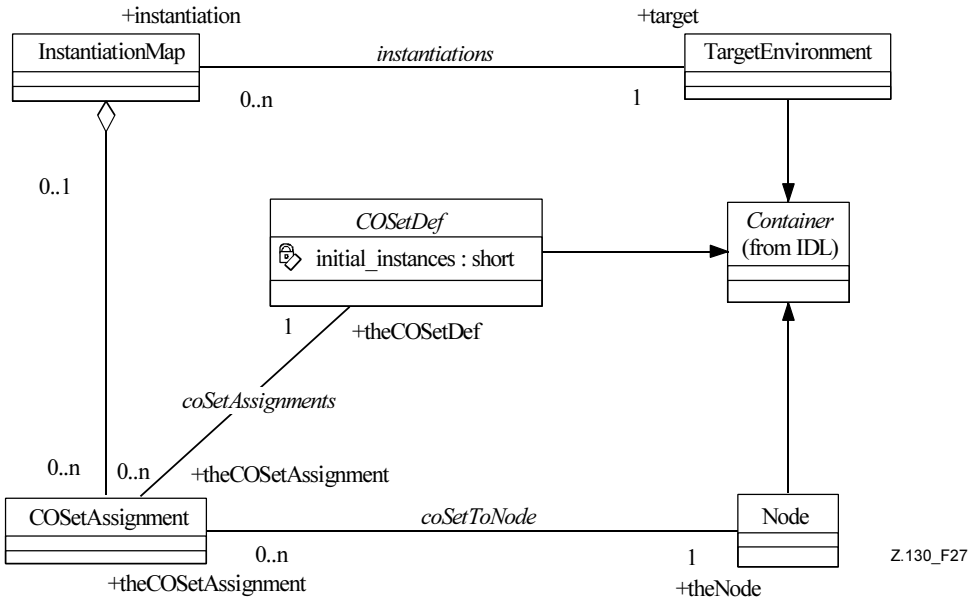


Рисунок 27/Z.130 – Отображение конкретизации

Семантика

Отображение конкретизации представляет собой способ, которым конкретные экземпляры **типов CO** распределяются в целевой окружающей среде. Оно состоит из множества присвоений конкретизации, ассоциирующих множество **COs** с одним узлом целевой окружающей среды. Отображение конкретизации относится к **типам CO**, определенным в контексте **сборки**, и к узлам выбранной целевой окружающей среды. Метаклассом, представляющим концепцию отображения конкретизации, является *InstantiationMap*. Присвоение **COs** представляется метаклассом *COSetAssignment* (см. рис. 27).

5.6.4 План внедрения

Метамодель

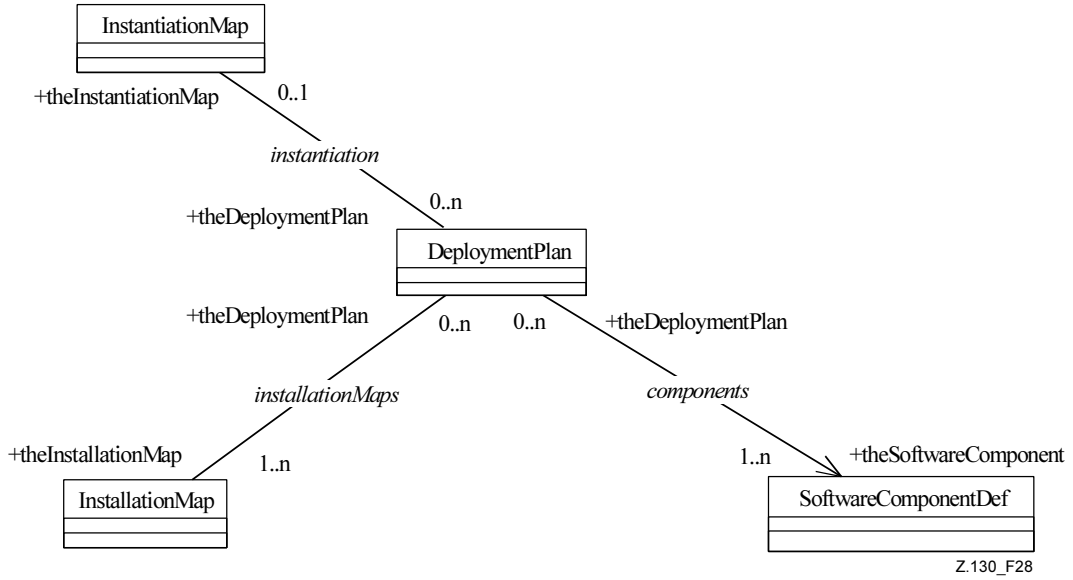


Рисунок 28/Z.130 – План внедрения

Семантика

План внедрения определяется выбором одного или более **программных компонентов**, содержащих реализации **типов CO**, одно или более отображений инсталляции, определяющих, где инсталлировать **программные компоненты**, а также ноль или одно отображение конкретизации, определяющее, где создать **COs** (см. рис. 28). Отображение конкретизации и все отображения инсталляций модели eODL должны обращаться к той же целевой окружающей среде и к той же **сборке**.

6 Библиография

[11] Формальный документ/00-03-02 OMG: Спецификация унифицированного языка моделирования OMG, Версия 1.3.

[12] Рекомендация W3C (2000 г.): Расширяемый язык разметки (XML) 1.0 (Второе издание).

[13] Документ ad/01-02-29 OMG: Профиль UML для MOF.

[14] Документ omg/ 00-11-05 OMG: Архитектура, управляемая моделями.

[15] Стандарт IEEE 754-1985: Стандарт для двоичной арифметики с плавающей запятой IEEE.

[16] SZYPERSKI (C.): Component Software – Beyond Object-Oriented Programming, Addison-Wesley, ISBN: 0-201-17888-5.

Приложение A

Синтаксис eODL

A.1 Введение

В данном приложении приводится текстовая нотация eODL по форме EBNF. Синтаксис для концепций, происходящих от OMG CORBA IDL 2.4.2, взят из [5].

A.2 Лексические соглашения и основа грамматики

В последующих параграфах применяются определения, приведенные в параграфе 3.1 (лексические соглашения для IDL) и параграфе 3.4 (грамматика IDL) документа спецификаций OMG CORBA 2.4.2.

A.3 Представление вычисления

A.3.1 Пространства имен, типы данных, исключения, операции и атрибуты

Метамодель основана на системе данных типа CORBA-IDL; язык eODL основан на CORBA-IDL. Эти основы обеспечивают каноническое отображение типов данных, **пространства имен** и **исключений** метамодели в eODL. **Элементы операционного взаимодействия** моделей, соответствующих метамодели, отражаются на **операции** CORBA-IDL и атрибуты таким же каноническим образом.

A.3.2 Сигналы и переносимые значения

Одним из расширений метамодели по сравнению с концептуальным обоснованием ODL-МСЭ является введение **Элементов сигнального взаимодействия**. Эти взаимодействующие элементы основаны на определении **сигналов**. В eODL **сигналы** определяются на основании следующей грамматики:

<signal_dcl> ::= "signal" <identifier> "{" <member_list> "}"

<member_list> ::= <member>+

<member> ::= <type_spec> <declarators> ";"

A.3.3 Тип носителя, носитель и набор носителей

В концептуальном обосновании ODL-МСЭ семантика потока взаимодействий оставлена открытой. Метамодель строго определяет семантику взаимодействий типа **непрерывных носителей**, которая заменяет потоковое взаимодействие в ODL-МСЭ. Представление концепций **носителя**, **типа носителя** и набора **носителей**, определенных в рамках метамодели в eODL, дается следующей грамматикой:

<mediaset_dcl> ::= "mediaset" <identifier> "{" <member_list> "}"

<mediatype_dcl> ::= "mediatype" <identifier>

<medium_dcl> ::= "medium" <identifier> "(" <scoped_name> { "," <scoped_name> }* ")"

A.3.4 Типы интерфейса и элементы взаимодействия

Тип интерфейс в метамодели объединяет **элементы взаимодействия** различных видов взаимодействия в рамках простого контекста взаимодействия. С целью синтаксического представления этой концепции в дополнение к **элементам операционного взаимодействия** конструкция интерфейс расширяется понятием **источник, сток, производить** и **потреблять элементы взаимодействия**.

<interface> ::= <interface_dcl>

| <forward_dcl>

<interface_dcl> ::= <interface_header> "{" <interface_body> "}"

<forward_dcl> ::= ["abstract"] "interface" <identifier>

<interface_header> ::= ["abstract"] "interface" <identifier> [<interface_inheritance_spec>]

<interface_body> ::= <export> *

```
<export>                ::=  <type_dcl> ";"
                        |    <const_dcl> ";"
                        |    <except_dcl> ";"
                        |    <attr_dcl> ";"
                        |    <op_dcl> ";"
                        |     <produce_dcl> ";"
                        |    <consume_dcl> ";"
                        |     <source_dcl> ";"
                        |    <sink_dcl> ";"
<produce_dcl>           ::=  "produce" <scoped_name> <identifier>
<consume_dcl>           ::=  "consume" <scoped_name> <identifier>
<source_dcl>            ::=  "source" <scoped_name> <identifier>
<sink_dcl>              ::=  "sink" <scoped_name> <identifier>
```

A.3.5 Тип вычислительного объекта

Уже ODL-МСЭ разрешает использование понятия **типа вычислительного объекта** (**типа CO**). Вследствие точного определения представления конфигурации на **тип CO**, концепцию исходного интерфейса употреблять не рекомендуется. Грамматика **типов CO**, принятая в ODL-МСЭ, в eODL изменилась и определяется следующими правилами:

```
<object_template>       ::=  <object_template_header> "{" <object_template_export> "}"
<object_template_header>::=  "CO" <identifier> [ <object_inheritance_spec> ]
<object_inheritance_spec>::= ":" <scoped_name> { "," <scoped_name> }*
<object_template_export>::=  <object_export>*
<object_export>         ::=  <export>
                        |    <reqrd_interf_templates> ";"
                        |    <suptd_interf_templates> ";"
                        |    <use_dcl> ";"
                        |    <provide_dcl> ";"
                        |    <implements_dcl> ";"
                        |    <state_def_dcl> ";"
                        |    <constraint_dcl> ";"
                        |    <property_list>
<reqrd_interf_templates>::=  "requires" <scoped_name> { "," <scoped_name> }*
<supd_interf_templates> ::=  "supports" <scoped_name> { "," <scoped_name> }*
```

A.3.6 Свойство

Свойство используется для определения доступных или необходимых свойств элементов модели. Понятие свойства используется для определения целевой окружающей среды и модуля программного обеспечения.

```
<property_list>         ::=  { <property_dcl> ";" }*
<property_dcl>          ::=  "property" <property_name> "=" <property_value>
<property_name>         ::=  <identifier>
<property_value>        ::=  <simple_property_value>
                        |    <structured_property_value>
                        |    <sequence_property_value>
```

```
<simple_property_value>      ::=   <string_literal>
                             |     <integer_literal>
                             |     <boolean_literal>
<structured_property_value>  ::=   "{"  <property_assign>*  "}"
<sequence_property_value >   ::=   "["  <property_value>*  "]"
<property_assign>            ::=   <property_name> "="  <property_value> ";"
<constraint_dcl>             ::=   "constraint" <identifier> "{" <constraint_body> "}"
<constraint_body>            ::=   "language" "=" <string_literal> "body" "=" <string_literal> ";"
```

A.3.7 Внешний тип

Тип внешний используется идентификатором для обращения к типу внешне предоставляемых данных.

```
<extern_type>                ::=   "extern" "type" <identifier> <string_literal>
```

A.4 Представление конфигурации

A.4.1 Порты

Основной концепцией представления конфигурации на объектах **CO** является концепция **порта**. Для простых и динамических **портов** нотация определяется следующими правилами:

```
<object_export>              ::=   <export>
                             |     <reqrd_interf_templates> ";"
                             |     <suptd_interf_templates> ";"
                             |     <use_dcl> ";"
                             |     <provide_dcl> ";"
                             |     <implements_dcl> ";"
                             |     <state_def_dcl> ";"
                             |     <constraint_dcl> ";"
                             |     <property_list>
<use_dcl>                    ::=   "use" [ "multiple" ] <scoped_name> <identifier>
<provide_dcl>                ::=   "provide" [ "multiple" ] <scoped_name> <identifier>
```

A.5 Представление реализации

A.5.1 Артефакты и элементы реализации

Артефакты абстрагируют из конструкций конкретных языков программирования, реализующих поведение объектов **CO**. В eODL представление **артефактов** и **элементов реализаций** задается следующими правилами:

```
<artefact>                   ::=   <artefact_dcl>
                             |     <artefact_forward_dcl>
<artefact_forward_dcl>       ::=   "artefact"  <identifier>
<artefact_dcl>               ::=   <artefact_header> "{" <artefact_body> "}"
<artefact_header>            ::=   "artefact" <identifier> [ <artefact_inheritance_spec> ]
<artefact_inheritance_spec>  ::=   ":" <scoped_name> { "," <scoped_name> }*
<artefact_body>              ::=   <impl_elem_dcl>*
<impl_elem_dcl>              ::=   <identifier> "implements" <impl_case_dcl> <scoped_name> ";"
<impl_case_dcl>              ::=   "supply" | "use"
```

A.5.2 Отношения реализаций и стратегии конкретизации

Отношение **implemented_by** определяет, какой **артефакт** используется для осуществления поведения **типа CO**. В eODL это отношение определяется в контексте определений **типа CO**:

```
<object_export>            ::=  <export>
                           |    <reqrd_interf_templates> ";"
                           |    <suptd_interf_templates> ";"
                           |    <use_dcl> ";"
                           |    <provide_dcl> ";"
                           |    <implements_dcl> ";"
                           |    <state_def_dcl> ";"
                           |    <constraint_dcl> ";"
                           |    <property_list>
<implements_dcl>           ::=  "implemented" "by" <artefact_with_policy>
                                { "," <artefact_with_policy> }*
<artefact_with_policy>     ::=  <scoped_name> [ "with" <instantiation_policy_dcl> ]
<instantiation_policy_dcl> ::=  "ArtefactPool"
                           |    "ArtefactPerRequest"
                           |    "Singleton"
                           |    "UserDefined"
```

A.5.3 Типы состояний

Во многих случаях реализации осуществлению **артефакта** необходим доступ к информации о состоянии объектов **CO**, которые он реализует. Информация о состоянии объектов **CO** определяется типами **CO** в соответствии со следующими правилами:

```
<object_export>            ::=  <export>
                           |    <reqrd_interf_templates> ";"
                           |    <suptd_interf_templates> ";"
                           |    <use_dcl> ";"
                           |    <provide_dcl> ";"
                           |    <implements_dcl> ";"
                           |    <state_def_dcl> ";"
                           |    <constraint_dcl> ";"
                           |    <property_list>
<state_def_dcl>            ::=  "state" <scoped_name>
                                [ "provided" "to" "(" <provided_to_dcl> ")" ]
<provided_to_dcl>          ::=  <scoped_name> { "," <scoped_name> }*
```

A.6 Представление внедрения

A.6.1 Softwarecomponent

Конкретная реализация **типа CO** представляется определением программного модуля. В рамках программного модуля могут быть **представлены** множественные **типы CO**. Модули программного обеспечения могут зависеть от других модулей и/или требовать других свойств и сервисов от конечной окружающей среды исполнения.

```
<softwarecomponent_dcl>     ::=     <softwarecomponent_header>
                                    "{" <softwarecomponent_body> "}"
<softwarecomponent_header>          ::=     "softwarecomponent"  <identifier>
                                            "realizes"  <cotype_identifier_list>
<cotype_identifier_list>    ::=     <cotype_identifier> { ","  <cotype_identifier> }*
<softwarecomponent_body>    ::=     <softwarecomponent_stmt>*
<softwarecomponent_stmt>    ::=     "dependent"  "{"  <softwarecomponent_list>  "}" ";"
                            |       "requires"  "{"  <property_list>  "}"  ";"
<softwarecomponent_list>    ::=     <softwarecomponent_identifier>
                                    {  ","  <softwarecomponent_identifier>  }*
<softwarecomponent_identifier>      ::=     <scoped_name>
```

A.6.2 Сборка

Сборка описывает набор компонентов, соединенных между собой, и не имеет отношения к конкретному распределению в распределенной окружающей среде обработки.

A.6.2.1 Определение сборки

Определение **сборки** содержит определения всех определений набора экземпляров, принадлежащих определениям **сборки** и **соединения** экземпляров в **сборке**.

```
<assembly_dcl>              ::=     <assembly_header> "{"  <assembly_body>  "}"
<assembly_header>           ::=     "assembly"  <identifier>
<assembly_body>             ::=     <assembly_stmt>*
<assembly_stmt>             ::=     <instance_set_dcl> ";"
                            |       <connect_dcl> ";"
                            |       <constraint_dcl> ";"
                            |       <property_list>
```

A.6.2.2 Определение множества экземпляров

Определение множества экземпляров описывает непустое множество экземпляров **типа CO**.

```
<instance_set_dcl>          ::=     <identifier> [ "(" <integer_literal> ")" ] ":" <cotype_identifier>
<cotype_identifier>         ::=     <scoped_name>
```

A.6.2.3 Определение соединения

Соединения между экземплярами и наборы экземпляров выражаются в определении соединения. Здесь **порты** соединяются между собой в соответствии с определением **типа CO**, причем один порт действует как **исток**, а другой как **сток**.

```
<connect_dcl>               ::=     "connect"  [ <identifier> ]  "{"  <connection_list>  "}"
<connection_list>           ::=     { <connection> ";" } +
<connection>                ::=     <instance_set_identifier>  "."  <port_identifier> "="
                                    <instance_set_identifier>  "."  <port_identifier>
<instance_set_identifier>   ::=     <scoped_name>
<port_identifier>           ::=     <scoped_name>
```

A.6.3 Определение отображения инсталляции

Определение отображения инсталляции описывает присвоение **типов CO** узлам целевой окружающей среды. В последующей операции инсталляции можно обращаться к определению отображения инсталляции и дать начало инсталляции модулей программного обеспечения в **узлах**.

```
<installation_map_dcl>        ::=  <installation_map_header>  "{" <installation_map_body> "}"
<installation_map_header>     ::=  "installation" <identifier>
                                   "uses"  "environment"  <environment_identifier>
<installation_map_body>       ::=  <install_stmt> *
<install_stmt>                ::=  <softwarecomponent_identifier>  "->" <node_identifier> ";"
<environment_identifier>      ::=  <scoped_name>
```

A.6.4 Определение отображения конкретизации

Определение отображения конкретизации описывает конкретное присвоение множества экземпляров **узлам** указанной целевой окружающей среды и **сборке**.

```
<instantiation_map_dcl>       ::=  <instantiation_map_header> "{"  <instantiation_map_body>  "}"
<instantiation_map_header>    ::=  "instantiation" <identifier> <instantiation_map_header_env>
                                   <instantiation_map_header_ass>
<instantiation_map_header_env>    ::=  "uses"  "environment"  <environment_identifier>
<instantiation_map_header_ass>    ::=  "uses"  "assembly"  <assembly_identifier>
<assembly_identifier>         ::=  <scoped_name>
<instantiation_map_body>      ::=  <assign_instance_stmt>*
<assign_instance_stmt>        ::=  <instance_set_identifier_list>  "->"  <node_identifier>  ";"
<instance_set_identifier_list> ::=  <instance_set_identifier> { "," <instance_set_identifier> }*
```

A.6.5 Операция внедрения

Операция внедрения является последовательностью операций инсталляции и конкретизации, которые должны исполняться в течение **внедрения**.

```
<deployment_action>           ::=  "deploy"  "{"  <deployment_body>  "}"  ";"
<deployment_body>             ::=  "install"  "{"  <install_list>  "}" ";"
                                   "instantiate"  "{"  <instantiation_list>  "}"  ";"
```

A.6.5.1 Операция инсталляции

Операция инсталляции устанавливает инсталляцию модуля программного обеспечения в узле исполнения целевой окружающей среды.

```
<install_list>                ::=  <install_member>*
<install_member>              ::=  <installation_map_identifier>  ";"
                               |   <qualified_install_stmt>
<qualified_install_stmt>      ::=  <softwarecomponent_identifier>  "->"
                                   <environment_identifier>  "."  <node_identifier>  ";"
<installation_map_identifier> ::=  <scoped_name>
```

A.6.5.2 Операция конкретизации

Операция конкретизации устанавливает конкретизацию множества **CO** в узле исполнения целевой окружающей среды.

```
<instantiation_list>          ::=  <instantiation_member>*
```

```
<instantiation_member>          ::=  <instantiation_map_identifier> ";"
                                |    <qualified_assign_instance_stmt> ";"
<instantiation_map_identifier>  ::=  <identifier>
<qualified_assign_instance_stmt> ::= <assembly_identifier> "." <instance_set_identifier>
                                     "->" <environment_identifier> "." <node_identifier>
```

A.7 Целевая окружающая среда

Целевая окружающая среда служит возможной исполнительной окружающей средой для **сборок**. Она отображает структуру и свойства этой окружающей среды. Текстовый синтаксис eODL может содержать более чем одну спецификацию целевой окружающей среды.

A.7.1 Определение окружающей среды

Определение окружающей среды описывает возможную исполнительную окружающую среду в терминах доступных **узлов** и **линий** связи.

```
<environment_dcl>       ::=  <environment_header> "{" <environment_body> "}"
<environment_header>    ::=  "environment" <identifier>
<environment_body>      ::=  <environment_stmt>+
<environment_stmt>      ::=  <node_dcl> ";"
                        |    <link_dcl> ";"
```

A.7.2 Определение узла

Определение узла отражает идентифицируемый исполнительный узел в целевой окружающей среде, которая может быть целевым объектом для инсталляции **типов CO** и конкретизации множества экземпляров. Свойства в определении узла характеризуют средства исполнительного узла.

```
<node_dcl>              ::=  "node" <identifier> "{" <property_list> "}"
```

A.7.3 Определение линии

Линии связи между исполнительными узлами в целевой окружающей среде представляются как определения линий. Свойства в определении линии относятся к характеристикам и виду линии связи.

```
<link_dcl>              ::=  <link_header>  "{" <link_body> "}"
<link_header>           ::=  "link" <identifier>
<link_body>             ::=  "node" <node_list> ";" <property_list> ";"
<node_list>             ::=  <node_identifier> { "," <node_identifier> }*
<node_identifier>       ::=  <scoped_name>
```

A.8 Синтаксис eODL

В данном параграфе дается полное множество порождающих правил eODL. Оно также включает все правила, наследованные от базового синтаксиса OMG IDL 2.4.2.

```
<specification>         ::=  <definition>+ [ <deployment_action> ]
<definition>            ::=  <type_dcl> ";"
                        |    <const_dcl> ";"
                        |    <except_dcl> ";"
                        |    <interface> ";"
                        |    <object_template> ";"
                        |    <artefact> ";"
```

```
                    |       <module> ";"
                    |        <value> ";"
                    |       <signal_dcl> ";"
                    |       <mediaset_dcl> ";"
                    |       <mediatype_dcl> ";"
                    |       <medium_dcl> ";"
                    |       <assembly_dcl> ";"
                    |       <softwarecomponent_dcl> ";"
                    |       <environment_dcl> ";"
                    |       <installation_map_dcl> ";"
                    |       <instantiation_map_dcl> ";"
<module>            ::=     "module" <identifier> "{" <definition> + "}"
<object_template>           ::=   <object_template_header> "{" <object_template_export> "}"
<object_template_header>    ::=   "CO" <identifier> [ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=   ":" <scoped_name> { "," <scoped_name> }*
<object_template_export>    ::=   <object_export>*
<object_export>     ::=     <export>
                    |       <reqrd_interf_templates> ";"
                    |       <suptd_interf_templates> ";"
                    |       <use_dcl> ";"
                    |       <provide_dcl> ";"
                    |       <implements_dcl> ";"
                    |       <state_def_dcl> ";"
                    |       <constraint_dcl> ";"
                    |       <property_list>

<reqrd_interf_templates>    ::=   "requires" <scoped_name> { "," <scoped_name> }*
<supd_interf_templates>     ::=   "supports" <scoped_name> { "," <scoped_name> }*
<use_dcl>       ::= "use" [ "multiple" ] <scoped_name> <identifier>
<provide_dcl>   ::= "provide" [ "multiple" ] <scoped_name> <identifier>
<artefact>      ::= <artefact_dcl>
                |   <artefact_forward_dcl>
<artefact_forward_dcl>      ::=   "artefact"  <identifier>
<artefact_dcl>  ::=         <artefact_header> "{" <artefact_body> "}"
<artefact_header>           ::=   "artefact" <identifier> [ <artefact_inheritance_spec> ]
<artefact_inheritance_spec>         ::=     ":" <scoped_name> { "," <scoped_name> }*
<artefact_body>     ::=     <impl_elem_dcl>*
<impl_elem_dcl>     ::=     <identifier> "implements" <impl_case_dcl> <scoped_name> ";"
<impl_case_dcl>     ::=     "supply" | "use"
<implements_dcl>    ::=     "implemented" "by" <artefact_with_policy>
                            { "," <artefact_with_policy> }*
<artefact_with_policy>      ::=   <scoped_name> [ "with" <instantiation_policy_dcl> ]
```

```
<instantiation_policy_dcl>     ::=     "ArtefactPool"
                                |       "ArtefactPerRequest"
                                |       "Singleton"
                                |       "UserDefined"
<state_def_dcl>        ::=      "state" <scoped_name> [ "provided" "to" "(" <provided_to_dcl> ")" ]
<provided_to_dcl>      ::=      <scoped_name> { "," <scoped_name> }*
<interface>        ::=  <interface_dcl>
                    |   <forward_dcl>
<interface_dcl>        ::=      <interface_header> "{" <interface_body> "}"
<forward_dcl>      ::=  [ "abstract" ] "interface" <identifier>
<interface_header>     ::=      [ "abstract" ] "interface"  <identifier> [ <interface_inheritance_spec> ]
<interface_body>       ::=      <export> *
<export>       ::=  <type_dcl> ";"
                |   <const_dcl> ";"
                |   <except_dcl> ";"`
                |   <attr_dcl> ";"
                |   <op_dcl> ";"
                |   <produce_dcl> ";"
                |   <consume_dcl> ";"
                |   <source_dcl> ";"
                |   <sink_dcl> ";"
<produce_dcl>          ::=      "produce" <scoped_name> <identifier>
<consume_dcl>          ::=      "consume" <scoped_name> <identifier>
<source_dcl>       ::=  "source" <scoped_name> <identifier>
<sink_dcl>         ::=  "sink" <scoped_name> <identifier>
<interface_inheritance_spec> ::=    ":" <interface_name> { "," <interface_name> } *
<interface_name>               ::=  <scoped_name>
<scoped_name>                  ::=  <identifier>
                                |   "::" <identifier>
                                |   <scoped_name> "::" <identifier>
<signal_dcl>       ::=  "signal" <identifier> "{" <member_list> "}"
<mediaset_dcl>         ::=      "mediaset" <identifier> "{" <member_list> "}"
<mediatype_dcl>        ::=      "mediatype" <identifier>
<medium_dcl>           ::=      "medium" <identifier> "(" <scoped_name> { "," <scoped_name> }* ")"
<value>        ::=     ( <value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)
<value_forward_dcl> ::=        [ "abstract" ] "valuetype" <identifier>
<value_box_dcl>        ::=      "valuetype" <identifier> <type_spec>
<value_abs_dcl>        ::=      "abstract"  "valuetype"  <identifier>  [  <value_inheritance_spec>  ]
                                "{" <export>* "}"
<value_dcl>        ::=  <value_header> "{" <value_element>* "}"
<value_header> ::=              ["custom" ] "valuetype" <identifier> [ <value_inheritance_spec> ]
<value_inheritance_spec>       ::=      [ ":" [ "truncatable" ] <value_name>
```

```
                        { "," <value_name> }* ]
                        [ "supports" <interface_name>
                        { "," <interface_name> }* ]
<value_name>    ::=  <scoped_name>
<value_element>     ::=     <export> | <state_member> | <init_dcl>
<state_member>      ::=     ( "public" | "private" ) <type_spec> <declarators> ";"
<init_dcl>    ::=  "factory" <identifier> "(" [ <init_param_decls> ] ")" ";"
<init_param_decls>  ::=     <init_param_decl> { "," <init_param_decl> }
<init_param_decl>   ::=     <init_param_attribute> <param_type_spec> <simple_declarator>
<init_param_attribute>      ::=     "in"
<const_dcl>     ::=         "const" <const_type> <identifier> "=" <const_exp>
<const_type>    ::=         <integer_type>
                |           <char_type>
                |           <wide_char_type>
                |           <boolean_type>
                |           <floating_pt_type>
                |           <string_type>
                |           <wide_string_type>
                |           <fixed_pt_const_type>
                |           <scoped_name>
                |           <octet_type>
<const_exp>     ::=         <or_expr>
<or_expr>       ::=         <xor_expr> | <or_expr> "|" <xor_expr>
<xor_expr>      ::=         <and_expr> | <xor_expr> "^" <and_expr>
<and_expr>      ::=         <shift_expr> | <and_expr> "&" <shift_expr>
<shift_expr>    ::=         <add_expr>
                |           <shift_expr> ">>" <add_expr>
                |           <shift_expr> "<<"<add_expr>
<add_expr>      ::=         <mult_expr>
                |           <add_expr> "+" <mult_expr>
                |           <add_expr> "-" <mult_expr>
<mult_expr>     ::=         <unary_expr>
                |           <mult_expr> "*" <unary_expr>
                |           <mult_expr> "/" <unary_expr>
                |           <mult_expr> "%" <unary_expr>
<unary_expr>    ::=         <unary_operator> <primary_expr> | <primary_expr>
<unary_operator>            ::=     "-"|"+"|"~"
<primary_expr>              ::=     <scoped_name> | <literal> | "(" <const_exp> ")"
<literal>       ::=         <integer_literal>
                |   <string_literal>
                |   <wide_string_literal>
                |   <character_literal>
```

```
                | <wide_character_literal>
                | <fixed_pt_literal>
                | <floating_pt_literal>
                | <boolean_literal>
<boolean_literal>     ::=       "TRUE"|"FALSE"
<positive_int_const> ::=        <const_exp>
<type_dcl>  ::=       "typedef" <type_declarator>
              | <struct_type>
              | <union_type>
              | <enum_type>
              | "native" <simple_declarator>
<type_declarator>     ::=       <type_spec> <declarators>
<type_spec>     ::= <simple_type_spec>
                | <constr_type_spec>
                | <extern_type>
<extern_type>         ::=       "extern" "type" <identifier> <string_literal>
<simple_type_spec> ::=        <base_type_spec>
                      |       <template_type_spec>
                      |       <scoped_name>
<base_type_spec>   ::=        <floating_pt_type>
                      |       <integer_type>
                      |       <char_type>
                      |       <wide_char_type>
                      |       <boolean_type>
                      |       <octet_type>
                      |       <any_type>
                      |       <object_type>
                      |       <value_base_type>
<template_type_spec>          ::=     <sequence_type>
                                |       <string_type>
                                |       <wide_string_type>
                                |       <fixed_pt_type>
<constr_type_spec> ::=        <struct_type>
                      |       <union_type>
                      |       <enum_type>
<declarators>    ::= <declarator> { "," <declarator> } *
<declarator>     ::= <simple_declarator> | <complex_declarator>
<simple_declarator> ::=       <identifier>
<complex_declarator>          ::=     <array_declarator>
<floating_pt_type>   ::=        "float"
                      |       "double"
```

```
                    |       "long" "double"
<integer_type>  ::= <signed_int> | <unsigned_int>
<signed_int>    ::= <signed_short_int>
                  | <signed_long_int>
                  | <signed_longlong_int>
<signed_short_int>  ::=     "short"
<signed_long_int>   ::=     "long"
<signed_longlong_int>       ::=     "long" "long"
<unsigned_int>      ::=     <unsigned_short_int>
                    |       <unsigned_long_int>
                    |       <unsigned_longlong_int>
<unsigned_short_int>        ::=     "unsigned" "short"
<unsigned_long_int>         ::=     "unsigned" "long"
<unsigned_longlong_int>             ::=     "unsigned" "long" "long"
<char_type>     ::= "char"
<wide_char_type>    ::=     "wchar"
<boolean_type>      ::=     "boolean"
<octet_type>    ::= "octet"
<any_type>      ::= "any"
<object_type>   ::= "Object"
<struct_type>   ::= "struct" <identifier> "{" <member_list> "}"
<member_list>   ::= <member>+
<member>   ::= <type_spec> <declarators> ";"
<union_type>    ::= "union" <identifier> "switch" "(" <switch_type_spec> ")" "{" <switch_body> "}"
<switch_type_spec>          ::=     <integer_type>
                            |       <char_type>
                            |       <boolean_type>
                            |       <enum_type>
                            |       <scoped_name>
<switch_body>               ::=     <case>+
<case>          ::=         <case_label>+ <element_spec> ";"
<case_label>    ::=         "case" <const_exp> ":"|"default" ":"
<element_spec> ::=          <type_spec> <declarator>
<enum_type>     ::=         "enum" <identifier> "{" <enumerator> { "," <enumerator> } * "}"
<enumerator>    ::=         <identifier>
<sequence_type>             ::=     "sequence" "<" <simple_type_spec> ","
                            <positive_int_const> ">" | "sequence" "<" <simple_type_spec> ">"
<string_type>   ::= "string" "<" <positive_int_const> ">" | "string"
<wide_string_type>  ::=     "wstring" "<" <positive_int_const> ">" | "wstring"
<array_declarator>  ::=     <identifier> <fixed_array_size>+
<fixed_array_size>  ::=     "[" <positive_int_const> "]"
```

```
<attr_dcl>    ::=       [ "readonly" ] "attribute"
                        <param_type_spec> <simple_declarator> { "," <simple_declarator> }*
<except_dcl>     ::=  "exception" <identifier> "{" <member>* "}"
<op_dcl>     ::=  [ <op_attribute> ] <op_type_spec>
                  <identifier> <parameter_dcls> [ <raises_expr> ] [ <context_expr> ]
<op_attribute>         ::=       "oneway"
<op_type_spec>      ::=       <param_type_spec> |"void"
<parameter_dcls>    ::=       "(" <param_dcl> { "," <param_dcl> } * ")"|"(" ")"
<param_dcl>     ::=  <param_attribute> <param_type_spec> <simple_declarator>
<param_attribute>   ::=       "in"|"out"|"inout"
<raises_expr>   ::=  "raises" "(" <scoped_name>{ "," <scoped_name> } * ")"
<context_expr>  ::=       "context" "(" <string_literal>{ "," <string_literal> } * ")"
<param_type_spec> ::=     <base_type_spec>
                        |       <string_type>
                        |       <wide_string_type>
                        |       <scoped_name>
<fixed_pt_type>       ::=       "fixed" "<" <positive_int_const> ","<positive_int_const> ">"
<fixed_pt_const_type>         ::=     "fixed"
<value_base_type>   ::=       "ValueBase"
<assembly_dcl> ::=  <assembly_header> "{"  <assembly_body>  "}"
<assembly_header> ::=     "assembly"  <identifier>
<assembly_body>   ::=     <assembly_stmt>*
<assembly_stmt>   ::=     <instance_set_dcl> ";"
                        |       <connect_dcl> ";"
                        |       <constraint_dcl> ";"
                        |       <property_list>
<instance_set_dcl>  ::=       <identifier> [ "(" <integer_literal> ")" ] ":" <cotype_identifier>
<cotype_identifier>  ::=      <scoped_name>
<connect_dcl>       ::=       "connect"  [ <identifier> ]  "{"  <connection_list>  "}"
<connection_list>   ::=       { <connection> ";" } +
<connection>        ::=       <instance_set_identifier>  "."  <port_identifier>
                              "=" <instance_set_identifier>  "."  <port_identifier>
<instance_set_identifier>       ::=     <scoped_name>
<port_identifier>     ::=       <scoped_name>
<softwarecomponent_dcl>     ::=     <softwarecomponent_header>
                                    "{" <softwarecomponent_body> "}"
<softwarecomponent_header>        ::=       "softwarecomponent"  <identifier>
                                            "realizes"  <cotype_identifier_list>
<cotype_identifier_list>        ::=     <cotype_identifier> { ","  <cotype_identifier> }*
<softwarecomponent_body>  ::=     <softwarecomponent_stmt>*
<softwarecomponent_stmt>  ::=     "dependent"  "{"  <softwarecomponent_list>  "}" ";"
```

```
                                |       "requires"  "{"  <property_list>  "}"  ";"
<softwarecomponent_list>        ::=     <softwarecomponent_identifier>
                                        {  ","  <softwarecomponent_identifier>  }*
<softwarecomponent_identifier>          ::=         <scoped_name>
<environment_dcl>  ::=          <environment_header> "{"  <environment_body>  "}"
<environment_header>            ::=     "environment"  <identifier>
<environment_body> ::=          <environment_stmt>+
<environment_stmt>  ::=         <node_dcl>  ";"
                        |       <link_dcl>  ";"
<node_dcl> ::=          "node"  <identifier>  "{"  <property_list>  "}"
<property_list>         ::=     { <property_dcl> ";" }*
<property_dcl>          ::=     <property_name>  "="  <property_value>
<property_name>         ::=     <identifier>
<property_value>        ::=     <simple_property_value>
                        |       <structured_property_value>
                        |       <sequence_property_value>
<simple_property_value>         ::=     <string_literal>
                        |       <integer_literal>
                        |       <boolean_literal>
<structured_property_value>  ::=    "{"  <property_assign>*  "}"
<sequence_property_value>  ::=      "["  <property_value>*  "]"
<constraint_dcl>        ::=     "constraint" <identifier> "{" <constraint_body> "}"
<constraint_body>       ::=     "language" "=" <string_literal> "body" "=" <string_literal> ";"
<property_assign>       ::=     <property_name> "="  <property_value> ";"
<link_dcl>          ::=  <link_header>  "{"  <link_body>  "}"
<link_header>       ::=  "link"  <identifier>
<link_body>         ::=  "node" <node_list>  ";" <property_list> ";"
<node_list>         ::=  <node_identifier>  {  ","  <node_identifier>  }*
<node_identifier>       ::=     <scoped_name>
<installation_map_dcl>          ::=     <installation_map_header>  "{" <installation_map_body> "}"
<installation_map_header>       ::=     "installation" <identifier>
                                        "uses" "environment" <environment_identifier>
<installation_map_body>         ::=     <install_stmt>*
<install_stmt>                  ::=     <softwarecomponent_identifier> "->" <node_identifier> ";"
<environment_identifier>        ::=     <scoped_name>
<instantiation_map_dcl>         ::=     <instantiation_map_header> "{"  <instantiation_map_body>  "}"
<instantiation_map_header>  ::=     "instantiation" <identifier>
                                        <instantiation_map_header_env>
                                        <instantiation_map_header_ass>
<instantiation_map_header_env>          ::=         "uses"  "environment"  <environment_identifier>
<instantiation_map_header_ass>          ::=         "uses" "assembly" <assembly_identifier>
<assembly_identifier>           ::=     <scoped_name>
```

```
<instantiation_map_body>     ::=    <assign_instance_stmt>*
<assign_instance_stmt>       ::=    <instance_set_identifier_list> "->" <node_identifier> ";"
<instance_set_identifier_list> ::=  <instance_set_identifier> { "," <instance_set_identifier> }*
<deployment_action> ::=      "deploy" "{" <deployment_body> "}" ";"
<deployment_body>  ::=       "install" "{" <install_list> "}" ";"
                             "instantiate" "{" <instantiation_list> "}" ";"
<install_list>     ::= <install_member>*
<install_member>   ::=       <installation_map_identifier>  ";"
                   |         <qualified_install_stmt>
<qualified_install_stmt>     ::=    <softwarecomponent_identifier>  "->"
                                    <environment_identifier>  "."  <node_identifier>  ";"
<installation_map_identifier> ::=   <scoped_name>
<instantiation_list>  ::=    <instantiation_member>*
<instantiation_member>       ::=    <instantiation_map_identifier>  ";"
                             |      <qualified_assign_instance_stmt> ";"
<instantiation_map_identifier>       ::=       <identifier>
<qualified_assign_instance_stmt>  ::=       <assembly_identifier>  "."
                                            <instance_set_identifier>  "->"
                                            <environment_identifier>  "."  <node_identifier>
```

Приложение B

Отображение метамодели на синтаксис

B.1 Введение

В данном приложении описываются соотношения между **метамоделью** eODL и конкретным текстовым синтаксисом eODL, определенным в Приложении A. Описание ограничивается теми концепциями **метамодели**, которые являются расширениями **метамодели** CORBA. Соотношения между текстовым синтаксисом OMG IDL 2.4.2 и **метамоделью** OMG CORBA вполне определены группой OMG, а поэтому здесь не повторяются.

В метамодели, определяемой в параграфе 5, используется графическая нотация. Соответствующий буквенно-цифровой синтаксис приводится под графиком в рамке с последующим текстовым пояснением.

B.2 Сигнал и параметр сигнала

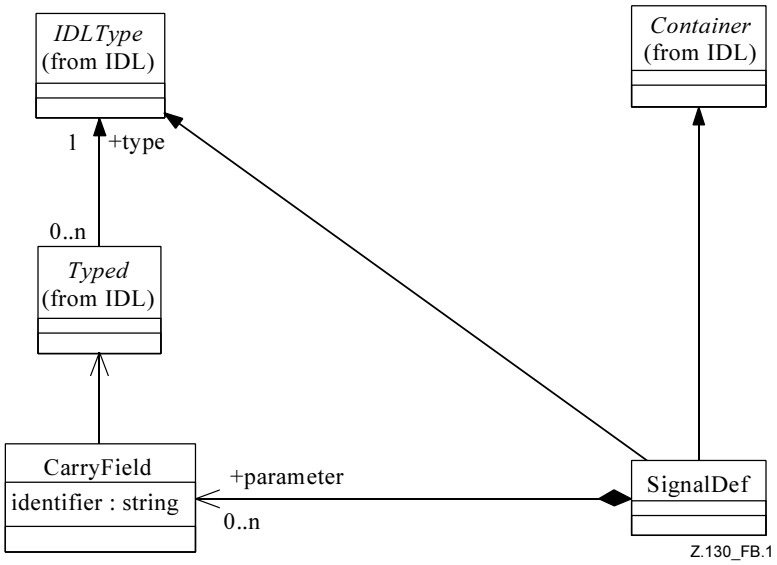


Рисунок B.1/Z.130 – Сигнал и параметр сигнала

В конкретном синтаксисе **(1)**, **(2)** и **(3)** отображаются на элементы *SignalDef*/*CarryField* модели (см. рис. B.1).

(1) **<signal_dcl>** **::=** **"signal" <identifier> "{" <member_list> "}"**

(2) **<member_list>** **::=** **<member>+**

(3) **<member>** **::=** **<type_spec> <declarators> ";"**

<identifier> из продукции **(1)** является именем элемента *SignalDef* модели. В пределах **<member_list>** **(2)** перечисляются все элементы *CarryField*, которые принимают участие в отношении **параметр** этого *SignalDef*. Продукции **<member>** **(3)** из конкретного синтаксиса отражаются как элементы *CarryField* модели. Здесь **<type_spec>** являются типами модели, ограниченными посредством концепции Typed из IDL. Для каждого оператора объявления в **<declarators>** существует элемент *CarryField* модели.

B.3 Тип носителя, носитель, множество носителей

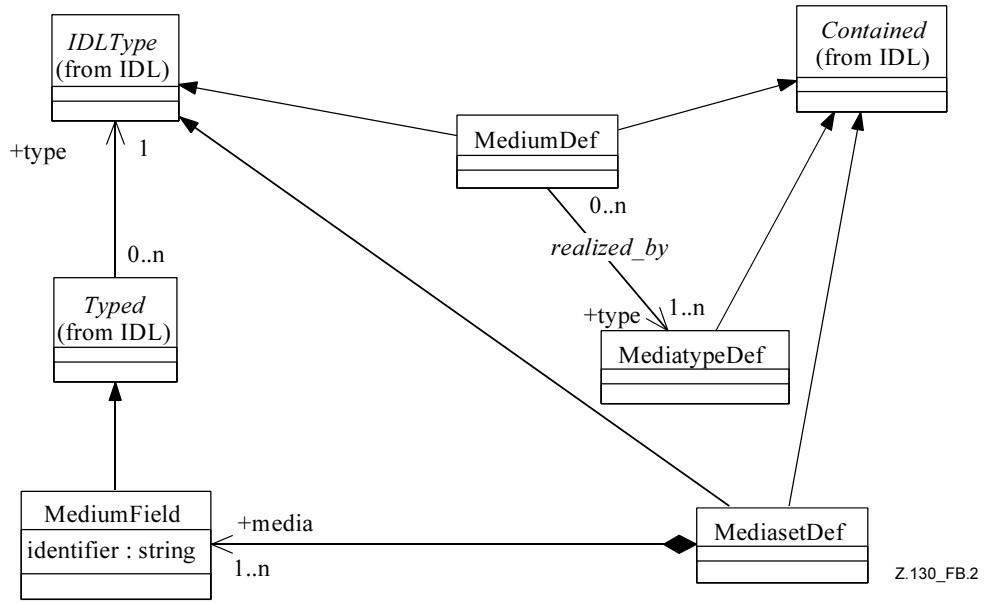


Рисунок B.2/Z.130 – Носитель, тип носителя, множество носителей

В конкретном синтаксисе продукции **(4)**, **(5)** и **(6)** отображаются на элементы *MediasetDef*, *MediatypeDef* и *MediumDef* модели (см. рис. B.2).

(4) **<mediaset_dcl> ::=** **"mediaset" <identifier> "{" <member_list> "}"**

(5) **<mediatype_dcl>::=** **"mediatype" <identifier>**

(6) **<medium_dcl> ::=** **"medium" <identifier>**
"(" <scoped_name> { "," <scoped_name> }* ")"

Продукция **<mediatype_dcl> (5)** представляется в модели элементом *MediatypeDef*; **<identifier>** является именем для концепции Named этого элемента. С продукцией **<medium_dcl>** конкретный синтаксис выражает элементы *MediumDef*; причем снова **<identifier>** является именем концепции Named. **<scoped_name>**, перечисленные в продукции **(6)**, должны всегда относиться к *MediatypeDef* и представляться как отношение *realized_by* к элементу *MediumDef* модели. Согласно продукции **(4) <mediaset_dcl>** представляется в модели как элемент *MediasetDef* под именем **<identifier>**. В пределах **<member_list>** в **(4)** перечисляются все элементы *MediumField*, принимающие участие в соотношении носителей этого *MediasetDef*. Относительно **<member>** из **<member_list>** продукции **(4)**, **<type_spec>** должен обращаться к *MediatypeDef*, а все операторы объявления в **<declarators>** должны быть простыми.

B.4 Поглощать и порождать

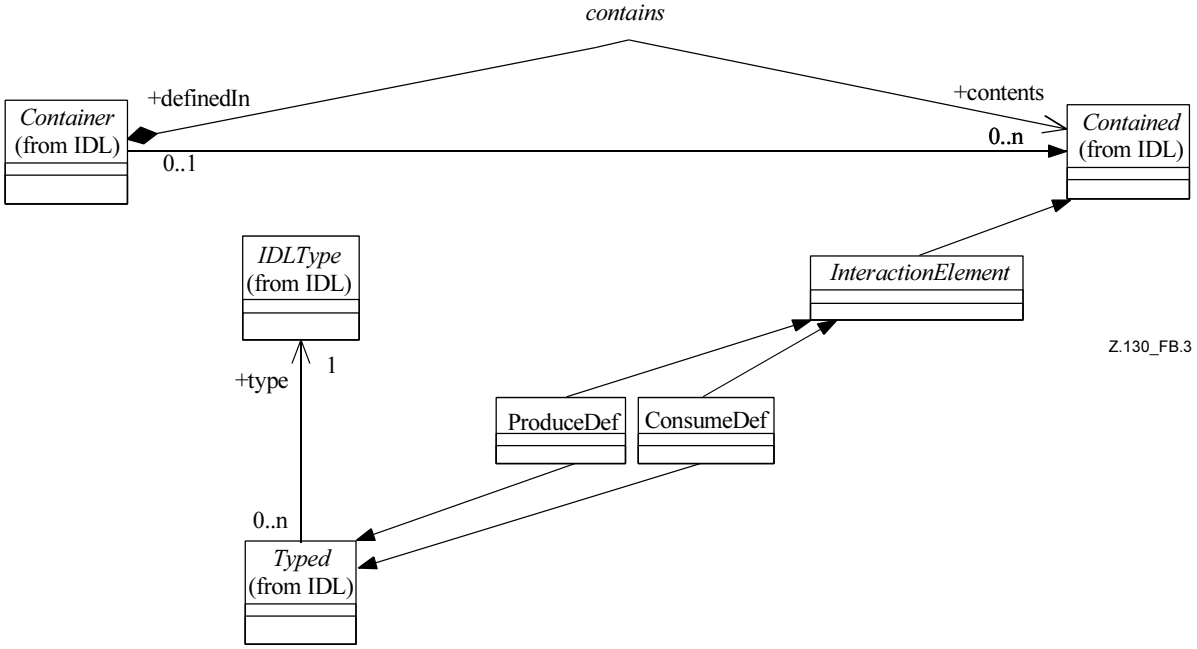


Рисунок B.3/Z.130 – Поглощать и порождать

В конкретном синтаксисе продукции **(7)** и **(8)** отображаются на элементы *ProduceDef/ConsumeDef* модели (см. рис. B.3).

(7) <produce_dcl> ::= "produce" <scoped_name> <identifier>

(8) <consume_dcl> ::= "consume" <scoped_name> <identifier>

В обоих объявлениях **<scoped_name>** обращается к *SignalDef*. Это отношение отображается в модели как отношение *Typed/IDLType*, где *ProduceDef/ConsumeDef* вовлекается через наследование, а *IDLType* в соответствии с **<signal_dcl>** является *SignalDef*. Оба *ProduceDef/ConsumeDef* являются концепциями Named из **метамодели**, а **<identifier>** в **(7)**/**(8)** является отображением элементов модели на атрибут имени.

B.5 Сток и исток

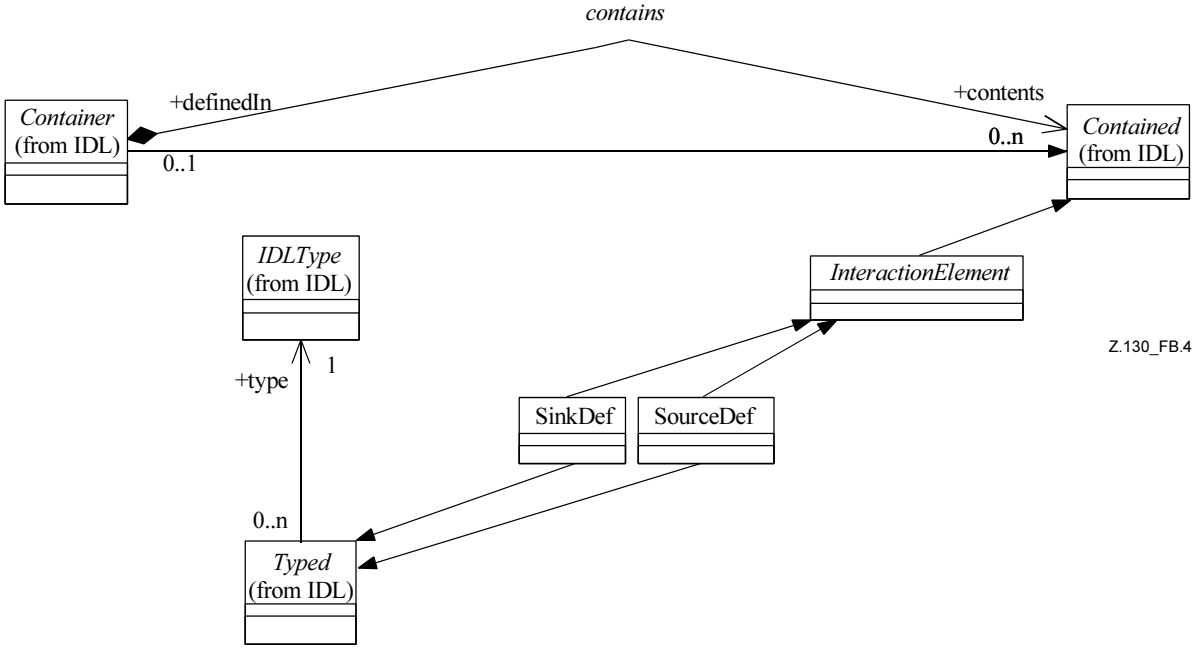


Рисунок B.4/Z.130 – Сток и исток

В конкретном синтаксисе продукции **(10)** и **(9)** отображаются на элементы *SinkDef/SourceDef* модели (см. рис. B.4).

(9) **<source_dcl>** **::=** **"source" <scoped_name> <identifier>**

(10) **<sink_dcl>** **::=** **"sink" <scoped_name> <identifier>**

В обоих объявлениях **<scoped_name>** относится к *MediasetDef*. Это отношение отображается в модели в виде отношения *Typed/IDLType*, где *SinkDef/SourceDef* вовлекается через наследование, а *IDLType* в соответствии с **<mediaset_dcl>** является *MediasetDef*. Оба *SinkDef/SourceDef* являются концепциями Named из **метамодели**, а **<identifier>** в **(10)/(9)** является отображением элементов модели на атрибут имени.

B.6 Тип интерфейса

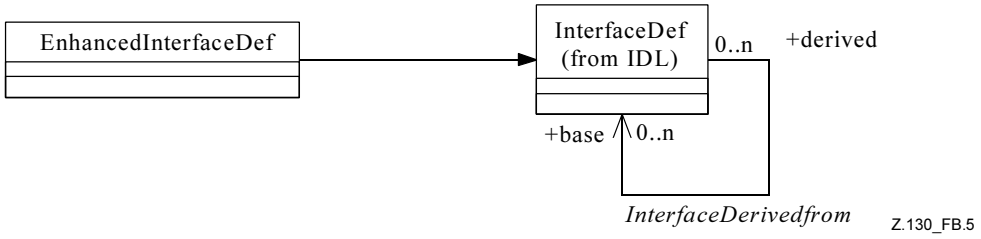


Рисунок B.5/Z.130 – Тип интерфейса

(11) **<interface_dcl>** **::=** **<interface_header> "{" <interface_body> "}"**

(12) **<interface_body>** **::=** **<export> ***

(13) **<export>** **::=** **<type_dcl> ";"**

| **<const_dcl> ";"**

| **<except_dcl> ";"**

| **<attr_dcl> ";"**

| **<op_dcl> ";"**

| **<produce_dcl> ";"**

| **<consume_dcl> ";"**

| **<source_dcl> ";"**

| **<sink_dcl> ";"**

В конкретном синтаксисе продукция **<interface_dcl> (11)** отображается на элемент *EnhancedInterfaceDef* модели. С **<interface_body> (12)** обращаются таким же образом, как и в IDL. По сравнению с *InterfaceDef* **<export> (13)** также разрешает **<produce_dcl>**, **<consume_dcl>**, **<source_dcl>** и **<sink_dcl>** в качестве содержащихся элементов. Для этого вида объявлений отображение определяется выше. Если **<interface_body>** не содержит новых элементов этого вида, то **<interface_dcl>** отражается на простой *InterfaceDef* (см. рис. B.5).

B.7 Типы CO, поддерживать и требовать

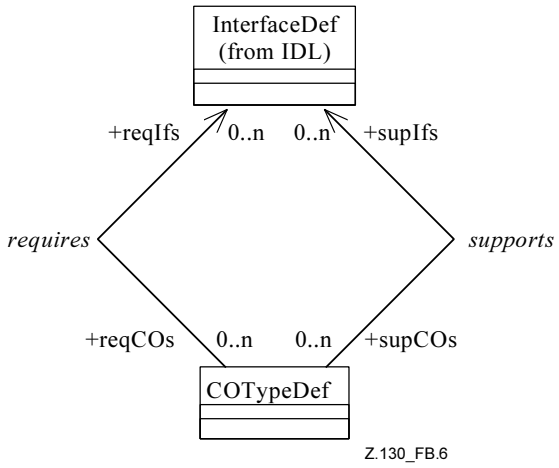


Рисунок B.6/Z.130 – Типы CO, поддерживать и требовать

В конкретном синтаксисе продукции **(14)**, **(15)** и **(16)** отображаются на элементы *COTypeDef* модели (см. рис. B.6).

(14) **<object_template>** **::=** **<object_template_header> "{" <object_template_export> "}"**

(15) **<object_template_header>** **::=** **"CO" <identifier> [<object_inheritance_spec>]**

(16) **<object_inheritance_spec>** **::=** **":" <scoped_name> { "," <scoped_name> }***

(17) **<object_template_export>** **::=** **<object_export>***

(18) **<object_export>** **::=** **<export>**

| **<reqrd_interf_templates> ";"**

| **<suptd_interf_templates> ";"**

| **<use_dcl> ";"**

| **<provide_dcl> ";"**

| **<implements_dcl> ";"**

| **<state_def_dcl> ";"**

| **<constraint_dcl> ";"**

| **<property_list>**

(19) **<reqrd_interf_templates>** **::=** **"requires" <scoped_name> { "," <scoped_name> }***

(20) **<supd_interf_templates>** **::=** **"supports" <scoped_name> { "," <scoped_name> }***

Продукции **(14)**, **(15)** и **(16)** отображают на элемент *COTypeDef* модели, где **<identifier>** из продукции **(15)** является именем концепции Named элемента *COTypeDef*. Как и конкретизация *InterfaceDef* в продукции **(16)**, перечисляются все *COTypeDef*, находящиеся в отношении наследования с текущим *COTypeDef*. Продукции **(17)** и **(18)** выражают концепцию емкости для этого *COTypeDef*. С использованием в **(18)** **<export>** обращаются таким же образом, как с *InterfaceDef* в IDL. Кроме того, **<reqrd_interf_templates>** и **<supd_interf_templates>** являются разрешенными содержащимися элементами для *COTypeDef*. В продукциях **(19)** и **(20)** символ **<scoped_name>** должен обращаться к элементам *InterfaceDef* или *EnhancedInterfaceDef* модели. Эти интерфейсные элементы находятся в отношении **требовать** и **поддерживать** с элементом, содержащим *COTypeDef*.

B.8 Порт предоставленный и порт используемый

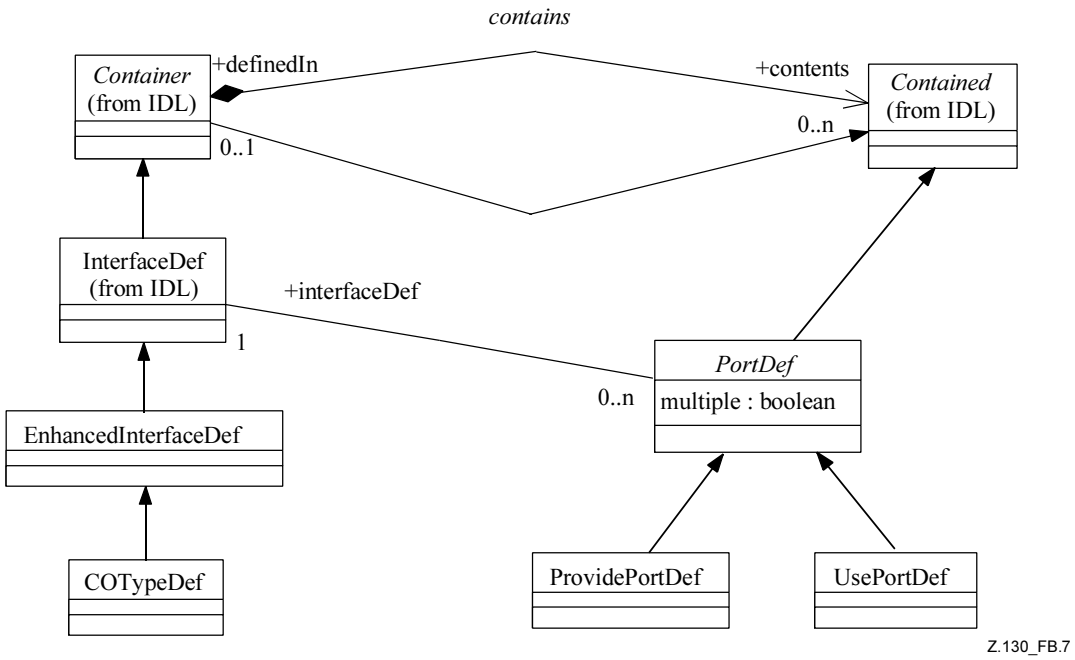


Рисунок B.7/Z.130 – Порт предоставленный и порт используемый

В конкретном синтаксисе продукции **(21)**/**(22)** отображаются на элементы *UsePortDef/ProvidePortDef* модели (см. рис. B.7).

(21) **<use_dcl>** **::=** **"use" ["multiple"] <scoped_name> <identifier>**

(22) **<provide_dcl>** **::=** **"provide" ["multiple"] <scoped_name> <identifier>**

Порты используемый и **предоставленный** выражаются продукциями **(21)** и **(22)**. Своим результатом в модели они имеют элементы *UsedPortDef* и *ProvidePortDef*. В обеих продукциях символ **<scoped_name>** должен обращаться к элементам *InterfaceDef* или *EnhancedInterfaceDef* модели. Если в конкретном синтаксисе используется "множественный", то в текущем *PortDef* множественное булево поле является истиной.

B.9 Артефакт и схема конкретизации

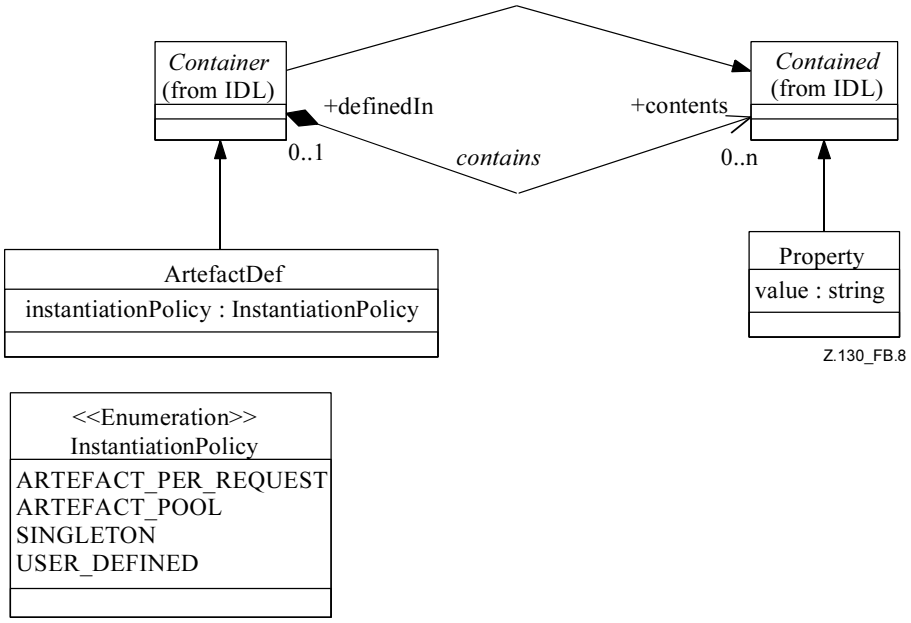


Рисунок B.8/Z.130 – Артефакт и схема конкретизации

(23)	**<artefact>**	**::=**	**<artefact_dcl>**
		|	**<artefact_forward_dcl>**
(24)	**<artefact_forward_dcl>**	**::=**	**"artefact" <identifier>**
(25)	**<artefact_dcl>**	**::=**	**<artefact_header> "{" <artefact_body> "}"**
(26)	**<artefact_header>**	**::=**	**"artefact" <identifier> [<artefact_inheritance_spec>]**
(27)	**<artefact_inheritance_spec>**	**::=**	**":" <scoped_name> { "," <scoped_name> }***
(28)	**<artefact_body>**	**::=**	**<impl_elem_dcl>***

Продукции **(23)** и **(24)** используются для соответствия с синтаксисом IDL, где **<identifier>**, используемый в **(24)**, имеет последующий *ArtefactDef* с этим именем. Продукции **(25)**, **(26)** и **(27)** находятся в модели в отношении с элементом *ArtefactDef*, где **<identifier>** из **(26)** является именем для концепции Named. Все перечисленные в **(27) <scoped_name>** должны обращаться к элементам *ArtefactDef* модели, находящимся в отношении наследования к текущему *ArtefactDef*. Как показывает **(28)**, лишь *ImplementationElementDef* может содержаться в *ArtefactDef* (см. рис. B.8).

B.10 Отношение реализаций

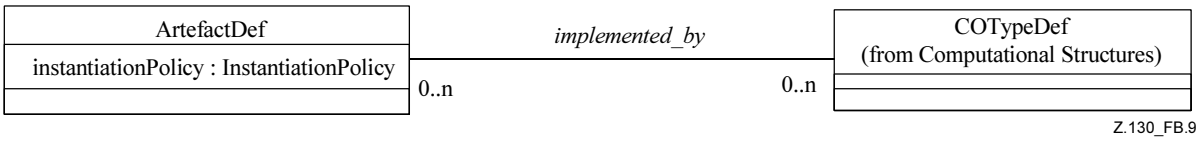


Рисунок B.9/Z.130 – Отношение реализаций

(29)	**<implements_dcl>**	**::=**	**"implemented" "by" <artefact_with_policy>**
			{ "," <artefact_with_policy> }*
(30)	**<artefact_with_policy>**	**::=**	**<scoped_name> ["with" <instantiation_policy_dcl>]**

(31) **<instantiation_policy_dcl>** **::=** **"ArtefactPool"**

| **"ArtefactPerRequest"**

| **"Singleton"**

| **"UserDefined"**

Продукции **(29)** и **(30)** находятся в отношении с элементом *ArtefactDef* модели (см. рис. B.9). Они выражают отношение *implemented_by* модели. Символ **<scoped_name>** в **(30)** должен обращаться только к элементу *ArtefactDef*. В конкретном синтаксисе с продукцией **(31)** выражается поле *instantiationPolicy* содержащего элемента *ArtefactDef*. Ключевые слова непосредственно относятся к перечислимым значениям поля.

B.11 Элемент реализации

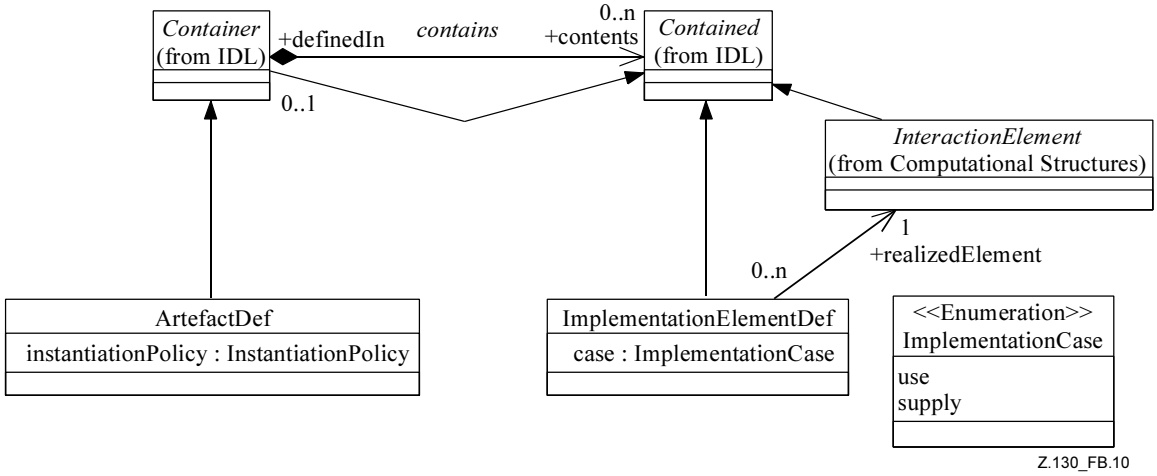


Рисунок B.10/Z.130 – Элемент реализации

(32) **<impl_elem_dcl>** **::=** **<identifier> "implements" <impl_case_dcl> <scoped_name> ";"**

(33) **<impl_case_dcl>** **::=** **"supply" | "use"**

Продукции **(32)** и **(33)** находятся в отношении с элементом *ImplementationElementDef* модели (см. рис. B.10). В **(32)** **<identifier>** является именем концепции Named. Элемент *ImplementationElementDef* может содержаться только в *ArtefactDef*. В конкретном синтаксисе с продукцией **(33)** выражается поле *case*, содержащего элемента *ImplementationElementDef*. Ключевые слова непосредственно относятся к перечислимым значениям поля.

B.12 Программный компонент

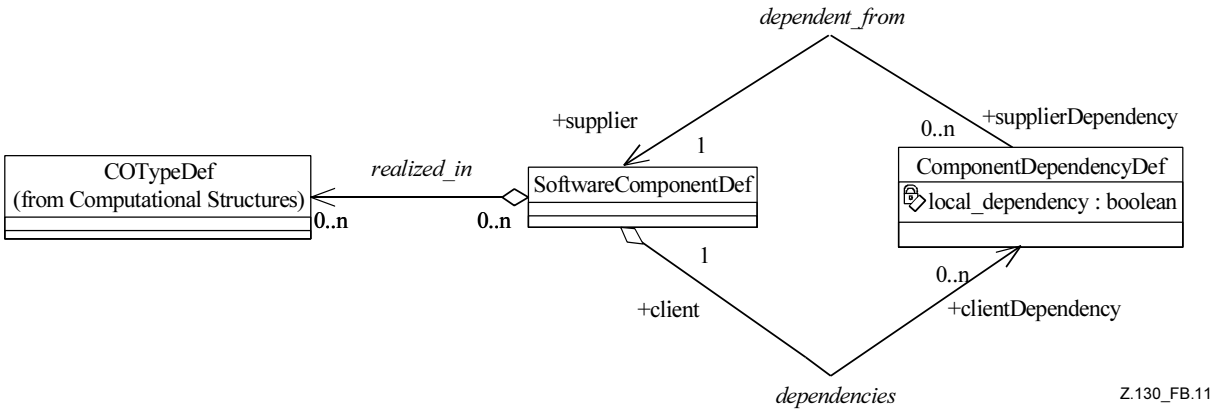


Рисунок B.11/Z.130 – Программный компонент

В конкретном синтаксисе продукция **(34)** отображается на элемент *SoftwareComponentDef* модели (см. рис. B.11).

(34) **<softwarecomponent_dcl>** **::=** **<softwarecomponent_header>**
"{" <softwarecomponent_body> "}"

(35) **<softwarecomponent_header>** **::=** **"softwarecomponent" <identifier> "realizes"**
<cotype_identifier_list>

(36) **<cotype_identifier_list>** **::=** **<cotype_identifier> { "," <cotype_identifier> }***

(37) **<softwarecomponent_body>** **::=** **<softwarecomponent_stmt>***

(38) **<softwarecomponent_stmt>** **::=** **"dependent" "{" <softwarecomponent_list> "}" ";"**
| **"requires" "{" <property_list> "}" ";**

(39) **<softwarecomponent_list>** **::=** **<softwarecomponent_identifier>**
{ "," <softwarecomponent_identifier> }*

Продукции **(34)**, **(35)** и **(36)** находятся в отношении с элементом *SoftwareComponentDef* модели, где **<identifier>** из **(35)** является именем концепции Named. Все **<scoped_name>**, перечисленные в **(36)**, должны обращаться к элементам *COTypeDef* модели, которые находятся в отношении *realized_in* с текущим *SoftwareComponentDef*. Продукции **(37)**, **(38)** и **(39)** находятся в отношении с элементом *SoftwareComponentDef* модели. Все перечисленные в **(36)** **<scoped_name>** должны обращаться к элементам *SoftwareComponentDef* модели.

<softwarecomponent_identifier> представляет собой **<scoped_name>**, который обращается только к элементу *SoftwareDependencyDef* модели.

B.13 Сборка и исходная конфигурация

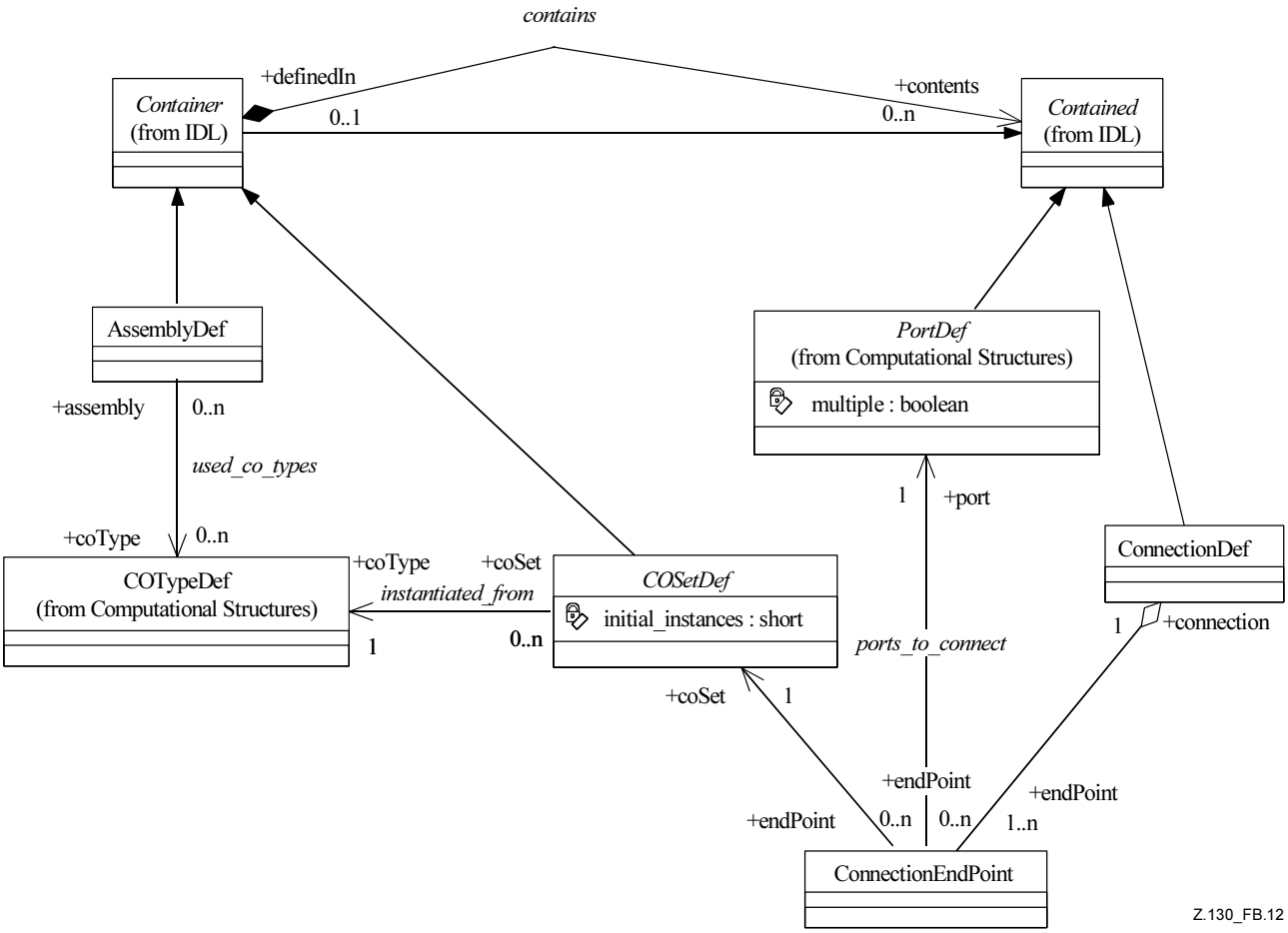


Рисунок B.12/Z.130 – Сборка и исходная конфигурация

(40)	**<assembly_dcl>**	**::=**	**<assembly_header> "{" <assembly_body> "}"**
(41)	**<assembly_header>**	**::=**	**"assembly" <identifier>**
(42)	**<assembly_body>**	**::=**	**<assembly_stmt>***
(43)	**<assembly_stmt>**	**::=**	**<instance_set_dcl>";"**
		|	**<connect_dcl> ";"**
		|	**<constraint_dcl> ";"**
		|	**<property_list>**
(44)	**<instance_set_dcl>**	**::=**	**<identifier> ["(" <integer_literal> ")"] ":" <cotype_identifier>**
(45)	**<connect_dcl>**	**::=**	**"connect" [<identifier>] "{" <connection_list> "}"**
(46)	**<connection_list>**	**::=**	**{ <connection> ";" } +**
(47)	**<connection>**	**::=**	**<instance_set_identifier> "." <port_identifier>**
			"=" <instance_set_identifier> "." <port_identifier>

Продукции **(40)** и **(41)** находятся в отношении с элементом *AssemblyDef* модели (см. рис. B.12), где **<identifier>** из **(41)** является именем концепции Named. Все перечисленные в **(44) <scoped_name>** должны обращаться к элементам *COTypeDef* модели, которые находятся в отношении *realized_in* с текущим *SoftwareComponentDef*. В **(45)**, **(46)** и **(47) <cotype_identifier>**, **<instance_set_identifier>** и **<port_identifier>** являются **<scoped_name>**s, которые обращаются только к элементам *COTypeDef*, *InstanceSetDef* and *PortDef* модели.

B.14 Ограничения и свойства

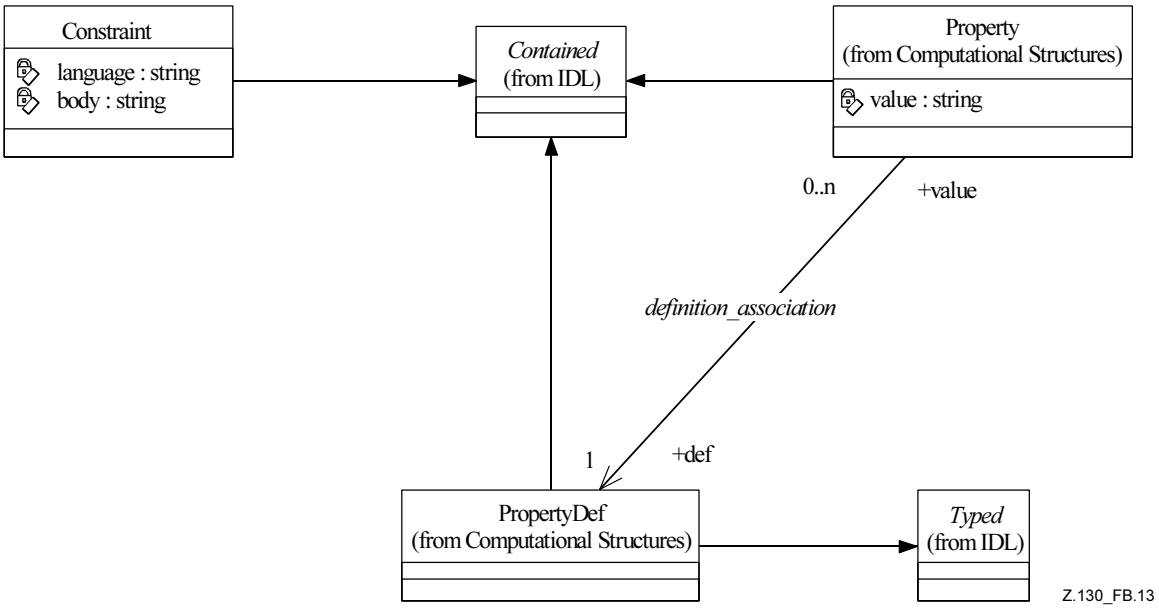


Рисунок B.13/Z.130 – Ограничения и свойства

(48) **<property_dcl>** **::=** **"property" <property_name> "=" <property_value>**

(49) **<property_name>** **::=** **<identifier>**

(50) **<property_value>** **::=** **<simple_property_value>**
| **<structured_property_value>**
| **<sequence_property_value>**

(51) **<simple_property_value>** **::=** **<string_literal>**
| **<integer_literal>**
| **<boolean_literal>**

(52) **<structured_property_value>::=** **"{" <property_assign>* "}"**

(53) **<sequence_property_value>::=** **"[" <property_value>* "]"**

(54) **<property_assign>** **::=** **<property_name> "=" <property_value> ";"**

(55) **<constraint_dcl>** **::=** **"constraint" <identifier> "{" <constraint_body> "}"**

(56) **<constraint_body>** **::=** **"language" "=" <string_literal> "body" "=" <string_literal> ";"**

Продукции **(48)** и **(49)** находятся в отношении с элементом *PropertyDef* модели (см. рис. B.13), где **<identifier>** из **(49)** является именем концепции Named. Продукция **(54)** отображается на элемент *Property* модели. В конкретном текстовом синтаксисе продукции **(50)**, **(51)**, **(52)** и **(53)** используются для нотации значений величины *field*.

Продукция **(55)** находится в отношении с элементом *Constraint* модели, где **<identifier>** из **(55)** является именем концепции Named. Продукция **(56)** обеспечивает значения для полей *language* и *body* элемента *Constraint*.

B.15 Целевая окружающая среда, Node и NodeLink

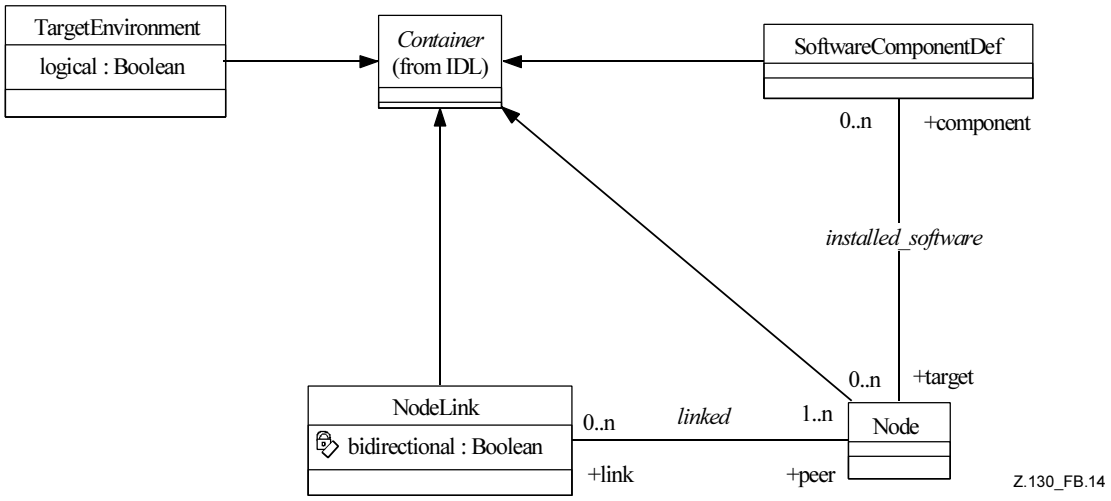


Рисунок B.14/Z.130 – Целевая окружающая среда назначения, Node и NodeLink

(57)	**<environment_dcl>**	**::=**	**<environment_header> "{" <environment_body> "}"**
(58)	**<environment_header>**	**::=**	**"environment" <identifier>**
(59)	**<environment_body>**	**::=**	**<environment_stmt>+**
(60)	**<environment_stmt>**	**::=**	**<node_dcl> ";"**
		|	**<link_dcl> ";"**
(61)	**<node_dcl>**	**::=**	**"node" <identifier> "{" <property_list> "}"**
(62)	**<link_dcl>**	**::=**	**<link_header> "{" <link_body> "}"**
(63)	**<link_header>**	**::=**	**"link" <identifier>**
(64)	**<link_body>**	**::=**	**"node" <node_list> ";" <property_list>**
(65)	**<node_list>**	**::=**	**<node_identifier> { "," <node_identifier> }+**

Продукции **(57)**, **(58)**, **(59)** и **(60)** находятся в отношении с элементом *TargetEnvironment* модели (см. рис. B.14), где **<identifier>** в продукции **(58)** является именем концепции Named. Продукция **(61)** отображается на элемент *Node* модели, где **<identifier>** в продукции **(61)** является именем концепции Named. Продукции **(62)**, **(63)** и **(64)** находятся в отношении с элементом *NodeLink* модели, где **<identifier>** в продукции **(63)** является именем концепции Named. В продукции **(65) <node_identifier>** является **<scoped_name>**, который должен обращаться к элементу *Node* модели.

B.16 InstallationMap

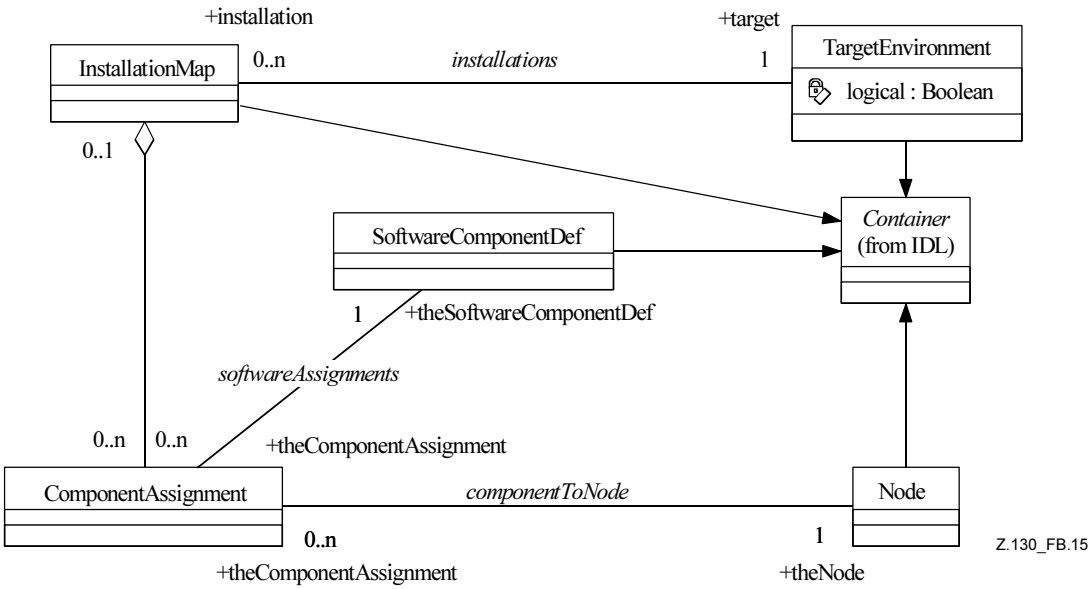


Рисунок B.15/Z.130 – InstallationMap

(66) **<installation_map_dcl>** **::=** **<installation_map_header>**
"{" <installation_map_body> "}"

(67) **<installation_map_header>** **::=** **"installation" <identifier> "uses" "environment"**
<environment_identifier>

(68) **<installation_map_body>** **::=** **<install_stmt>***

(69) **<install_stmt>** **::=** **<softwarecomponent_identifier> "->" <node_identifier> ";"**

Продукции **(66)**, **(67)** и **(68)** находятся в отношении с элементом *InstallationMap* модели (см. рис. B.15), где **<identifier>** в продукции **(67)** является именем концепции Named. В продукции **(67)** **<environment_identifier>** является **<scoped_name>**, которое обращается только с элементом *TargetEnvironment* модели. В продукции **(69)** **<softwarecomponent_identifier>** и **<node_identifier>** являются **<scoped_name>**s, которые обращаются только к элементам *SoftwareComponentDef* и *Node* модели.

B.17 InstantiationMap

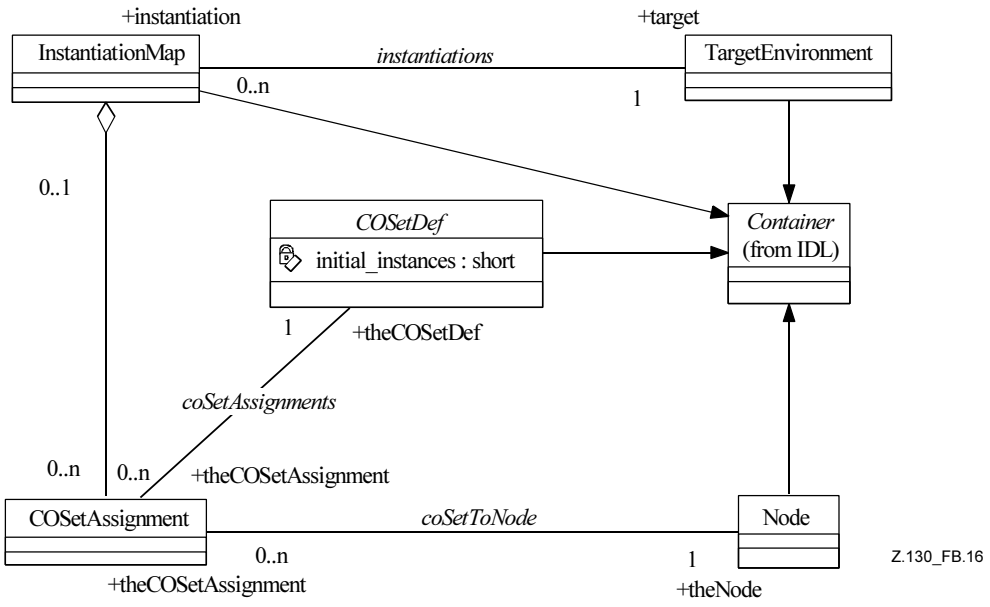


Рисунок B.16/Z.130 – InstantiationMap

(70)	**<instantiation_map_dcl>**	**::=**	**<instantiation_map_header>**
			"{" <instantiation_map_body> "}"
(71)	**<instantiation_map_header>**	**::=**	**"instantiation" <identifier>**
			<instantiation_map_header_env>
			<instantiation_map_header_ass>
(72)	**<instantiation_map_header_env>**	**::=**	**"uses" "environment" <environment_identifier>**
(73)	**<instantiation_map_header_ass>**	**::=**	**"uses" "assembly" <assembly_identifier>**
(74)	**<instantiation_map_body>**	**::=**	**<assign_instance_stmt>***
(75)	**<assign_instance_stmt>**	**::=**	**<instance_set_identifier_list> "->"**
			<node_identifier> ";"
(76)	**<instance_set_identifier_list>**	**::=**	**<instance_set_identifier>**
			{ "," <instance_set_identifier> }*

Продукции **(70)**, **(71)**, **(72)**, **(73)** и **(74)** находятся в отношении с элементом *InstantiationMap* модели (см. рис. B.16), где **<identifier>** в продукции **(71)** является именем концепции Named. **<environment_identifier>** в продукции **(72)** и **<assembly_identifier>** в продукции **(73)** являются **<scoped_name>**s, которые обращаются только к элементам *TargetEnvironment* и *AssemblyDef* модели. **<node_identifier>** в продукции **(75)** является **<scoped_name>**, который обращается только к элементу *Node* модели, содержащемуся в элементе *TargetEnvironment*, квалифицированном продукцией **(72)**. **<instance_set_identifier>**s в продукции **(76)** являются **<scoped_name>**s, которые обращаются только к элементам *COSetDef* модели, которые содержатся в *AssemblyDef*, квалифицированном продукцией **(73)**.

B.18 План внедрения

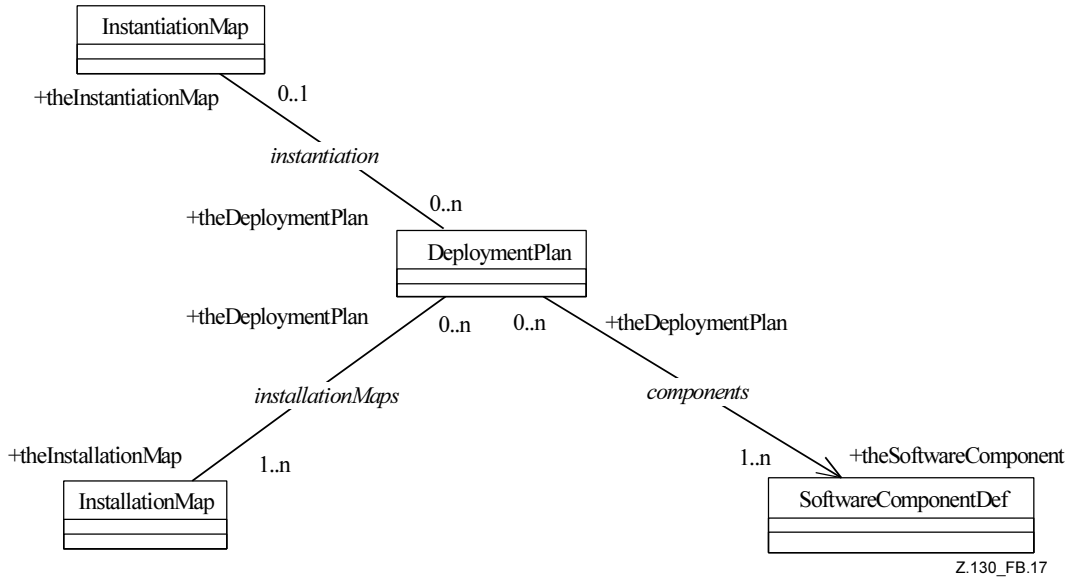


Рисунок B.17/Z.130 – План внедрения

(77) **<deployment_action>** **::=** **"deploy" "{" <deployment_body> "}" ";"**

(78) **<deployment_body>** **::=** **"install" "{" <install_list> "}" ";" <instantiation_action>+**

(79) **<install_list>** **::=** **<install_member>***

(80) **<install_member>** **::=** **<installation_map_identifier> ";"**

(81) **<instantiation_action>** **::=** **"instantiate" <instantiation_map_identifier> ";"**

В конкретном синтаксисе продукция **(77)** **<deployment_action>** отображается на элемент *DeploymentPlan* модели (см. рис. B.17). Списки в теле **<deployment_action>**, построенные продукциями **(78)**, **(79)**, **(80)** и **(81)**, выражают отношения *InstallationMaps* и конкретизации модели. **<instantiation_map_identifier>** и **<installation_map_identifier>** являются **<scoped_name>**s, которые обращаются к элементам *InstallationMap* и *InstantiationMap* модели. Каждый из перечисленных идентификаторов соответствует отношению в модели.

B.19 Тип внешний

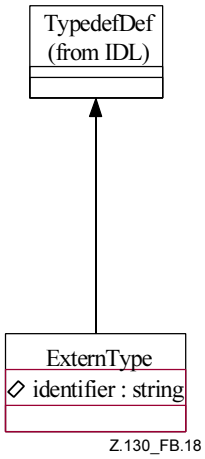


Рисунок B.18/Z.130 – Тип внешний

(82) <extern_type> ::= "extern" "type" <identifier> <string_literal>";"

В конкретном синтаксисе продукция **(82) <extern_type>** отображается на элемент *ExternType* модели (см. рис. B.18). Значение строкового литерала непосредственно отражается на идентификатор атрибута элемента модели. В продукции **(82) <identifier>** отображается на атрибут имени концепции *Contained*.

Приложение C

Отображение на ЯСО-2000

C.1 Введение

Рекомендуемый расширенный язык описания объектов МСЭ (eODL-МСЭ) обеспечивает возможность описания компонентно-ориентированных распределенных систем. В данном приложении вводится отображение eODL-МСЭ на ЯСО-2000. Это отображение позволяет автоматически генерировать код ЯСО-2000, исходя из описания, приведенного в eODL-МСЭ. Используется текстовое представление (ЯСО/PR) ЯСО-2000 [8].

Отображение eODL-МСЭ на ЯСО-2000 позволяет пользователям автоматически генерировать скелет ЯСО-2000, исходя из данной модели eODL. Отображение поддерживает почти все концепции вычисления и реализации. Единственным важным **исключением** является то, что не поддерживается концепция **непрерывных носителей**. Сверх того, не поддерживаются концепции **внедрения** и целевой окружающей среды.

Типы eODL отображаются на соответствующие типы в ЯСО-2000. Концепции, определяющие аспекты поведения типов, отображаются на автоматически реализуемые агенты ЯСО. Заданная полная модель eODL, инструмент отображения генерирует скелет ЯСО-2000. Пользователь должен реализовать бизнес-логику и быть способным использовать **COs**, определенные в ЯСО-2000 в качестве стандартных блоков системы ЯСО-2000.

Отображение имеет целью наиболее полную поддержку концепций, даже за счет создания в ЯСО-2000 до какой-то степени сложных структур. Например, множественное наследование **COs** поддерживается ценой более сложных структур и менее эффективного поведения, т. к. язык ЯСО-2000 не поддерживает множественных наследований для типов агентов.

C.2 Пакет eodl

Отображение eODL-МСЭ на ЯСО-2000 определяет пакет ЯСО `eodl`. Он содержит определения **типов данных**, которые используются моделями, генерируемыми из моделей eODL. Типы, называемые предопределенными, определяются в пакете `eodl`. Полный листинг пакета `eodl` приведен в C.10.

Текстовая нотация eODL наследует свои лексические правила от OMG CORBA-IDL. В CORBA-IDL неквалифицированные идентификаторы начинаются с буквенного символа, за которым следует любое число буквенных символов, цифр или символов подчеркивания, причем буквенными символами являются строчные и прописные буквы от 'A' до 'Z' английского алфавита. Кроме того, в eODL строчные и прописные буквы в идентификаторах не различаются.

В соответствии с лексическими правилами ЯСО-2000, все (неквалифицированные) идентификаторы eODL являются идентификаторами ЯСО-2000.

Квалифицированные идентификаторы приведены в параграфе C.4 (Имена с установленными областями определения).

C.3 Структура

Каждый файл eODL-МСЭ отображается на два пакета ЯСО:

- `<name>_interface` (называемый "пакет интерфейса "); и

- `<name>_definition` (называемый "пакет определения "),

где `<name>` - имя спецификации eODL-МСЭ (например, имя файла), как показано на рис. C.1.

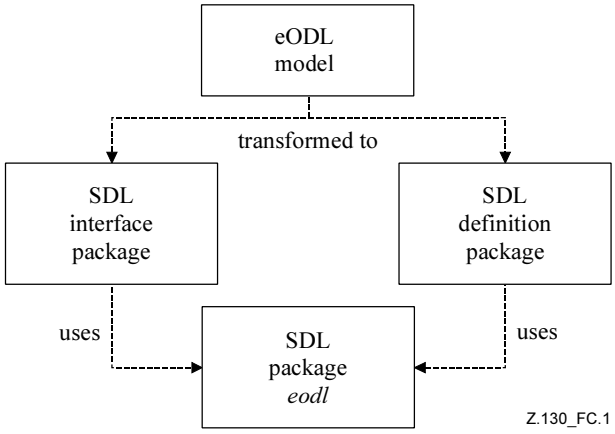


Рисунок C.1/Z.130 – Структура преобразований из модели eODL в ЯСО

Пакет интерфейса содержит всю информацию, относящуюся как к клиентской, так и к серверной стороне системы. Более подробно, это:

- определения типа данных;
- определения постоянных; а также
- определения интерфейсов, представляющие регулярные **интерфейсы** и **интерфейсы COs**.

Пакет определения содержит скелеты серверных сторон системы. Фактически это:

- типы блоков, представляющих **типы CO**; а также
- типы процессов, представляющих **артефакты**.

C.4 Имена с установленными областями определения

Квалификация представляет собой концепцию, существующую как в eODL, так и в ЯСО-2000. Таким образом, отображение является каноническим: квалифицированные имена в eODL отображаются на квалифицированные имена в ЯСО.

C.5 Отображение концепций вычисления

C.5.1 Модули

Модуль eODL является контейнером для всех других элементов eODL и открывает namespace. Эта концепция отображается на концепцию пакета ЯСО.

Модуль eODL пакета ЯСО отображается на то, что может содержаться либо в пакете интерфейса ЯСО, либо в пакете определения ЯСО, или в обоих пакетах в зависимости от сущностей, вложенных в модуль eODL.

C.5.2 Определения типов

Конструкция eODL `typedef` присваивает (другое) имя данному типу. Конструкция `typedef` отображается на конструкцию ЯСО `syntype`.

C.5.3 Предопределенные типы данных

C.5.3.1 Типы данных для целых чисел

Типы данных eODL `unsigned short`, `unsigned long` и `unsigned long long` отображаются на следующие сорты ЯСО. Эти сорты определяются в пакетах `eodl`.

- Тип eODL `unsigned short` отображается на сорт `ushort` типа `Integer`, находящийся в диапазоне от 0 до $2^{16} – 1$.
- Тип eODL `unsigned long` отображается на сорт `ulong` типа `Integer`, находящийся в диапазоне от 0 до $2^{32} – 1$.
- Тип eODL `unsigned long long` отображается на сорт `ulong_long` типа `Integer`, находящийся в диапазоне от 0 до $2^{64} – 1$.

Типы данных eODL `signed short`, `signed long` и `signed long long` отображаются на следующие сорты ЯСО.

- Тип eODL `signed short` отображается на сорт `short` типа `Integer`, находящийся в диапазоне от$–2^{15}$ до $2^{15} – 1$.
- Тип eODL `signed long` отображается на сорт `long` типа `Integer`, находящийся в диапазоне от $–2^{31}$ до $2^{31} – 1$.
- Тип eODL `signed long long` отображается на сорт `long_long` типа `Integer`, находящийся в диапазоне от $–2^{63}$ до $2^{63} – 1$.

C.5.3.2 Типы данных для чисел с плавающей запятой

Типы eODL с плавающей запятой `float`, `double` и `long double` отображаются на предопределенный сорт `Real` ЯСО. Заметим, что в соответствии с Рекомендацией МСЭ-Т Z.100, сорт `Real` не удовлетворяет IEEE 754 [15], тогда как типы eODL с плавающей запятой удовлетворяют.

C.5.3.3 Типы данных для символов

Тип eODL `char` отображается на предопределенный сорт ЯСО `Character`. Тип eODL `wchar` отображается на предопределенный сорт ЯСО `Natural`.

C.5.3.4 Тип данных булевый

Тип eODL `boolean` отображается на сорт ЯСО `Boolean`, а константы `TRUE` и `FALSE` типа eODL `boolean` отображаются на литералы `true` и `false` сорта ЯСО `Boolean`, соответственно.

C.5.3.5 Тип данных октетный

Тип eODL `octet` отображается на сорт `Octet` ЯСО.

C.5.3.6 Тип данных любой

Тип eODL `any` отображается на сорт ЯСО `Any`. Следует заметить, что в eODL семантика любого является той же самой, что и любого в ЯСО (см. § 3.10.1.7 редакции 2.4.2 IDL OMG).

C.5.3.7 Идентификация типа с помощью атрибута типа TypeCode

В **метамодели** eODL атрибут `typeCode` экземпляра *IDLType* не отображается на ЯСО. Однако в пакете ЯСО тип `TypeCode` отображается на сорт ЯСО `TypeCode`. Сорт `TypeCode` является абстрактным типом данных. Это означает, что в соответствии с данным отображением значения этого типа данных отсутствуют. Хотя это не ограничивает реализации по использованию производного конкретного типа данных.

C.5.4 Сконструированные типы данных

C.5.4.1 Перечисления

В eODL перечисления отображаются на тип данных значения ЯСО, который содержит лишь литералы.

C.5.4.2 Структуры

В eODL структуры отображаются на типы данных значения ЯСО с конструктором типа данных открытой структуры. Элементы структуры eODL являются полями типа данных ЯСО.

C.5.4.3 Объединения

В eODL объединения отображаются на вложенный тип данных значения ЯСО. В ЯСО внешний тип данных значения имеет конструктор типа данных открытой структуры с двумя полями:

1) поле `tag`, которое представляет дискриминатор объединения-eODL; а также

2) поле `union`, которое представляет само объединение.

Объединение поля представляет собой тип данных значения `<name-of-eODL-union>_union`. Этот тип объявляется в пределах области определения внешнего типа объединения ЯСО. Он имеет конструктор типа данных открытого выбора, а его список выбора представляет элементы объединения eODL.

C.5.4.4 Массивы

В eODL массивы отображаются на сорт ЯСО, который наследует предопределенный сорт ЯСО `Vector`.

Многомерные массивы отображаются на тип данных значения ЯСО, который поддерживает операторы `Make`, `Modify` и `Extract`.

C.5.5 Типы значения

В eODL типы значения отображаются на типы данных объекта ЯСО с конструктором типа структуры данных. Все элементы данных типа значения eODL являются полями типов данных ЯСО.

Если тип значения eODL объявляется абстрактным, соответствующие данные объекта ЯСО также объявляются абстрактными. Он не имеет конструктора типа данных.

Простое наследование типов значения в eODL отображается на простое наследование типов данных в ЯСО. Для абстрактных типов значения в eODL допускается множественное наследование. В ЯСО это достигается копированием объявления операции из базовых типов данных в производный тип данных.

Типы значения в eODL могут иметь фабрики (элементы установки в исходное состояние). В ЯСО они отображаются на операторы, которые возвращают значение типа данных. Если объявляется точно одна фабрика, оператор `Make` автоматически реализуется посредством вызова этой фабрики. Другими словами, он реализуется не точно.

Если тип значения eODL поддерживает интерфейс, **операции** этого интерфейса становятся операциями типа данных ЯСО. Атрибуты в поддерживаемом интерфейсе отображаются на пару операций get/set и на поле, переносящее атрибут.

Типы значения, обрамленные рамкой, отображаются таким же образом, как и (конкретные) типы значения.

C.5.6 Параметризованные типы данных

C.5.6.1 Последовательности

Если в eODL последовательности ограничены, то они отображаются на сорт ЯСО `Vector`, в противном случае на сорт ЯСО `Array`.

C.5.6.2 Символьные строки

Тип eODL `string` отображается на предопределенный сорт ЯСО `Charstring`.

Тип eODL `wstring` отображается на сорт ЯСО `wstring` в пакете `eodl`. Этот сорт наследует `String<Natural>`. Он также прибавляет операцию из `Charstring` для преобразования значения `Charstring` в значение `wstring`.

C.5.6.3 Числа с фиксированной запятой

Пакет ЯСО `eodl` содержит тип `fixedpt`, параметризованный **параметрами** разрядности и масштаба. Оба принадлежат к сорту `Natural`. Семантики параметров идентичны семантикам параметров в eODL.

Тип `fixedpt` определяет операторы "+","–","*","/", которые суммируют, вычитают, умножают и делят два значения типа `fixedpt` и возвращают значения типа `fixedpt`. Также, оператор "=" производит сравнение со значениями `fixedpt` и возвращает значение `Boolean`: истина, если оба операнда представляют одно и то же число, и ложь, если оба операнда представляют различные числа.

Для преобразования значения `fixedpt` в значение `Real` имеется метод `toReal`. Используя оператор `Make`, можно из значения `Real` сконструировать значение `fixedpt`.

C.5.7 Константы, литералы типа данных и выражения констант

C.5.7.1 Константы

Константы eODL отображаются на синонимы ЯСО.

C.5.7.2 Литералы типа данных

В этом параграфе перечисляются типы данных, не отображаемые на предопределенные сорты ЯСО.

C.5.7.2.1 Литералы типов целых чисел

Десятичные и шестнадцатеричные литералы целых чисел поддерживаются в отображении ЯСО: десятичные литералы поддерживаются сортом ЯСО `Integer`, а шестнадцатеричные литералы целых чисел могут быть записаны, как литералы типа ЯСО `Integer`, а затем преобразованы в `Integer`. Восьмеричные литералы целых чисел должны быть преобразованы либо в десятичные, или в шестнадцатеричные литералы.

C.5.7.2.2 Литералы типов символов

Литерал типа eODL `char` отображается так, что он формирует допустимый литерал предопределенного сорта ЯСО `Character`. Если литерал eODL является символом со значением большим 127, он не может отображаться. Вместо этого рекомендуется использовать тип `wchar`.

Литерал типа eODL `wchar` отображается таким образом, чтобы он формировал допустимый литерал `Natural`.

C.5.7.2.3 Литералы строковых символов

Литерал типа eODL `string` отображается так, что он формирует допустимый литерал предопределенного сорта ЯСО `Charstring`. Если литерал eODL содержит символ со значением большим 127, он не может отображаться. Вместо этого рекомендуется использовать тип `wstring`.

Литерал типа eODL `wstring` отображается таким образом, чтобы он формировал допустимый литерал `wchar` сорта ЯСО. Это включает возможность преобразования литерала `Charstring` в значение ЯСО `wstring`.

C.5.7.2.4 Литералы типа числа с фиксированной запятой

Значения с фиксированной запятой могут создаваться только путем преобразования действительных чисел.

C.5.7.3 Выражения постоянных

В eODL выражения постоянных отображаются на выражения постоянных в ЯСО. Оба выражения должны представляться теми же самыми значениями.

C.5.8 Типы сигналов

Типы сигналов отображаются на типы данных значения.

C.5.9 Исключения

Как eODL-МСЭ, так и ЯСО поддерживают **исключения**. Единственная разница состоит в том, что **исключения** ЯСО не именуют элементы **исключений**. **Исключения** определяются в пакете интерфейса.

C.5.10 Интерфейсы и элементы взаимодействия

C.5.10.1 Интерфейсы

Группы интерфейсов eODL:

- **операции**;
- потоки **сигналов**;
- взаимодействия (потоки) типа **непрерывных носителей**;
- атрибуты.

Взаимодействия типа **непрерывных носителей** не отображаются на ЯСО.

Интерфейс eODL не отображается на интерфейсы ЯСО `exported_I` и `imported_I`. Эти интерфейсы определяются в пакете ЯСО `I` пакета интерфейса. Интерфейс `exported_I` содержит

- все **операции**; и
- поглощенный **сигнал**.

Тогда как интерфейс `imported_I` содержит порожденные **сигналы**.

Это отображение является таким, что тип интерфейса `exported_<interface-name>` содержит все, что нужно клиенту интерфейса для вызова службы, которую представляет интерфейс.

Данное отображение подробно описывается в последующих параграфах.

C.5.10.2 Элементы операционного взаимодействия

Операция eODL отображается на удаленную процедуру, которая объявляется в интерфейсе ЯСО `exported_<interface name>`. Обе концепции поддерживают не более одного типа возврата, параметры, которого могут быть in, out и inout, а также **исключениями**. Экземпляры концепции "контекст" не отображаются.

C.5.10.3 Элементы сигнального взаимодействия

Если интерфейс eODL объявляет **элемент взаимодействия** для поглощения **сигнала** `A`, соответствующий интерфейс ЯСО `exported_<interface-name>` объявляет использование сигнала `A`.

Если интерфейс eODL объявляет **элемент взаимодействия** для порождения **сигнала** `A`, соответствующий интерфейс ЯСО `imported_<interface-name>` объявляет использование сигнала `A`.

C.5.10.4 Элементы взаимодействия типа атрибута

В eODL атрибуты отображаются на пару удаленных процедур set/get в интерфейсе ЯСО `exported_<interface-name>`. Если атрибут является доступным только для чтения, удаленная процедура set не генерируется.

C.5.10.5 Ссылки интерфейса

Тип интерфейса ЯСО неявно определяет специальный сорт `PId`. Экземпляр этого сорта `PId` служит в качестве ссылки интерфейса. Более того, эта концепция обеспечивает тип безопасности: клиент, который получил такой `PId`, может отправлять сигналы и вызывать удаленные процедуры в агенте, реализующим этот интерфейс. Типом интерфейса ЯСО, служащим типом ссылки интерфейса, является `exported_<interface name>`.

C.5.10.6 Наследование

Как eODL-МСЭ, так и ЯСО поддерживают множественное наследование интерфейсов. Таким образом, отображение является каноническим.

C.5.11 Объекты вычисления

В eODL **COs** инкапсулируют состояние и поведение. Они предоставляют интерфейсы (в качестве ссылок интерфейса) окружающей среде и могут сами использовать интерфейсы (ссылки) других **COs**.

В ЯСО **CO** является агентом типа процесса. Каждый **CO** имеет три шлюза: один - входящий, называемый `initial`, другой - называется `provides` и третий - называется `uses`. Как обсуждается в последующих параграфах, эти шлюзы поддерживают несколько интерфейсов. Все три шлюза соединяются с окружающей средой процесса, потому что они представляют собой интерфейс между **CO** и окружающей средой.

Сам тип процесса **CO** определяется в пакете определения.

Для создания процессов **CO** определяется фабрика **CO**. Фабрика **CO** представляется как тип процесса, определенного в той же области определения, как и процесс типа **CO**. Как сам **тип CO**, так и его фабрика являются типами процессов, определенных внутри типа агента вида блока. Этот блок называется `SDL component`. Заметим, что `SDL component`, как определен в данном отображении, является концепцией представления вычисления, тогда как компонент, как определен в данной Рекомендации, является концепцией представления **внедрения**. Компонент ЯСО является типом блока с вполне определенным интерфейсом.

C.5.11.1 Интерфейс CO

Отдельно от предоставления и использования интерфейсов **тип CO** открывает свой собственный интерфейс. Этот интерфейс состоит из трех компонентов: определенный пользователем интерфейс, неявный компонент, идентифицирующий интерфейс, и конфигурационный интерфейс.

Метамодель определяет **тип CO** в виде интерфейса и ограничивает содержащиеся **элементы взаимодействия** экземплярами *AttributeDef*. При рассмотрении **типа CO** в виде лишь интерфейса, он называется "определенный пользователем интерфейс".

Для идентификации **COs** каждый **тип CO** имеет неявный ключ атрибута только для чтения предопределенного типа `ComponentKey`. Этот атрибут является единственным **элементом взаимодействия** в предопределенном интерфейсе `ComponentBase`.

Интерфейс `ComponentBase` определяет процедуру доступа к ключу **CO**. Ключ **CO** реализует свою идентичность и должен быть однозначно определяемым, по крайней мере, что касается компонента ЯСО, в котором содержится **CO**. В пакете `eodl` имеется объявленная внешняя процедура `compute_co_key`, возвращающая такой однозначно определяемый ключ. Реализация этого отображения должно обеспечить либо реализацию такой процедуры, или генерировать код, вычисляющий однозначно определяемый ключ другим образом.

Конфигурационный интерфейс определяет процедуры для поддержки операций **порта**. Далее этот интерфейс уточняется в параграфе, посвященном отображению портов. Конфигурационный интерфейс наследует предопределенный интерфейс `ConfigBase`. Интерфейс `ConfigBase` объявляет общие операции **порта**.

Все эти интерфейсы определяются в пакете интерфейса. Они содержатся в пакете с названием **CO**. Определенный пользователем интерфейс называется `<CO-name>_attributes`. Он наследует предопределенный интерфейс `ComponentBase`. Конфигурационный интерфейс называется `<CO-name>_config` и наследует предопределенный интерфейс `ConfigBase`. В заключение, интерфейс `<CO-name>` определяется также просто и наследует `<CO-name>_attributes` и `<CO-name>_config`.

Интерфейс `<CO-name>` поддерживается шлюзом `initial`. Определяется канал от окружающей среды к типу процесса **CO**.

C.5.11.2 Поддерживаемые интерфейсы

Шлюз `provides` обеспечивает ссылками все интерфейсы, которые **поддерживает CO**. Через этот шлюз клиенты могут пользоваться **CO**. Интерфейс ЯСО `exported_<name>` поддерживается в направлении от окружающей среды к типу процесса, а интерфейс ЯСО `imported_<name>` поддерживается в направлении от типа процесса к окружающей среде.

C.5.11.3 Требуемые интерфейсы

Шлюз `uses` обеспечивает ссылками все интерфейсы, в которых **CO нуждается** и использует. Через этот шлюз **CO** (в роли клиента) может использовать другие **COs**. Интерфейс `imported_<name>` ЯСО

поддерживается в направлении от окружающей среды к типу процесса, а интерфейс exported_<name> ЯСО поддерживается в направлении от типа процесса к окружающей среде.

C.5.11.4 Наследование

Так как ЯСО не поддерживает множественное наследование, наследование представляется делегированием. Оно представляется следующим образом:

1) Определенный пользователем интерфейс **типа CO** и конфигурационный интерфейс **типа CO** наследуют соответствующие интерфейсы супер **CO**(s).

2) Шлюз provides поддерживает все интерфейсы, которые поддерживаются супер **CO**(s). Он также поддерживает все интерфейсы, которые поддерживаются самим **типом CO**.

3) Шлюз uses поддерживает все интерфейсы, которые требуются супер **CO**(s). Он также поддерживает все интерфейсы, требуемые самим **типом CO**.

Относительно аспектов поведения множественного наследования см. C.7.2.5.

C.5.11.5 Фабрики CO

Для каждого **типа CO** существует ассоциированная фабрика **CO** <CO-name>_factory. Она реализует интерфейс фабрики. Этот интерфейс определяется в пакете <CO-name>_factory, который, в свою очередь, определяется в том же пакете, где определяются интерфейсы **CO**. Интерфейс фабрики наследует предопределенный интерфейс CoFactoryBase.

Интерфейс CoFactoryBase содержит следующие процедуры. Процедура get_co_type возвращает полностью квалифицированное имя **типа CO**. Символом квалификации является десятичная точка. Процедура generic_create конкретизирует ассоциированный **тип CO**. Процедура list_cos возвращает список значений ComponentKey конкретизированных **COs**. Процедура resolve_co выбирает значение ComponentKey и возвращает ассоциированный **CO**.

Интерфейс фабрики объявляет процедуру create_<CO-name>, которая создает ассоциированный **тип CO**. Агент фабрики имеет шлюз factory, поддерживающий во входящем направлении интерфейс CoFactoryBase.

C.5.11.6 Инкапсуляция типа CO и фабрики CO – компонент ЯСО

Как **тип CO**, так и тип фабрики **CO** определяются в типе блока <CO-name>_CO. Этот тип блока называется компонентом ЯСО. В каждом компоненте ЯСО имеется множество экземпляров factory типа <CO-name>_factory, а также множество экземпляров cos типа <CO-name>. Множество экземпляров factory содержит точно один экземпляр. В исходном положении множество экземпляров cos не содержит ни одного экземпляра и ограничения относительно максимального числа экземпляров. Шлюзы initial, provides и uses **типа CO** дублируются компонентом ЯСО так же, как шлюз factory типа фабрики и те шлюзы, которые соединены каналами.

Компонент ЯСО определяется в пакете определения. Он представляет собой **тип CO**.

C.6 Отображение концепций представления конфигурации

C.6.1 Порты предоставленные

В eODL концепция **портов предоставленных** является механизмом раздачи ссылок интерфейса, предоставленных **CO**, клиентам этого **CO**.

Эта концепция отображается на множество удаленных процедур, которые объявляются в конфигурационном интерфейсе **CO**.

Порт предоставленный foo типа bar отображается на удаленную процедуру provide_foo, возвращающую ссылку (PId) типу bar. Если foo является атрибутом **single**, от каждого вызова

provide_foo требуется возврат той же PId. Если foo является атрибутом **multiple**, пользователь должен сам реализовать семантику (относительно управления **портом** см. C.7.4.3).

Конфигурационный интерфейс наследует предопределенный интерфейс ConfigBase. Этот интерфейс объявляет процедуру provide. Она выбирает последовательность в качестве аргумента. Фактический **параметр** в вызове процедуры указывает **порт**. Если этот порт существует, ссылка возвращается в виде PId. Если **порт** не существует, устанавливается **исключение** NoSuchPort. **Исключение** NoSuchPort является предопределенным.

C.6.2 Порты используемые

В eODL концепция **порта используемого** является механизмом, дающим возможность **CO** хранить ссылки интерфейса других **COs**.

Эта концепция отображается на множество удаленных процедур, которые объявляются в конфигурационном интерфейсе **CO**.

Порт используемый foo типа bar отображается на удаленную процедуру link_foo, которая в качестве **параметра** выбирает ссылку к bar. Если **порт** имеет атрибут **single**, а какая-либо ссылка уже хранится в этом **порту**, устанавливается предопределенное **исключение** AlreadyConnected. Кроме того, объявляется удаленная процедура unlink_foo, которая удаляет находящуюся на хранении ссылку из **порта** foo. Если ссылка на хранении в foo отсутствует, то устанавливается предопределенное **исключение** NotConnected. Если **порт** имеет атрибут **multiple**, последовательность ссылок запоминается. **Исключение** AlreadyConnected никогда не устанавливается.

Предопределенный интерфейс ConfigBase, который наследуется конфигурационным интерфейсом, объявляет процедуру link и процедуру unlink. Эти процедуры могут использоваться обычным образом для хранения или удаления ссылок в **порту используемом**. Как прототипы для **портов предоставленных**, они вызывают аргумент последовательности, указывающий имя **порта**. Снова, если **порт** с указанным именем не существует, устанавливается **исключение** NoSuchPort. Общая процедура соединения в качестве второго аргумента вызывает PId и при возможности запоминает его в указанном **порту** или, в противном случае, устанавливает AlreadyConnected. Для решения о возможности хранения ссылки она использует ту же самую семантику, как и процедура соединения, характерная для **порта**. Если ссылка на хранении в указанном **порту** отсутствует, то общая процедура разъединения устанавливает **исключение** NotConnected.

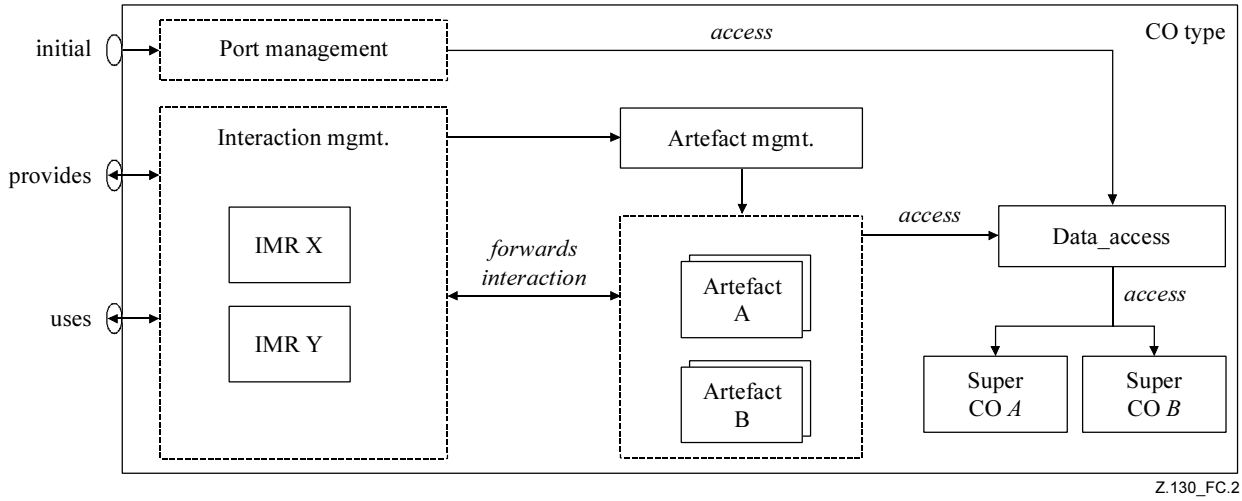
C.6.3 Услуга присваивания имен ЯСО

С целью динамического обнаружения других компонентов для **CO** определяется услуга присваивания имен. Услуга присваивания имен реализуется процессом типа SDL_Component_Register_Type в пакете eodl. Требуется, чтобы каждая система ЯСО, полученная из модели eODL, имела бы множество экземпляров SDL_Component_Registry, содержащее точно один экземпляр типа SDL_Component_Register_Type.

Как только компонент ЯСО конкретизирован, он регистрирует себя, используя экспортируемую процедуру register_SDLComponent. Любой **CO** может запрашивать услугу присваивания имен, используя query_SDLComponent. Ключом для поиска компонента ЯСО является полностью квалифицированное имя, использующее точку в качестве символа квалификации. Процедура запроса возвращает ссылку экземпляру компонента ЯСО запрошенного типа. Используя эту ссылку, любой клиент может запросить фабрику компонента ЯСО для **COs**.

C.7 Отображение концепций реализации

На рис. C.2 приведена внутренняя структура процесса ЯСО, представляющая в общем виде **тип CO** с различными факультативными представлениями ЯСО.



IMR Interaction Management Representation

Рисунок C.2/Z.130 – Процесс ЯСО, представляющий тип СО

Прямоугольник представляет процесс ЯСО. Пунктирный прямоугольник представляет концепцию, а не процесс. Например, "управление взаимодействиями" представляет собой концепцию управления взаимодействиями и содержит конкретные процессы управления взаимодействиями, например, процессы ЯСО.

На рис. C.3 в графической нотации представлен конкретный пример типа процесса СО. Процедуры `get_key`, `provide_SamplePort` и `provide` составляют управление портом. Процессы `interaction_interfaceX` и `interaction_interfaceY` соответствуют "IMR X" и "IMR Y" на рис. C.2. Множества `A` и `B` экземпляра процесса являются экземплярами **артефактов**. Множество экземпляра `base` соответствует прямоугольнику "Super CO A" на рис. C.2.

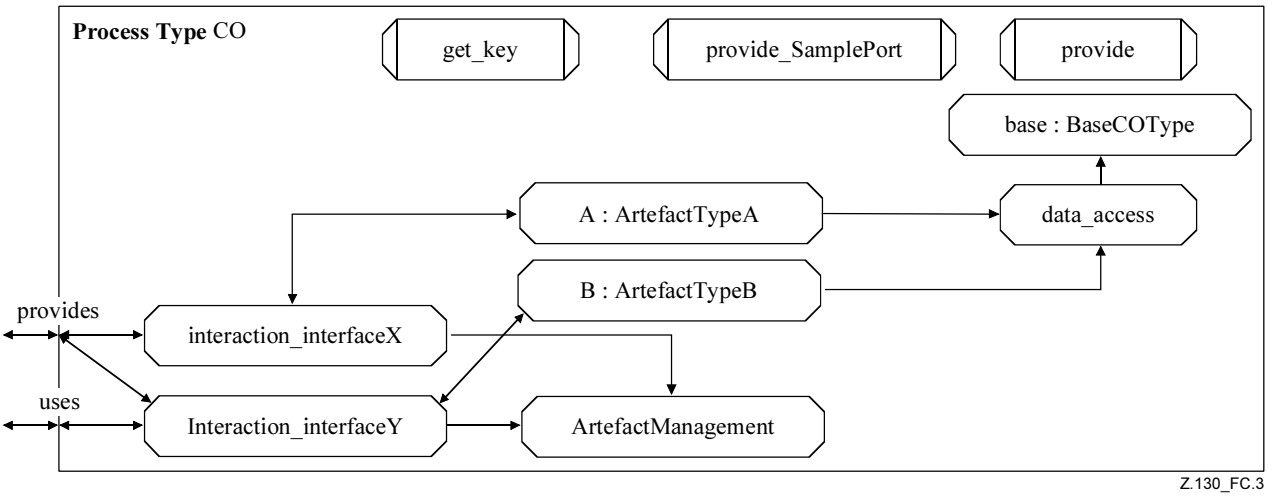


Рисунок C.3/Z.130 – Пример типа процесса СО

C.7.1 Доступ к данным

Данные СО хранятся в процессе `data_access`, который реализует процедуры get/set, чтобы позволить артефактам доступ к этим данным. Данные состоят из ссылок, используемых управлением портом и управлением взаимодействиями.

Для обеспечения типового доступа к таким данным в пакете описания объявляется пакет <CO-Name>_data. Этот пакет данных содержит интерфейс internal_data, который объявляет все процедуры get/set.

C.7.2 Артефакты и элементы реализации

Артефакты представляют собой конструкции языка программирования, содержащие **элементы реализации**. В ЯСО-2000 они отображаются на ссылочные типы процессов, определяемые в пакетах определений. **Артефакты** конкретизируются подобно множеству экземпляров в пределах типа процесса **CO**, который оно реализует.

Для доступа к данным в **CO**, определяется канал от множества экземпляров **артефакта** к процессу data_access. Он переносит все вызовы процедур к процессу data_access.

Элементы реализации ассоциируют **артефакты** и **элементы реализации**, которые реализует **артефакт**. Они не имеют представления в ЯСО.

Реализация **элемента взаимодействия** зависит от способа его реализации. Существует два способа реализации:

- способ поставки;
- способ использования.

В табл. C.1 приведена семантика видов **элементов взаимодействия** и способов реализации.

Таблица C.1/Z.130 – Семантика видов элементов взаимодействия и способов реализации

Тип элемента взаимодействия в модели проекта	Способ определения элемента реализации в модели проекта	Семантика элементов реализации
операция/атрибут	поставка	реализация поведение **операции** /поставка **операций** доступа атрибуту
операция/атрибут	использование	вызов явной операции возможен
поглощать	поставка	реализация поглощения сигнала
поглощать	использование	реализация отправления сигнала
порождать	поставка	реализация отправления сигнала
порождать	использование	реализация поглощения сигнала

Подобно отображению интерфейсов, описанному в C.5.10 относительно интерфейсов и элементов взаимодействия, каждый интерфейс eODL дополнительно отображается в пакете определения на пакет того же названия, что и интерфейс, содержащий два интерфейса ЯСО: exported_<Interface-name> и imported_<Interface-name>. Отображение **элементов взаимодействия** является точно таким, как описано в C.5.10, за исключением процедуры и сигнала. Каждая процедура и каждый сигнал несут дополнительный формальный **параметр** типа PId. Этот параметр используется для переноса информации отправителя/ получателя, соответственно. Для дополнительной информации см. C.7.4. Типы сигналов, на которые эти интерфейсы ссылаются (несущие дополнительный формальный **параметр** типа PId), определяются в пакете определения.

Последующие параграфы содержат дополнительные детали реализации **элементов взаимодействия**.

C.7.2.1 Операция реализации

Для того чтобы реализовать **элемент операционного взаимодействия** интерфейса, **артефакт** должен содержать экспортируемую процедуру, которая реализует процедуру, определенную в интерфейсе exported_<Interface-Name> ЯСО соответствующего интерфейса в пакете определения (см. последний параграф). Поэтому он реализует процедуру, содержащую дополнительный **параметр** типа PId. **Артефакт** может использовать этот **параметр** для получения информации об исходном отправителе вызова процедуры.

C.7.2.2 Вызов операции из артефакта

Вызовы операций других **COs** представляются следующим образом: **артефакт** посылает вызов процедуры того представления управления взаимодействиями, которое реализует `imported_<interface-name>` интерфейса, содержащего операцию вызова. Эта процедура определяется в пакете определения и содержит дополнительный формальный **параметр** типа `PId`. Фактический **параметр** `PId` используется для указания получателя вызова процедуры. Представление управления взаимодействиями отвечает за пересылку вызова процедуры к своему получателю.

C.7.2.3 Отправление сигнала

Аналогично вызову процедуры, **артефакт** отправляет сигнал не непосредственно к своему получателю, а к представлению управления взаимодействием. Сигнал содержит дополнительный формальный **параметр**, который указывает получателя сигнала и определяется в пакете определений. Представление управления взаимодействиями отвечает за пересылку сигнала к своему получателю.

C.7.2.4 Поглощение сигнала

Для того чтобы реализовать поглощение сигнала, **артефакту** необходимо реализовать обработчик сигнала, который принимает соответствующий сигнала, определенный в пакете определений.

C.7.2.5 Наследование артефактов

Для **артефактов** допускается множественное наследование. Так как в ЯСО множественное наследование для типов агентов не допускается, оно представляется делегированием. Базовые **артефакты** содержатся в производном **артефакте**, а соответствующие шлюзы каждого базового **артефакта** и производного **артефакта** соединяются посредством канала. Все вызовы процедур и сигналы, поступающие к непереопределяемым **элементам реализации**, являются прямо прошедшими из окружающей среды производного **артефакта** к соответствующему базовому **артефакту**. Если некий **элемент реализации** переопределяется в производном **артефакте**, переопределенный **элемент реализации** определяется в производном **артефакте**.

C.7.3 Управление артефактами и схема конкретизации

Управление **артефактами** отвечает за создание и управление экземплярами **артефакта**. Для каждого **артефакта**, реализующего **CO**, в пределах **CO** существует множество экземпляров `artefact_<CO-name>`. Однако, схема конкретизации, которая комментируется в модели, определяет, что экземпляр **артефакта** используется во взаимодействии. Управление **артефактом** реализует схему конкретизации.

Управление **артефактом** представляется как процесс `artefactmanagement`, содержащийся в типе процесса **CO**. Множество экземпляров этого процесса содержит точно один экземпляр. Для каждого типа **артефакта**, реализующего данный **CO**, существует экспортируемая удаленная процедура `get_artefact_<artefact-name>`, которая возвращает ссылку экземпляру этого типа **артефакта**. Эти процедуры реализуются автоматически. Их реализация зависит от используемой схемы конкретизации:

- *Артефакт по требованию*: В течение каждого вызова процедуры создается новый экземпляр и возвращается ссылка к нему;
- *Пул артефактов*: Создается ограниченное число экземпляров ("пул") и возвращается ссылка к одному из этих экземпляров;
- *Singleton*: Существует только один экземпляр **артефакта** и возвращается ссылка к нему;
- *Определенный пользователем*: Процедура не может автоматически исполняться, так как семантика определяется пользователем. Вместо этого объявляется ссылочная процедура и пользователь должен реализовать ее самостоятельно.

C.7.4 Представление управления взаимодействием

При отображении на ЯСО представление управления взаимодействием действует в качестве посредника между элементами реализации и окружающей средой **СО**. Каждое представление **элемента взаимодействия** имеет дело, как с входящими, так и с исходящими взаимодействиями (называемыми **СО**).

Каждый интерфейс, который требуется или поддерживается **СО**, представляется процессом в рамках типа процесса **СО**. Этот процесс реализует каждый **элемент взаимодействия** посредством пересылки запроса взаимодействия либо:

- от окружающей среды **СО** к соответствующему **элементу реализации**, тем самым, соблюдая схему конкретизации **артефакта** посредством использования представления управления **артефактом**; или
- от **элемента реализации** к окружающей среде **СО**.

Существует один и только один экземпляр процесса на интерфейс. Однако, если есть **порт множественный** этого типа, то существует исключение из этого правила. В этом случае есть определенное число экземпляров. Предметом реализации является конкретное число. Кроме того, для каждого процесса управления взаимодействиями существует неявная переменная `<process-name>_reference` внутреннего **СО**, которая хранит `PId` этого процесса. Если есть более чем одна ссылка экземпляра для хранения, используется строковый `PIds`.

C.7.4.1 Представление управления взаимодействиями, реализующее взаимодействие от СО к окружающей среде

Для взаимодействий к окружающей среде представление управления взаимодействием реализует:

- все **операционные** вызовы (только для требуемых интерфейсов); и
- все сигналы, которые могут быть отправлены.

Оно поддерживает интерфейс `imported_<interface-name>` пакета определения во входящем направлении (от артефактов), а в исходящем направлении (к окружающей среде) - интерфейс `imported_<interface-name>` определения интерфейсов.

Операция (в особенности, использование **операции**) реализуется следующим образом:

1) Значение `sender` запоминается в виде временной переменной.
2) Путем вызова соответствующей процедуры представления управлением **артефактом** запрашивается ссылка на экземпляр **артефакта**.
3) Со ссылкой на экземпляр **артефакта**, как место назначения, вызывается процедура (запомненное значение `sender` добавляется к списку параметров).
4) Если **операция** имеет объявленные **исключения**, эти **исключения** должны быть получены и обеспечена их установка вновь.

Отправление сигнала реализуется следующим образом: при получении представлением элемента взаимодействия определенного сигнала (с дополнительным параметром, указывающим место назначения), оно отправляет в место назначения определенный сигнал (без дополнительного **параметра**).

C.7.4.2 Представление управления взаимодействиями от окружающей среды к СО

Для взаимодействий в этом направлении, представление управления взаимодействиями реализует:

- все **операции**, объявленные в интерфейсе; и
- все порожденные и поглощенные **сигналы**.

Оно поддерживает интерфейс `exported_<interface-name>` пакета интерфейса во входящем направлении (от окружающей среды), а в исходящем направлении (к **артефактам**) – интерфейс `exported_<interface-name>` пакета определения.

Операция (более точно, поставка **операции** окружающей среде) реализуется следующим образом. При получении вызова процедуры посредством вызова управления **артефактом** запрашивается

ссылка экземпляра **артефакта**. Затем, делается вызов процедуры к этому экземпляру, при этом в качестве дополнительного **параметра** поставляется значение переменной `sender`. Если **операция** имеет **параметр** возврата, возвращаемое значение отдается обратно отправителю. Если устанавливается **операция**, представление управления взаимодействиями устанавливается снова.

Поглощение **сигнала** реализуется следующим образом: при получении сигнала от окружающей среды представлением **элемента взаимодействия**, оно отправляет **артефакту** соответствующий сигнал (с дополнительным **параметром**, содержащим значение переменной `sender`).

C.7.4.3 Управление портами

Управление портами является ответственным за:

- создание представлений **элементов взаимодействия**;
- управление ссылками к интерфейсам;
- реализация аксессорных операций, характерных для **портов**; и
- реализация общих аксессорных операций **портов**.

Все эти обязанности реализуются автоматически.

C.7.4.4 Представление управления портами

Управление портами реализуется несколькими экспортируемыми процедурами и переходом начала конечного автомата процесса **типа CO**.

C.7.4.5 Создание представлений управления взаимодействиями

Управление портами отвечает за создание процессов, представляющих управление взаимодействиями. Оно представляется в начале перехода процесса **типа CO**. После создания процесса управления взаимодействиями управление портами, используя переменную `interaction_<interface-name>`, обеспечивает хранение ссылок к каждому процессу в процессе доступа к данным.

C.7.4.6 Управление ссылками к интерфейсам

Для каждого **порта** имеется неявная переменная `port_<port-name>` местного **CO**. Тип этой переменной является либо ссылочным типом интерфейса, типизирующего порт (когда атрибут **port** является **single**), либо в виде строки PId (когда атрибут **port** является **multiple**). Операции **портов** напрямую манипулируют этими внутренними переменными.

C.7.4.7 Реализация аксессорных операций, характерных для портов

Отдельные операции реализуются следующим образом.

Порт предоставляемый простой: Эта операция возвращает ссылку, которая хранится во внутренней переменной.

Порт предоставляемый множественный: Должна выбираться одна ссылка из множества ссылок. Для принятия этого решения вызывается процедура `choose_provide_<port-name>` (определенная в пределах процесса управления портами). Эта процедура должна реализоваться пользователем, а поэтому является ссылочной.

Порт используемый простой: Операция `link` хранит данную ссылку в процессе доступа к данным, используя переменную `port_<port-name>`. Если там уже имеется сохраняемая ссылка, то устанавливается **исключение** `AlreadyConnected`. Операция `unlink` присваивает значение `Null` переменной местного **CO**. Если значение `Null` уже ей присвоено, операция устанавливает **исключение** `NotConnected`.

Порт используемый множественный: Операция `link` хранит данную ссылку в последовательности ссылок. Для того чтобы определить, где точно поместить ссылку в пределах последовательности, пользователь должен реализовать процедуру `choose_link_<port-name>` (определенную в рамках процесса управления портами), которая выбирает ссылку и должна сохранить ее где-то в последовательности. Аналогично, операция разъединения вызывает `choose_link_<port-name>`, чтобы удалить соответствующую ссылку.

C.8 Пропуск автоматически генерируемого поведения

Отображение eODL-ЯСО, представленное до сих пор, управляется мыслью о том, что пользователь хочет реализовать только бизнес-логику. Однако, если пользователь захочет генерировать из ЯСО код продукции, он или она могут захотеть избежать автоматического генерирования кода, а пожелают реализовать его самостоятельно.

Для обеспечения этого отображение предоставляет возможность пропустить автоматически генерируемый код. Более подробно это означает, что отсутствует генерирование следующих сущностей:

- управление **артефактами** и множествами экземпляров **артефактов**;
- представления управления взаимодействиями; и
- управление портами.

Вкратце, тип процесса **CO** не содержит никаких сущностей. Пользователь может реализовать тип процесса любым путем, которым он или она захотят.

C.9 Не отображаемые концепции eODL

Следующие концепции eODL не отображаются на ЯСО-2000:

- идентификация **рабочего цикла** любого типа;
- взаимодействие типа **непрерывных носителей**;
- **программные компоненты**;
- **сборки**;
- ограничения и сборки;
- концепции целевой окружающей среды;
- план внедрения.

C.10 Предопределенный пакет eodl

Ниже приведено полное содержание пакета **`eodl`**.

```
package eODL;
  syntype unsigned_short = Integer constants (0:65535);
  endsyntype;

  syntype unsigned_long = Integer constants (0:4294967295);
  endsyntype;

  syntype unsigned_long_long = Integer constants (0:18446744073709551615);
  endsyntype;

  syntype short = Integer constants (-32768:32767);
  endsyntype;

  syntype long = Integer constants (-2147483648:2147483647);
  endsyntype;

  syntype long_long = Integer constants
    (9223372036854775808:9223372036854775807);
  endsyntype;

  syntype char = Character endsyntype;
  syntype wchar = Natural endsyntype;

  syntype float = Real endsyntype;
  syntype double = Real endsyntype;
  syntype long_double = Real endsyntype;
```

```
value type wstring
    inherits String < wchar >;
  endvalue type wstring;

  value type wstring_bounded < synonym length Natural >
     inherits Vector < wchar, length>;
  endvalue type wstring_bounded;

  abstract value type TypeCode;
  endvalue type;

  value type fixedpt < synonym Width Natural; synonym scale Natural >;
    struct
      private unscaled_int Integer;
    operators
      Make ( Real ) -> this fixedpt;
      Make ( Integer ) -> this fixedpt;
       "+"  ( this fixedpt, this fixedpt ) -> this fixedpt;
       "-"  ( this fixedpt, this fixedpt ) -> this fixedpt;
       "*"  ( this fixedpt, this fixedpt ) -> this fixedpt;
       "/"  ( this fixedpt, this fixedpt ) -> this fixedpt;
       "="  ( this fixedpt, this fixedpt ) -> Boolean;
       ">"  ( this fixedpt, this fixedpt ) -> Boolean;
    methods
      toReal -> Real;

    OPERATOR Make (r Real ) -> this fixedpt  {
      DCL retVal this fixedpt;
      r := r * power(10,scale);
      retVal.unscaled_int := fix(r);
      return retVal;
    }
    OPERATOR Make (n Integer ) -> this fixedpt  {
      DCL retVal this fixedpt;
      retVal.unscaled_int := n * power(10,scale);
      return retVal;
    }
    OPERATOR  "+"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
      DCL retVal this fixedpt;
      retVal.unscaled_int := a.unscaled_int + b.unscaled_int;
      return retVal;
    }
    OPERATOR  "-"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
      DCL retVal this fixedpt;
      retVal.unscaled_int := a.unscaled_int - b.unscaled_int;
      return retVal;
    }
    OPERATOR  "*"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
      DCL retVal this fixedpt,
          t Real;
      t := float(a.unscaled_int * b.unscaled_int);
       t := t / float(power(10,2*scale));
      retVal.unscaled_int := Make( t );
      return retVal;
    }
    OPERATOR  "/"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
      DCL retVal this fixedpt,
           t Real;
      t := float(a.unscaled_int)/float(a.unscaled_int);
      retVal.unscaled_int := Make( t );
      return retVal;
    }
```

```
    OPERATOR  "="  (a this fixedpt, b this fixedpt ) -> Boolean {
      return a.unscaled_int = b.unscaled_int;
    }
    OPERATOR  ">"  (a this fixedpt, b this fixedpt ) -> Boolean {
      return a.unscaled_int > b.unscaled_int;
    }
    METHOD toReal -> Real {
      return float(unscaled_int)/float(power(10,scale));
    }
  endvalue type;

  package ComponentModel;
    value type ComponentKey;
       struct
         the_key string;
       methods
         virtual equal (this ComponentKey) -> Boolean;
    endvalue type;
    interface ComponentBase;
      procedure get_key -> ComponentKey;
    endinterface ComponentBase;

    procedure generate_CO_key -> ComponentKey external;

    value type ComponentKeySeq
      inherits String < ComponentKey >;
    endvalue type;
    interface CoFactoryBase;
      procedure get_co_typ -> string;
      procedure generic_create -> ComponentBase;
      procedure resolve_CO (ComponentKey) -> ComponentBase;
      procedure list_cos -> ComponentKeySeq;
    endinterface CoFactoryBase;

    exception NotConnected;
    interface ConfigBase;
      procedure provide(in string) -> PId
           raise NoSuchPort;
      procedure link(in string, in PId)
           raise AlreadyConnected, NoSuchPort;
      procedure unlink(in string, in ComponentBase)
           raise NotConnected, NoSuchPort;
    endinterface ConfigBase;

  endpackage ComponentModel;

  interface SDLComponent_Registry_IF;
    procedure register_SDLComponent(in string, in PId);
    procedure query_SDLComponent(in string) -> PId;
  endinterface;

  process type SDLComponent_Registry_Type;
    gate registry in with SDLComponent_Registry_IF;
    channel nodelay
      from env  to this via registry;
    endchannel;

    value type registry_store
      inherits Array < string, Pid >;
    endvalue type registry_store;

    dcl store registry_store := Make;

    exported as <<package eodl>>register_SDLComponent
```

```
procedure register_SDLComponent(in key string, in item Pid);
      start;
      task registry_store := Modify(store,key,item);
      return;
    endprocedure;

    exported as <<package eodl>>query_SDLComponent
    procedure register_SDLComponent(in key string) ->  Pid
           raise InvalidIndex;
      dcl retval Pid;
      start;
      task retval := Extract(store,key);
      return retval;
    endprocedure;

  endprocess type SDLComponent_Registry_Type;


endpackage eODL;
```

Приложение D

Представление XML метамодели eODL

Метамодель была определена с использованием UML. Ее представление XML согласно OMG XMI [6] предназначается для считывания инструментальными средствами и составляет Приложение D. Фактические данные содержаться в пакете программного обеспечения "Z.130 Annex D.xml".

ПРИМЕЧАНИЕ. – Пакет программного обеспечения Z.130 Annex D.xml имеется бесплатно в базе данных формального языка МСЭ-Т по адресу

http://www.itu.int/ITU-T/formal-language/xml/database/itu-t/z/z130/2003/.

Дополнение I

Пример: Обедающие философы

I.1 Введение

Целью этого дополнения является демонстрация примера того, как eODL может быть использован при проектировании, реализации и **внедрении** распределенной системы.

Проблема обедающих философов была впервые описана Эдсгером Дейкстрой в 1965 году. Она представляет собой модель и универсальный метод тестирования и сравнения теорий по распределению ресурсов. Дейкстра надеялся использовать ее, чтобы помочь создать многоуровневую операционную систему путем создания машины, которая может рассматриваться полностью детерминированным автоматом.

Реконфигурируемое число философов (процессов) сидят за круглым столом; конечное число вилок (ресурсов) находится на столе. Философы совершают действия – думают, едят и спят. Им не нужны никакие ресурсы для того, чтобы думать или спать, но для еды им нужны две вилки каждому: одна вилка для левой руки и одна – для правой. Поэтому перед началом еды любой философ старается взять две вилки, которые находятся перед ним. Это означает, что соседние философы не могут есть в одно и то же время.

При изменении деятельности, т. е. в моменты времени, когда какой-либо из философов начинает есть, думать или спать, все философы будут информировать наблюдателя. Кроме того, наблюдатель также информируется о наступлении критического состояния голода.

I.2 Описание

Проблема состоит в конечном множестве процессов, совместно использующих конечное множество ресурсов, каждый из которых в данный момент может использоваться только одним процессом, что приводит к потенциально тупиковым и смертельным ситуациям.

Конечное множество процессов, ресурсов и динамических взаимодействий между ними образует распределенную систему. Задача состоит в том, чтобы распределить реализацию ресурсов и процессов в целевой окружающей среде. Кроме того, ресурсы должны быть связаны с процессами.

Пример плана действий включает три различных **типа СО**:

– философ;

– вилка;

– наблюдатель.

Для того чтобы спроектировать, реализовать и внедрить пример необходимо сделать следующие шаги:

Фаза проектирования

– Определение модели элементов примера, включающих типы СО, порты и интерфейсы.

– Определение модели реализационной структуры.

Фаза реализации

– Реализация **артефактов** в соответствии с моделью (обеспечение бизнес-логики).

– Формирование **программных компонентов** в соответствии с моделью.

– Определение модели исходной структуры системы (**исходной конфигурации**) путем определения сборки.

– Пакетирование **программных компонентов** и относящейся к ним информационной модели для отправки реализации клиентам.

Фаза интеграции

– Доставка пакета клиенту.

– Моделирование целевой окружающей среды клиентских условий выполнения.

– Определение должного присвоения **программных компонентов**, содержащихся в пакете, целевой окружающей среде.

– Инсталляция присвоенных **программных компонентов** в установленных целевых узлах.

– Создание **исходной конфигурации** посредством взаимного соединения всех **исходных СОs** в соответствии с **исходной конфигурацией**.

В параграфе I.3 приведен пример "Обедающих философов" с использованием eODL. В спецификации определяются три **типа СО**:

– Тип объекта o_Philosopher представляет философа.

– Тип объекта o_Fork представляет вилку.

– Тип объекта o_Observer представляет наблюдателя.

Параграф I.4 содержит отображение модели eODL в соответствии с правилами отображения, приведенными в Приложении C. Отображаются только концепции представлений вычисления, конфигурации и реализации, так как отображение концепций представления внедрения отсутствует. Модель ЯСО состоит из двух основных пакетов:

– пакет интерфейса ЯСО phil_interface;

– пакет описания ЯСО phil_definition.

I.3 Пример на языке eODL

```
module DiningPhilosophers {
     CO o_Philosopher;
     CO o_Fork;

     interface i_Fork;
     interface i_Philosopher;
     interface i_Observer;

     exception ForkNotAvailable {};
     exception NotTheEater {};

     enum e_ForkState {
          UNUSED,
          USED,
          WASHED
     };

     enum e_Pstate {
          EATING,
          THINKING,
          SLEEPING,
          DEAD,
          CREATED,
          HUNGRY
     };

     interface i_Fork {
        void obtain_fork ( in o_Philosopher eater )
        raises ( ForkNotAvailable );
        void release_fork ( in o_Philosopher eater ) raises ( NotTheEater );
     };

     artefact a_ForkImpl {
          obtain_fork implements supply i_Fork::obtain_fork;
          release_fork implements supply i_Fork::release_fork;
     };

     CO o_Fork {
          supports i_Fork;
          provide i_Fork fork;
          implemented by a_ForkImpl with ArtefactPool(2);
     };

     interface i_Philosopher {
          void set_name ( in string name);
     };

     artefact a_PhilosopherImpl {
          set_name_impl implements supply i_Philosopher::set_name;
          pstate_impl implements use i_Observer::pstate;
     };

     CO o_Philosopher {
          implemented by a_PhilosopherImpl with Singleton;
          supports i_Philosopher;
          requires i_Fork, i_Observer;
          use I_observer observer;
          use i_Fork left;
```

```
          use i_Fork right;
     };

     valuetype Pstate {
          public e_PState state;
          public string name;
          public i_Philosopher philosoph;
          factory create (
               in e_PState state,
               in string name,
               ini_Philosopher philo);
     };

     signal PhilosopherState {
          PState carry_pstate;
     };

     interface i_Observer {
          consume PhilosopherState pstate;
     };

     artefact a_Observer {
          pstate_Impl implements supply i_Observer::pstate;
     };

     CO o_Observer {
          implemented by a_Observer with Singleton;
          supports i_Observer;
          provide i_Observer observer;
     };
};
softwarecomponent Philosopher
realizes o_Philosopher, o_Observer
{
     requires {
          property os =  [
                              { name = "WINNT"; version = "4,0,0,0"; },
                              { name = "WIN98"; }
          ];
     };
};

softwarecomponent Fork
realizes o_Fork;
{
     requires {
          property os =  [
                              { name = "WINNT"; version = "4,0,0,0"; },
                              { name = "WIN98"; }
          ];
     };
};

assembly ass1 {
     p (3) : o_Philosopher;
     f1 : o_Fork;
     f2 : o_Fork;
     o : o_Observer;
     connect c1 {
          p.left  = f1.fork;
          p.right = f2.fork;
     };
```

```
     connect c2 { o.observer = p.observer; };
};

environment myenv_1 {
     node n1 {
          property os =  { name = "WINNT"; version = "4,0,0,0"; };
          property memory = 256;
     };

     node n2 {
          property os =  { name = "WINNT"; version = "4,0,0,0"; };
          property memory = 128;
     };

     link l1 { node n1, n2; };
};

installation install1
uses environment myenv_1 {
     Philosopher ->n2;
     Fork ->n1;
};

instantiation instantiate1
uses environment myenv_1
uses assembly ass1 {
     p,  o  -> n2;
     f1, f2 ->  n1;
};

deploy {
     install { install1; };
     instantiate { instantiate1; };
};
```

I.4 Пример на языке ЯСО-2000

```
use eODL;
/* /--------------------------------\ */
/*     типы данных и интерфейс,       */
/*     требуемые клиентами           */
package phil_interface;

package DiningPhilosophers;

  /* исключения */
  exception ForkNotAvailable;
  exception NotTheEater;

  /* перечисления */
  value type e_ForkState;
    literals
      UNUSED, USED;
  endvalue type;

  value type e_ForkState;
    literals
      EATING, THINKING, SLEEPING,
      DEAD, CREATED, HUNGRY;
  endvalue type;

  /* интерфейс i_Fork */
  package i_Fork;
```

```
    /* объявление экспортируемых процедур */
    interface exported_i_Fork;
      procedure obtain_fork(in o_Philosopher)
             raise ForkNotAvailable;
      procedure release_fork(in o_Philosopher)
               raise NoTheEater;
    endinterface;

     /* объявление поглощенных сигналов */
    interface imported_i_Fork;
         /* отсутствие объявления поглощенных сигналов */
    endinterface;
  endpackage;

  /* определение типа CO o_Fork */
  use i_Fork;
  package o_Fork;

    /* содержит атрибуты, определенные в типе CO o_Fork */
    interface o_Fork_attributes
        inherits <<package eODL/package ComponentModel>>ComponentBase;
    endinterface;

    /* операции порта */
    interface o_Fork_config
        inherits <<package eODL/package ComponentModel>>ConfigBase adding;
      /* порт предоставленный "fork" */
      procedure provide_fork -> exported_i_Fork;
    endinterface;

    /* объединение combine интерфейсов конфигурации и атрибутов */
    interface o_Fork
      inherits o_Fork_attributes, o_Fork_config;
    endinterface;

    /* объявление интерфейса фабрики CO */
    package factory;
      interface o_Fork_factory
       inherits <<package eODL/package ComponentModel>>CoFactoryBase adding;
        procedure create_o_Fork -> o_Fork;
      endinterface;
    endpackage;

  endpackage o_Fork;

package i_Philosopher;

    interface exported_i_Philosopher;
      procedure set_name(in string);
    endinterface;

    interface imported_i_Philosopher;
    endinterface;

  endpackage;

  use i_Philosopher;
  use o_Philosopher;
  object type Pstate;
    struct
      public eodl_state e_PState; /* состояние -> eodl_state ! */
      public name string;
       public philosoph exported_i_Philosopher;
    operators
```

```
      /* создавать -> eodl_create) */
      eodl_create(e_PState, string, exported_i_Philosopher) -> Pstate;
      make(e_PState, string, exported_i_Philosopher) -> PState;

    operator eodl_create(eodl_state e_PState,
                         name string,
                         philo exported_i_Philosopher) {
      dcl retval Pstate;
       retval.eodl_state := eodl_state;
      retval.name := name;
      retval.philosoph := philo;
       return retval;
    }
    operator make(eodl_state e_PState,
                  name string,
                  philo exported_i_Philosopher) {
       return eold_create(eodl_state,name,philo);
    }
  endobject type;

  use i_Philosopher;
  package o_Philosopher;

    /* содержит атрибуты, определенные в типе CO o_Fork */
    interface o_Philosopher_attributes
        inherits <<package eODL/package ComponentModel>>ComponentBase;
    endinterface;

    interface o_Philosopher_config
     inherits <<package eODL/package ComponentModel>>ConfigBase adding;
      procedure link_observer(exported_i_Observer) raise AlreadyConnected;
      procedure link_left(exported_i_Fork) raise AlreadyConnected;
      procedure link_right(exported_i_Fork) raise AlreadyConnected;
      procedure unlink_observer raise NotConnected;
      procedure unlink_left raise NotConnected;
      procedure unlink_right raise NotConnected;
    endinterface;

    interface o_Philosopher
         inherits o_Philosopher_attributes, o_Philosopher_config;
    endinterface;

    use eODL / package ComponentModel;
    package factory;
      interface o_Fork_factory
         inherits <<package eODL/package ComponentModel>>CoFactoryBase adding;
          procedure create_o_Philosopher -> o_Philosopher;
      endinterface;
    endpackage factory;

  endpackage;

  signal PhilosopherState(PState);

package i_Observer;

    interface exported_i_Observer;
      use PhilosopherState;
    endinterface;

    interface imported_i_Observer;
    endinterface;

  endpackage;
```

```
  /* CO o_Observer */
  use i_Observer;
  package o_Observer;

    interface o_Observer_attributes
        inherits <<package eODL/package ComponentModel>>ComponentBase adding;
    endinterface;

    interface o_Observer_config
        inherits <<package eODL/package ComponentModel>>ConfigBase adding;
    endinterface;

    interface o_Observer inherits o_Observer_attributes, o_Observer_config;
    endinterface;

    package factory;
      interface o_Observer_factory
       inherits <<package eODL/package ComponentModel>>CoFactoryBase adding;
        procedure create_o_Observer -> o_Observer;
      endinterface;
    endpackage;

  endpackage;

endpackage DiningPhilosophers;

endpackage phil_interface;
/* \--------------------------------/ */

/* /--------------------------------\ */
/*     пакет реализации               */
use eODL;
package phil_definition;

package DiningPhilosophers;


  /* используемый при определении операций, реализованных или */
  /* используемых артефактами */
  package i_Fork;
    /* операции, реализованные артефактами */
    interface exported_i_Fork;
      procedure obtain_fork(in o_Philosopher, in Pid)
             raise ForkNotAvailable;
      procedure release_fork(in o_Philosopher, in Pid)
                raise NoTheEater;
    endinterface;

    /* операции, используемые артефактами */
    interface imported_i_Fork;
    endinterface;
  endpackage;


  /* атрибуты состояния */
  package o_Fork_data;

    interface internal_data;
      procedure get_port_fork -> exported_i_Fork;
      procedure get_interaction_i_Fork -> exported_i_Fork;
      procedure set_port_fork(exported_i_Fork);
      procedure set_interaction_i_Fork(exported_i_Fork);
```

```
  endinterface;
endpackage;

signallist a_ForkImpl_in =
     procedure <<package phil_definition/package DiningPhilosophers/
                   package i_Fork>>obain_fork,
      procedure <<package phil_definition/package DiningPhilosophers/
                  package i_Fork>>release_fork;

/* артефакт: справочное определение */
use a_PhilosopherImpl;
use o_Fork_state;
process type a_ForkImpl with
   use (a_ForkImpl_in);;
  referenced;

/* /--------------------------------\ */
/*     компонент SDL o_Fork_CO        */
use a_PhilosopherImpl;
use o_Fork_state;
use a_ForkImpl;
block type o_Fork_CO;

  /* определения шлюза */
  gate factory
    in with <<package phil_interface/package DiningPhilosopher/
        package o_Fork/package factory>>o_Fork_factory;
  gate initial
    in with  (<<package phil_interface/
        package DiningPhilosopher/package o_Fork>>o_Fork);
  gate provides
    in with <<package phil_interface/
        package DiningPhilosopher/package i_Fork>>exported_i_Fork;
    out with <<package phil_interface/
        package DiningPhilosopher/package i_Fork>>imported_i_Fork;

  /* определяет процесс фабрики */
  process type o_Fork_factory;
    gate factory
      in with <<package phil_interface/package DiningPhilosopher/
       package o_Fork/package factory>>o_Fork_factory; /* ; zuviel  */
    channel nodelay
      from this via factory to env;
    endchannel;

    /* хранит здесь ключи компонент */
    dcl keys ComponentKeysSeq;

    /* создает CO и возвращает ссылку на него */
    exported as <<package phil_interface/package DiningPhilosopher/
        package o_Fork/package factory>>generic_create
    procedure generic_create -> ComponentBase;
   dcl key ComponentKey;
       start;
      create co_instance;
       task key := call get_key to offspring;
        task keys := Modify(keys, key, offspring);
      return offspring;
    endprocedure;

    /* создает CO и возвращает ссылку на него */
    exported as <<package phil_interface/package DiningPhilosopher/
        package o_Fork/package factory>>create_o_Fork
    procedure create_o_Fork -> o_Fork;
```

```
      dcl key ComponentKey;
      start;
    create co_instance;
      task key := call get_key to offspring;
      task keys := Modify(keys, key, offspring);
    return offspring;
  endprocedure;

  /* возвращает имя типа CO */
  exported as <<package phil_interface/package DiningPhilosopher/
     package o_Fork/package factory>>get_co_type
  procedure get_co_type -> string;
    start;
    return 'DiningPhilosophers.o_Fork';
  endprocedure;

  /* возвращает ссылку на CO */
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>resolve_co
   procedure resolve_co(in key ComponentKey) -> ComponentBase
        raise InvalidIndex;
      dcl returnValue Pid;
      dcl i Integer := 1;
      start;
      task returnValue := Extract(keys,key);
      return returnValue;
   endprocedure;

  /* возвращает список ключей CO */
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>list_co
   procedure list_co -> ComponentKeySeq;
      start;
      return keys;
   endprocedure;

endprocess type o_Fork_factory;
/* \------------------------------/ */

/* определяет сам тип CO */
process type o_Fork;

  /* шлюзы, используемые для comm. с окружающей средой */
  gate initial
    in with  (<<package phil_interface/package DiningPhilosopher
          /package o_Fork>>o_Fork);
  gate provides
    in with <<package phil_interface/package DiningPhilosopher
          /package i_Fork>>exported_i_Fork; /* NOS */
    out with <<package phil_interface/package DiningPhilosopher
          /package i_Fork>>imported_i_Fork;

  /* /------------------------------\ */
  /*     инкапсулирует переменные состояния     */
  process data_access(1,1);

    dcl reference_interaction_i_Fork exported_i_Fork;
    dcl reference_port_fork exported_i_Fork;
    dcl the_key string;


  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_port_fork
  procedure get_port_fork -> exported_i_Fork;
```

```
    start;
    return reference_port_fork;
  endprocedure;
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_interaction_i_Fork
  procedure get_interaction_i_fork -> exported_i_Fork;
    start;
    return reference_interaction_i_Fork;
  endprocedure;
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>set_port_fork
  procedure set_port_fork(in ref exported_i_Fork);
    start;
    reference_port_fork := ref;
  endprocedure;
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>set_interaction_i_Fork
  procedure set_interaction_i_fork(in ref exported_i_Fork);
    start;
    reference_interaction_i_Fork := ref;
  endprocedure;

endprocess data_access;
/* \--------------------------------/ */

/* артефакт канала <--> state_access */
channel nodelay
  from artefact_a_ForkImpl to data_accessor
    with internal_state;
endchannel;


/* в случае наследования типа CO экземпляр этого типа */
/* содержится в типе "eODL-inherited". Следующий */
/* шлюз используется в качестве интерфейса с процессом state_access */
/* наследуемого типа. */
gate data_accessor
  in with internal_data;
channel route_state_access nodelay
  from
    env via data_accessor to data_access;
endchannel;


/* /--------------------------------\ */
/*     управляет экземплярами артефактов  */
process artefactmanagement(1,1);
  /* сигналы для делегирования создания артефактов */
  signal create_a_ForkImpl_req;
  signal create_a_ForkImpl_res(Pid);
  /* пул артефактов и указатель в пуле */
  dcl a_ForkImpl_seq PIdSeq := (. .);
  dcl a_ForkImpl_ptr Integer := 0;
  /* возвращает ссылку экземпляра артефакта */
  /* реализует пул размером POOLSIZE */
  /* создает новый артефакт, если пул еще не имеет*/
  /* размера POOLSIZE, в противном случае возвращает "следующий" */
  /* артефакт. */
  exported procedure get_artefact_a_ForkImpl -> PId;
    dcl new_pid PId;
    start;
     decision length(a_ForkImpl_seq);
      (0:1):
        /* делегировать создание артефакта */
```

```
          output create_a_ForkImpl;
            nextstate wait4response;
        else:
          task a_ForkImpl_ptr := a_ForkImpl_ptr+1;
          task { if (a_ForkImpl_ptr>length(a_ForkImpl_seq))
                 a_ForkImpl_ptr := 1;
               };
          return extract(a_ForkImpl_seq, a_ForkImpl_ptr);
      enddecision;
      /* делегирование продолжается ... */
      state wait4response;
      input create_a_ForkImpl_res(new_pid);
      task a_ForkImpl_seq := a_ForkImpl_seq // new_pid;
      return offspring;
    endprocedure;
    start;
    nextstate wait_for_signal;
    /* создать экземпляр артефакта для процедуры */
    state wait_for_signal;
    input create_a_ForkImpl_req;
    create artefact_a_ForkImpl;
     output create_a_ForkImpl_res(offspring) to sender;
    nextstate -;
  endprocess;
  /* \------------------------------/ */

  /* это - представление управления взаимодействием */
  /* для интерфейса i_fork. имеет дело с вызовами процедур */
  /* из окружающей среды */
  process interaction_i_Fork(0,1);
    exported as <<interface exported_i_Fork>>obtain_Fork
    procedure obtain_fork(in eater o_Philosopher)
            raise ForkNotAvailable;
      dcl p PId;
      start;
      task p := get_artefact_a_ForkImpl;
      call obtain_Fork(who, sender) to p;
      return;
        exceptionhandler defhandler;
        handle ForkNotAvailable;
        raise ForkNotAvailable;
      endexceptionhandler defhandler;
    endprocedure;
    exported as <<interface exported_i_Fork>>release_Fork
    procedure release_fork(in eater o_Philosopher)
      raise NotTheEater;
      dcl p PId;
      start;
      task p := get_artefact_a_ForkImpl;
      call obtain_Fork(who,sender) to p;
      return;
        exceptionhandler defhandler;
        handle NotTheEater;
        raise NotTheEater;
      endexceptionhandler defhandler;
    endprocedure;
  endprocess;

  /* управление портом */
  exported as <<package phil_interface/
       package DiningPhilosopher>>provide_fork
  procedure provide_fork -> exported_i_Fork;
    start;
    return call provide_fork;
```

```
endprocedure;
exported as <<package philo_interface/
   package DiningPhilosopher/package o_Fork>>provide
procedure provide(s string) -> PId
     raise NoSuchPort;
  start;
  decision s;
     (='fork'): return call provide_fork;
    else: raise NoSuchPort;
  enddecision;
endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Fork>>port_connect
procedure port_connect(s string) -> PId
   raise NoSuchPort,AlreadyConnected;
  start;
  raise NoSuchPort;
endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Fork>>port_disconnect
procedure port_disconnect(s string) -> PId
    raise NoSuchPort,NotConnected;
  start;
  raise NoSuchPort;
endprocedure;
  /* get/set для внутренних переменных */
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_key
  procedure get_key -> ComponentKey;
    start;
    return the_key;
  endprocedure;


/* вычисляет ключ и конкретизирует       */
/* представления управления взаимодействием */
start;
task the_key := <<package eODL>>generate_key;
create interaction_i_Fork;
call set_interaction_i_Fork(offspring);
task reference_port_fork := offspring;
nextstate initial_state;

/* обрабатывать множество экземпляров артефакта a_ForkImpl */
process  artefact_a_ForkImpl(0,2): a_ForkImpl;

/* соединяет артефакт и imr */
channel interaction_i_fork_a_fork_impl nodelay
   from interaction_i_Fork to artefact_a_ForkImpl
      with procedure <<package phil_definition/package DiningPhilosophers
          /package i_Fork>>obtain_fork,
          procedure <<package phil_definition/package DiningPhilosophers
          /package i_Fork>>release_fork;
endchannel;

channel ch_i_Fork nodelay
   from env via provides to interaction_i_Fork
      with exported_i_Fork;
   from interaction_i_Fork to env via provides
      with imported_i_Fork;
endchannel;

 channel ch_initial nodelay
  from env via initial to this
```

```
        with config_o_Fork;
    endchannel;

  endprocess type o_Fork;
  /* \--------------------------------/ */

  /* множества экземпляров фабрики и типа co */
  process factory(1,1): o_Fork_factory;
  process cos(0,): o_Fork;

  channel factory_to_co nodelay
    from factory to co via initial;
  endchannel;
  channel initial_to_env nodelay
    from env via initial to co via initial;
  endchannel;
  channel provides_to_env nodelay
    from cos via provides to env via provides;
    from env via provides to cos via provides;
  endchannel;
  channel uses_to_env nodelay
    from cos via uses to env via uses;
    from env via uses to cos via uses;
  endchannel;


endblock type o_Fork_CO;
/* \--------------------------------/ */

package i_Philosopher;

    interface exported_i_Philosoper;
      procedure set_name(in string, in Pid);
    endinterface;

    interface imported_i_Philosoper;
    endinterface;

  endpackage i_Philosopher;

  package o_Philosopher_data;

    interface internal_data;
      procedure get_port_observer -> i_Observer;
      procedure get_port_left -> i_Fork;
      procedure get_port_right -> i_Fork;
      procedure get_interaction_i_Fork -> imported_i_Fork;
      procedure get_interaction_i_Observer -> imported_i_Observer;
      procedure get_interaction_i_Philosopher -> exported_i_Philosopher;
      procedure set_port_observer(i_Observer);
      procedure set_port_left(i_Fork);
      procedure set_port_right(i_Fork);
      procedure set_interaction_i_Fork(imported_i_Fork);
      procedure set_interaction_i_Observer(imported_i_Observer);
      procedure set_interaction_i_Philosopher(exported_i_Philosopher);
    endinterface;

  endpackage;

  signallist a_PhilosopherImpl :=
    procedure <<package phil_definition/package DiningPhilosophers/
                    package i_Philosopher>>set_name;
```

```
process type a_PhilosopherImpl with
   use (a_PhilosopherImpl_in);;
  referenced;


block type o_Philosopher_CO;

  gate factory
    in with <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>o_Philosopher_factory;
  gate initial
    in with  <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher>>o_Philosopher;
  gate provides
    in with <<package phil_interface/package DiningPhilosophers/
            package i_Philosopher>>exported_i_Philosopher;
    out with <<package phil_interface/package DiningPhilosophers/
            package i_Philosopher>>imported_i_Philosopher;
  gate uses
    out with <<package phil_interface/package DiningPhilosophers/
            package i_Fork>>exported_i_Fork,
             <<package phil_interface/package DiningPhilosophers
            /package i_Observer>>exported_i_Observer;

  process type o_Philosopher_factory;
    gate factory
      in with <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>o_Philosopher_factory;
    channel nodelay
      from this via factory to env;
     endchannel;

     dcl keys ComponentKeysSeq;

    exported as <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>generic_create
     procedure generic_create -> ComponentBase;
       dcl key ComponentKey;
        start;
       create co_instance;
        task key := call get_key to offspring;
        task keys := keys // key;
       return offspring;
    endprocedure;

    exported as <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>create_o_Philosopher
     procedure create_o_Philosopher -> o_Philosopher;
       dcl key ComponentKey;
       start;
      create co_instance;
       task key := call get_key to offspring;
       task keys := keys // key;
      return offspring;
    endprocedure;

    exported as <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>get_co_type
    procedure get_co_type -> string;
      start;
      return 'o_Philosopher';
    endprocedure;

    /* возвращает ссылку на CO */
```

```
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>resolve_co
   procedure resolve_co(in key ComponentKey) -> ComponentBase
      raise InvalidIndex;
      dcl returnValue Pid;
      dcl i Integer := 1;
      start;
      task returnValue := Extract(keys,key);
      return returnValue;
   endprocedure;

  /* возвращает список ключей CO */
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>list_co
   procedure list_co -> ComponentKeySeq;
      start;
      return keys;
   endprocedure;

endprocess type;


process type o_Philosopher;
  gate initial
    in with  <<package phil_interface/package DiningPhilosophers/
          package o_Philosopher>>o_Philosopher;

  channel ch_initial nodelay
    from env via initial to this with o_Philosopher;
  endchannel;

  gate provides
    in with <<package phil_interface/package DiningPhilosophers/
          package i_Philosopher>>exported_i_Philosopher;
    out with <<package phil_interface/package DiningPhilosophers/
          package i_Philosopher>>imported_i_Philosopher;
  channel ch_provides nodelay
    from env via provides to interaction_i_Philosopher
          with exported_i_Philosopher;
    from interaction_i_Philosopher via provides to env
          with imported_i_Philosopher;
  endchannel;

  gate uses
    out with <<package phil_interface/package DiningPhilosophers/
          package i_Fork>>exported_i_Fork,
              <<package phil_interface/package DiningPhilosophers/
          package i_Observer>>exported_i_Observer;

  dcl the_key string;

  process data_access(1,1);
   dcl reference_interaction_i_Philosopher exported_i_Philosopher;
   dcl reference_interaction_i_Fork imported_i_Fork;
   dcl reference_interaction_i_Observer imported_i_Observer;
   dcl reference_port_observer exported_i_Observer := Null;
   dcl reference_port_left exported_i_Fork := Null;
   dcl reference_port_right exported_i_Fork := Null;

 exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>get_port_observer
    procedure get_port_observer -> i_Observer;
        start;
        return reference_port_observer;
```

```
     endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>set_port_observer
    procedure set_port_observer(ref i_Observer);
  start;
  task reference_port_observer := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>get_port_left
    procedure get_port_left -> i_Fork;
  start;
  return reference_port_left;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>set_port_left
    procedure set_port_left(ref i_Fork);
  start;
  task reference_port_left := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>get_port_right
    procedure get_port_right -> i_Fork;
  start;
  return reference_port_right;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>set_port_right
    procedure set_port_right(ref i_Fork);
  start;
  task reference_port_right := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>get_interaction_i_Fork
procedure get_interaction_req_i_Fork -> imported_i_Fork;
  start;
  return reference_interaction_i_Fork;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>set_interaction_i_Fork
procedure set_interaction_i_Fork(ref imported_i_Fork);
  start;
  task reference_interaction_i_Fork := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>get_interaction_i_Observer
    procedure get_interaction_i_Observer -> imported_i_Observer;
  start;
  return reference_interaction_i_Observer;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>set_interaction_i_Observer
    procedure set_interaction_i_Observer(ref imported_i_Observer);
  start;
  task reference_interaction_i_Observer := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>get_interaction_i_Philosopher
    procedure get_interaction_i_Philosopher -> exported_i_Philosopher;
  start;
  return reference_interaction_i_Philosopher;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>set_interaction_i_Philosopher
    procedure set_interaction_i_Philosopher(ref exported_i_Philosopher);
```

```
   start;
   task reference_interaction_i_Philosopher := ref;
 endprocedure;

  endprocess;


/* артефакт канала <--> state_access */
channel nodelay
  from artefact_a_ForkImpl to data_accessor
    with internal_state;
endchannel;


/* в случае типа CO наследование экземпляра этого типа */
/* содержится в типе "наследованный eODL" . Следующий */
/* шлюз используется для связи с процессом state_access */
/* наследованного типа. */
gate data_accessor
  in with internal_data;
channel route_state_access nodelay
  from
    env via data_accessor to data_access;
endchannel;

  process artefactmanagement(1,1);
    signal create_a_PhilosopherImpl_req;
    signal create_a_PhilosopherImpl_res(Pid);
    dcl a_PhilosopherImpl_ptr PId := Null;
    exported procedure get_artefact_a_PhilosopherImpl;
      dcl new_pid PId;
      start;
       decision a_PhilosopherImpl_ptr;
        (Null):
           output create_a_PhilosopherImpl;
            nextstate wait4response;
        else:
          return a_PhilosopherImpl_ptr;
      enddecision;

      state wait4response;
       input create_a_PhilosopherImpl_res(new_pid);
      task a_PhilosopherImpl_ptr := new_pid;
      return offspring;
    endprocedure;
    start;
    nextstate wait_for_signal;
    state wait_for_signal;
    input create_a_PhilosopherImpl;
    create artefact_a_PhilosopherImpl;
    nextstate -;
  endprocess;

  channel nodelay
  from
    artefact_a_PhilosopherImpl
    to state_accessor
    with internal_state;
  endchannel;

  process interaction_i_Philosopher(0,1);
     exported as <<package phil_definition/package DiningPhilosophers/
           package i_Philosopher>>set_name
    procedure set_name(in name string);
```

```
    dcl p PId;
    start;
     task p := get_artefact_a_PhilosopherImpl;
     call set_name(name,sender) to p;
     return;
   endprocedure;
 endprocess;

process interaction_i_Fork(0,1);
   exported as <<package phil_definition/package DiningPhilosophers/
        package i_Fork>>release_fork
  procedure obtain_fork(in eater o_Philosopher, in server PId);
    start;
     call obtain_fork(eater) to server;
     return;
   endprocedure;

  exported as <<package phil_definition/package DiningPhilosophers/
   package i_Fork>>release_fork
  procedure release_fork(in eater o_Philosopher, in server PId);
    start;
     call obtain_fork(eater) to server;
     return;
   endprocedure;
 endprocess;

 process interaction_i_Observer(0,1);
   dcl carry_PhilosopherState PhilosopherState;
   dcl consumer PId;
   start;
   nextstate signal_handler;
   state signal_handler;
   input PhilosopherState(carry_PhilosopherState, consumer);
   output PhilosopherState(carry_PhilosopherState) to consumer;
   nextstate -;
 endprocess;


process artefact_a_PhilosopherImpl(0,1): a_PhilosopherImpl;

  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_observer
  procedure link_observer -> exported_i_Observer;
     start;
      decision call get_port_observer;
     (Null): call set_port_observer(ref);
      else: raise AlreadyConnected;
      enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_observer
  procedure unlink_observer -> exported_i_Observer;
     start;
      decision call get_port_observer;
     (Null): raise NotConnected;
      else: call set_port_observer(ref);
      enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_left
  procedure link_left(ref imported_i_Fork)
        raise AlreadyConnected;
     start;
      decision call get_port_left;
```

```
    (Null): call set_port_left(ref);
     else: raise AlreadyConnected;
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_left
  procedure unlink_left
        raise NotConnected;
     start;
     decision call get_port_left;
    (Null): raise NotConnected;
     else: call set_port_left(Null);
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_right
  procedure link_right(ref imported_i_Fork)
        raise AlreadyConnected;
     start;
     decision call get_port_right;
    (Null): call set_port_right ( ref );
     else: raise AlreadyConnected;
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_right
  procedure unlink_right
        raise NotConnected;
     start;
     decision call get_port_right;
    (Null): raise NotConnected;
     else: call set_port_right(Null);
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_observer
  procedure link_observer(ref imported_i_Observer)
        raise AlreadyConnected;
     start;
     decision call get_port_observer;
    (Null): call set_port_observer(ref);
     else: raise AlreadyConnected;
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_observer
  procedure unlink_observer
        raise NotConnected;
     start;
     decision call get_port_observer;
    (Null): raise NotConnected;
     else: call set_port_observer(Null);
     enddecision;
  endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Philosopher>>provide
procedure provide(s string) -> PId
     raise NoSuchPort;
  start;
   enddecision;
endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Philosopher>>port_connect
procedure link(ref Pid, s string)
```

```
      raise NoSuchPort,AlreadyConnected;
    start;
    decision s;
      (='observer'): call link_observer(ref);
      (='left'): return call link_left(ref);
      (='right'): return call link_right(ref);
      else: raise NoSuchPort;
    return ;
  endprocedure;
  exported as <<package philo_interface/
       package DiningPhilosopher/package o_Philosopher>>port_disconnect
  procedure unlink(s string) -> PId
      raise NoSuchPort,NotConnected;
    start;
    decision s;
      (='observer'): call unlink_observer;
      (='left'): return call unlink_left;
      (='right'): return call unlink_right;
      else: raise NoSuchPort;
    return ;
  endprocedure;

 start;
    task the_key := <<package eODL>>generate_key;
 create interaction_i_Fork;
    call set_interaction_i_Fork(offspring);
 create interaction_i_Philosopher;
 call set__interaction_i_Philosopher(offspring);
 create interaction_i_Observer;
 call set__interaction_i_Observer(offspring);
 nextstate initial_state;

  exported as <<package phil_interface/package DiningPhilosophers/
       package o_Philosophers>>get_key
  procedure get_key -> ComponentKey;
    start;
    return the_key;
  endprocedure;

endprocess type;

process factory(1,1): o_Fork_factory;
process cos(0,): o_Fork;


channel factory_to_co nodelay
  from factory to co via initial;
endchannel;

channel initial_to_env nodelay
  from co via initial to env via initial;
endchannel;

channel provides_to_env nodelay
  from co via provides to env via provides;
endchannel;

channel uses_to_env nodelay
  from co via uses to env via uses;
endchannel;

channel factory_to_env nodelay
  from env via factory to  factory;
endchannel;
```

```
  endblock type;

package o_Observer_data;
    interface internal_data;
      procedure get_interaction_i_Observer -> exported_i_Observer;
      procedure get_port_observer -> exported_i_Observer;
      procedure set_interaction_i_Observer(exported_i_Observer);
      procedure set_port_observer(exported_i_Observer);
    endinterface;
  endpackage;

  signal PhilosopherState(PState, Pid);

  signallist a_Observer_in =
       <<package phil_definition/package DiningPhilosophers>>PhilosopherState;

  /* артефакт: справочное определение */
  process type a_ForkImpl with
     use (a_Observer_in);;
    referenced;



  block type o_Observer_CO;


    gate factory
      in with <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>o_Observer_factory;
    gate initial
      in with  <<package phil_interface/package DiningPhilosophers/
              package o_Observer>>o_Observer;
    gate provides
      in with <<package phil_interface/package DiningPhilosophers/
              package i_Observer>>exported_i_Observer;
      out with <<package phil_interface/package DiningPhilosophers/
              package i_Observer>>imported_i_Observer;

    process type o_Observer_factory;
      gate factory
        in with <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>o_Observer_factory;
      channel nodelay
        from this via factory to env;
       endchannel;

       dcl keys ComponentKeysSeq;

      exported as <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>generic_create
       procedure generic_create -> ComponentBase;
        dcl key ComponentKey;
         start;
        create co_instance;
         task key := call get_key to offspring;
         task keys := keys // key;
        return offspring;
      endprocedure;

      exported as <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>create_o_Observer
       procedure create_o_Observer -> o_Observer;
```

```
        dcl key ComponentKey;
        start;
       create co_instance;
        task key := call get_key to offspring;
        task keys := keys // key;
       return offspring;
     endprocedure;

     exported as <<package phil_interface/package DiningPhilosophers/
             package o_Observer/package factory>>get_co_type
     procedure get_co_type -> string;
       start;
       return 'o_Observer';
     endprocedure;

      exported as <<package phil_interface/package DiningPhilosophers/
             package o_Observer/package factory>>resolve_co
      procedure resolve_co(in key ComponentKey) raise InvalidIndex;
         dcl returnValue Pid;
         dcl i Integer := 1;
         start;
         task returnValue := Extract(keys,key);
         return returnValue;
      endprocedure;

      exported as <<package phil_interface/package DiningPhilosophers/
             package o_Observer/package factory>>list_co
      procedure list_co -> ComponentKeySeq;
         start;
         return keys;
      endprocedure;

  endprocess type;

  process type o_Observer;
    gate initial
      in with  <<package phil_interface/
          package DiningPhilosopher/package o_Observer>>o_Observer;
    gate provides
      in with <<package phil_interface/
         package DiningPhilosopher/package o_Observer>>exported_i_Observer;
      out with <<package phil_interface/package DiningPhilosopher/
          package o_Observer>>imported_i_Observer;


  dcl reference_interaction_i_Observer exported_i_Fork;
  dcl reference_port_observer exported_i_Fork;
  dcl the_key string;

  process data_access(1,1);
    exported as <<package philo_definition/
       package DiningPhilosopher/package o_Fork_data>>get_port_observer
    procedure get_port_fork -> exported_i_Observer;
      start;
      return reference_port_observer;
    endprocedure;
    exported as <<package philo_definition/
       package DiningPhilosopher/package
o_Fork_state>>get_interaction_i_Observer
    procedure get_interaction_i_Observer -> exported_i_Observer;
      start;
      return reference_interaction_i_Observer;
    endprocedure;
  endprocess;
```

```
      exported as <<package philo_definition/
         package DiningPhilosopher/package o_Fork_data>>set_port_observer
      procedure set_port_fork(in ref exported_i_Observer);
        start;
        reference_port_observer := ref;
      endprocedure;
      exported as <<package philo_definition/
         package DiningPhilosopher/package
o_Fork_state>>set_interaction_i_Observer
      procedure set_interaction_i_Observer(exported_i_Observer);
        start;
        reference_interaction_i_Observer := ref;
      endprocedure;
    endprocess;

    gate state_accessor
      in with internal_state;
    channel route_state_access nodelay
      from
        state_accessor via state_accessor to env;
    endchannel;
    channel nodelay
      from artefact_a_Observer to state_accessor
        with internal_state;
    endchannel;


    process artefactmanagement(1,1);
      signal create_a_Observer_req;
      signal create_a_Observer_res(Pid);
      dcl a_Observer Pid := Null;
      exported procedure get_artefact_a_Observer;
        dcl new_pid PId;
        start;
         decision a_ForkImpl;
          (Null):
             output create_a_ForkImpl;
              nextstate wait4response;
          else:
            return a_Observer;
        enddecision;

        state wait4response;
        input create_a_ForkImpl_res(new_pid);
        task a_ForkImpl_seq := a_ForkImpl_seq // new_pid;
        return offspring;
      endprocedure;
      start;
      nextstate wait_for_signal;
      state wait_for_signal;
      input create_a_Observerl;
      create artefact_a_Observer;
      nextstate -;
    endprocess;

    process interaction_i_Observer(0,1);
      dcl p PId;
       dcl e_PState pstate;
      start;
      nextstate wait4signal;
       state wait4signal;
      input PhilosopherState(pstate);
      task p := get_artefact_a_Observer;
      output PhilosopherState(pstate, sender) to p;
```

```
    nextstate -;
  endprocess;

process  artefact_a_Observer(0,1): a_Observer;

    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>provide_observer
    procedure provide_observer -> exported_i_Observer;
      start;
      return call get_provide_observer to data_access;
    endprocedure;
    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>provide
    procedure provide(s string) -> PId
         raise NoSuchPort;
      start;
      decision s;
       (='observer'): return reference_port_observer;
       else: raise NoSuchPort;
       enddecision;
    endprocedure;
    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>port_connect
    procedure port_connect(s string) -> PId
        raise NoSuchPort,AlreadyConnected;
      start;
      raise NoSuchPort;
    endprocedure;
    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>port_disconnect
    procedure port_disconnect(s string) -> PId
        raise NoSuchPort,NotConnected;
      start;
      raise NoSuchPort;
    endprocedure;


  start;
  task the_key := <<package eODL>>generate_key;
  create interaction_i_Observer;
  call set_interaction_i_Observer(offspring) to data_access;
  call set_port_observer(offspring) to data_access;
  nextstate initial_state;

  channel ch_provides nodelay
     from env via provides to interaction_i_Observer
        with exported_i_Observer;
     from interaction_i_Fork to env via provides
        with imported_i_Observer;
  endchannel;

   channel ch_initial_port nodelay
    from env via initial to portmanagement
       with config_o_Observer;
  endchannel;

   channel ch_initial nodelay
     from env via initial to this with imported_o_Observer;
     from this via initial to env with exported_o_Observer;
   endchannel;

  exported as <<package phil_interface/package DiningPhilosophers/
       package o_Observer>>get_key
  procedure get_key -> ComponentKey;
```

```
      start;
      return the_key;
    endprocedure;
  endprocess type;

  process factory(1,1): o_Observer_factory;
  process co(0,): o_Observer;

  channel factory_to_co nodelay
    from factory to co via initial;
  endchannel;

  channel initial_to_env nodelay
    from co via initial to env via initial;
  endchannel;

  channel provides_to_env nodelay
    from co via provides to env via provides;
  endchannel;

  channel uses_to_env nodelay
    from co via uses to env via uses;
  endchannel;

  endblock type;

endpackage;

endpackage DiningPhilosophers;

endpackage phil_definition;
```

Дополнение II

Обработка информации и инструментальная поддержка

II.1 Введение

Наличие простой и законченной **метамодели** обеспечивает прочную основу для разработки программного обеспечения даже в сложных прикладных областях. Для того чтобы сделать методику удобной и, в частности, обеспечить простоту использования проектировщиками, необходима соответствующая инструментальная поддержка. В общем, такие инструментальные средства могут обеспечивать поддержку самого процесса моделирования, подобно редактору или/и имитатору, или они могут охватывать больше фаз, подобно реализации и **внедрению** на **целевых платформах**.

Вследствие широко распространенного многообразия задач, которые должны поддерживаться инструментальными средствами обработки простой модели, начиная от спецификации до **внедрения** и конкретизации, ожидается, что различные инструментальные средства будут использоваться в последовательности инструментальных средств. Таким образом, возникает вопрос наличия формата обмена при переходе от одного инструментального средства к другому. Для этого, по умолчанию, существует, по крайней мере, одна стандартизованная нотация. В связи с использованием OMG MOF, для **метамодели** предполагается представление, основанное на XML, которое может использоваться в качестве формата обмена между различными инструментальными средствами.

Конкретное определение инструментальных средств и выполняемые ими функции не могут являться предметом стандартизации. Хотя на практике простые инструментальные средства могут разрабатываться произвольно или фактически состоять из последовательности инструментальных средств, определенные аспекты инструментальных средств рассматриваются в последующих параграфах. Фактические инструментальные средства могут охватывать несколько таких аспектов.

II.2 Инструментальные средства моделирования

Инструментальные средства, имеющие дело с работой информационной модели, могут использовать произвольное представление модели с единственным ограничением, состоящим в том, что каждое представление должно иметь соответствующее отображение **метамодели**. Такие представления могут простираться от языков программирования до графических нотаций, как UML. Могут использоваться любые инструментальные средства, поддерживающие обработку такого представления, и никаких ограничений не делается.

Так как **метамодель** охватывает почти полный жизненный цикл программного обеспечения, имеется множество возможных проблем моделирования, каждая из которых, безусловно, решается инструментальной поддержкой. Модель может постепенно расширяться путем добавления дополнительной информации в нескольких итерациях моделирования в разные моменты времени. Таким образом, совокупность уже существующих **типов СО** может позже использоваться для указания **сборки**, а после этого может даваться конкретное присвоение этой **сборки целевой платформе**. Ниже приведены вопросы моделирования в порядке их применения:

– спецификация типа **СО**;

– спецификация **сборки**;

– пакетирование реализации (**программных компонентов**);

– моделирование окружающей среды;

– присвоение **программных компонентов** на **целевой платформе**;

– конкретизация соответствующих **сборок типов СО**.

Первым шагом, который надо сделать, является спецификация **типов СО** в соответствии с **метамоделью**. Каждый тип может использоваться в произвольном числе сборок. Следующим шагом является обеспечение реализации для всей **сборки**, содержащей код реализации для используемых **типов СО**, сгруппированных в **программные компоненты**. Пакетирование реализаций может быть выполнено инструментальными средствами архивирования, аналогичными zip. После того, как все это обеспечено, модель в сочетании с пакетом реализации **сборки** может отправляться для внедрения на клиентских платформах. Для того чтобы внедрить **сборку**, прежде всего, необходимо определить распределение **СОs**. В течение этого процесса модель обогащается дополнительной информацией, которая относится главным образом к конкретной окружающей среде и специальной бизнес-ситуации клиента. Модель целевой окружающей среды и модель **сборки** сравниваются для нахождения правильного присвоения каждого **СО** узлу платформы. Это может быть сделано вручную, предпочтительно системным администратором, или полуавтоматически посредством автоматической функции, обеспечивающей решение для **исходной конфигурации сборки**. В любом случае, должны выполняться требования каждого **СО сборки** в целевой окружающей среде. При этом не указывается, где получить модель окружающей среды. Она может быть получена непосредственно из целевой окружающей среды, причем в этом случае потребуется специальная архитектура. В результате шага присвоения модель содержит информацию о том, какой **программный компонент** и где инсталлировать. Фактическая конкретизация **типов СО** или сборок может являться частью модели. Подходящее инструментальное средство будет поддерживать модель в течение **рабочего цикла** на уровне современных требований.

II.3 Инструментальные средства генерации

Инструментальные средства, обеспечивающие поддержку реализации сущностей модели, могут сэкономить много усилий по сравнению с созданием реализации вручную. Генерация кода может выполняться для всех языков реализации, имеющих отображение от **метамодели**. Ниже приведена информация, которая может генерироваться из модели:

– скелеты **типов СО**;

– код для качества спецификаций обслуживания.

Генерированный код может предложить проектировщику структуру, служащую основой для реализаций **типа СО**. Пока в **метамодели** отсутствуют поведенческие аспекты, реализация бизнес-логики автоматически генерироваться не может. Вместо этого, она должна вставляться ручным способом.

II.4 Инструментальные средства внедрения

Внедрение, включающее конкретизацию сборок на клиентской **целевой платформе**, требует поддержки подходящими инструментальными средствами, а также зависит от соответствующей поддержки со стороны самой платформы. Поэтому фактические инструментальные средства для **внедрения** тесно связаны с конкретными архитектурами платформ, с которыми они должны взаимодействовать. Инструментальные средства не могут быть независимыми, пока не существует стандартизованной архитектуры платформы, с которой они могут работать совместно.

В общих чертах, процесс **внедрения** включает несколько задач, начиная от определения правильного распределения до инсталляции и конкретизации программного обеспечения. Следующие инструментальные средства могут иметь дело с общими задачами **внедрения**:

– моделирование окружающей среды;

– присвоение **программных компонентов** на **целевой платформе**;

– инсталляция **программных компонентов** на **целевой платформе**;

– **сборка** соответствующей конкретизации **типа СО**;

– обработка ограничений и операций.

Задачи моделирования окружающей среды и присваивания **программных компонентов** на **целевой платформе** уже упоминались в контексте вопросов, относящихся к инструментальным средствам моделирования. В действительности, во время этих задач модели расширяются, что является причиной, почему с ними там имеют дело. Фактически, в большинстве случаев ожидается, что эти задачи будут исполняться как часть **внедрения сборки**. Как уже указывалось, платформа может поддерживать инструментальные средства для получения информации моделирования окружающей среды. Выполнив эти задачи и определив правильное присвоение **программных компонентов** на **целевой платформе**, следующим шагом является загрузка и инсталляция программного обеспечения на указанном узле. После этого, с помощью другого инструментального средства или возможностей платформы **сборка** или **типы СО** могут быть конкретизированы. И, наконец, ограничения и операции, содержащиеся в модели, каким-то образом должны обрабатываться в **рабочем цикле**.

На рис. II.1 приведена последовательность действий от моделирования до **внедрения**, относящихся к инструментальным средствам.

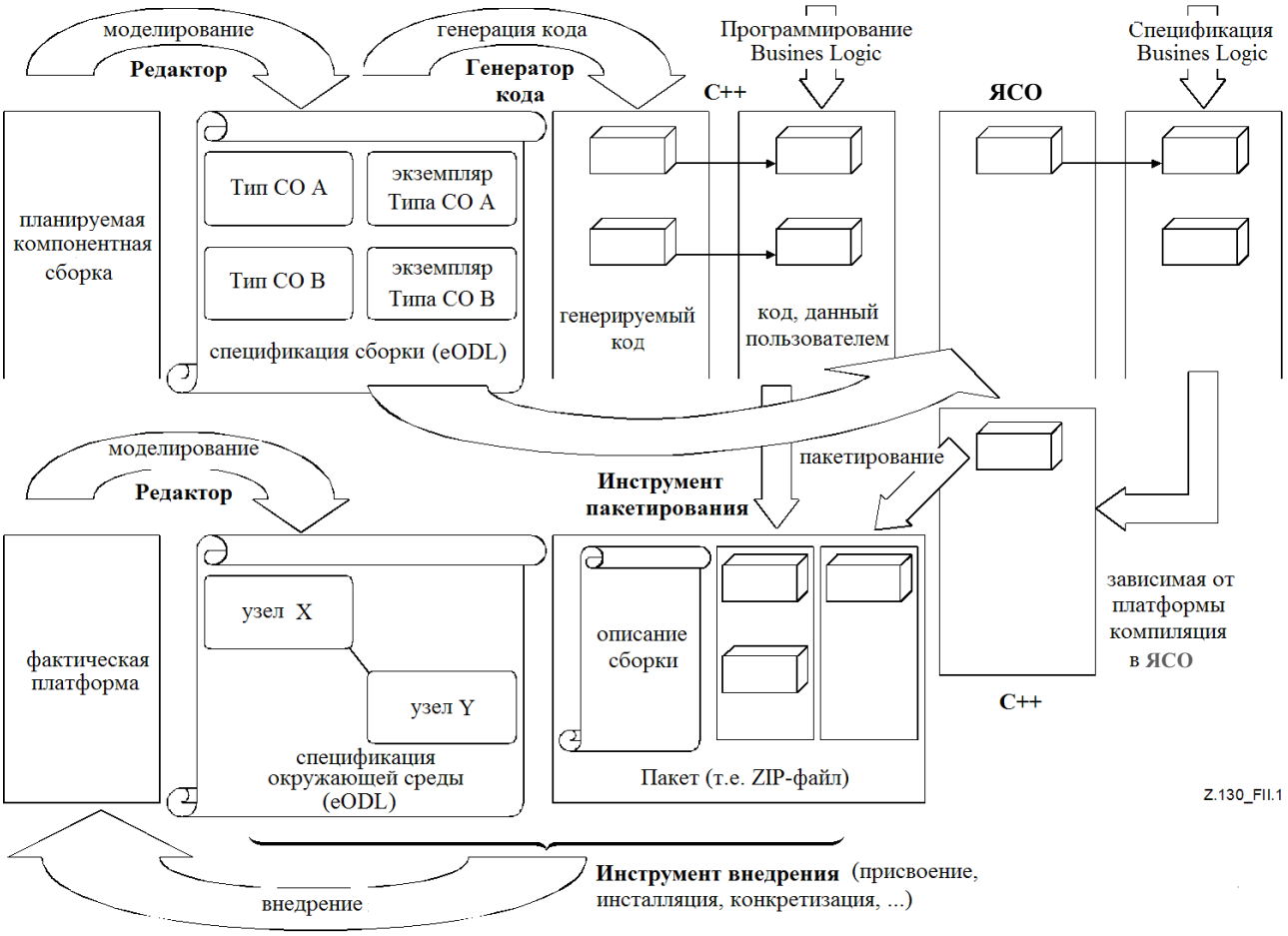


Рисунок II.1/Z.130 – Обработка информации

СЕРИИ РЕКОМЕНДАЦИЙ МСЭ-Т

Серия A	Организация работы МСЭ-Т
Серия B	Средства выражения: определения, символы, классификация
Серия C	Общая статистика электросвязи
Серия D	Общие принципы тарификации
Серия E	Общая эксплуатация сети, телефонная служба, служба эксплуатации и человеческие факторы
Серия F	Нетелефонные службы электросвязи
Серия G	Система 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 телевизионных и звуковых программ и других мультимедийных сигналов
Серия K	Защита от помех
Серия L	Конструкция, прокладка, и защита кабелей и других элементов линейно-кабельных сооружений
Серия M	TMN и техническое обслуживание сетей: международные системы передачи, телефонные, телеграфные, факсимильные и арендованные каналы
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 и взаимосвязь открытых систем
Серия Y	Глобальная информационная инфраструктура и аспекты межсетевого протокола (IP)
Серия Z	**Языки и общие аспекты программного обеспечения для систем электросвязи**

Отпечатано в Швейцарии
Женева, 2004 г.