UNIÓN INTERNACIONAL DE TELECOMUNICACIONES

UIT-T Z.130

SECTOR DE NORMALIZACIÓN
DE LAS TELECOMUNICACIONES
DE LA UIT

(07/2003)

SERIE Z: LENGUAJES Y ASPECTOS GENERALES DE SOPORTE LÓGICO PARA SISTEMAS DE TELECOMUNICACIÓN

Técnicas de descripción formal – Lenguaje ampliado de definición de objetos

Lenguaje ampliado de definición de objetos: Técnicas de desarrollo de componentes de soporte lógico distribuido – Bases conceptuales, notaciones y correspondencias tecnológicas

Recomendación UIT-T Z.130

RECOMENDACIONES UIT-T DE LA SERIE Z

LENGUAJES Y ASPECTOS GENERALES DE SOPORTE LÓGICO PARA SISTEMAS DE TELECOMUNICACIÓN

TÉCNICAS DE DESCRIPCIÓN FORMAL	
Lenguaje de especificación y descripción	Z.100–Z.109
Aplicación de técnicas de descripción formal	Z.110–Z.119
Gráficos de secuencias de mensajes	Z.120–Z.129
Lenguaje ampliado de definición de objetos	**Z.130–Z.139**
Notación de prueba y de control de prueba	Z.140–Z.149
Notación de requisitos de usuarios	Z.150–Z.159
LENGUAJES DE PROGRAMACIÓN	
CHILL: el lenguaje de alto nivel del UIT-T	Z.200–Z.209
LENGUAJE HOMBRE-MÁQUINA	
Principios generales	Z.300–Z.309
Sintaxis básica y procedimientos de diálogo	Z.310–Z.319
LHM ampliado para terminales con pantalla de visualización	Z.320–Z.329
Especificación de la interfaz hombre-máquina	Z.330–Z.349
Interfaces hombre-máquina orientadas a datos	Z.350–Z.359
Interfaces hombre-máquina para la gestión de las redes de telecomunicaciones	Z.360–Z.369
CALIDAD	
Calidad de soportes lógicos de telecomunicaciones	Z.400–Z.409
Aspectos de la calidad de las Recomendaciones relativas a los protocolos	Z.450–Z.459
MÉTODOS	
Métodos para validación y pruebas	Z.500–Z.519
SOPORTE INTERMEDIO	
Entorno del procesamiento distribuido	Z.600–Z.609

Para más información, véase la Lista de Recomendaciones del UIT-T.

Recomendación UIT-T Z.130

Lenguaje ampliado de definición de objetos: Técnicas de desarrollo de componentes de soporte lógico distribuido – Bases conceptuales, notaciones y correspondencias tecnológicas

Resumen

Esta Recomendación se dirige a los diseñadores, implementadores y gestores de sistemas distribuidos y a los desarrolladores de herramientas de soporte de los sistemas distribuidos.

Esta Recomendación especifica el lenguaje ampliado de definición de objetos de la UIT (UIT-eODL). El UIT-eODL se utiliza para el desarrollo por componentes de sistemas distribuidos desde cuatro perspectivas distintas pero relacionadas entre sí: la computacional, la de la implementación, la del despliegue y la del entorno objetivo. Cada perspectiva está vinculada a un objetivo específico de modelado expresado mediante conceptos de abstracción dedicados. Los tipos de objetos computacionales con interfaces (operacionales, de tren, de señal) y puertos son los principales conceptos de la perspectiva computacional que describen abstractamente los componentes distribuidos de soporte lógico en términos de sus interfaces potenciales. La perspectiva de la implementación está integrada por artefactos tales como las abstracciones de contextos de lenguajes de programación concretos y su relación con las interfaces. La perspectiva del despliegue describe las entidades de soporte lógico (componentes de soporte lógico) en representación binaria y las entidades computacionales que aquéllas realizan. La perspectiva del entorno objetivo proporciona los conceptos de modelado de una red física sobre la que de deberá realizarse el despliegue de los componentes de soporte lógico. Todos los conceptos de las perspectivas están relacionados entre sí. Estas relaciones constituyen la base esencial de las técnicas y herramientas que soportan el proceso de desarrollo de soporte lógico desde el diseño hasta el despliegue, pasando por la implementación y la integración. La fase de pruebas aún no forma parte de la presente Recomendación.

El UIT-eODL es una ampliación del lenguaje de definición de objetos de la UIT UIT-ODL [1] al que sustituye. Originalmente el UIT-ODL se diseñó como ampliación del ODP-IDL [9] y definía conceptos computacionales basados en la terminología ODP [2], [3]. El eODL respeta este principio. No obstante, las definiciones se basan en un metamodelo en vez de en la solución tradicional de sintaxis abstracta. Una ventaja de la solución del metamodelo es que permite la utilización de herramientas relacionadas con MOF [4] para soportar la automatización de las transiciones de los modelos entre las distintas fases de desarrollo del soporte lógico. Otra ventaja es sin duda la capacidad de ejemplificar modelos concretos a partir del metamodelo, que pueden representarse por lenguajes existentes, de modo que se puede llegar a la integración de diferentes soluciones de diseño.

Se supone que el lector de esta Recomendación esta familiarizado con IDL [5], UML [11] y MOF.

La definición del eODL se basa en los siguientes anexos y apéndices:

- El anexo A introduce la sintaxis textual del eODL para la representación de las especificaciones eODL. Esta sintaxis se define con el estilo EBNF.
- El anexo B define la correspondencia entre el metamodelo eODL y la sintaxis textual definida en el anexo A.
- El anexo C presenta una correspondencia entre el eODL y el UIT SDL-2000 que permite la transformación automática de un modelo eODL en otro SDL-2000.

- El anexo D contiene una referencia de soporte lógico para la representación en XML [12] del metamodelo eODL con arreglo al formato de intercambio meta de XML (XMI) [6]. Se facilita en un fichero independiente para poder importar y procesar el metamodelo eODL con las herramientas UML.
- La cláusula 1 presenta una sinopsis de cómo deben utilizar el eODL los diseñadores, implementadores y gestores de sistemas distribuidos. El apéndice I proporciona un ejemplo concreto de su uso.
- El apéndice II describe el proceso global de desarrollo cuando se utiliza eODL y el soporte de herramientas posible.

Orígenes

La Recomendación UIT-T Z.130 fue aprobada el 22 de julio de 2003 por la Comisión de Estudio 17 (2001-2004) del UIT-T por el procedimiento de la Recomendación UIT-T A.8.

PREFACIO

La UIT (Unión Internacional de Telecomunicaciones) es el organismo especializado de las Naciones Unidas en el campo de las telecomunicaciones. El UIT-T (Sector de Normalización de las Telecomunicaciones de la UIT) es un órgano permanente de la UIT. Este órgano estudia los aspectos técnicos, de explotación y tarifarios y publica Recomendaciones sobre los mismos, con miras a la normalización de las telecomunicaciones en el plano mundial.

La Asamblea Mundial de Normalización de las Telecomunicaciones (AMNT), que se celebra cada cuatro años, establece los temas que han de estudiar las Comisiones de Estudio del UIT-T, que a su vez producen Recomendaciones sobre dichos temas.

La aprobación de Recomendaciones por los Miembros del UIT-T es el objeto del procedimiento establecido en la Resolución 1 de la AMNT.

En ciertos sectores de la tecnología de la información que corresponden a la esfera de competencia del UIT-T, se preparan las normas necesarias en colaboración con la ISO y la CEI.

NOTA

En esta Recomendación, la expresión "Administración" se utiliza para designar, en forma abreviada, tanto una administración de telecomunicaciones como una empresa de explotación reconocida de telecomunicaciones.

La observancia de esta Recomendación es voluntaria. Ahora bien, la Recomendación puede contener ciertas disposiciones obligatorias (para asegurar, por ejemplo, la aplicabilidad o la interoperabilidad), por lo que la observancia se consigue con el cumplimiento exacto y puntual de todas las disposiciones obligatorias. La obligatoriedad de un elemento preceptivo o requisito se expresa mediante las frases "tener que, haber de, hay que + infinitivo" o el verbo principal en tiempo futuro simple de mandato, en modo afirmativo o negativo. El hecho de que se utilice esta formulación no entraña que la observancia se imponga a ninguna de las partes.

PROPIEDAD INTELECTUAL

La UIT señala a la atención la posibilidad de que la utilización o aplicación de la presente Recomendación suponga el empleo de un derecho de propiedad intelectual reivindicado. La UIT no adopta ninguna posición en cuanto a la demostración, validez o aplicabilidad de los derechos de propiedad intelectual reivindicados, ya sea por los miembros de la UIT o por terceros ajenos al proceso de elaboración de Recomendaciones.

En la fecha de aprobación de la presente Recomendación, la UIT no ha recibido notificación de propiedad intelectual, protegida por patente, que puede ser necesaria para aplicar esta Recomendación. Sin embargo, debe señalarse a los usuarios que puede que esta información no se encuentre totalmente actualizada al respecto, por lo que se les insta encarecidamente a consultar la base de datos sobre patentes de la TSB.

ÍNDICE

Recomendación UIT-T Z.130

Lenguaje ampliado de definición de objetos: Técnicas de desarrollo de componentes de soporte lógico distribuido – Bases conceptuales, notaciones y correspondencias tecnológicas

1 Alcance

La disponibilidad de técnicas eficientes y de un soporte instrumental para el desarrollo e ingeniería de los sistemas distribuidos es un factor clave para la evolución de las tecnologías de la información. Los sistemas de telecomunicación son sistemas distribuidos especiales integrados por componentes distribuidos a lo largo de redes, que tienen que afrontar problemas de concurrencia, autonomía, sincronización y comunicación. El desarrollo de sistemas escalables de alta eficiencia es una tarea ardua y compleja, en la que deben utilizarse instrumentos de ayuda en todas las fases del proceso de desarrollo: desde la definición de requisitos hasta la integración, prueba y despliegue, pasando por el diseño y la implementación.

La generación de código a partir de modelos de diseño orientados a objetos produce componentes ejecutables reutilizables. Estos componentes integran aspectos dependientes del entorno de ejecución y de la tecnología de la plataforma de soporte intermedio con el modelo de diseño orientado a objetos específico de la empresa. Cada componente de soporte lógico tiene una representación física (por ejemplo un fichero binario), que ha de estar disponible para ser ejecutado en un nodo especial del sistema distribuido. El principal objetivo de esta Recomendación es el diseño de dichos componentes.

Las técnicas para el desarrollo de sistemas distribuidos contribuyen significativamente a reducir el tiempo de comercialización de las aplicaciones distribuidas y de los servicios de telecomunicación. El adecuado tratamiento de todos los aspectos de la comunicación reside en la propia naturaleza del dominio de aplicación que se contempla. Estos aspectos van desde los requisitos transaccionales para las interacciones entre objetos hasta las políticas de seguridad, pasando por las cuestiones de calidad de servicio. Teniendo en cuenta la amplia aceptación de la tecnología del soporte intermedio de objetos, las plataformas de soporte intermedio constituyen un entorno de implementación ideal para estos diseños. Entre estas tecnologías se pueden citar CORBA [5] básica, los componentes CORBA [7] y otras plataformas distribuidas de procesamiento.

Esta Recomendación se centra en todos los procesos de desarrollo del soporte lógico, contemplando especialmente las siguientes fases del ciclo vital del soporte lógico:

- diseño,
- implementación,
- integración,
- explotación.

Esta Recomendación no contempla la fase de definición de requisitos.

Esta Recomendación subraya muy especialmente el soporte tecnológico de las transiciones entre fases para conseguir su automatización. La tecnología clave es una solución basada en un modelo que se basa en un metamodelo definido (véase [11]). Este metamodelo permite la integración de varios lenguajes de diseño existentes, tales como SD [8], UML y CORBA-IDL. El metamodelo es la definición de los conceptos de las fases contempladas en el ciclo vital del soporte lógico. Los modelos que se ejemplifican en base al metamodelo pueden representarse por medio de lenguajes existentes. Dado que algunos conceptos no están contemplados en los lenguajes existentes, esta Recomendación define además una sintaxis concreta: el eODL (ODL ampliado). La solución basada en metamodelos sustituye a la conocida solución de sintaxis abstracta para la definición de

lenguajes. El UIT-eODL es una versión del UIT-ODL. La definición de su sintaxis se presenta en el anexo A.

Así pues, el metamodelo es independiente de cualquier notación de diseño específica. Los modelos de diseño pueden desarrollarse utilizando distintas notaciones, aunque basándose siempre en los mismos conceptos. La información del diseño puede intercambiarse con arreglo al metamodelo. Tanto la notación como el propio metamodelo son independientes del entorno de ejecución específico. El mismo diseño puede trasladarse a distintos entornos objetivo. Esto proporciona una gran flexibilidad y es además importante en lo que se refiere al aspecto de reutilización de los modelos de diseño de componentes.

La integración de esta Recomendación en los procesos de desarrollo de soporte lógico se representa en la figura 1. La especificación de un modelo de diseño comienza con una definición exacta de sus requisitos. Esta Recomendación define conceptos que se corresponden con las distintas perspectivas del modelo de diseño. Cada perspectiva contempla aspectos diferentes del sistema a desarrollar. Los conceptos del metamodelo permiten desarrollar modelos suficientemente potentes para obtener automáticamente el esqueleto de los componentes de soporte lógico. Los esqueletos forman el punto de arranque de la fase de implementación, en la que han de completarse con la lógica de la empresa. En la fase de integración los componentes de soporte lógico completos se integrarán en el entorno objetivo.

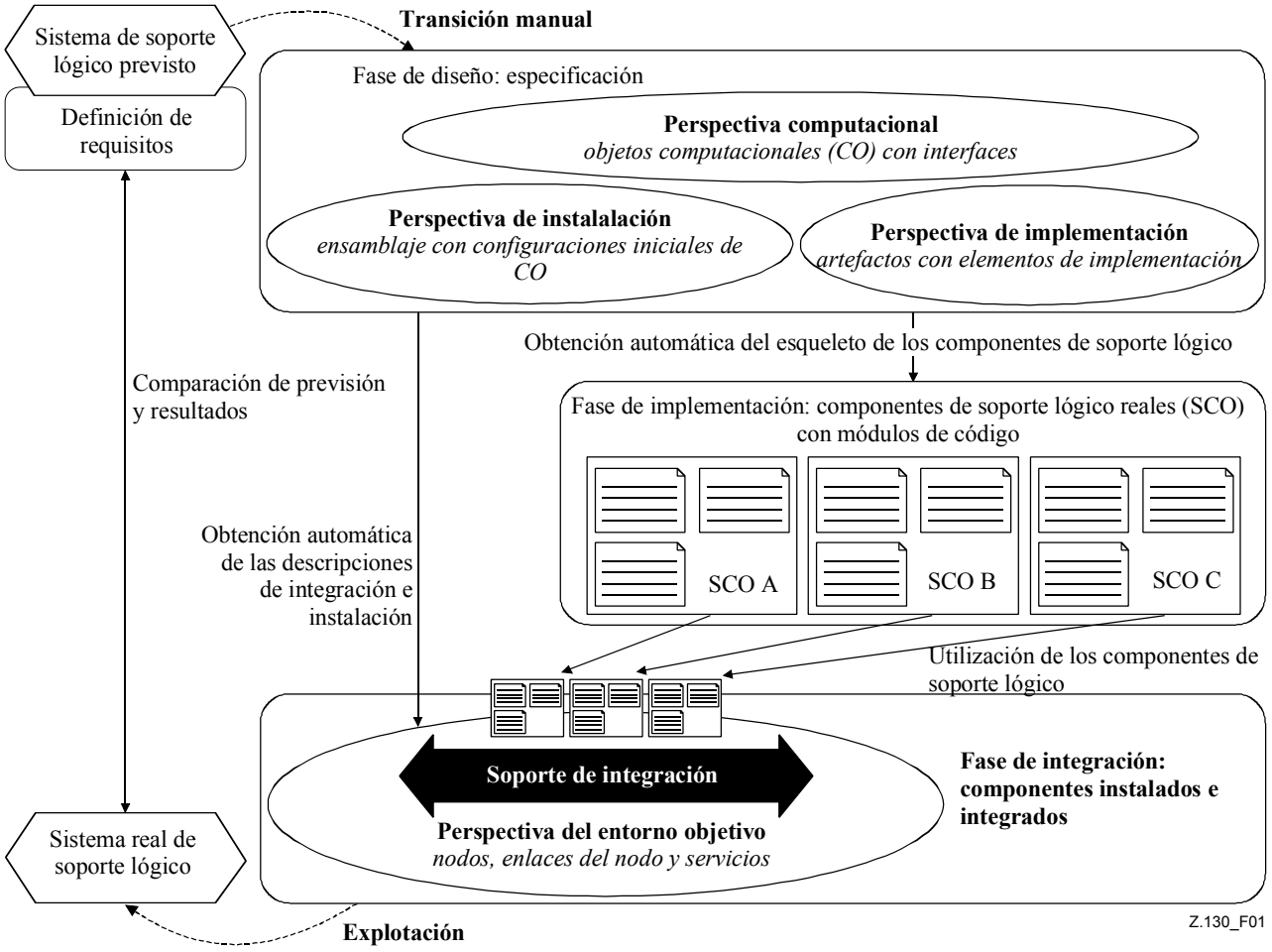


Figura 1/Z.130 – Desarrollo del soporte lógico

Esta Recomendación contiene asimismo conceptos que permiten describir la topología y propiedades del entorno objetivo. Junto con las descripciones del despliegue y de la integración, generadas automáticamente en la fase de diseño, la descripción del entorno objetivo permite la

automatización del despliegue. Tras la fase de integración, el sistema de soporte lógico desarrollado puede ponerse en explotación.

2 Referencias

Las siguientes Recomendaciones del UIT-T y otras referencias contienen disposiciones que, mediante su referencia en este texto, constituyen disposiciones de la presente Recomendación. Al efectuar esta publicación, estaban en vigor las ediciones indicadas. Todas las Recomendaciones y otras referencias son objeto de revisiones por lo que se preconiza que los usuarios de esta Recomendación investiguen la posibilidad de aplicar las ediciones más recientes de las Recomendaciones y otras referencias citadas a continuación. Se publica periódicamente una lista de las Recomendaciones UIT-T actualmente vigentes. En esta Recomendación, la referencia a un documento, en tanto que autónomo, no le otorga el rango de una Recomendación.

[1] Recomendación UIT-T Z.130 (1999), *Lenguaje de definición de objetos de la UIT*.

[2] Recomendación UIT-T X.902 (1995) | ISO/CEI 10746-2:1996, *Tecnología de la información – Procesamiento distribuido abierto – Modelo de referencia: Fundamentos*.

[3] Recomendación UIT-T X.903 (1995) | ISO/CEI 10746-3:1996, *Tecnología de la información – Procesamiento distribuido abierto – Modelo de referencia: Arquitectura*.

[4] OMG Document formal/00-04-03, *Meta Object Facility (MOF) Specification*, Version 1.3.

[5] OMG Document formal/01-02-33, *The Common Object Request Broker Architecture and Specification*, Revision 2.4.2.

[6] OMG Document formal/00-11-02, *XML Metadata Interchange (XMI) Specification*, Version 1.1.

[7] OMG Document formal/02-06-65, *CORBA Component*, Version 3.0.

[8] Recomendación UIT-T Z.100 (2002), *Lenguaje de especificación y descripción*.

[9] Recomendación UIT-T X.920 (1997) | ISO/CEI 14750:1999, *Tecnología de la información – Procesamiento distribuido abierto – Lenguaje de definición de interfaz*.

[10] Recomendación UIT-T Z.600 (2000), *Arquitectura de entorno de procesamiento distribuido*.

3 Abreviaturas

En esta Recomendación se utilizan las siguientes siglas.

API Interfaz de programación de aplicaciones (*application programming interface*)

ASN.1 Notación de sintaxis abstracta uno (*abstract syntax notation one*)

CCM Modelo de componentes CORBA (*CORBA component model*)

CO Objeto computacional (*computational object*)

COM Modelo de objeto de componente (*component object model*)

CORBA Arquitectura de intermediario de petición de objeto común (*common object request broker architecture*)

CWM Modelo de almacenamiento común (*common warehouse metamodel*)

DTD Definición de tipo de documento para XML (*document type definition for XML*)

EBNF Forma Backus-Naur ampliada (*extended Backus-Naur form*)

EJB Enterprise Javabeans TM

IDL Lenguaje de definición de interfaz (*interface definition language*)

MDA Arquitectura modelizada (*model driven architecture*)

MOF Facilidad metaobjeto (*meta object facility*)

OCL Lenguaje de constricción de objetos (*object constraint language*)

ODL Lenguaje de definición de objetos (*object definition language*)

OMG Grupo de gestión de objetos (*object management group*)

OSD Descripción de soporte lógico abierto (*open software description*)

RM-ODP Modelo de referencia de procesamiento distribuido abierto (*reference model for open distributed processing*)

SDL Lenguaje de especificación y descripción (*specification and description language*)

TINA Arquitectura de funcionamiento en red para información de telecomunicación (*telecommunication information networking architecture*)

UML Lenguaje de modelado unificado (*unified modelling language*)

XMI Intercambio de metadatos XML (*XML metadata interchange*)

XML Lenguaje de marcaje extensible (*extensible markup language*)

4 Definiciones

En esta Recomendación se definen los términos siguientes.

4.1 artefacto: Abstracción de construcciones en un lenguaje de programación concreto, por ejemplo, la clase en los lenguajes orientados a objetos (realizada en forma de módulos de código), encerrada por un **componente de soporte lógico**. Un ejemplar de **artefacto realiza** (en términos del modelo) el estado, comportamiento e identidad de un **CO**.

4.2 ensamblaje: Descripción de un sistema de soporte lógico distribuido por el conjunto de todos los **tipos de CO** participantes y de su **configuración inicial**. (Se utiliza en CCM.)

4.3 excepción: Tipo especial de **terminación** de la operación en caso de error. (Definido por RM-ODP.)

4.4 clase: Concepto de clasificación de objetos. Los objetos pueden ejemplificarse con arreglo a la descripción de su clase. (Definido por MOF.)

4.5 conexión: Concepto para el intercambio de **referencias de interfaz**, que corresponden a definiciones de puertos conforme a su tipo de definición especial (*utilizar* relación o *proporcionarla*). (Definido como vinculación computacional por RM-ODP utilizado por CCM.)

4.6 arquitectura de componentes: **Entorno de procesamiento distribuido** que soporta la interacción de **componentes de soporte lógico distribuido**.

4.7 plataforma objetivo: **Arquitectura de componentes** con soporte de **despliegue** y ejecución distribuida, en la que se pretende instalar los componentes.

4.8 objeto computacional (CO, *computational objetc*): Unidad funcional, resultante de la descomposición funcional del sistema de soporte lógico modelado. (Definido por RM-ODP.)

4.9 tipo de objeto computacional (tipo de CO): Plantilla para la ejemplificación de **objetos computacionales**. (Definido por RM-ODP identificado por CCM como componente CORBA.)

4.10 consumir: Concepto para el modelado de la posible señal recibida. (Utilizado por CCM.)

4.11 interacción de medio continuo, de señal y **operacional**: **Interacción** entre **CO** mediante **elementos de interacción** de *medio continuo*, de señal u operacionales: **operación**, **atributo**, **consumir**, **producir**, **sumidero**, **fuente**. [Definido por RM-ODP y TINA (interacción operacional y de *medios continuos*).]

4.12 tipo de datos: Prescripción de la estructura, contenido y comportamiento permisibles de los datos; se trata de un elemento del modelo de tipo de datos (es decir CORBA-IDL) que es la base de los modelos de información. (Definido por CORBA.)

4.13 entorno de procesamiento distribuido: Base tecnológica que soporta las interacciones entre los objetos de un sistema distribuido. (Definido por TINA.)

4.14 despliegue: Proceso de crear representaciones físicas de los **componentes de soporte lógico** existentes en los **nodos**, instalándolos de modo que queden listos para su ejecución y estableciendo la **configuración inicial**.

4.15 elemento de implementación: Relación entre un **elemento de interacción** y un **artefacto**, en la que un ejemplar del **artefacto** es responsable del comportamiento del **elemento de interacción**.

4.16 implementa: Relación entre **artefactos** y **tipos de CO**, en la que los ejemplares del **artefacto realizan** el comportamiento del **tipo de CO**.

4.17 CO inicial: **CO** que se crea al comenzar la **ejecución** de un sistema de soporte lógico distribuido.

4.18 configuración inicial: Conjunto de **CO iniciales** y **conexiones iniciales**. (Utilizado por CCM.)

4.19 conexión inicial: Vinculación que se establece inicialmente al comienzo de la **ejecución** de un sistema de soporte lógico distribuido. (Utilizado por CCM.)

4.20 política de ejemplificación: Descripción política de la ejemplificación de diversos conceptos de la implementación modelados por **artefactos**.

4.21 interacción: Acción en la que se ve implicado el entorno de un objeto. Definido por RM-ODP. (Definido por RM-ODP.)

4.22 elemento de interacción: Generalización de los conceptos **operación**, **atributo**, **sumidero**, **fuente**, **consumir** y **producir**.

4.23 interfaz: Agregación referenciable de las posibles interacciones de un **CO**. (Definido por RM-ODP.)

4.24 atributo de interfaz: Tipo especial de operación como método abreviado de las operaciones *obtener (get)* y *poner (set)* para un determinado **tipo de datos**. (Utilizado por CORBA.)

4.25 referencia de interfaz: Referencia a una **interfaz**. (Corresponde a la referencia de objetos CORBA.)

4.26 tipo de interfaz: Descripción de un conjunto de **elementos de interacción** nombrados y de puntos extremos identificables de la posible comunicación. (Definido por RM-ODP corresponde a la interfaz OMG IDL.)

4.27 conjunto de medios: Agregación de los medios.

4.28 tipo de medio: Declaración a utilizar para la codificación, transmisión y decodificación de los datos de un **medio**.

4.29 medio: Tren de datos continuo, unidireccional y atómico. (Sustituye a la notación de tren de la Rec. UIT-T Z.600 [10].)

4.30 metamodelo: Definición de conceptos de modelado para la construcción de modelos de un dominio específico. (Definido por MOF.)

4.31 meta-metamodelo: Definición de conceptos de modelado para la construcción de **metamodelos**. (Definido por MOF.)

4.32 puerto múltiple: **Puerto** que soporta dinámicamente el registro y recuperación de varias **referencias de interfaz**. (Definido por CCM.)

4.33 espacio de nombres: Concepto destinado a estructurar los identificadores de los elementos del modelo. (Definido por RM-ODP.)

4.34 nodo: Dispositivo para la interpretación de los módulos de código de un **componente de soporte lógico**. (Definido por RM-ODP.)

4.35 objeto: Modelo de una entidad, siendo ésta cualquier fenómeno de interés del dominio examinado; los objetos tienen identidad, estado y comportamiento. (Definido por RM-ODP.)

4.36 operación: Elemento de una interacción operacional descrito por un conjunto de parámetros y terminaciones posibles. (Definido por RM-ODP y CORBA.)

4.37 parámetro: Elemento de la invocación de una operación descrito por el sentido de un intercambio de información y un **tipo de datos**. (Definido por CORBA.)

4.38 puerto: Entidad para el registro y recuperación de **referencias de interfaz** de un **CO**. (Definido por CCM.)

4.39 producir: **Elemento de interacción** para enviar **señales**. (Definido por CCM.)

4.40 puerto suministrado: **Puerto** en el que pueden recuperarse las **referencias de interfaz** del **CO** correspondiente. (Definido por CCM.)

4.41 Realizar: Correspondencia entre los **componentes de soporte lógico** y los **tipos CO**. (Definido por UML.)

4.42 requiere: Relación entre un **tipo de interfaz** y un **tipo de CO** en virtud de la cual los **CO** de este **tipo de CO** requieren **referencias de interfaz** del **tipo de interfaz** del entorno del **CO**. (Definido por TINA.)

4.43 ejecución: Tiempo de ejecución de un **componente de soporte lógico**.

4.44 señal: **Elemento de interacción** para el intercambio asíncrono de mensajes atómicos. (Definido por RM-ODP.)

4.45 puerto único: **Puerto** en el que sólo puede registrarse y recuperarse una única **referencia de interfaz**. (Definido por CCM.)

4.46 sumidero: **Elemento de interacción** para recibir un **conjunto de medios**. (Definido por TINA.)

4.47 componente de soporte lógico: Entidad integrada por secuencias de instrucciones (módulos de código), con representación física (que se plasma en formatos de datos especiales) y que puede ensamblarse en **componentes de soporte lógico** estructurados o en un sistema de soporte lógico; para poder crear **componentes de soporte lógico**, se suministra la funcionalidad a través de **tipos de interfaz** definidos.

Durante la ejecución de un **componente de soporte lógico**, los objetos se materializan como ejemplares de clases (realizadas como módulos de código). (Definido por [16].)

4.48 lotes de soporte lógico: Lote de **componentes de soporte lógico**. (Definido por CCM.)

4.49 fuente: **Elemento de interacción** para enviar un **conjunto de medios**. (Definido por TINA.)

4.50 soporta: Relación entre un **tipo de interfaz** y un **tipo de CO**, en virtud de la cual los **CO** de este **tipo de CO** soportan las **referencias de interfaz** del **tipo de interfaz** del entorno del **CO**. (Definido por TINA.)

4.51 terminación: Final de una **operación** invocada. (Definido por RM-ODP.)

4.52 parámetro de señal: Elemento de una **señal** para transportar información; se refiere a un **tipo de datos**. (Definido por SDL.)

4.53 puerto utilizado: **Puerto** en el que pueden registrarse **referencias de interfaz**. (Definido por CCM.)

5 El metamodelo

Un **metamodelo** define conceptos de modelado para la construcción de modelos de un dominio específico. En esta Recomendación el **metamodelo** es conforme con la facilidad metaobjeto (MOF, *meta-object facility*). El **metamodelo** se describe por medio de diagramas de clase UML. La semántica se presenta en lenguaje natural. Cuando es necesario, se añaden reglas de formación correcta. MOF es la norma adoptada para el metamodelado en OMG (grupo de gestión de objetos). MOF define un marco genérico para la descripción y representación de metainformación.

MOF define una arquitectura de cuatro niveles, que se representa en la figura 2:

- En el nivel M3 hay un **meta-metamodelo** único (el modelo MOF) que define los conceptos básicos necesarios para la descripción orientada a objetos de cualquier **metamodelo**. Los elementos constructivos básicos son: clase, asociación, tipo de datos, atributo de clase y herencia de clase.
- En el nivel M2 están los **metamodelos** (lenguajes). Un **metamodelo** proporciona las abstracciones necesarias para construir modelos. Se describe en términos de los elementos constructivos básicos de M3 (en la práctica la sintaxis abstracta de un **metamodelo** se suministra como colección de diagramas de clase). Ejemplos de **metamodelos** son UML, CWM (almacén de datos), y el **metamodelo** CCM.
- En el nivel M1 están los *modelos*. Éstos se describen en términos de uno de los **metamodelos** definidos en el nivel M2. Un ejemplo de éstos sería un modelo a nivel de red en UML que definiese lo que es un camino.
- En el nivel M0 hay datos que no son sino ejemplares de un modelo del nivel M1. Un ejemplo de datos sería una lista de registros representando caminos.

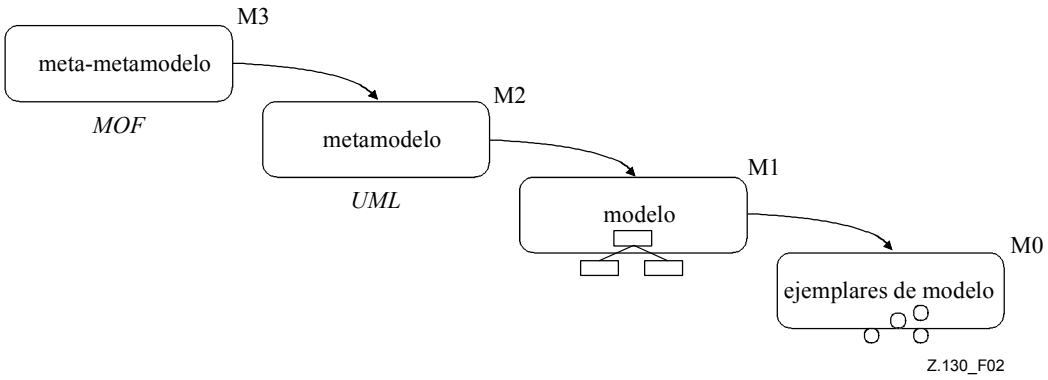


Figura 2/Z.130 – Niveles MOF

Además de proporcionar elementos constructivos básicos para la descripción orientados a objetos de **metamodelos**, MOF normaliza los **tipos de interfaz** que pueden utilizarse para funcionar en las entidades modelo (interfaces OMG IDL). Además, XMI [6] relacionada normaliza la

externalización de los modelos en el formato de tren XML [12]. El vocabulario XML utilizado para la externalización depende únicamente de las entidades **metamodelo**.

5.1 Definiciones y convenios

En esta cláusula se definen los conceptos del modelo (**meta-metamodelo**) MOF que se utilizan para definir los conceptos del eODL (**metamodelo**). Véase la figura 3. En dicha figura se introduce la notación utilizada para la visibilidad de los atributos y operaciones. Esta notación es la misma en todas las figuras UML. Para mayor información puede consultarse [4].

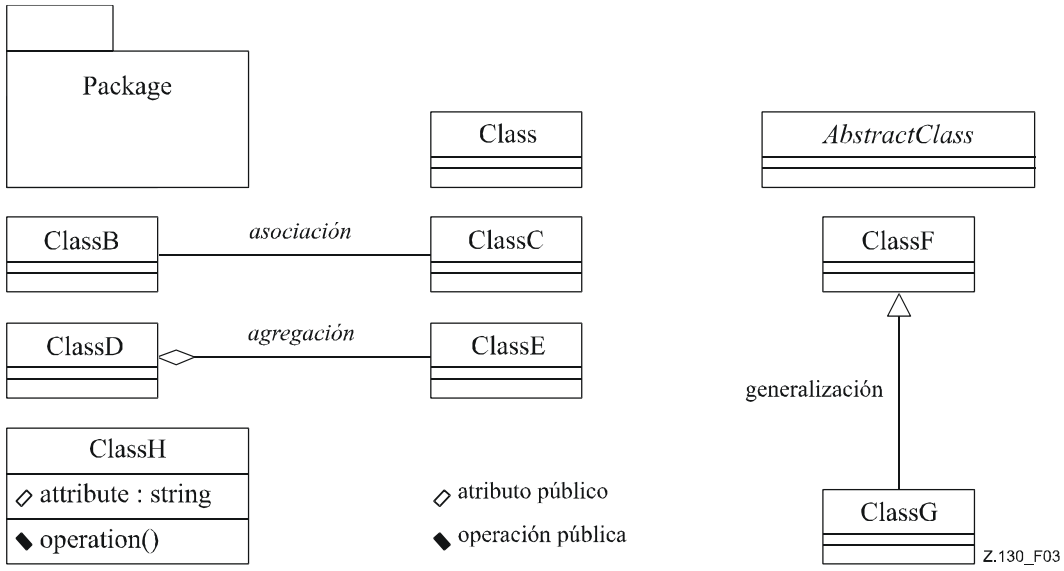


Figura 3/Z.130 – Notación UML para los conceptos MOF

5.1.1 clase y objeto: La clase es la descripción de un conjunto de objetos que tienen las mismas características peculiares de clase. La clase se utiliza para la clasificación y constituye el concepto básico de la construcción. Un objeto es un ejemplar de una cierta clase.

5.1.2 generalización: La generalización es una relación unidireccional entre dos clases. La generalización asocia la clase especial con la general. La clase especial hereda todas las características de la clase general.

5.1.3 asociación: La asociación es la relación entre dos clases. Cuando hay ejemplares de las dos clases, la asociación puede estar ejemplificada entre ellas o no. La asociación es navegable desde uno de los objetos implicados al otro.

5.1.4 agregación: La agregación es una relación dirigida entre clases en la que los ejemplares de las clases agregadas (partes) se consideran contenidos en ejemplares de la clase contenedora (o sea, de la clase agregativa). Su semántica es la de una relación 'tiene un'. Se puede establecer una distinción entre la agregación fuerte y la débil en relación con el ciclo vital de las partes y de su contenedor. En esta Recomendación se aplica siempre la agregación fuerte. Se utiliza para indicar composición frente a descomposición.

El **metamodelo** de eODL se define mediante la notación UML para MOF (véase la figura 4). Las restricciones y reglas de formación correcta que forman parte de este **metamodelo** y que son indispensables para su semántica se facilitan como texto en inglés. Los diagramas UML de las figuras de las siguientes cláusulas sólo muestran algunas partes del **metamodelo** eODL. Para entender la semántica hay que leer el texto explicativo, especialmente el relativo a los requisitos. Las restricciones ya descritas como parte del **metamodelo** IDL no se reproducen en esta Recomendación. Sírvase consultar [7] a tal efecto.

El **metamodelo** completo, incluidas las restricciones, se referencia como un tren XMI en el anexo D.

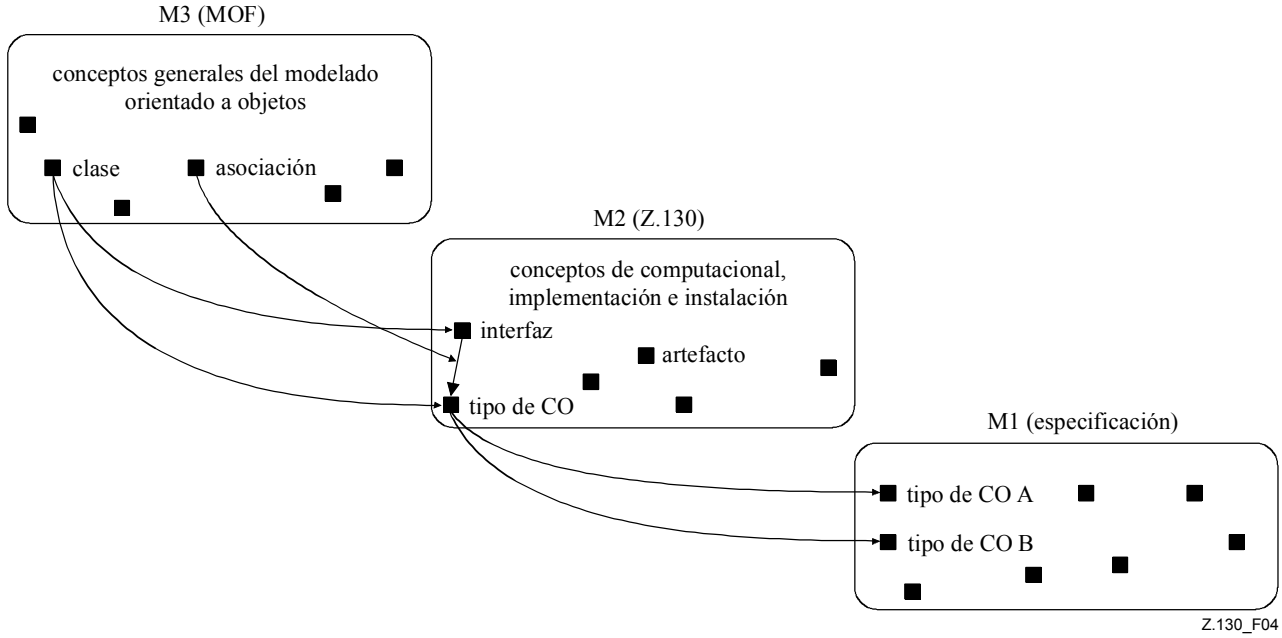


Figura 4/Z.130 – Conceptos orientados a objeto

5.2 Denominación y delimitación

Las reglas de denominación y delimitación se definen para poder identificar sin ambigüedades los elementos del modelo.

El modelo en su totalidad constituye un ámbito de nombres. Cada entidad que forma un nuevo ámbito es un ejemplar de la metaclase abstracta *Container* (*contenedora*). Los elementos contenidos en un ámbito son ejemplares de la metaclase abstracta *Contained* (*contenida*). La definición de las metaclases *Container* y *Contained* como abstractas implica que todas los ejemplares se derivan de metaclases no abstractas. La relación *Container-Contained* se utiliza con frecuencia en el **metamodelo**.

Cada entidad con nombre (ejemplar de la metaclase *Contained*) tiene un identificador que representa el nombre. Este identificador es un atributo de la metaclase *Contained*. Los identificadores de dos entidades diferentes con nombre, que pertenezcan al mismo *Container* (definedIn apunta al mismo elemento del modelo) deben ser diferentes.

Para que un modelo pueda tener ámbitos puros, se introduce la metaclase *ModuleDef*. *ModuleDef* forma parte del **metamodelo** CORBA-IDL en el que se basa el **metamodelo** de esta Recomendación. Es pues una metaclase concreta que puede ejemplificarse. No ostenta propiedades adicionales.

Cada ejemplar de *Container* forma un espacio de nombres. La generalización de *Contained* a *Container* expresa la posibilidad de anidar ámbitos de nombres (véase la figura 5).

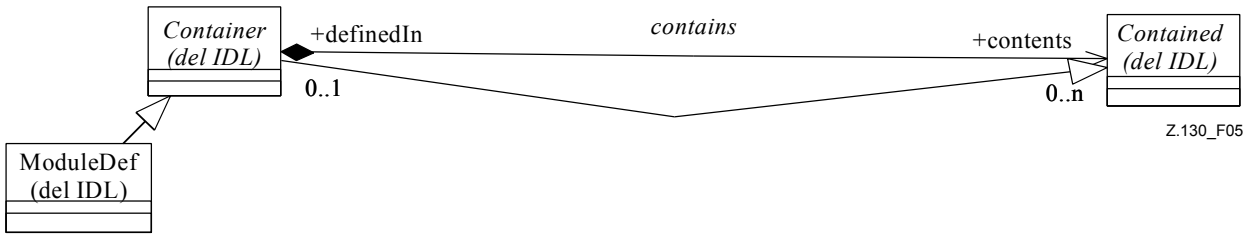


Figura 5/Z.130 – Denominación y delimitación

5.3 Conceptos computacionales

5.3.1 Conceptos CORBA IDL utilizados

Para introducir los **tipos de datos**, **operaciones**, **atributos**, **excepciones** y **tipos de interfaz** como conceptos de modelado, el **metamodelo** eODL se basa en el **metamodelo** CORBA-IDL.

Todos estos conceptos de modelado permiten definir bloques constructivos básicos para las especificaciones de computación. Uno de los propósitos de una especificación computacional es definir las signaturas de los **objetos computacionales** (**CO**) en sus **puertos**. Dado que esta Recomendación presenta un modelado basado en tipos, los **tipos de datos** son indispensables para la descripción de estas signaturas.

5.3.1.1 Tipos de datos, tipos de interfaz, operaciones, atributos, excepciones

Los **tipos de datos** de los modelos son ejemplares de metaclases derivados de la metaclase abstracta *IDLType*. Esto supone la inclusión de todo el sistema de tipo de datos CORBA-IDL. Al utilizar la metaclase abstracta *IDLType* se garantiza la posibilidad de intercambio del sistema de tipo de datos para conseguir que esta Recomendación sea abierta. La figura 6 muestra un subconjunto del **metamodelo** CORBA IDL correspondiente a los **tipos de datos**. En [7] se presenta el metamodelo junto con una descripción adicional.

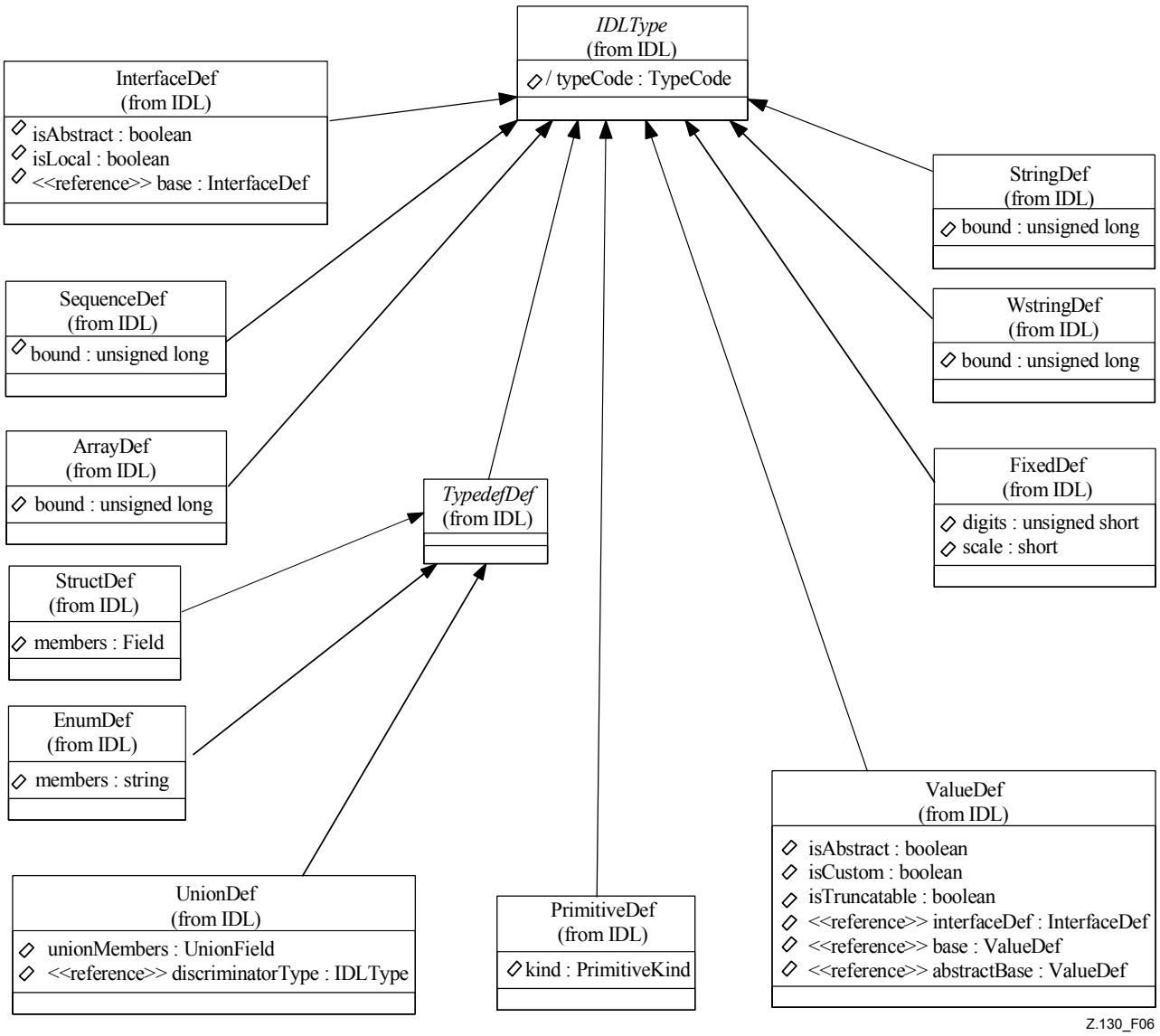


Figura 6/Z.130 – Tipos de datos

El sistema de tipos de datos CORBA contiene todos los **tipos de datos** primitivos comúnmente utilizados tales como *long, float, char, string,* etc. Además, estos conceptos sirven para describir **tipos de datos** estructurados tales como *array, struct, sequence, union,* etc. El **tipo de datos** *value* es un tipo adicional que se utiliza para pasar objetos utilizando una semántica de objeto por valor. Al igual que las clases en los lenguajes de programación, un ejemplar del tipo *value* puede agregar **atributos** y **operaciones**.

Para utilizar otras definiciones de tipos de datos distintas de las incluidas en CORBA, se introduce el elemento Extern Type (tipo exterior), que es una particularización del concepto TypedefDef de CORBA. El identificador de atributos se refiere aquí a que la definición del tipo de datos se suministra desde el exterior. Véase la figura 7.

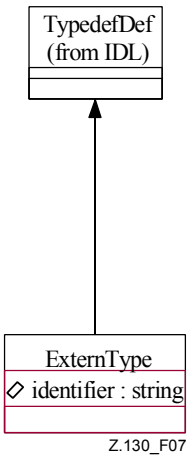


Figura 7/Z.130 – Tipo exterior

Los **atributos**, **operaciones**, **excepciones** y **parámetros** son conceptos necesarios para la definición de las interacciones operacionales. Una **operación** definida en un modelo contiene una lista de parámetros, un tipo para el posible valor de retorno y una lista de finales insatisfactorios modelados por **excepciones**. Las **excepciones** transportan información, se definen del mismo modo que el **tipo de datos** *StructDef* y contienen una lista de miembros para dicha información. Un **atributo** es una notación abreviada de las operaciones de modelado utilizadas para obtener o dar valor a una variable con nombre de un cierto tipo. Además, pueden añadirse **excepciones** a los **atributos**. En tal caso, se distingue entre aquellas **excepciones** que pueden plantearse durante una **operación** *set* de dar valor a dicho **atributo** y las aparecidas durante una **operación** *obtención*. La figura 8 representa el **metamodelo** de las **excepciones**.

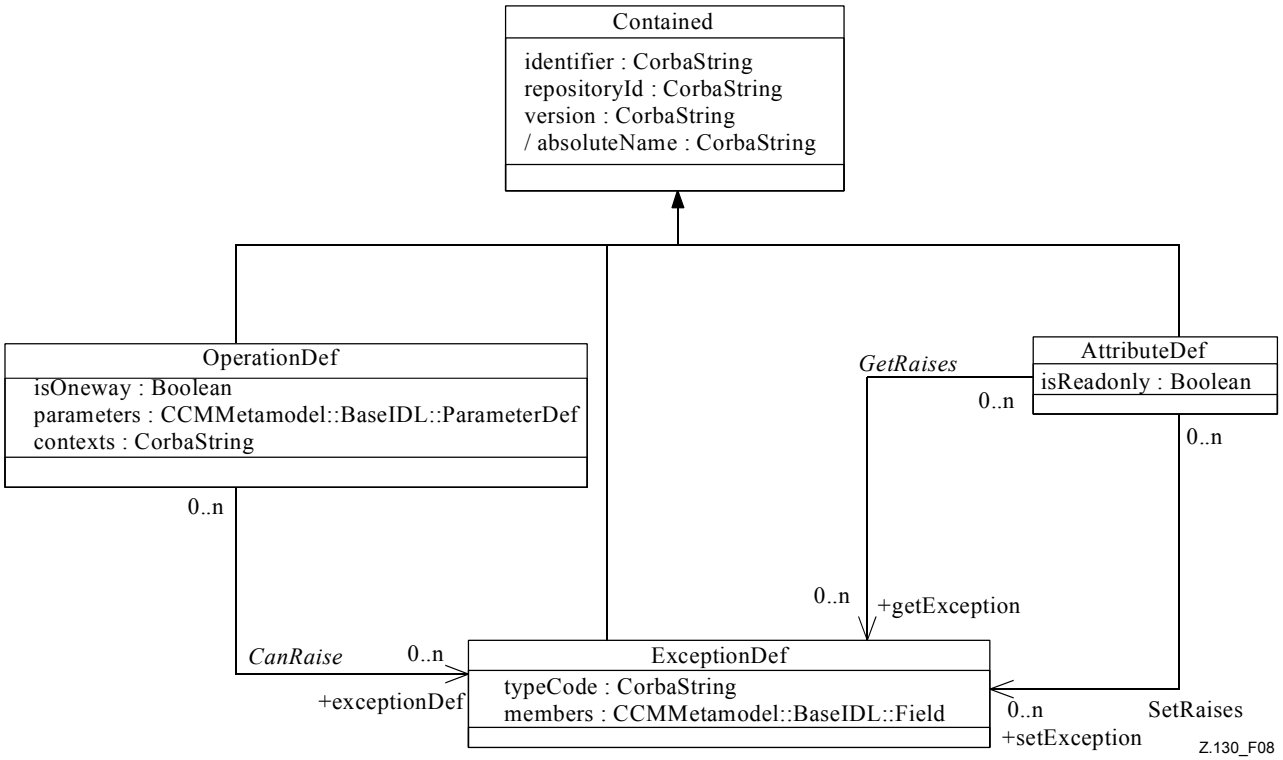


Figura 8/Z.130 – Excepciones

Las metaclases *OperationDef, AttributeDef, ParameterDef* y *ExceptionDef* proceden del **metamodelo** CORBA-IDL y se describen con detalle en el mismo. El **metamodelo** correspondiente a **atributos** y **operaciones** se representa a continuación (véanse las figuras 9 y 10).

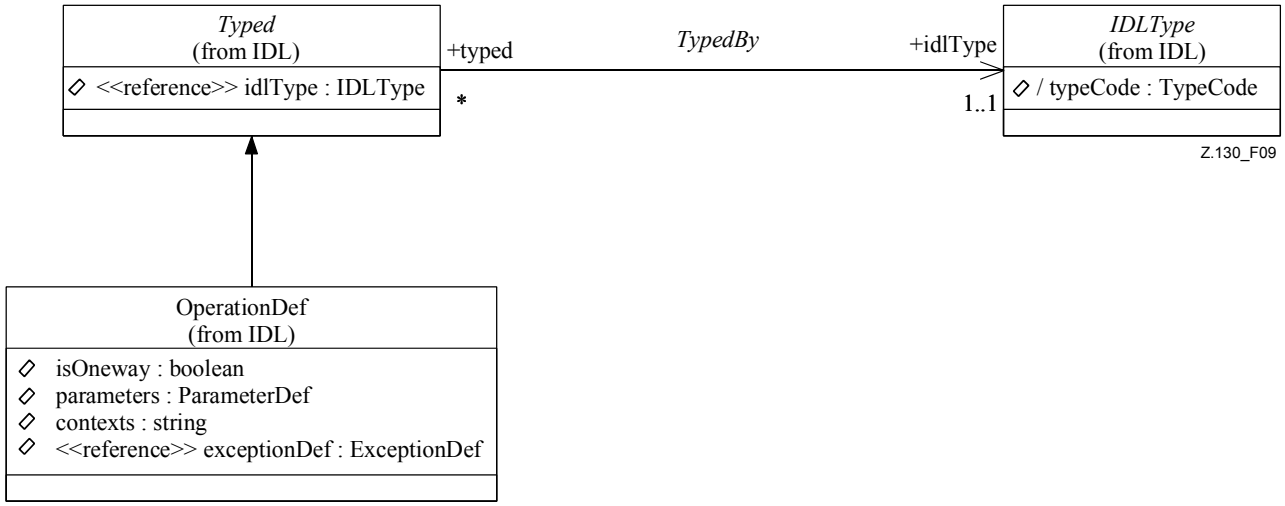


Figura 9/Z.130 – Operaciones

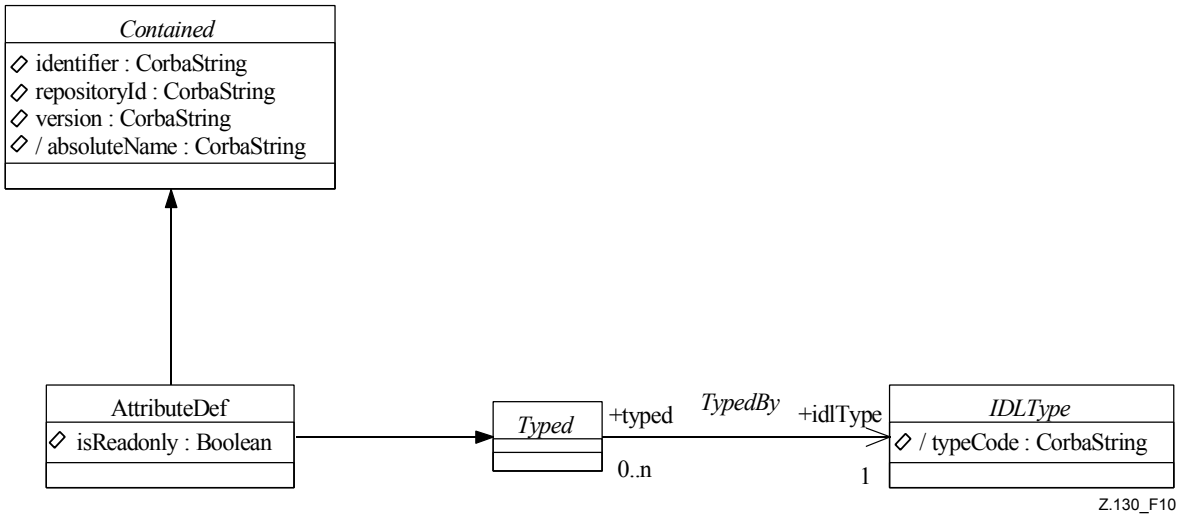


Figura 10/Z.130 – Atributos

Los **tipos de interfaz** se utilizan para definir signaturas destinadas a las posibles **interacciones** de un sistema. El concepto de **tipo de interfaz** ya se conoce de OMG IDL, donde se denomina interfaz. En OMG IDL, las interfaces sólo agregan **elementos de interacción** operacionales. Eso significa que son contenedores de **atributos** y **operaciones**. Esto se muestra en la figura 11.

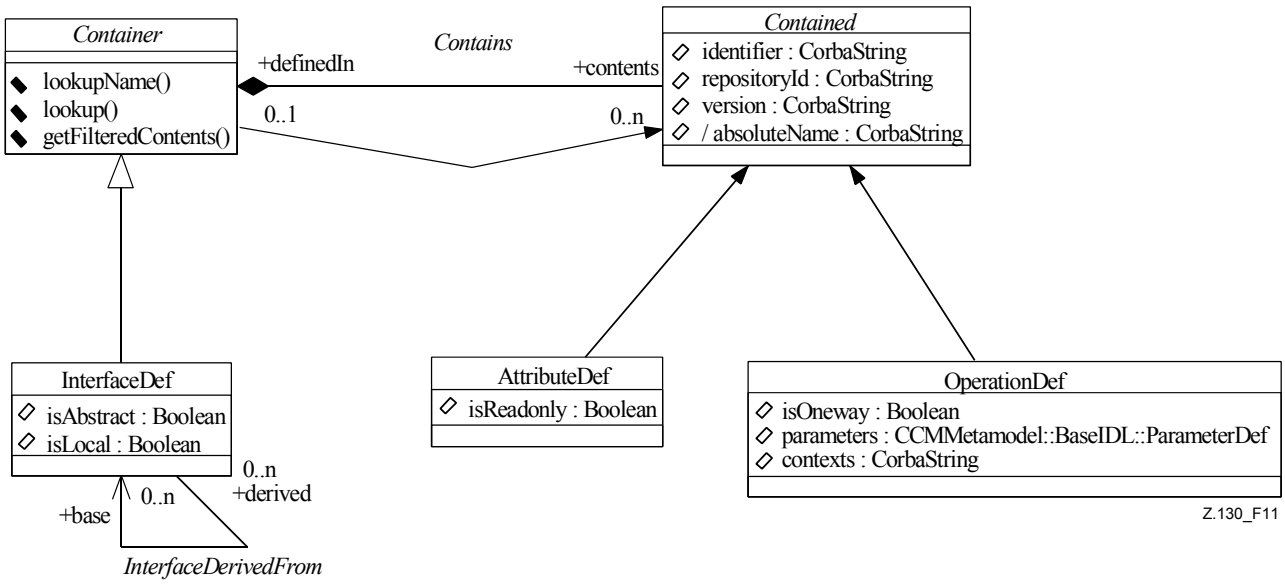


Figura 11/Z.130 – Interfaces operacionales

5.3.2 Señales y parámetros de señal

Metamodelo

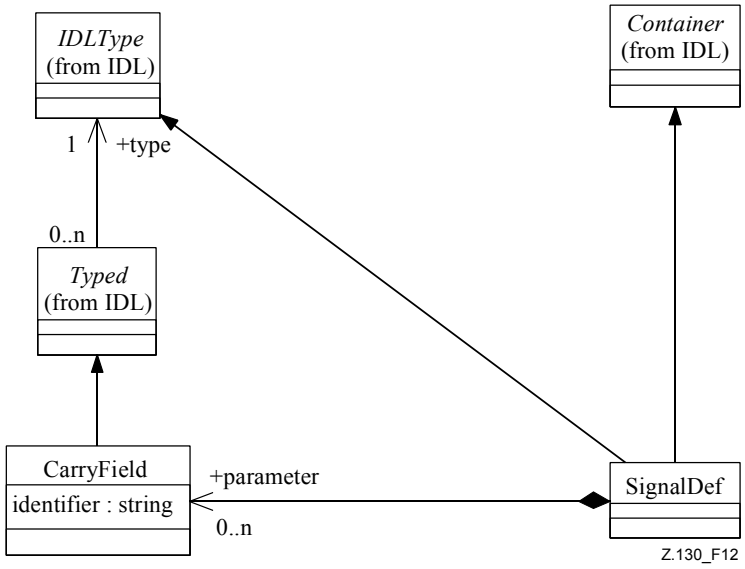


Figura 12/Z.130 – Señal y parámetros de señal

Semántica

Para ampliar los conceptos de modelado ofrecidos por CORBA IDL y para incluir otros tipos de interacción en el modelado de sistemas distribuidos, es necesario definir conceptos de modelado adicionales (véase la figura 12).

Las **señales** se utilizan para modelar la interacción basada en **señales**, es decir el intercambio asíncrono de mensajes desacoplados entre entidades del sistema. Las **señal**es transportan información. Las **señales** se modelan como ejemplares de la metaclase *SignalDef*. La información transportada (denominada **parámetros de señal**) se modela como ejemplares de *CarryField*, cada uno de ellos con referencia a un ejemplar de *ValueDef*, que es un **tipo de datos** especial de CORBA-IDL. Cada **parámetro de señal** se puede identificar por medio de un nombre en el contexto de una definición de **señal**. Los nombres han de ser únicos.

Una definición de **señal** prescribe la estructura y propiedades de la información transportada en una interacción de señal concreta entre entidades del sistema. Sin embargo no se asocia a ningún **tipo de interfaz**. Las definiciones de **señal** pueden reutilizarse en distintas definiciones de **tipo de interfaz** y desempeñan el mismo papel que los **tipos de datos** en la definición de **operaciones**. Son los bloques constructivos de la **interacción** basada en **señal**.

Las definiciones de las **señales** en los modelos sólo pueden aparecer en **espacios de nombre** que son o bien módulos o el **espacio de nombres** global formado por la propia especificación.

Los tipos IDL utilizados para especificar los parámetros de las **señales** han de ser ejemplares del tipo *value*.

5.3.3 Tipo de medio, medio y conjunto de medios

Metamodelo

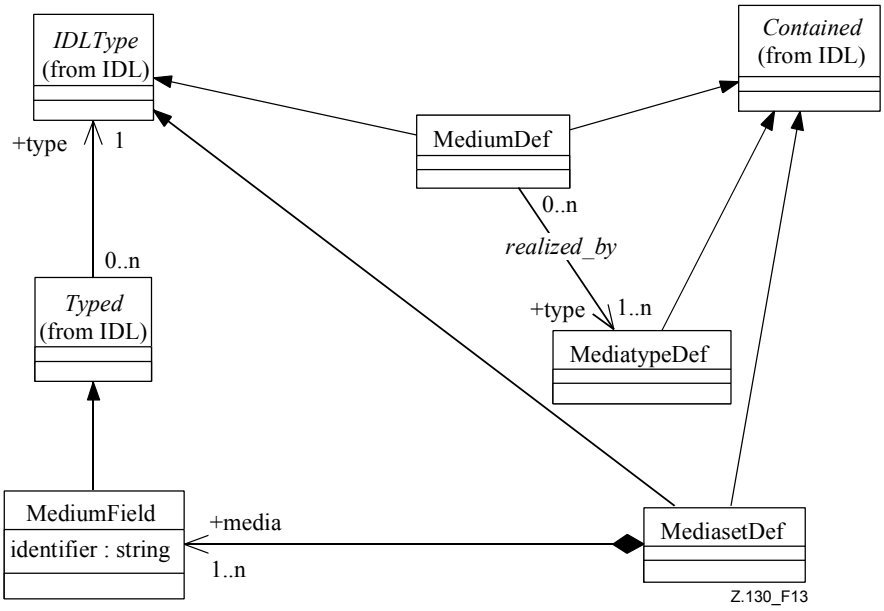


Figura 13/Z.130 – Medio, tipo de medio y conjunto de medios

Semántica

Además de la interacción **operacional** y la basada en **señales**, el intercambio de **medios continuos** también es un tipo importante de interacción en los sistemas distribuidos. Han de facilitarse los conceptos de modelado correspondientes a este tipo de interacción. Esto se lleva a cabo en la presente Recomendación exactamente de la misma manera que para la interacción **operacional** y la interacción de **señales**. En primer lugar hay que definir los bloques constructivos básicos de los **elementos de interacción**. Éstos son el **medio**, el **conjunto de medios** y el **tipo de medios** (véase la figura 13).

El concepto **conjunto de medios** se utiliza para modelar la interacción de los **medios continuos**. En el **metamodelo** esto se realiza por la definición de la clase *MediasetDef*. Los ejemplares de esta clase agregan ejemplares de la clase *MediumDef* a una lista con nombre en la que cada elemento es del tipo *MediumField*. El concepto de **medio** se utiliza para modelar un flujo de datos atómico entre dos entidades. Un **medio** tiene el significado de información multimedios tal como películas o secuencias de audio. El intercambio requiere la existencia de codificación, decodificación y formatos de transmisión, modelados por ejemplares de la clase *MediatypeDef*. Un **medio** puede realizarse por uno o más **tipos de medios**. Los **conjuntos de medios** son necesarios para modelar el intercambio de dos medios diferentes que vayan juntos (como los datos de vídeo y audio de una película) y en los que las interacciones deben tener las mismas propiedades para ambos.

Las definiciones de **medios**, **conjuntos de medios** y **tipos de medios** en los modelos sólo pueden existir en los espacios con nombre que son módulos o el espacio de nombres global constituido por la propia especificación.

5.3.4 Consumir y producir

Metamodelo

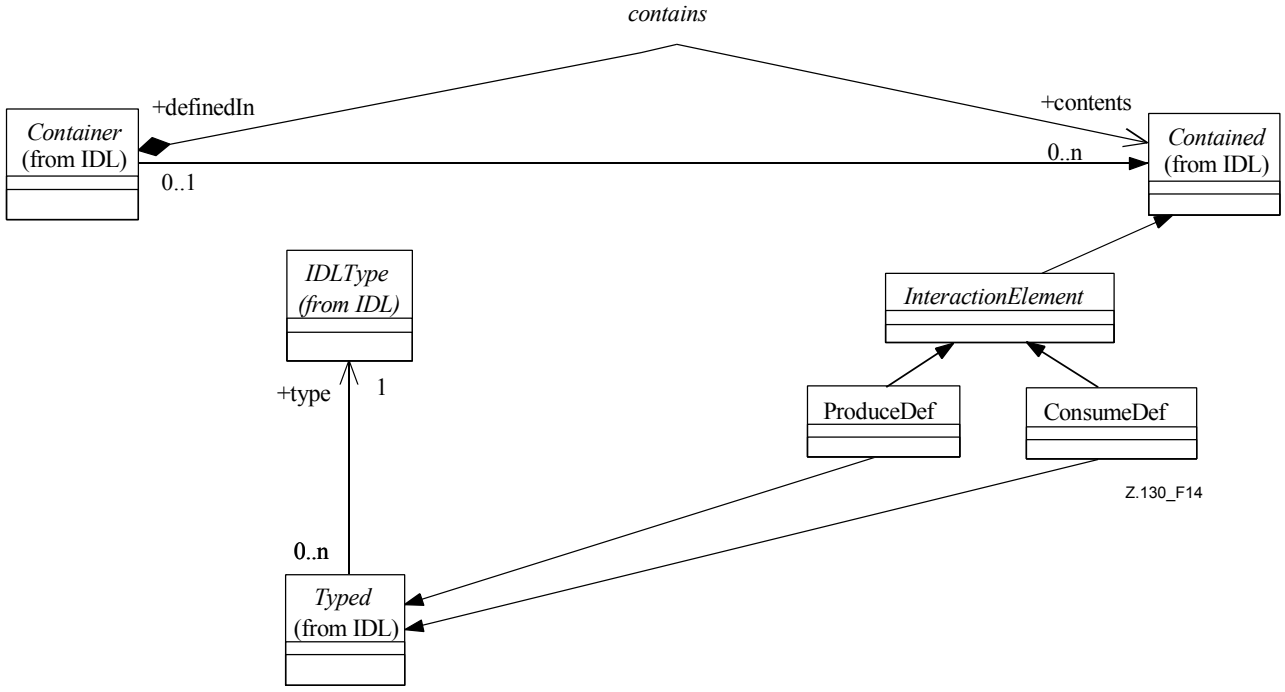


Figura 14/Z.130 – Consumir y producir

Semántica

Las **señales** se intercambiarán entre las entidades funcionales durante la **ejecución** a través de sus correspondientes **interfaces**. Por dicho motivo, los **tipos de interfaz** ofrecen las signaturas necesarias. En el caso de comunicación de **señales**, la signatura contiene el tipo (definición de la **señal**), un nombre para el **elemento de interacción** y la indicación de si la **señal** se produce o consume a través de esta interfaz. Por consiguiente, los conceptos de **consumir** y **producir** son **elementos de interacción** y se utilizan para modelar el consumo y la producción de **señales** (véase la figura 14). Se trata de elementos de interacción de **señales** en el mismo sentido que las **operaciones** y los **atributos** son los elementos de la interacción **operacional**. Al igual que los **elementos de interacción**, **consumir** y **producir** son elementos identificables. Se incluyen en el **metamodelo** con la definición de las metaclases *ConsumeDef* y *ProduceDef*, que se heredan de la metaclase abstracta *InteractionElement*, que a su vez se hereda de *Contained*. *ConsumeDef* y *ProduceDef* son subclases de *Typed*. Esta herencia se utiliza para establecer la relación con la *SignalDef* objeto de **producir** o **consumir**.

Obsérvese que *SignalDef* es una subclase de *IDLType*. Es necesario que el ejemplar de *IDLType* asociado a una definición de **producir** o **consumir** sea una **señal**. El concepto de **señal** se define como una especialización de *IDLType* a tal efecto.

5.3.5 Sumidero y fuente

Metamodelo

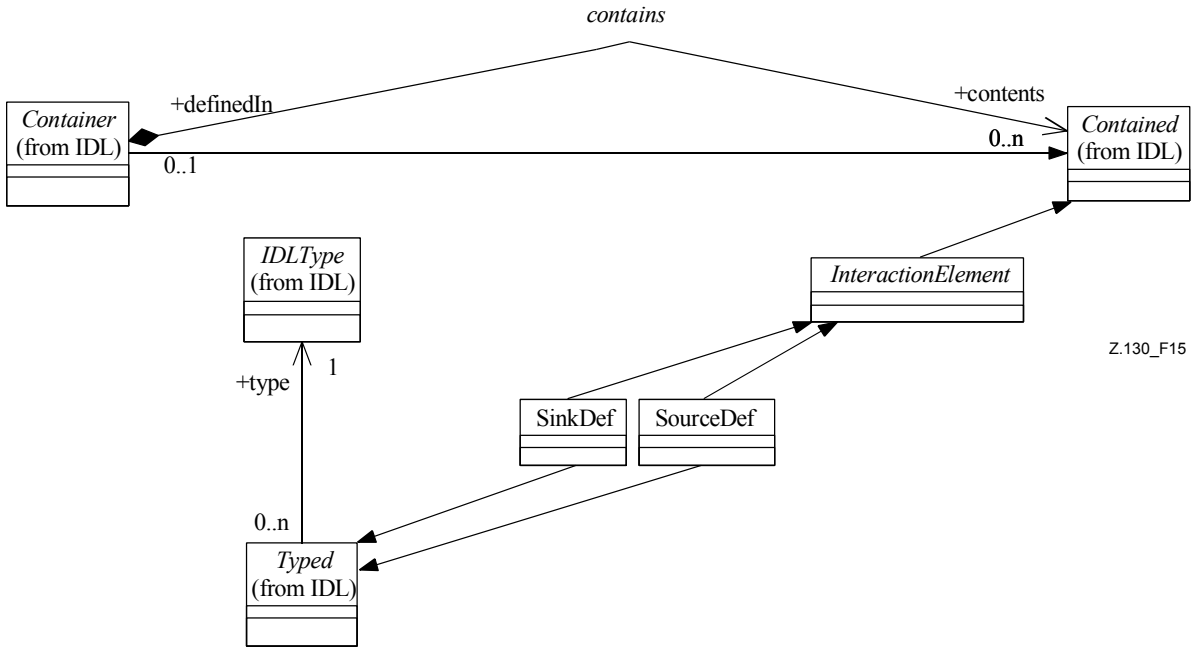


Figura 15/Z.130 – Sumidero y fuente

Semántica

Para completar la lista de los posibles **elementos de interacción** utilizados para especificar signaturas de **tipos de interfaz**, han de definirse **elementos de interacción** para las interacciones de **medios continuos** que se caracterizan, como las interacciones de las **señales**, por la información que se intercambia cuando se produce una interacción durante la **ejecución** (**conjunto de medios**), por un identificador y por el sentido de la comunicación, es decir si la **interfaz** es **sumidero** o **fuente** con respecto a la interacción. Por consiguiente, los conceptos de **sumidero** y **fuente** son **elementos de interacción** para modelar el consumo o producción de **conjuntos de medios** (véase la figura 15). Se trata de elementos de interacción de **medios continuos** en el mismo sentido que las **operaciones** y **atributos** son elementos de la interacción **operacional**. Como ocurre con todos los **elementos de interacción**, **sumidero** y **fuente** son elementos identificables. Se incluyen en el **metamodelo** con la definición de las metaclases *SinkDef* y *SourceDef*, que se heredan de la metaclase abstracta *InteractionElement*, que a su vez se hereda de *Contained*. *SinkDef* y *SourceDef* son subclases de *Typed*. Esta herencia se utiliza para establecer la relación con los *MediasetDef*, objeto del concepto de **sumidero** y **fuente**.

Obsérvese que *MediasetDef* es una subclase de *IDLType*. El único *IDLType* concreto que se permite asociar a un **sumidero** o **fuente** es **tipo de medio**.

5.3.6 Tipo de interfaz

Metamodelo

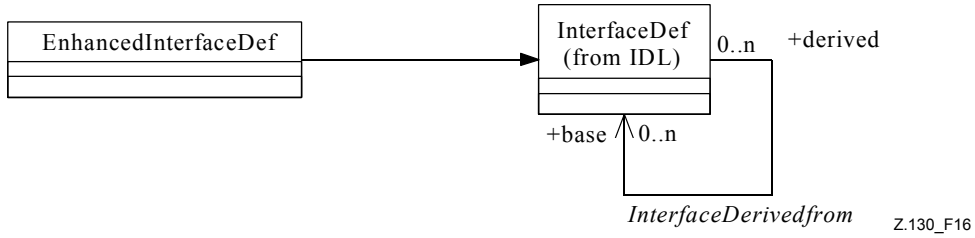


Figura 16/Z.130 – Tipo de interfaz

Semántica

El concepto **tipo de interfaz** se utiliza para especificar un subconjunto de **interacciones** potenciales de **CO** de **tipos de CO**. Los **tipos de interfaz** agregan **elementos de interacción** de los tipos de interacción **operación**, **señal** y **medios continuos**. Con esto, la semántica de un **tipo de interfaz** se amplía con respecto a RM-ODP: los **tipos de interfaz** proporcionan un contexto común para los **elementos de interacción** de tipos de interacción *diferentes*. Para los clientes, que tengan una referencia a dicha **interfaz**, es posible utilizar todos los **elementos de interacción** con independencia del tipo de interacción utilizada. El entorno de ejecución es el que se ocupa de este aspecto.

En el **metamodelo**, el **tipo de interfaz** es un ejemplar de la clase denominada *EnhancedInterfaceDef* (véase la figura 16). Dado que el **metamodelo** de IDL ya contiene un significado de agregación de **elementos de interacción operacional**, la clase *EnhancedInterfaceDef* se hereda de *InterfaceDef*. De esto se deduce que las reglas de herencia y las restricciones de *InterfaceDef* también son aplicables a *EnhancedInterfaceDef*.

Además, los **tipos de interfaz** son contenedores para los **elementos de interacción producir, consumir**, **sumidero** y **fuente**.

5.3.7 Tipos de CO y relaciones soporta y requiere

Metamodelo

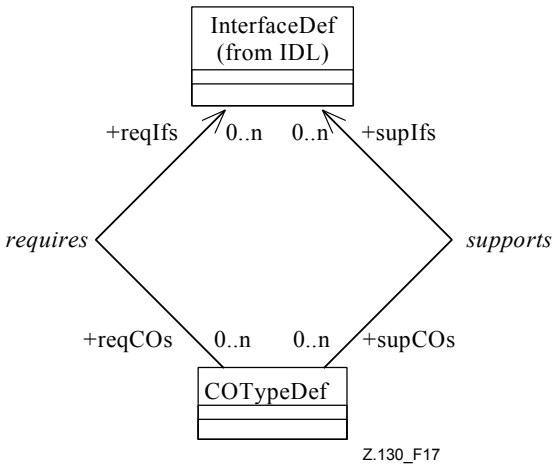


Figura 17/Z.130 – Tipos de CO, soporta y requiere

Semántica

El concepto de **tipo de CO** se utiliza para especificar la descomposición funcional de un sistema. Los ejemplares de un **tipo de CO (CO)** son entidades interactivas autónomas que encapsulan estado y comportamiento. Los **CO** interaccionan con su entorno a través de **interfaces** definidas. Estas **interfaces** se especifican mediante el concepto **tipo de interfaz** descrito anteriormente.

Un **tipo de CO** puede soportar **(soporta)** o requerir **(requiere)** un **tipo de interfaz**. Soportar un **tipo de interfaz** significa que los **CO** de dicho **tipo de CO** proporcionan **interfaces** de dicho **tipo de interfaz**. Requerir un **tipo de interfaz** significa que los **CO** de dicho **tipo de CO** utilizan **interfaces** de dicho **tipo de interfaz**. Un **tipo de CO** es un ejemplar de la clase *COTypeDef* del **metamodelo**. Las etiquetas soporta y requiere identifican las asociaciones entre *COTypeDef* e *InterfaceDef* (véase la figura 17).

Para acceder a los **CO** en tiempo de **ejecución** se obtiene *COTypeDef* de *InterfaceDef* (como se representa en la figura 17). De este modo, los ejemplares pueden configurarse mediante atributos definidos por este **tipo de CO**. Es importante señalar que a un **tipo de CO** sólo se le permite contener **elementos de interacción** del tipo de atributo. No se permite ningún otro **elemento de interacción**. Además, las relaciones de herencia entre **tipos de CO** y **tipos de interfaz** no pueden mezclarse, o sea los **tipos de CO** sólo pueden heredarse de **tipos de CO** y los **tipos de interfaz** de **tipos de interfaz**.

5.3.8 Definición de puerto suministrado y utilizado

Metamodelo

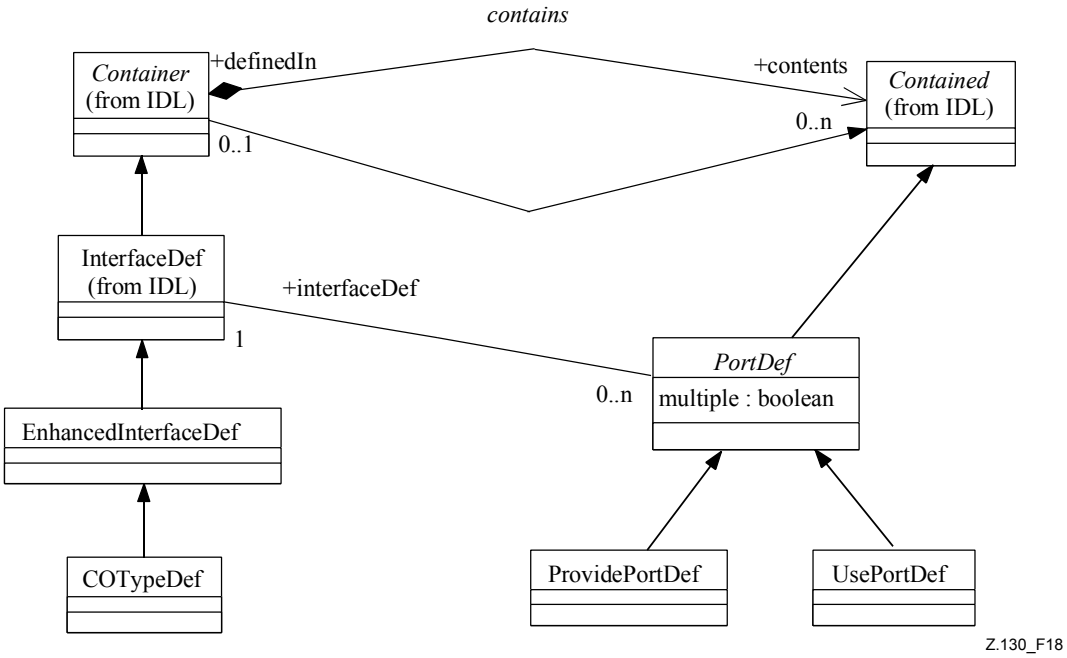


Figura 18/Z.130 – Puerto suministrado y utilizado

Semántica

Los **CO** son entidades funcionales de un sistema distribuido especificado mediante la presente Recomendación que se comunican a través de sus **interfaces** soportadas y requeridas. No obstante, la configuración de sistemas distribuidos siempre plantea problemas, especialmente en relación con la obtención e intercambio de **referencias de interfaz**, lo que constituye un requisito previo para la **interacción**. Por este motivo, la presente Recomendación introduce el concepto de **puerto** como

punto de interacción con nombre en el que obtener la referencia de una **interfaz** soportada de un **CO** o registrar en tiempo de **ejecución** la referencia de una **interfaz** utilizada.

Los conceptos de **puerto suministrado** y **utilizado** sirven para modelar **puertos** de un **tipo de CO** utilizados por el entorno para obtener una referencia a una **interfaz (puerto suministrado)** o para guardar una referencia a una **interfaz (puerto utilizado)** con arreglo a un nombre. Con los conceptos de **soporta** y **requiere** sólo puede expresarse el posible suministro o utilización de **tipos de interfaz** en el contexto de un **tipo de CO**, aunque no los mecanismos concretos que permiten al entorno de un **CO** tener acceso a estos contextos de interacción. Los conceptos **puerto suministrado** y **puerto utilizado** se definen como ejemplares de las clases *ProvidePortDef* y *UsePortDef*. Estas dos clases se heredan de la clase abstracta *PortDef* que a su vez se hereda de *Contained*, lo que significa que un ejemplar de *COTypeDef* puede contener definiciones de **puerto suministrado** y de **puerto utilizado**. Las definiciones de **puerto suministrado** y de **puerto utilizado** se asocian siempre a la definición de **interfaz** (véase la figura 18).

Las definiciones de **puerto** sólo se permiten en las definiciones de **tipo de CO**. Una **interfaz** para la que se defina un **puerto suministrado** se convierte automáticamente en una **interfaz** soportada. Una **interfaz** para la que se defina un **puerto utilizado** se convierte automáticamente en una **interfaz** requerida.

5.4 Conceptos de implementación

5.4.1 Artefactos y diagrama de ejemplificación

Metamodelo

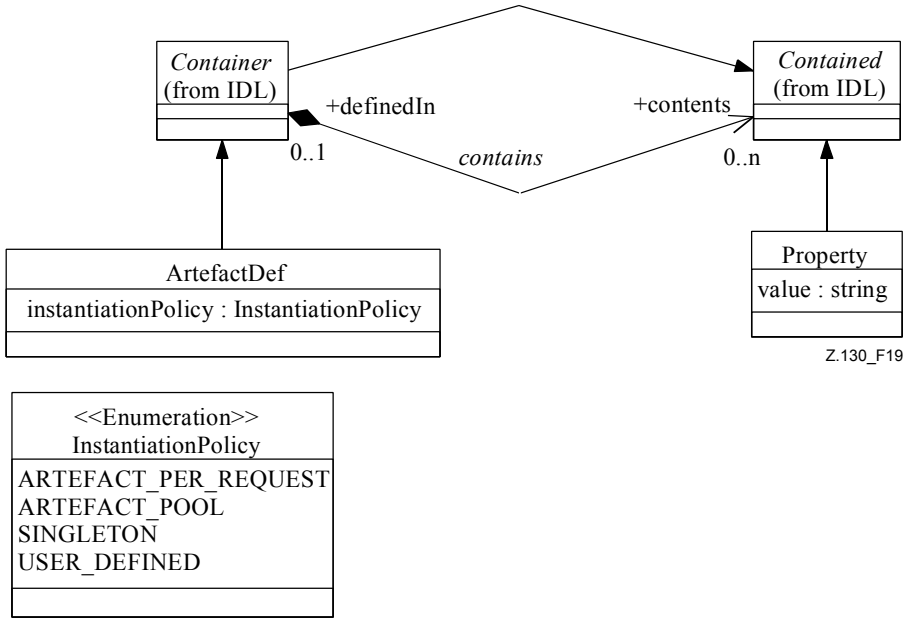


Figura 19/Z.130 – Artefactos y diagrama de ejemplificación

Semántica

El concepto de **artefacto** sirve para describir un contexto de lenguaje de programación (tal como una clase de un lenguaje de programación orientado a objetos) en un modelo. Los ejemplares del concepto **artefacto** realizan el comportamiento de los **CO** y por consiguiente suministran la lógica de la empresa de los **tipos de CO**. Las relaciones entre los **artefactos** y las partes comportamentales de los **CO** se definen mediante asociaciones entre los **artefactos** y los **elementos de interacción** de los **tipos de interfaz**. Los contextos de lenguaje de programación modelados por ejemplares del concepto **artefacto** se ejemplificarán en tiempo de **ejecución** para procesar, por ejemplo,

invocaciones de **operaciones**, entradas de **señal** o datos de **medios continuos**. Las políticas a utilizar para la ejemplificación se especifican mediante ejemplares del concepto diagrama de ejemplificación. Los diagramas permitidos se representan en la figura 19. Si fuera necesario se podrían añadir más diagramas. La separación de los conceptos **artefacto** y **tipo de CO** confiere plena flexibilidad al diseño de la aplicación distribuida:

- la perspectiva externa (el modo en que el entorno puede interaccionar con un **CO**) está separada de la perspectiva interna (el modo en el que se suministra el comportamiento de un **CO**);
- pueden utilizarse árboles de herencia diferentes para las perspectivas externa e interna;
- la reutilización del comportamiento y de las definiciones de **interfaz** existentes son posibles y recíprocamente independientes.

El concepto de **artefacto** se expresa en el **metamodelo** mediante un ejemplar de la clase denominada *ArtefactDef*. El concepto de diagrama de ejemplificación se modela como atributo de dicha clase de enumeración de tipo con los enumeradores representados en la figura 19.

5.4.2 Relación implementa

Metamodelo

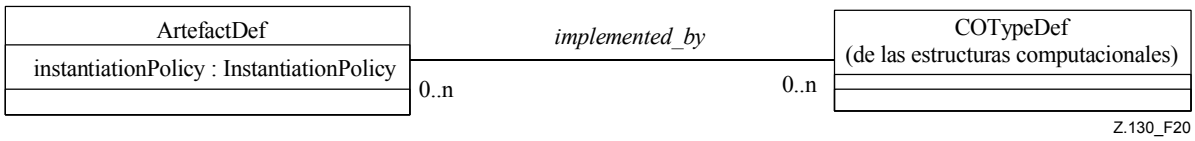


Figura 20/Z.130 – Relación implementa

Semántica

Los ejemplares del concepto **artefacto** describen la realización del comportamiento previsto de los **elementos de interacción** de los **tipos de interfaz** soportados o requeridos por los **tipos de CO**. Como se ha explicado anteriormente, las perspectivas externa e interna de un **CO** están totalmente separadas entre sí. No obstante, debe describirse la relación entra ellas para elegir el comportamiento correcto cuando el **CO** se ve implicado en una determinada interacción. Por consiguiente, el modelo describe qué conjunto de **artefactos** proporciona el comportamiento para los **CO** de dicho **tipo de CO**. Esta relación se define mediante una asociación *implemented_by* entre *COTypeDef* y *ArtefactDef* en el **metamodelo** (véase la figura 20).

5.4.3 El elemento implementación

Metamodelo

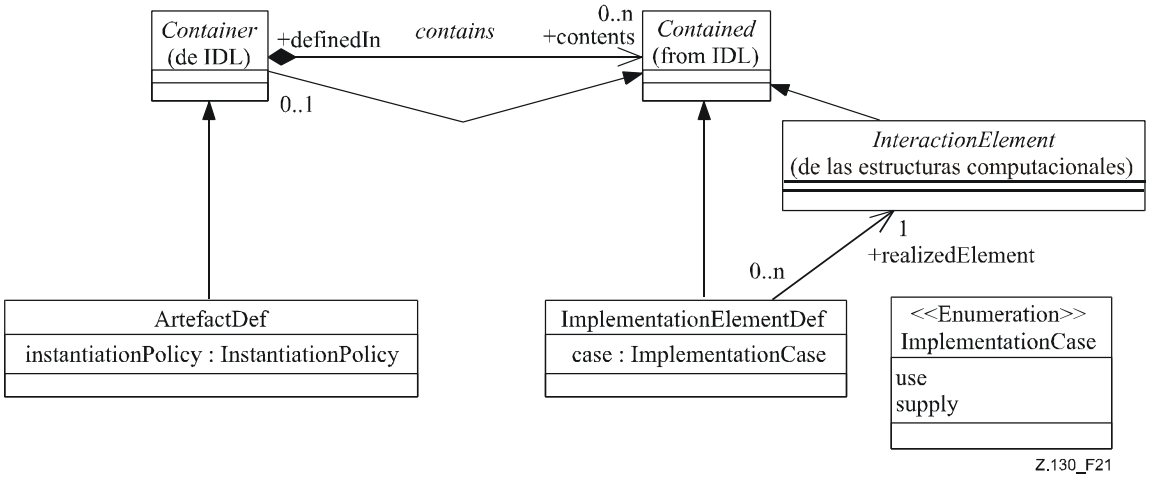


Figura 21/Z.130 – El elemento implementación

Semántica

En el contexto de un ejemplar de **artefacto**, el concepto de **elemento de implementación** se utiliza para indicar que una parte comportamental de un **artefacto** (por ejemplo un método de una clase, modelado por un ejemplar del concepto **artefacto**) **realiza** un **elemento de interacción** particular de un **tipo de interfaz**. Este concepto es necesario para proporcionar más detalles a la relación *implemented_by* como se ha indicado anteriormente; *implemented_by* especifica que un **artefacto** contribuye al comportamiento de un **CO** sin explicitar qué parte del **artefacto** es responsable de cuál otra del comportamiento del **CO**. Estos detalles se suministran con los **elementos de implementación** del **artefacto**. Esta información es necesaria para asociar el comportamiento a una interacción en tiempo de **ejecución**.

El concepto de **elemento de implementación** se especifica como ejemplar de la clase denominada *ImplementationElementDef*. Esta clase se hereda de *Contained* y puede haber ejemplares de *ImplementationElementDef* contenidos en ejemplares de **artefactos** (véase la figura 21). La *implementación case* define el sentido de implementación de un **elemento de implementación**: o bien la utilización o bien el suministro del comportamiento pueden **realizarse** por un **elemento de implementación** con respecto a un determinado **elemento de interacción**.

5.5 Conceptos de despliegue

5.5.1 Componentes de soporte lógico y dependencia de los componentes

Metamodelo

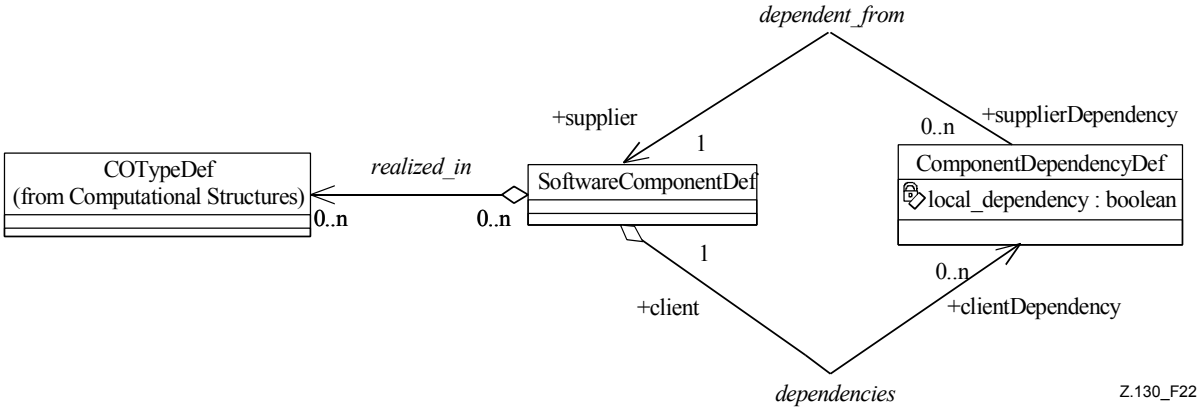


Figura 22/Z.130 – Componentes de soporte lógico y dependencia de los componentes

Semántica

El concepto de **componente de soporte lógico** refleja el soporte lógico real del modelo de diseño. Identifica una entidad de **instalación** y permite ampliar la descripción mediante la utilización de propiedades. Un **componente de soporte lógico** puede, aunque no forzosamente, **realizar** un número arbitrario de **tipos de CO**. Por consiguiente, contiene secuencias de instrucciones que al ejecutarse en un nodo materializan **CO**, es decir proporcionan el comportamiento, estado e identidad de los **CO**. En el **metamodelo**, este concepto se introduce mediante la metaclase *SoftwareComponentDef*. Para indicar qué **tipos de CO** se **realizan** por un determinado **componente de soporte lógico**, se utiliza el concepto de la relación **realizar**, introducido en el **metamodelo** mediante la asociación entre la metaclase *COTypeDef* y la *SoftwareComponentDef* (véase la figura 22).

Los **componentes de soporte lógico** pueden requerir otros **componentes de soporte lógico** a fin de poder ser ejecutados correctamente. Para poder reflejar esto en el modelo, se define la metaclase *SoftwareDependencyDef*. Ésta contiene el atributo *local_dependency* que expresa si el **componente de soporte lógico** requerido debe estar disponible localmente. Un *SoftwareComponentDef* puede contener un número arbitrario de *SoftwareDependencyDefs,* teniendo cada *SoftwareDependencyDef* una asociación con otro *SoftwareComponentDef* que indica el soporte lógico requerido.

5.5.2 Ensamblaje y configuración inicial

Metamodelo

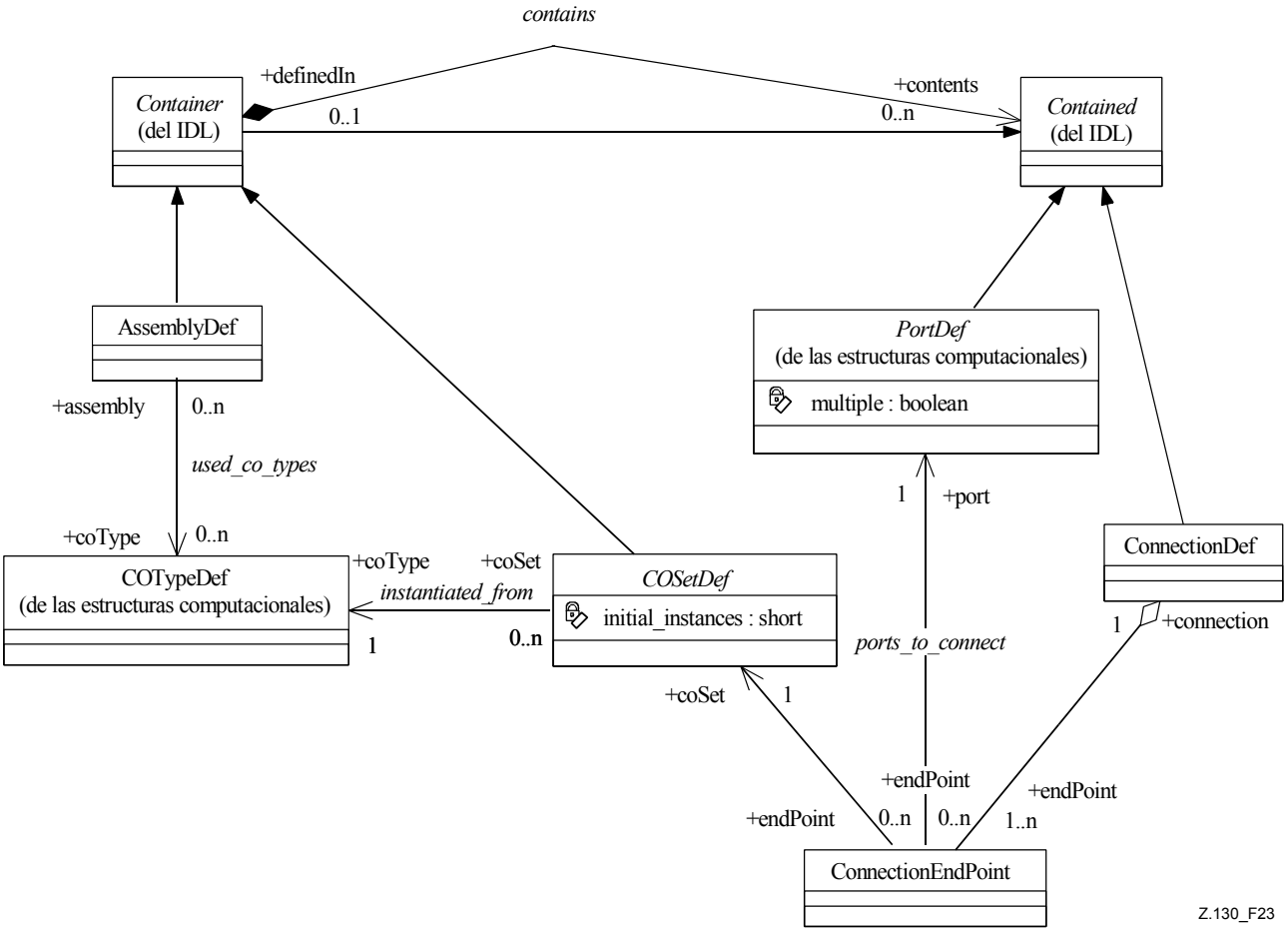


Figura 23/Z.130 – Ensamblaje y configuración inicial

Semántica

El concepto de **ensamblaje** se utiliza para modelar sistemas de soporte lógico mediante la especificación de los **tipos de CO** implicados en el sistema y para modelar la **configuración inicial** del sistema. La **configuración inicial** es la configuración establecida al comenzar el tiempo de ejecución del sistema de soporte lógico y consiste en los **CO iniciales** y en sus **conexiones iniciales**. En el **metamodelo**, el concepto de **ensamblaje** se modela mediante la metaclase *AssemblyDef*. Los **tipos de CO** se asocian mediante la introducción de una asociación entre las metaclases *AssemblyDef* y *COTypeDef* (véase la figura 23).

Para modelar los **CO iniciales**, el **metamodelo** contiene la metaclase *COSetDef*. Una *COSetDef* define la creación de un número arbitrario de ejemplares del **tipo de CO** asociado. Este número viene determinado por el atributo *initial_instances*. Un *COSetDef* está contenido en un *AssemblyDef*.

Para modelar las **conexiones iniciales**, el **metamodelo** contiene la metaclase *ConnectionDef*. Una **conexión** se establece entre los puertos de los **CO** participantes mediante el intercambio de las **referencias de interfaz** de los **CO**. Estas referencias se obtienen de un **CO** en el que el **tipo de CO** tiene una definición de **puerto suministrado** y se transfiere a un **CO** cuyo tipo tiene una definición de **puerto utilizado**. En el **metamodelo**, la *ConnectionDef* consiste en un conjunto de *ConnectionEndPoints*. Un *ConnectionEndPoint* está asociado a un *PortDef* de un *COTypeDef* y a

un *COSetDef*. Cada **CO** de un *COSetDef* asociado a un *ConnectionEndPoint* se conecta a cada **CO** de cada *COSetDef* asociado a otros *ConnectionEndPoints* agregados a la misma *ConnectionDef*.

5.5.3 Propiedades y restricciones

Metamodelo

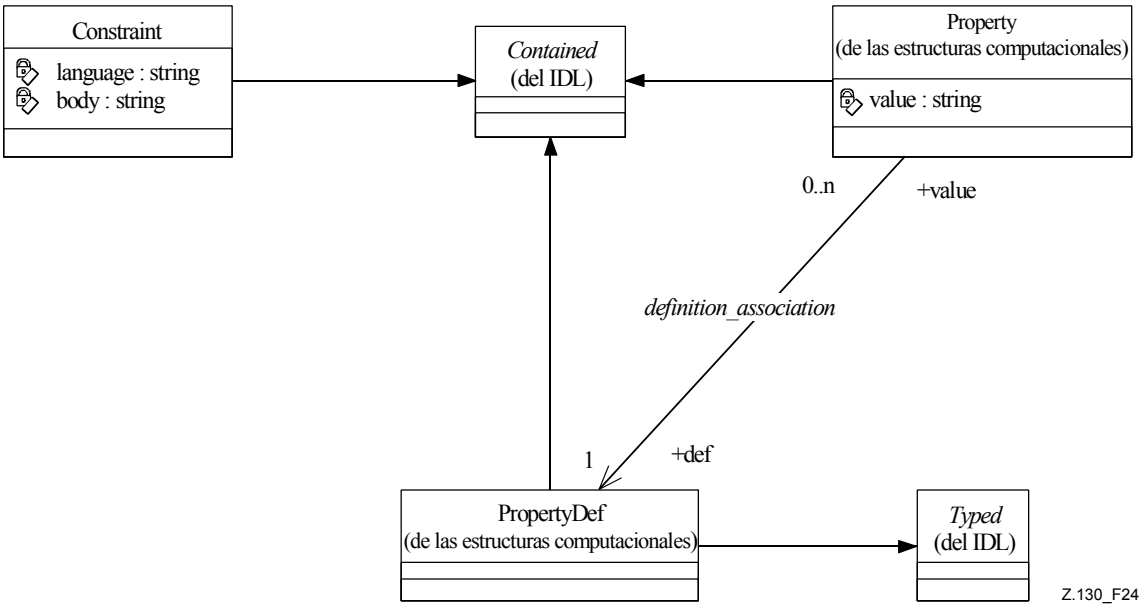


Figura 24/Z.130 – Propiedades y restricciones

Semántica

El concepto de **propiedades**, que puede unirse a los elementos del modelo se refleja en el **metamodelo** mediante las metaclases *PropertyDef* y *Property* (véase la figura 24). El **metamodelo** distingue entre una definición de propiedad y un valor de propiedad. Una *Property* contiene un valor representado por la cadena valor de atributo mientras que una *PropertyDef* contiene la especificación del **tipo de datos** correspondiente a dicho valor, heredada de la metaclase *Typed*. Por ejemplo un **tipo de CO** puede definir las propiedades necesarias para su configuración, mientras que un **CO** puede definir los valores de propiedad adecuados.

El concepto de **restricción** se refleja en el **metamodelo** mediante la metaclase *Constraint* y tiene dos atributos. El atributo *language* determina el lenguaje en el que viene escrita la restricción y debe utilizarse para la evaluación, mientras que el atributo *body* contiene la representación real de la cadena de caracteres de la restricción. La elección de un lenguaje para la restricción se deja a criterio del usuario, por lo que no se prescribe en esta Recomendación. De este modo, puede elegirse cualquier lenguaje apropiado, por lo que no se define aquí la semántica de las restricciones, sino que se deja para las herramientas de procesamiento. Un **CO** puede definir un conjunto de restricciones para expresar las combinaciones permitidas de valores de propiedad. Un **ensamblaje** puede definir restricciones de colocación de los componentes en ejecución. Las definiciones y los atributos de propiedad de las referencias se califican por sus propios nombres.

5.6 Conceptos del entorno objetivo

La especificación de la distribución y **despliegue** se efectúa mediante modelado de los entornos objetivo reales, de los entornos objetivo lógicos y mediante la conversión de las entidades lógicas en nodos reales de los entornos. En ciertos casos es posible recuperar automáticamente el entorno objetivo real mediante una herramienta.

5.6.1 Entorno objetivo, nodo y enlace de nodo

Metamodelo

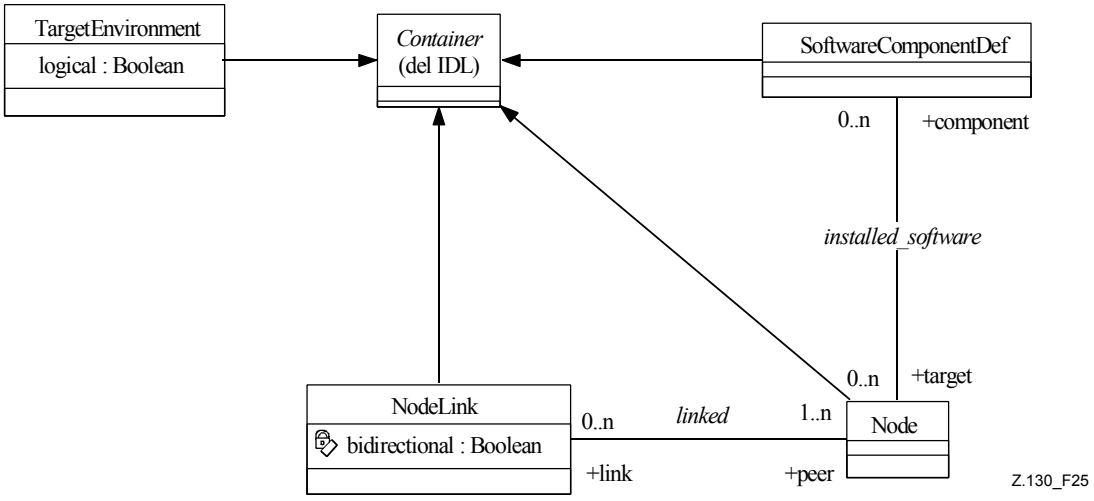


Figura 25/Z.130 – Entorno objetivo, nodo y enlace de nodo

Semántica

Un entorno objetivo modela un entorno físico distribuido en tiempo de ejecución, por ejemplo una red de telecomunicación consistente en nodos y los enlaces entre éstos. En el **metamodelo**, la metaclase *TargetEnvironment* es un contenedor de *Node* y *NodeLink* (véase la figura 25). El atributo *logical* de *TargetEnvironment* se utiliza para indicar si el modelo corresponde a un entorno existente o posible.

La metaclase *Node* representa el concepto de nodo, que se define como un elemento del entorno objetivo dotado de varios procesadores o al menos de uno solo, de una unidad de memoria y de un sistema operativo. Obsérvese que una determinada máquina física puede servir de anfitrión de varios *Node* lógicos. El nodo se refiere al soporte lógico instalado (**componentes de soporte lógico**, compiladores, intérpretes, etc.) y al soporte físico instalado (representado como soporte lógico controlador). Las propiedades tales como el sistema operativo o el procesador se describen como propiedades predefinidas y se relacionan a continuación.

El concepto de enlace entre nodos se presenta mediante la metaclase *NodeLink* como enlace físico entre dos o más *Nodes* (por ejemplo, un bus compartido). El atributo booleano *bidirectional* de *NodeLink* se aplica únicamente al caso en el que el *NodeLink* está asociado exactamente a dos *Nodes*. Cuando *bidirectional* es falso, el orden de los dos *Nodes* asociados al *NodeLink* se interpreta del siguiente modo: el primer ejemplar de *Node* corresponde al nodo origen mientras que el segundo representa el nodo de destino. Por consiguiente, la asociación *linked* está ordenada.

Un *Node* y un *NodeLink* son contenedores de *PropertyDef* y de *Property*. Esto significa que las propiedades predefinidas y las propiedades definidas por el usuario pueden aplicarse directamente a los ejemplares de nodo y a los ejemplares de enlace de nodo.

5.6.1.1 Propiedades predefinidas de los nodos y de los enlaces de nodo

Existen ciertas propiedades predefinidas para *Node* y para *NodeLink* (véase el cuadro 1). A tal efecto, se introducen los siguientes **tipos de datos** mediante IDL:

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

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

Cuadro 1/Z.130 – Propiedades predefinidas

Entidad	Nombre de la propiedad predefinida	Tipo	Descripción	Obligatoria
Node	Procesador	Tipo de procesador	Procesador del nodo	Sí
	Memoria	Entero	Máximo tamaño de la memoria del nodo (en kilobytes)	No
	OS	Tipo de OS	Identificación del sistema operativo del nodo	Sí
NodeLink	Anchura de banda	Entero	Máxima velocidad binaria del enlace (en kilobytes por segundo)	No

5.6.2 Mapa de instalación

Metamodelo

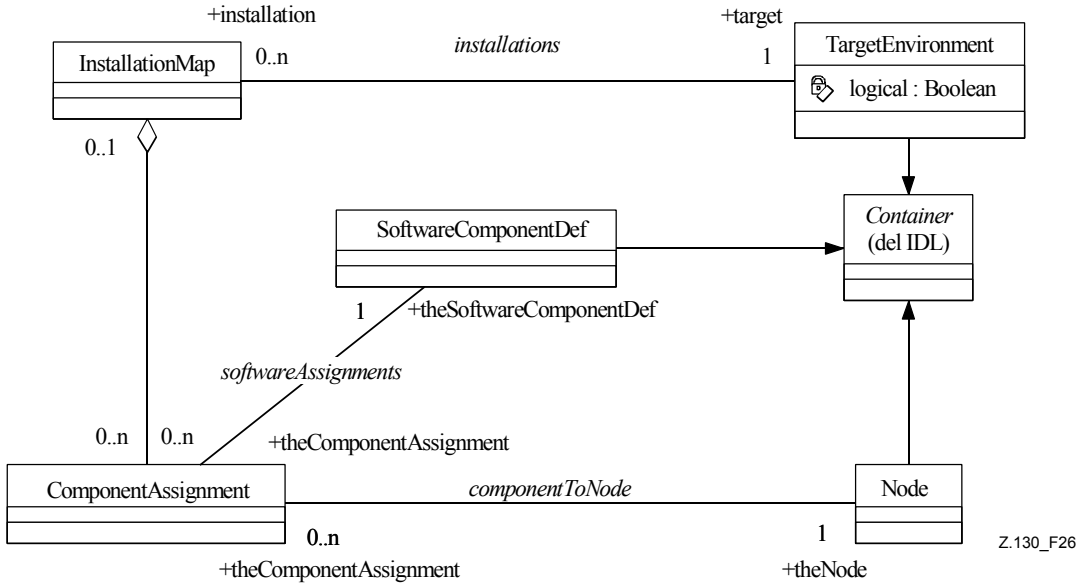


Figura 26/Z.130 – Mapa de instalación

Semántica

Una vez modelado un *TargetEnvironment* pueden asignarse las entidades adecuadas a sus nodos. Hay dos tipos de entidades: unidades de soporte lógico, que representan el soporte lógico necesario en un nodo, y conjuntos de **CO**, que representan ejemplares concretos de **tipos de CO**.

El mapa de instalación representa la manera en que se distribuyen las implementaciones en un entorno objetivo. Consta de un conjunto de asignaciones de instalación, en las que cada una de ellas asocia un **componente de soporte lógico** a un nodo. Un mapa de instalación se refiere a los nodos de un entorno objetivo. La representación del mapa de instalación es la metaclase *InstallationMap*.

La representación de la asignación de instalación es la metaclase *ComponentAssignment* (véase la figura 26).

5.6.3 Mapa de ejemplificación

Metamodelo

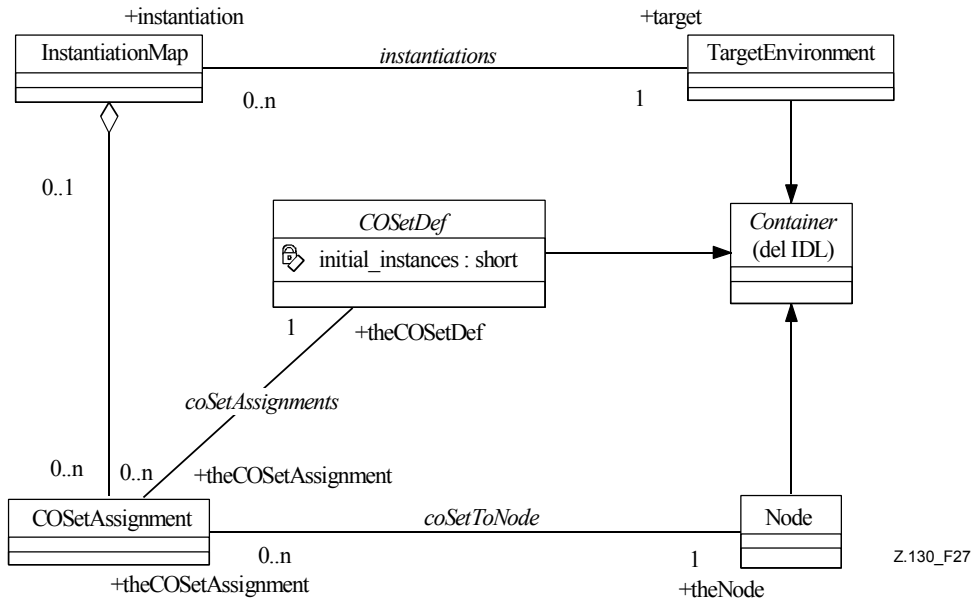


Figura 27/Z.130 – Mapa de ejemplificación

Semántica

El mapa de ejemplificación representa el modo en que los ejemplares concretos de los **tipos de CO** se distribuyen en un entorno objetivo. Consta de un conjunto de asignaciones de ejemplificación, que asocia un conjunto de **CO** a un nodo del entorno objetivo. Un mapa de ejemplificación se refiere a los **tipos de CO** definidos en el contexto de un **ensamblaje** y a los nodos de un entorno objetivo seleccionado. La metaclase que representa el concepto de mapa de ejemplificación es *InstantiationMap*. La asignación de **CO** se representa mediante la metaclase *COSetAssignment* (véase la figura 27).

5.6.4 Plan de despliegue

Metamodelo

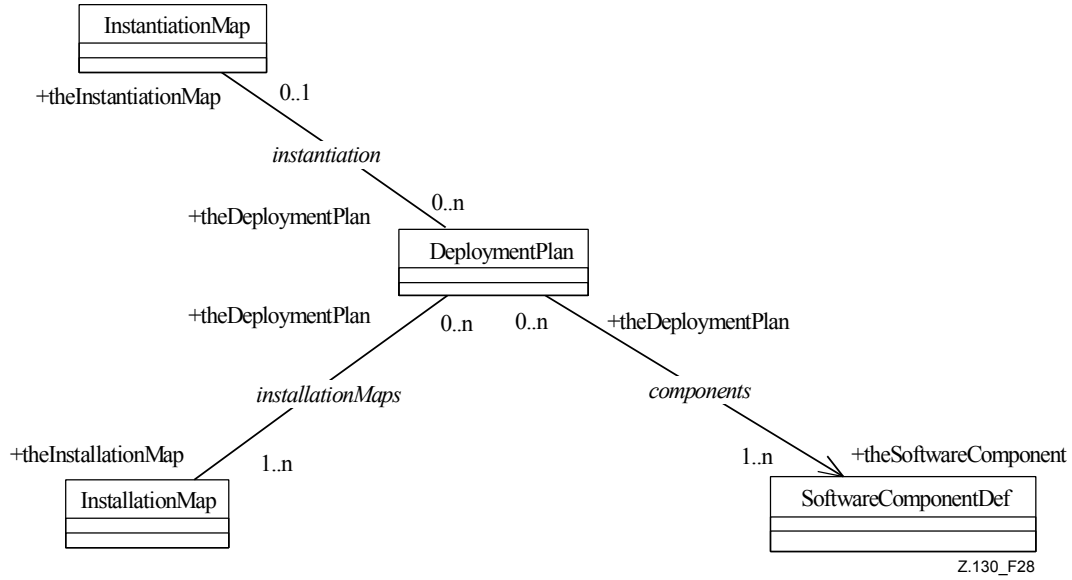


Figura 28/Z.130 – Plan de despliegue

Semántica

Un plan de despliegue se define mediante la selección de uno o más **componentes de soporte lógico**, conteniendo implementaciones correspondientes a **tipos de CO**, uno o más mapas de la instalación, que determinan dónde han de instalarse los **componentes de soporte lógico** y cero o un mapa de ejemplificación, que determinan dónde han de crearse los **CO** (véase la figura 28). El mapa de ejemplificación y todos los mapas de instalación de un modelo eODL han de referirse al mismo entorno objetivo y al mismo **montaje**.

6 Bibliografía

[11] OMG Document formal/00-03-02, *OMG Unified Modeling Language Specification*, Version 1.3.

[12] W3C Recommendation (2000), Extensible Markup Language (XML) 1.0 (Second edition).

[13] OMG Document ad/01-02-29, *UML Profile for MOF*.

[14] OMG Document omg/ 00-11-05, *Model Driven Architecture*.

[15] IEEE Std. 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*.

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

Anexo A

Sintaxis del eODL

A.1 Introducción

Este anexo proporciona la notación textual del eODL en estilo EBNF. La sintaxis de los conceptos cuyo origen es OMG CORBA IDL 2.4.2 procede de [5].

A.2 Convenios léxicos y base gramatical

En las cláusulas siguientes se aplican las definiciones de la cláusula 3.1 (convenios léxicos para el IDL) y 3.4 (gramática del IDL) del documento de especificación OMG CORBA 2.4.2.

A.3 Perspectiva computacional

A.3.1 Espacios de nombres, tipos de datos, excepciones, operaciones y atributos

El metamodelo se basa en el sistema de tipo de datos CORBA IDL, mientras que el lenguaje eODL se basa en CORBA IDL. Estos fundamentos proporcionan una correspondencia canónica de los tipos de datos metamodelos, **espacios de nombres** y **excepciones** con el eODL. Se establece una correspondencia igualmente canónica entre los **elementos de interacción operacionales** de los modelos conformes con CMM y las **operaciones** y atributos CORBA IDL.

A.3.2 Señales y valores transportados

Una de las ampliaciones del metamodelo en comparación con el fundamento conceptual del UIT-ODL es la introducción de **elementos de interacción de señal**. Estos **elementos de interacción** se basan en la definición de **señales**. Las **señales** del eODL se definen de acuerdo con la siguiente gramática:

```
<signal_dcl>      ::=   "signal" <identifier> "{" <member_list> "}"
<member_list>     ::=   <member>+
<member>          ::=   <type_spec> <declarators> ";"
```

A.3.3 Tipo de medios, medio y conjunto de medios

La semántica de las interacciones de los trenes queda abierta en el fundamento conceptual del UIT-ODL. El metamodelo define con exactitud la semántica de las interacciones de los **medios continuos** que sustituye a la interacción de trenes del UIT-ODL. La representación de los conceptos **medio, tipo de medio** y **conjunto de medios** definidos en el CMM en eODL viene dada por la siguiente gramática:

```
<mediaset_dcl>    ::=   "mediaset" <identifier> "{" <member_list> "}"
<mediatype_dcl>   ::=   "mediatype" <identifier>
<medium_dcl>      ::=   "medium" <identifier>  "(" <scoped_name> { "," <scoped_name> }* ")"
```

A.3.4 Tipos de interfaz y elementos de interacción

Un **tipo de interfaz** en el metamodelo combina los **elementos de interacción** de diferentes tipos de interacción en un único contexto de interacción. Para la representación sintáctica de este concepto, el elemento constructivo interfaz se amplía mediante los conceptos de **elementos de interacción**, **fuente**, **sumidero**, **producir** y **consumir** así como por los **elementos de interacción operacionales**.

```
<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 Tipo de objeto computacional

El UIT-ODL ya dispone del concepto de **tipo de objeto computacional (tipo de CO)**. Mediante la definición precisa de la perspectiva de configuración de un **tipo de CO**, se abandona el concepto de interfaz inicial. La gramática correspondiente a los **tipos de CO** del UIT-ODL se modifica en eODL, y viene definida por las siguientes reglas:

```
<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 Propiedad

La propiedad se utiliza para definir las propiedades disponibles para los elementos del modelo o necesitadas por éstos. El concepto de propiedad se utiliza para el entorno objetivo y para la definición de las unidades de soporte lógico.

```
<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 Tipo externo

El tipo externo se utiliza para referirse a un tipo de datos suministrado externamente, mediante un identificador.

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

A.4 Perspectiva de configuración

A.4.1 Puertos

El principal concepto de la perspectiva de configuración de los **CO** es el de **puerto**. La notación correspondiente a los **puertos** único y dinámico se define mediante las siguientes reglas:

```
<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 Perspectiva a la implementación

A.5.1 Artefactos y elementos de implementación

Los **artefactos** abstractos proceden de elementos constructivos de lenguajes de programación concretos que implementan el comportamiento de los **CO**. La representación de los **artefactos** y de los **elementos de implementación** en el eODL viene dada por las siguientes reglas:

```
<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 Relaciones implementa y políticas de ejemplificación

La relación **implemented_by** define qué **artefacto** se utiliza para la realización de un comportamiento de los **tipos de CO**. Esta relación se define en el eODL en el contexto de las definiciones del **tipo de CO** del siguiente modo:

```
<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 Tipos de estado

En muchos casos de implementación la realización de un **artefacto** necesita información del estado de acceso de los **CO** implementados. La información de estado de los **CO** se define en los **tipos de CO** de acuerdo con las siguientes reglas:

```
<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 Perspectiva del despliegue

A.6.1 Componentes de soporte lógico

La implementación concreta de un **tipo de CO** se representa mediante su definición de unidad de soporte lógico. En una definición de unidad de soporte lógico pueden **realizarse** varios **tipos de CO**. Las unidades de soporte lógico pueden depender a su vez de otras unidades de soporte lógico y requerir otras propiedades o servicios del entorno final de ejecución.

```
<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 Ensamblaje

El **ensamblaje** describe el conjunto de componentes interconectados y no tiene relación con una distribución concreta en un entorno de procesamiento distribuido.

A.6.2.1 Definición del ensamblaje

La definición del **ensamblaje** contiene definiciones para todos los conjuntos de ejemplares que pertenecen a las definiciones de **ensamblaje** y **conexión** correspondientes a los ejemplares del **ensamblaje**.

```
<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 Definición de conjunto de ejemplares

La definición del conjunto de ejemplares describe un conjunto no vacío de ejemplares de un **tipo de CO**.

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

A.6.2.3 Definición de conexión

Las **conexiones** entre ejemplares y conjuntos de ejemplares se expresan mediante la definición de conexión. En éstas se interconectan los **puertos** con arreglo a la definición de **tipo de CO**, comportándose un **puerto** como **fuente** y el otro como **sumidero**.

```
<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 Definición del mapa de instalación

La definición del mapa de instalación describe la asignación de los **tipos de CO** a los nodos de un entorno objetivo. En una actuación posterior de instalación, es posible referirse a la definición del mapa de instalación y desencadenar la instalación de unidades de soporte lógico en los **nodos**.

```
<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 Definición del mapa de ejemplificación

La definición del mapa de ejemplificación describe una asignación concreta de conjuntos de ejemplares a **nodos** del entorno objetivo especificado y del **ensamblaje**.

```
<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 Acción de despliegue

Una acción de despliegue es una secuencia de acciones de instalación y de ejemplificación a ejecutar durante el **despliegue**.

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

A.6.5.1 Acción de instalación

La acción de instalación especifica la instalación de una unidad de soporte lógico en un nodo de ejecución de un entorno objetivo.

```
<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 Acción de ejemplificación

La acción de ejemplificación especifica la ejemplificación de un conjunto de **CO** en un nodo de ejecución de un entorno objetivo.

```
<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 Entorno objetivo

El entorno objetivo sirve de posible entorno de ejecución para los **ensamblajes**, reflejando la estructura y propiedades de dicho entorno. Una sintaxis textual eODL puede contener más de una especificación de entorno objetivo.

A.7.1 Definición de entorno

La definición de entorno describe el posible entorno de ejecución en términos de **enlaces** de comunicación y **nodos** disponibles.

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

A.7.2 Definición de nodo

La definición de nodo refleja un **nodo** de ejecución identificable en un entorno objetivo, que puede estar destinado a la instalación de **tipos de CO** y a la ejemplificación de conjuntos de ejemplares. Las propiedades de la definición del nodo caracterizan las facilidades del nodo de ejecución.

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

A.7.3 Definición de enlace

Los enlaces de comunicación entre los nodos de ejecución del entorno objetivo se representan como definiciones de enlace. Las propiedades de la definición de enlace están relacionadas con las características y tipo del enlace de comunicación.

```
<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 Sintaxis del eODL

Esta cláusula proporciona el conjunto completo de reglas de producción del eODL. Se incluyen asimismo todas las reglas heredadas de la sintaxis de base de 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>
```

Anexo B

Metamodelo de correspondencia intáctica

B.1 Introducción

Este anexo describe la relación entre el **metamodelo** eODL y la sintaxis textual concreta del eODL definida en el anexo A. La descripción se restringe a los conceptos del **metamodelo** que son ampliaciones del **metamodelo** CORBA. La relación entre la sintaxis textual OMG IDL 2.4.2 y el **metamodelo** OMG CORBA está perfectamente definida en el OMG y por consiguiente no se reproduce en este contexto.

El **metamodelo** se ajusta a la especificación de la cláusula 5 y utiliza la notación gráfica contenida en el mismo. La sintaxis alfanumérica correspondiente se indica en la casilla que hay debajo del gráfico, seguida de una explicación textual.

B.2 Señal y parámetro de señal

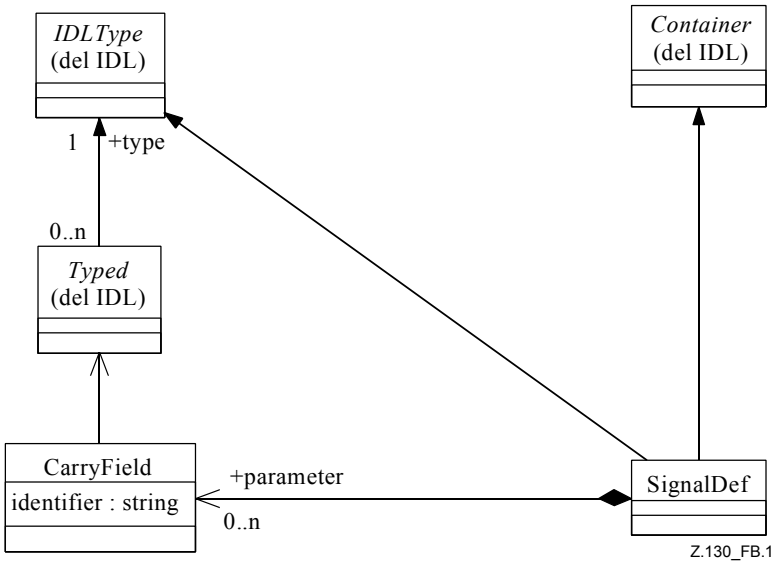


Figura B.1/Z.130 – Señal y parámetro de señal

(1), **(2)** y **(3)** de la sintaxis concreta corresponden a los elementos *SignalDef/CarryField* del modelo (véase la figura B.1).

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

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

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

El **<identifier>** de producción **(1)** es el nombre del elemento *SignalDef* del modelo. En **<member_list> (2)** se listan todos los elementos *CarryField* que intervienen en la relación **parámetro** de dicha *SignalDef*. Las producciones **<member> (3)** de la sintaxis concreta se reflejan como elementos *CarryField* en el modelo. **<type_spec>** en este contexto corresponde a los tipos del modelo vinculados mediante el concepto Typed de IDL. Para cada declarador de **<declarators>** existe un elemento *CarryField* en el modelo.

B.3 Tipo de medio, medio y conjunto de medios

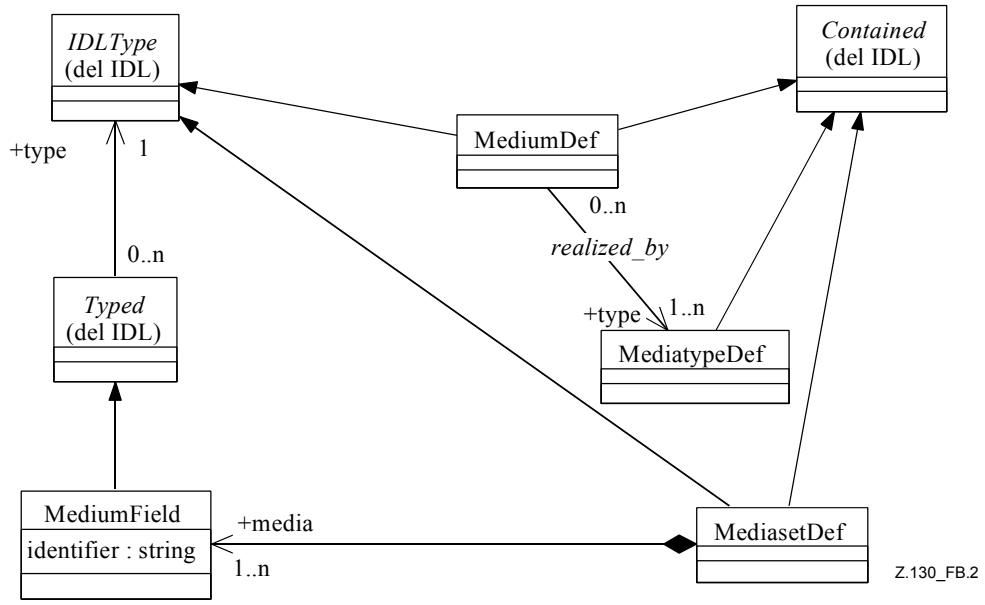


Figura B.2/Z.130 – Medio, tipo de medios y conjunto de medios

Las producciones **(4)**, **(5)** y **(6)** de la sintaxis concreta se corresponden con los elementos *MediasetDef, MediatypeDef* y *MediumDef* del modelo (véase la figura B.2).

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

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

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

El **<mediatype_dcl> (5)** se representa como elemento *MediatypeDef* en el modelo, **<identifier>** es el nombre del concepto denominado de este elemento. Con **<medium_dcl>** la sintaxis concreta expresa elementos *MediumDef*, en este contexto **<identifier>** es una vez más el nombre del concepto denominado. El **<scoped_name>** listado en la producción **(6)** ha de referirse siempre a *MediatypeDef* y se representa en el modelo como una relación *realized_by* con el elemento *MediumDef*. De acuerdo con la producción **(4) <mediaset_dcl>** se representa como elemento *MediasetDef* en el modelo, siendo su nombre **<identifier>**. En la **<member_list>** de **(4)** se listan todos los elementos *MediumField*, que toman parte en la relación de medios de dicho *MediasetDef*. Para el **<member>** de **<member_list> (4)** el **<type_spec>** ha de referirse al *MediatypeDef* y todos los declaradores de **<declarators>** deben ser sencillos.

B.4 Consumir y producir

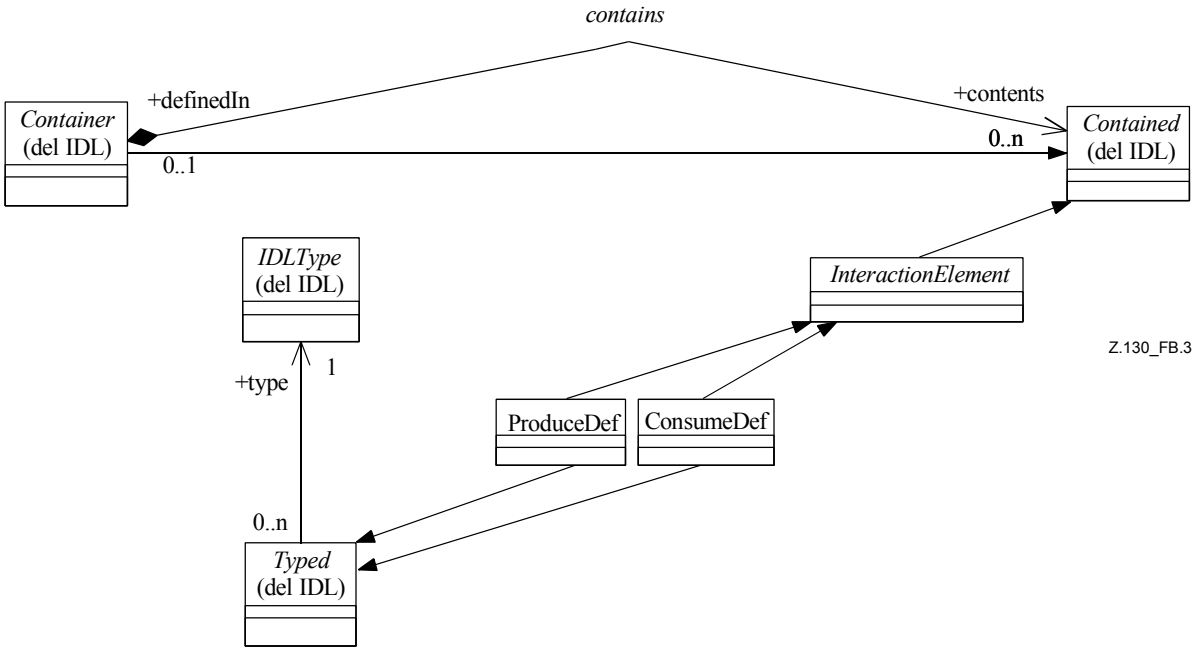


Figura B.3/Z.130 – Consumir y producir

Las producciones entre **(7)** y **(8)** de la sintaxis concreta se corresponden con los elementos *ProduceDef/ConsumeDef* del modelo (véase la figura B.3).

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

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

El **<scoped_name>** de ambas declaraciones se refiere a una *SignalDef*. Esta relación se refleja en el modelo como relación *Typed/IDLType* en la que se ven afectados *ProduceDef/ConsumeDef* mediante herencia y el *IDLType* es una *SignalDef* de acuerdo con la **<signal_dcl>**. *ProduceDef/ConsumeDef* son ambos conceptos con nombre del **metamodelo** y el **<identifier>** de **(7)**/**(8)** se corresponde con el atributo o nombre de los elementos del modelo.

B.5 Sumidero y fuente

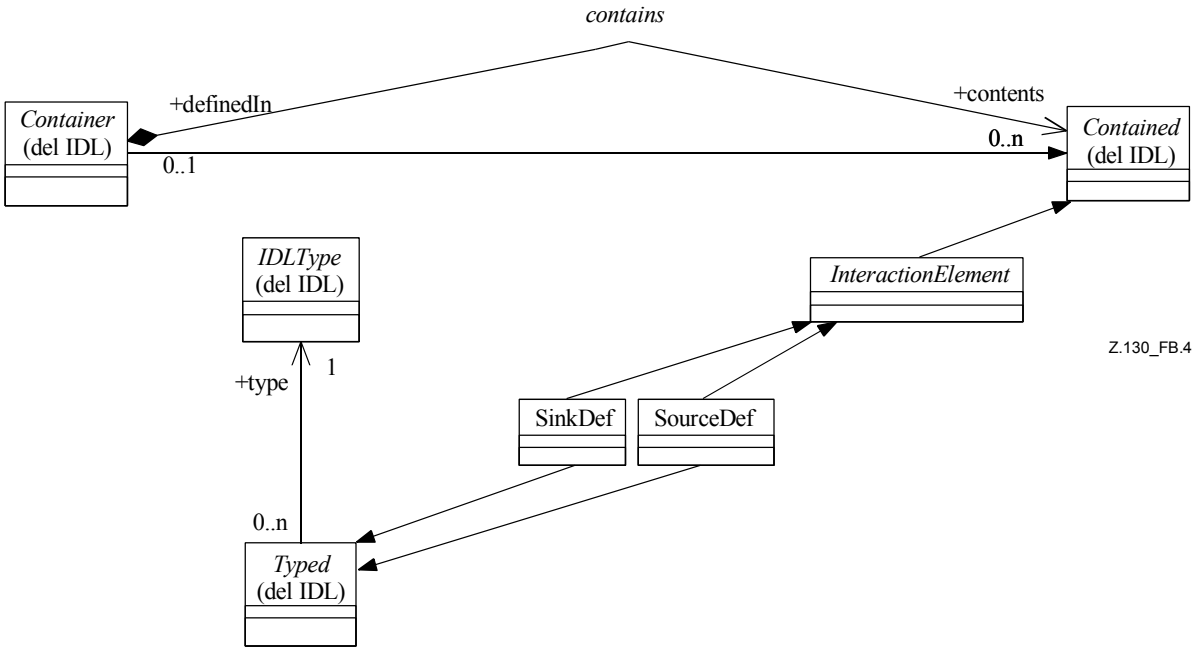


Figura B.4/Z.130 – Sumidero y fuente

Las producciones **(10)** y **(9)** de la sintaxis concreta se corresponden con los elementos *SinkDef/SourceDef* del modelo (véase la figura B.4).

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

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

El **<scoped_name>** de ambas declaraciones se refiere a una *MediasetDef*. Esta relación se refleja en el modelo como la relación *Typed/IDLType* en la que se ven implicados *SinkDef/SourceDef* por herencia y el *IDLType* es una *MediasetDef* de acuerdo con la **<mediaset_dcl>**. *SinkDef/SourceDef* son ambos conceptos con nombre del **metamodelo** y el **<identifier>** en **(10)/(9)** se corresponde con el atributo nombre de los elementos del modelo.

B.6 Tipo de interfaz

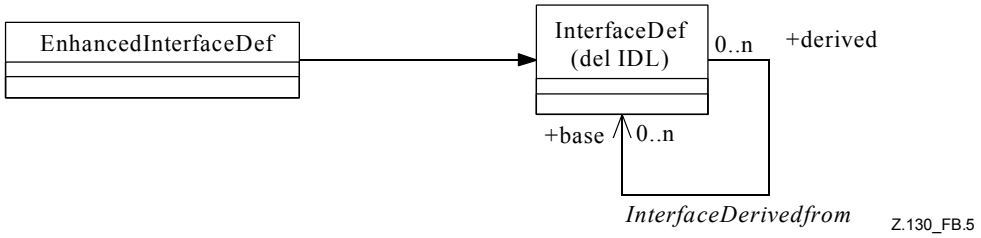


Figura B.5/Z.130 – Tipo de interfaz

(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> ";"**

La producción **<interface_dcl> (11)** en la sintaxis concreta se corresponde con el elemento *EnhancedInterfaceDef* del modelo. **<interface_body> (12)** se maneja del mismo modo que en IDL. Al igual que *InterfaceDef* **<export> (13)** admite también **<produce_dcl>**, **<consume_dcl>**, **<source_dcl>** y **<sink_dcl>** como elementos contenidos. La correspondencia para este tipo de declaraciones se define *supra*. Si el **<interface_body>** no contiene este tipo de nuevos elementos la **<interface_dcl>** se corresponde con una *InterfaceDef ordinaria* (véase la figura B.5).

B.7 Tipos de CO, soporta y requiere

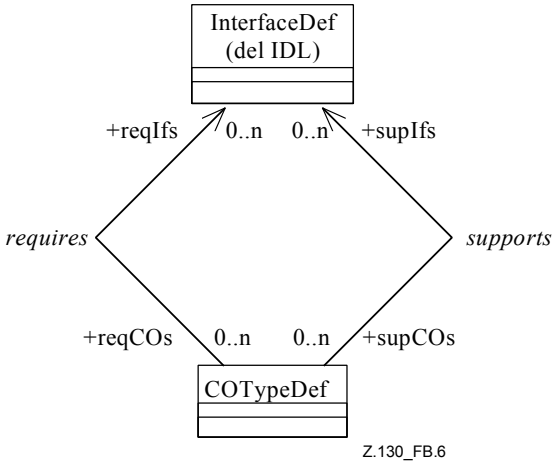


Figura B.6/Z.130 – Soporta y requiere

Las producciones **(14)**, **(15)** y **(16)** de la sintaxis concreta se corresponden con los elementos *COTypeDef* del modelo (véase la figura 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> }*

me> { "," <scoped_name> }*

Las producciones **(14)**, **(15)** y **(16)** se corresponden con un elemento *COTypeDef* del modelo, en el que **<identifier>** de la producción **(15)** es el nombre del concepto denominado de *COTypeDef*. Como particularización de la *InterfaceDef* de la producción **(16)** se listan todas las *COTypeDef* que se encuentran en una relación de herencia con la *COTypeDef actual*. Las producciones **(17)** y **(18)** expresan el concepto contenido correspondiente a esta *COTypeDef*. La utilización de **<export>** en **(18)** se maneja del mismo modo que en la *InterfaceDef* en IDL. Además, **<reqrd_interf_templates>** y **<supd_interf_templates>** son elementos contenidos admitidos para *COTypeDef*. El **<scoped_name>** en la producción **(19)** y **(20)** ha de referirse a los elementos *InterfaceDef* o *EnhancedInterfaceDef* del modelo. Estos elementos de interfaz se encuentran en la relación **requiere** o **soporta** con el elemento *COTypeDef* continente.

B.8 Puerto suministrado y puerto utilizado

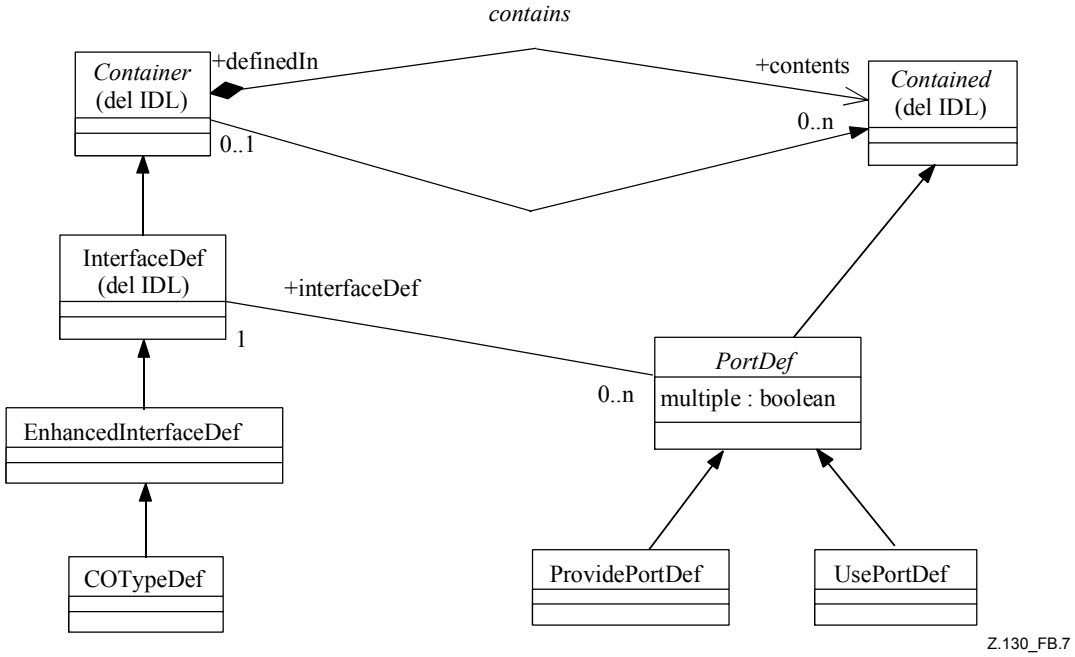


Figura B.7/Z.130 – Puerto suministrado y puerto utilizado

Las producciones **(21)/(22)** de la sintaxis concreta se corresponden con los elementos *UsePortDef/ProvidePortDef* del modelo (véase la figura B.7).

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

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

Los **puertos utilizados** y **suministrados** se expresan mediante las producciones **(21)** y **(22)**. Se traducen en los elementos *UsedPortDef* y *ProvidePortDef del modelo*. El **<scoped_name>** de ambas producciones ha de referirse a los elementos *InterfaceDef* o *EnhancedInterfaceDef* del modelo. Si se utiliza "multiple" en la sintaxis concreta, el campo booleano múltiple en la *PortDef* actual es verdadero.

B.9 Diagrama de ejemplificación y artefactos

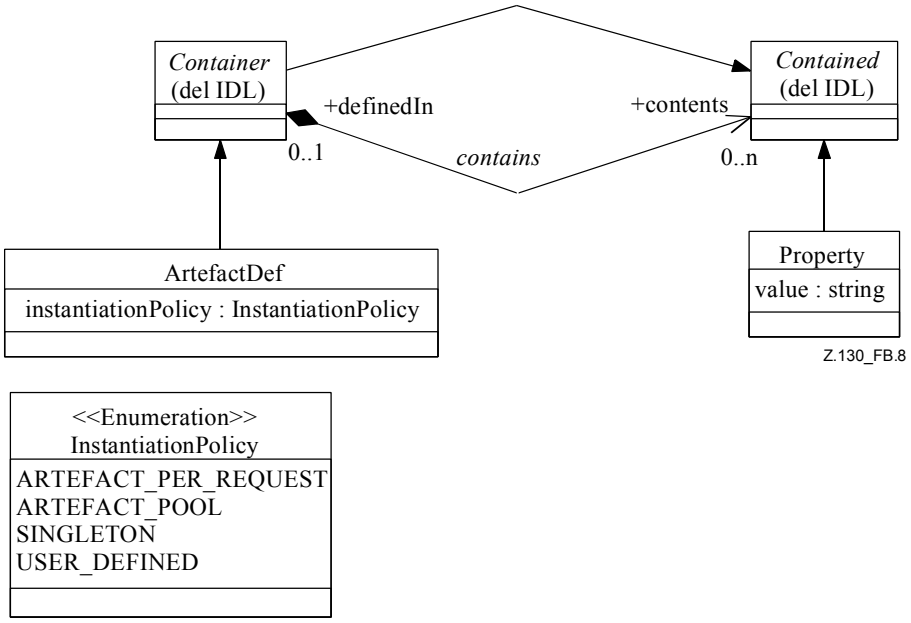


Figura B.8/Z.130 – Diagrama de ejemplificación y artefactos

(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>***

Las producciones **(23)** y **(24)** se utilizan para continuar la sintaxis del IDL, donde el **<identifier>** utilizado en **(24)** tiene una *ArtefactDef* siguiente con este nombre. Las producciones **(25)**, **(26)** y **(27)** están relacionadas con un elemento *ArtefactDef* del modelo, donde **<identifier>** de **(26)** es el nombre del concepto denominado. Todos los **<scoped_name>** listados en **(27)** han de referirse a los elementos *ArtefactDef* del modelo que se encuentran en una relación de herencia con el *ArtefactDef* actual. Como puede apreciarse en **(28)**, sólo se permite al elemento *ImplementationElementDef* estar contenido en *ArtefactDef* (véase la figura B.8).

B.10 Relación implementa

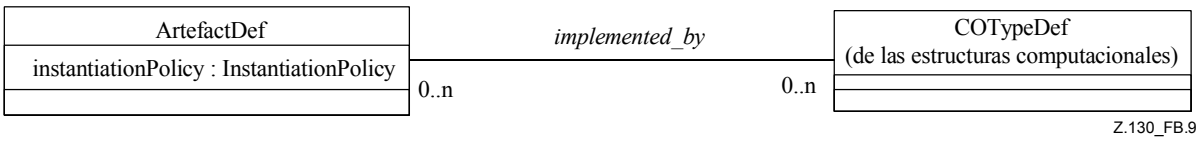


Figura B.9/Z.130 – Relación implementa

(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"**

Las producciones **(29)** y **(30)** están relacionadas con el elemento *ArtifactDef* del modelo (véase la figura B.9). Esto expresa la relación *implemented_by* en el modelo. El **<scoped_name>** de **(30)** ha de referirse sólo a un elemento *ArtefactDef*. Con la producción **(31)** de la sintaxis concreta, se expresa el campo *instantiationPolicy* del elemento *ArtefactDef* continente. Las palabras clave se relacionan directamente con los valores de enumeración correspondientes al campo.

B.11 Elemento implementación

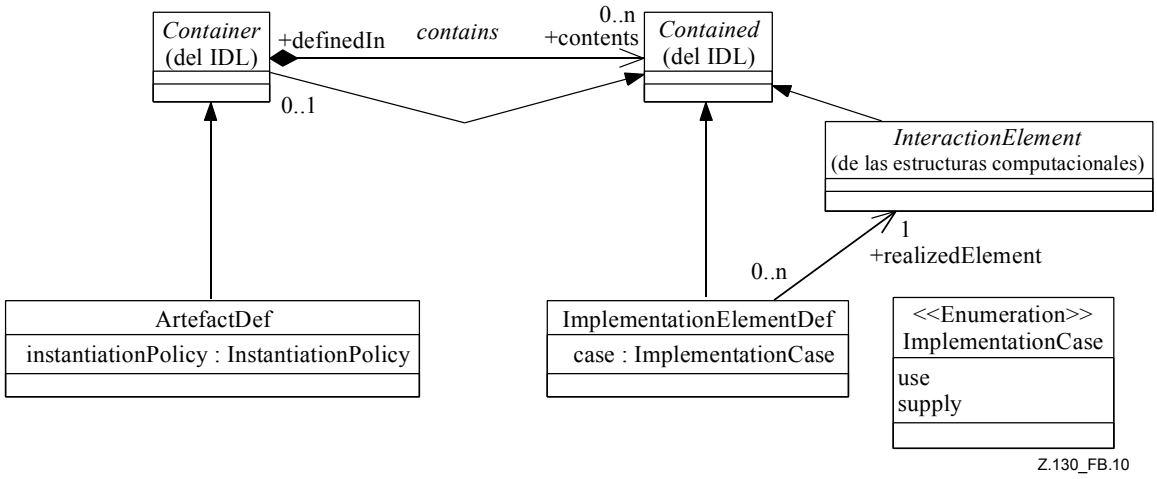


Figura B.10/Z.130 – Elemento implementación

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

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

Las producciones **(32)** y **(33)** están relacionadas con el elemento *ImplementationElementDef* del modelo (véase la figura B.10). El **<identifier>** de **(32)** es el nombre correspondiente al concepto denominado. *ImplementationElementDef y* sólo puede estar contenido en *ArtefactDef*. Con la producción **(33)** de la sintaxis concreta se expresa el campo *case* del elemento *ImplementationElementDef* continente. Las palabras clave se relacionan directamente con los valores de enumeración correspondientes al campo.

B.12 Componente de soporte lógico

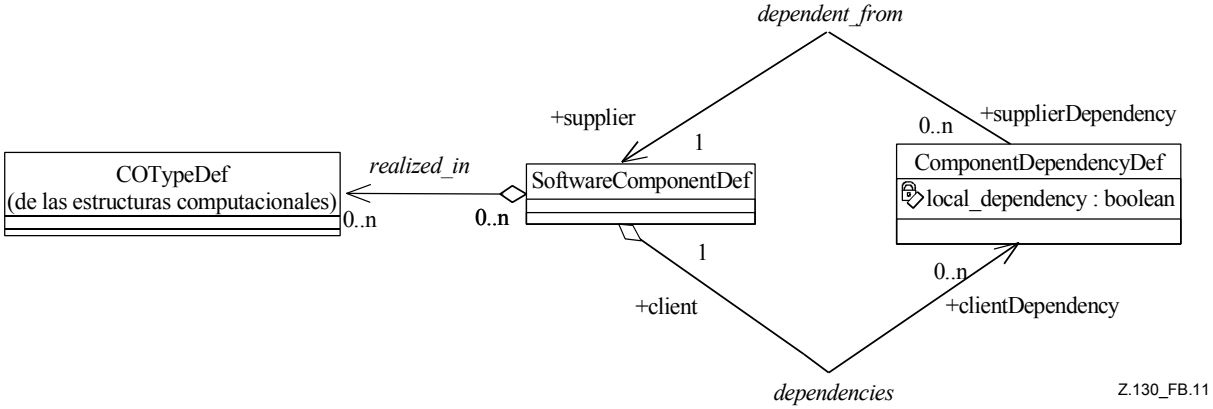


Figura B.11/Z.130 – Componente de soporte lógico

La producción **(34)** de la sintaxis concreta se corresponde con el elemento *SoftwareComponentDef* del modelo (véase la figura 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> }*

Las producciones **(34)**, **(35)** y **(36)** están relacionadas con el elemento *SoftwareComponentDef* del modelo, donde **<identifier>** de **(35)** es el nombre del concepto denominado. Todos los **<scoped_name>** listados en **(36)** han de referirse a elementos *COTypeDef* del modelo, que se encuentran en una relación *realized_in* con el *SoftwareComponentDef* actual. Las producciones **(37)**, **(38)** y **(39)** están relacionadas con un elemento *SoftwareComponentDef* del modelo. Todos los **<scoped_name>** de **(36)** han de referirse a elementos *SoftwareComponentDef* del modelo.

El **<softwarecomponent_identifier>** es un **<scoped_name>** que únicamente se refiere al elemento del modelo *SoftwareDependencyDef*.

B.13 Ensamblaje y configuración inicial

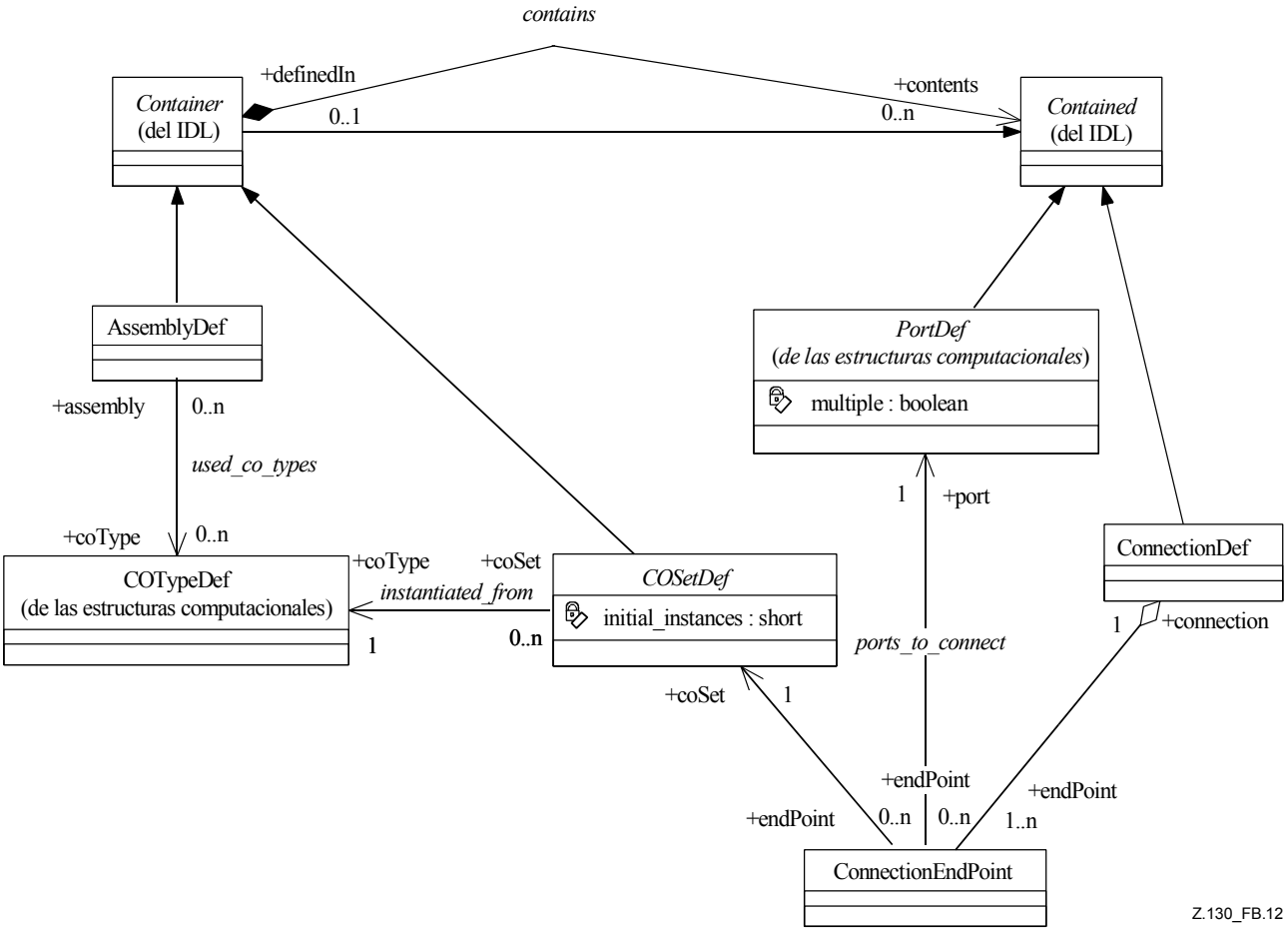


Figura B.12/Z.130 – Ensamblaje y configuración inicial

(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>

Las producciones **(40)** y **(41)** están relacionadas con un elemento *AssemblyDef* del modelo (véase la figura B.12), donde **<identifier>** de **(41)** es el nombre correspondiente al concepto denominado. Todos los **<scoped_name>** listados en **(44)** han de referirse a elementos *COTypeDef* del modelo, que guardan una relación *realized_in* con la *SoftwareComponentDef actual*. El **<cotype_identifier>**, el **<instance_set_identifier>** y el **<port_identifier>** de **(45)**, **(46)** y **(47)** son **<scoped_name>** que se refieren sólo a los elementos del modelo *COTypeDef*, *InstanceSetDef* y *PortDef*.

B.14 Restricciones y propiedades

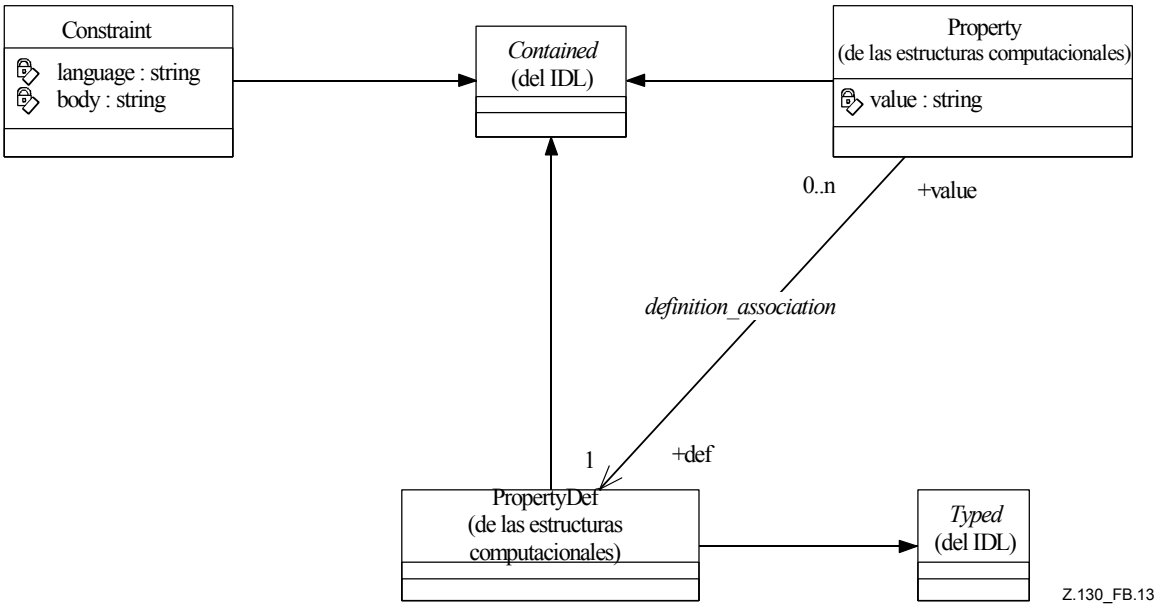


Figura B.13/Z.130 – Restricciones y propiedades

(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> ";"**

Las producciones **(48)** y **(49)** están relacionadas con un elemento *PropertyDef* del modelo (véase la figura B.13), donde **<identifier>** de **(49)** es el nombre correspondiente al concepto denominado. La producción **(54)** se corresponde con un elemento *Property* del modelo. Las producciones **(50)**, **(51)**, **(52)** y **(53)** de la sintaxis textual concreta se utilizan para anotar valores correspondientes al valor *field*.

La producción **(55)** se relaciona con el elemento *Constraint* del modelo, donde **<identifier>** de **(55)** es el nombre correspondiente al concepto denominado. La producción **(56)** proporciona los valores correspondientes a los campos *language* y *body* del elemento *Constraint*.

B.15 Entorno objetivo, nodo y enlace de nodo

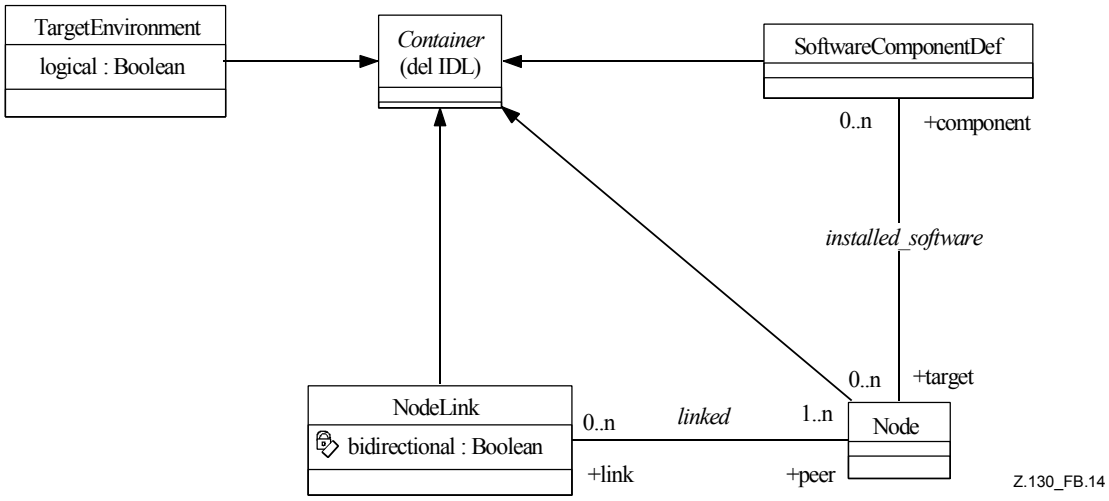


Figura B.14/Z.130 – Entorno objetivo, nodo y enlace de nodo

(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> }+**

Las producciones **(57)**, **(58)**, **(59)** y **(60)** están relacionadas con un elemento *TargetEnvironment* del modelo (véase la figura B.14), donde el **<identifier>** de la producción **(58)** es el nombre correspondiente al concepto denominado. La producción **(61)** se corresponde con un elemento *Node* del modelo, donde el **<identifier>** de la producción **(61)** es el nombre correspondiente al concepto denominado. Las producciones **(62)**, **(63)** y **(64)** están relacionadas con un elemento *NodeLink* del modelo, donde el **<identifier>** de la producción **(63)** es el nombre correspondiente al concepto denominado. El **<node_identifier>** de la producción **(65)** es un **<scoped_name>** que ha de referirse a un elemento *Node* del modelo.

B.16 Mapa de instalación

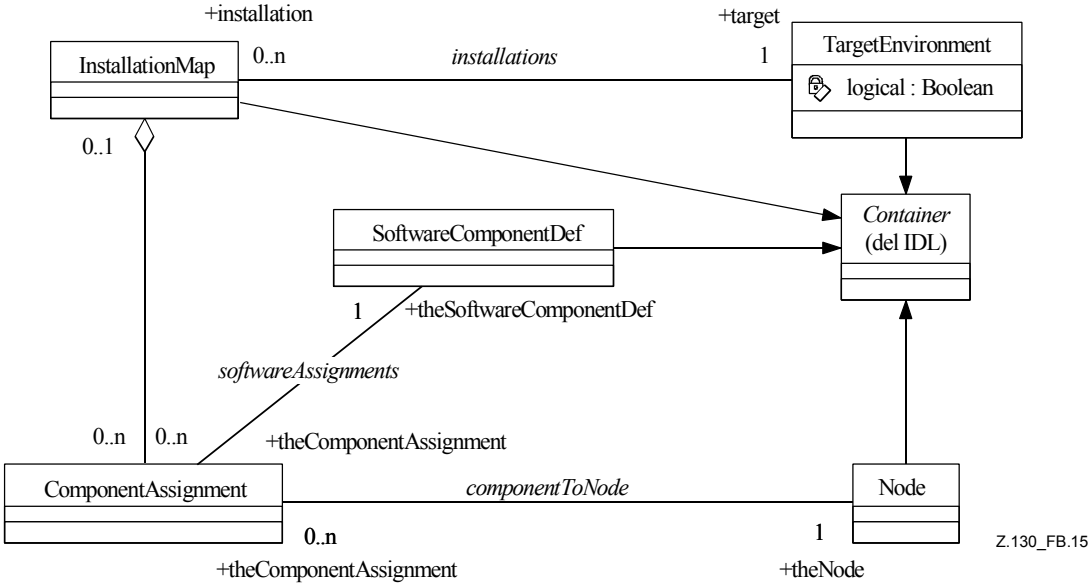


Figura B.15/Z.130 – Mapa de instalación

(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> ";"**

Las producciones **(66)**, **(67)** y **(68)** están relacionadas con un elemento *InstallationMap* del modelo (véase la figura B.15), donde el **<identifier>** de la producción **(67)** es el nombre correspondiente al concepto denominado. El **<environment_identifier>** de la producción **(67)** es un **<scoped_name>**, que se refiere únicamente al elemento *TargetEnvironment* del modelo. El **<softwarecomponent_identifier>** y el **<node_identifier>** de la producción **(69)** son **<scoped_name>** que se refieren únicamente a los elementos *SoftwareComponentDef* y *Node* del modelo.

B.17 Mapa de ejemplificación

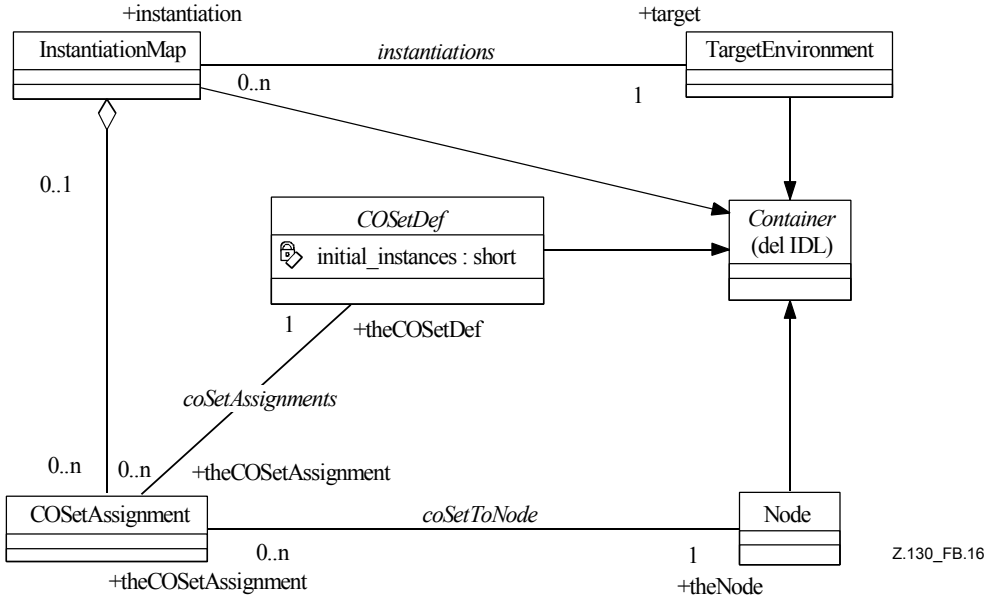


Figura B.16/Z.130 – Mapa de ejemplificación

(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> }*

Las producciones **(70)**, **(71)**, **(72)**, **(73)** y **(74)** están relacionadas con un elemento *InstantiationMap* del modelo (véase la figura B.16), donde el **<identifier>** de la producción **(71)** es el nombre correspondiente al concepto denominado. El **<environment_identifier>** de la producción **(72)** y el **<assembly_identifier>** de la producción **(73)** son **<scoped_name>** y se refieren únicamente a los elementos *TargetEnvironment* y *AssemblyDef* del modelo. El **<node_identifier>** de la producción **(75)** es un **<scoped_name>** que se refiere únicamente al elemento del modelo *Node* que está contenido en el *TargetEnvironment* calificado por la producción **(72)**. Los **<instance_set_identifier>** de la producción **(76)** son **<scoped_name>** que se refieren únicamente a los elementos del modelo *COSetDef* que están contenidos en la *AssemblyDef* calificada por la producción **(73)**.

B.18 Plan de despliegue

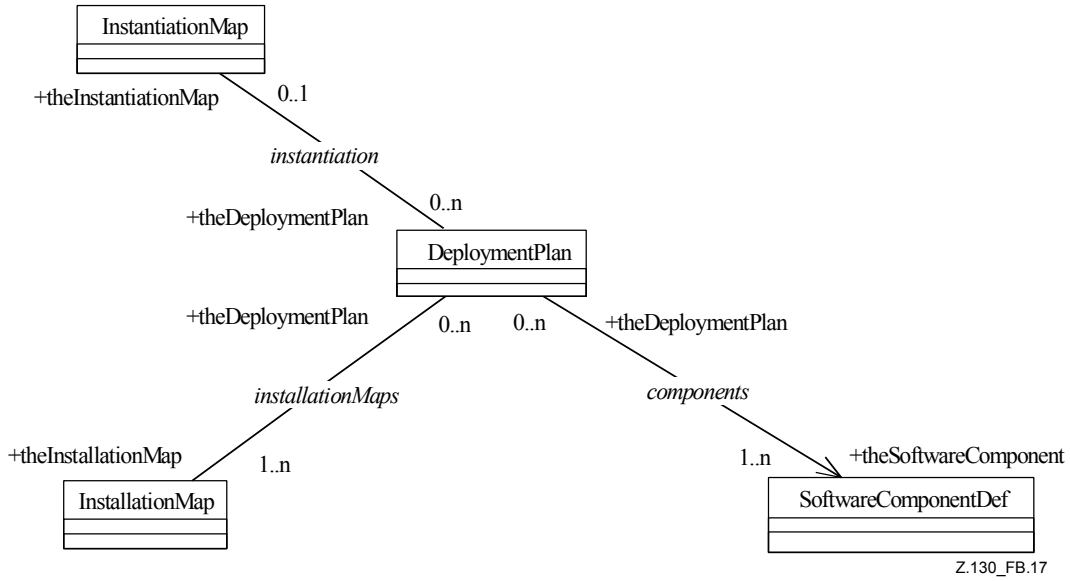


Figura B.17/Z.130 – Plan de despliegue

(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> ";"**

La producción **(77) <deployment_action>** de la sintaxis concreta se corresponde con el elemento del modelo *DeploymentPlan* (véase la figura B.17). Las listas del cuerpo de la **<deployment_action>,** construidas por las producciones **(78)**, **(79)**, **(80)** y **(81)**, expresan las relaciones del *InstallationMap* y las de ejemplificación en el modelo. El **<instantiation_map_identifier>** y el **<installation_map_identifier>** son **<scoped_name>** que se refieren a los elementos del modelo *InstallationMap* e *InstantiationMap*. Cada uno de los identificadores listados corresponde a una relación en el modelo.

B.19 Tipo externo

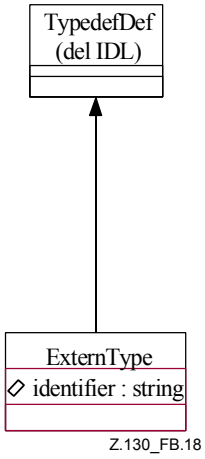


Figura B.18/Z.130 – Tipo externo

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

La producción **(82) <extern_type>** de la sintaxis concreta se corresponde con el elemento del modelo *ExternType* (véase la figura B.18). El valor del literal de la cadena de caracteres se corresponde directamente con el identificador de atributo del elemento del modelo. El **<identifier>** de la producción **(82)** se corresponde con el atributo de nombre del concepto *Contained*.

Anexo C

Correspondencia con el SDL-2000

C.1 Introducción

El lenguaje de definición de objetos ampliado del UIT (UIT-eODL) recomendado ofrece la posibilidad de describir sistemas distribuidos orientados a componentes. En este anexo, se introduce una correspondencia entre el UIT-eODL y el SDL-2000. Esta correspondencia permite generar automáticamente código SDL-2000 a partir de una descripción elaborada en UIT-eODL. Se utiliza la representación de frase textual (SDL/PR) de SDL-2000 [8].

La correspondencia entre el UIT-eODL y el SDL-2000 permite a los usuarios generar automáticamente un esqueleto SDL-2000 a partir de un determinado modelo eODL. Esta correspondencia soporta casi todos los conceptos computacionales y de implementación. La única **excepción** importante es que no se soporta el concepto de **medios continuos**. Además, tampoco se soportan los conceptos de **despliegue** y entorno objetivo.

Los tipos de eODL se corresponden con los tipos adecuados de SDL-2000. Los conceptos que definen aspectos del comportamiento de los tipos se corresponden con agentes SDL implementados automáticamente. Dado un modelo eODL completo, la herramienta de correspondencia genera un esqueleto SDL-2000. El usuario tiene que implementar la lógica de la empresa y es capaz de utilizar **CO** definidos en SDL-2000 como elementos constructivos del sistema SDL-2000.

La correspondencia pretende soportar la mayor parte de los conceptos posible incluso a expensas de producir estructuras de SDL-2000 un tanto complejas. Por ejemplo, se soporta la herencia múltiple de **CO** a expensas de estructuras más complejas y de un comportamiento menos eficiente, porque SDL-2000 no soporta herencias múltiples para los tipos de agente.

C.2 El lote eodl

La correspondencia entre el UIT-eODL y el SDL-2000 define el lote eODL de SDL. Contiene definiciones de **tipos de datos** utilizados por los modelos generados a partir de modelos eODL. Los tipos referidos como predefinidos se definen en el lote eODL. En C.10 puede consultarse el listado completo del lote eODL.

La notación textual del eODL hereda sus reglas léxicas de OMG CORBA IDL. En CORBA IDL, los identificadores no calificados comienzan con un carácter alfabético seguido de cualquier número de caracteres alfabéticos, cifras o subrayados, siendo los caracteres alfabéticos las letras inglesas 'A' a 'Z' tanto mayúsculas como minúsculas. Además, los identificadores en eODL no distinguen entre mayúsculas y minúsculas.

De acuerdo con las reglas léxicas del SDL-2000, todos los identificadores eODL (no calificados) son identificadores SDL-2000.

Los identificadores calificados se estudian en la cláusula C.4 (Nombres con ámbito).

C.3 Estructura

Cada fichero UIT-eODL se corresponde con dos paquetes SDL:

- `<name>_interface` (denominado "lote de la interfaz") y
- `<name>_definition` (denominado "lote de definición")

siendo `<name>` el nombre de la especificación UIT-eODL (por ejemplo, el nombre del fichero), representado en la figura C.1.

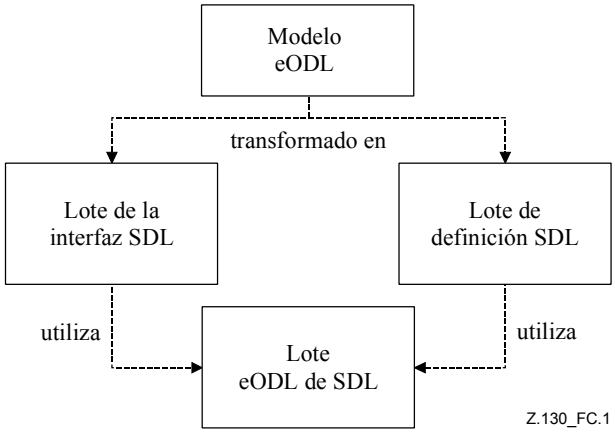


Figura C.1/Z.130 – Estructura de transformación de un modelo eODL en SDL

El lote de la interfaz contiene toda la información pertinente tanto al lado del cliente como al servidor del sistema. El detalle de la misma es el siguiente:

- definiciones de tipos de datos;
- definiciones de constantes; y
- definiciones de interfaz representando **interfaces** ordinarias e **interfaces** de **CO**.

El lote de definición contiene esqueletos para los lados servidor del sistema. A saber:

- tipos de bloque representando **tipos de CO**; y
- tipos de procesos representando **artefactos**.

C.4 Nombres con ámbito

La calificación es un concepto que existe tanto en eODL como en SDL-2000. Por consiguiente la correspondencia es canónica. Los nombres calificados en eODL se corresponden con nombres calificados en SDL.

C.5 Correspondencia de conceptos computacionales

C.5.1 Módulos

Un módulo eODL es un contenedor para todos los demás elementos eODL y abre un espacio de nombres. Este concepto se corresponde con el concepto de lote en SDL.

El lote SDL al que corresponde un módulo eODL puede estar contenido ya sea en el lote de interfaz SDL o en el lote de definición SDL o en ambos lotes, dependiendo de las entidades incluidas en el módulo eODL.

C.5.2 Definiciones de tipo

El elemento constructivo `typedef` de eODL asigna un nombre (diferente) a un tipo determinado. El elemento constructivo `typedef` se corresponde con el elemento constructivo `syntype` de SDL.

C.5.3 Tipos de datos predefinidos

C.5.3.1 Tipos de datos para números enteros

Los tipos de datos `unsigned short` (precisión corta sin signo), `unsigned long` (precisión larga sin signo) y `unsigned long long` (doble precisión larga sin signo) de eODL se corresponden con los géneros de SDL que se indican a continuación. Estos tipos se definen en el lote `eodl`.

- El tipo `unsigned short` de eODL se corresponde con un género `Integer ushort` que va de 0 a $2^{16} – 1$.
- El tipo `unsigned long` de eODL se corresponde con un género `Integer ulong` que va de 0 a $2^{32} – 1$.
- El tipo `unsigned long long` de eODL se corresponde con un género `Integer ulong_long` que va de 0 a $2^{64} – 1$.

Los tipos de datos `signed short` (precisión corta con signo), `signed long` (precisión larga con signo) y `signed long long` (doble precisión larga con signo) de eODL se corresponden con los siguientes géneros de SDL.

- El tipo `signed short` de eODL se corresponde con un género `Integer short` que va de $–2^{15}$ a $2^{15} – 1$.
- El tipo `signed long` de eODL se corresponde con un género `Integer long` que va de $–2^{31}$ a $2^{31} – 1$.
- El tipo `signed long long` de eODL se corresponde con un género `Integer long_long` que va de $–2^{63}$ a $2^{63} – 1$.

C.5.3.2 Tipos de datos correspondientes a los números en coma flotante

Los tipos en coma flotante `float`, `double` y `long double` de eODL se corresponden con el género predefinido de SDL `Real`. Obsérvese que `Real` no se ajusta a IEEE 754 [15] con arreglo a la Rec. UIT-T Z.100 mientras que los tipos en coma flotante de eODL sí se ajustan.

C.5.3.3 Tipos de datos correspondientes a los caracteres

El tipo `char` de eODL se corresponde con el género predefinido de SDL `Character`. El tipo `wchar` de eODL se corresponde con el género predefinido de SDL `Natural`.

C.5.3.4 Tipo de datos booleano

El tipo `boolean` de eODL se corresponde con el género SDL `Boolean`, mientras que las constantes `boolean` de eODL `TRUE` y `FALSE` se corresponden con los literales `Boolean true` y `false`, respectivamente.

C.5.3.5 Tipo de datos octeto

El tipo `octet` de eODL se corresponde con el género de SDL `Octet`.

C.5.3.6 Tipo de datos cualquiera

El tipo `any` de eODL se corresponde con el género de SDL `Any`. Cabe observar que la semántica de `any` en eODL coincide con la semántica de any en SDL (véase OMG IDL 2.4.2 sección 3.10.1.7).

C.5.3.7 Identificación del tipo por medio del atributo de tipo TypeCode

El atributo `typeCode` de un ejemplar de *IDLType* en el **metamodelo** de eODL no tiene correspondencia en SDL. No obstante, el tipo `TypeCode` se corresponde con el género SDL `TypeCode` en el paquete SDL. El tipo `TypeCode` es un tipo de datos abstracto, lo que significa que no hay valores de este tipo de datos según esta correspondencia. Esto no impide, sin embargo, que las implementaciones utilicen un tipo concreto de datos derivado.

C.5.4 Tipos de datos construidos

C.5.4.1 Enumeraciones

Las enumeraciones en eODL se corresponden con un tipo de valor de SDL que contiene únicamente letras.

C.5.4.2 Estructuras

Las estructuras eODL se corresponden con tipos de datos de valor de SDL, con un constructor de tipos de datos de estructura pública. Los miembros de la estructura eODL son campos del tipo de datos SDL.

C.5.4.3 Uniones

Las uniones eODL se corresponden con tipos de datos de valores anidados SDL. El tipo de datos de valor exterior en SDL tiene un constructor de tipos de datos de estructura pública con dos campos:

1) Un campo `tag` (rótulo) que representa el discriminador de la unión eODL y

2) un campo `union` que representa la propia unión.

El campo `union` es del tipo de datos de valor `<name-of-eODL-union>_union`. Este tipo se declara en el ámbito del tipo de unión SDL exterior. Tiene un constructor de tipo de datos de opción pública y su lista de opciones representa los miembros de la unión eODL.

C.5.4.4 Matrices

Las matrices eODL se corresponden con un género de SDL que se hereda del género predefinido de SDL `Vector`.

Las matrices multidimensionales se corresponden con un tipo de datos de valor SDL que soporta los operadores `Make`, `Modify` y `Extract`.

C.5.5 Tipos de valor

Los tipos de valor eODL se corresponden con tipos de datos de objeto SDL con un constructor de tipos de datos de estructura. Todos los miembros de datos del tipo valor eODL son campos de los tipos de datos SDL.

Si un tipo de valor eODL se declara abstracto, los datos de objeto SDL correspondiente también se declaran abstractos. No tiene constructor de tipo de datos.

La herencia sencilla de tipos de valor eODL se corresponde con la herencia sencilla de tipo de datos en SDL. La herencia múltiple se permite para los tipos de valor abstracto en eODL. En SDL esto se efectúa mediante copia de la declaración de operación de los tipos de datos de base a los tipos de datos derivados.

Los tipos de valor de eODL pueden tener fábricas (elementos de inicialización). Éstas se corresponden con los operadores de SDL que devuelven un valor del tipo de datos. Si se declara exactamente una fábrica, el operador `Make` se implementa automáticamente llamando a esta fábrica. De lo contrario, no se implementa exactamente.

Si el tipo de valor eODL soporta una interfaz, las **operaciones** de dicha interfaz se convierten en operaciones del tipo de datos SDL. Los atributos en una interfaz soportada se corresponden con un par de operaciones get/set y un campo que contiene el atributo.

Los tipos de valor encasillados se corresponden del mismo modo que los tipos de valor (concreto).

C.5.6 Tipos de datos parametrizados

C.5.6.1 Secuencias

Las secuencias en eODL se corresponden con el género SDL `Vector` si están limitadas o con el género SDL `Array` en caso contrario.

C.5.6.2 Cadenas de caracteres

El tipo `string` de eODL se corresponde con el género SDL predefinido `Charstring`.

El tipo `wstring` de eODL se corresponde con el género SDL `wstring` en el lote `eodl`. Este tipo se hereda de `String<Natural>`. Se añade asimismo una operación de `Charstring` para convertir un valor `Charstring` en un valor `wstring`.

C.5.6.3 Números en coma fija

El paquete SDL `eodl` contiene un tipo `fixedpt` determinado por un **parámetro** de anchura y otro de escala. Ambos son del tipo `Natural`. La semántica de los parámetros es idéntica a la semántica de los parámetros en eODL.

El tipo `fixedpt` define los operadores "+","–","*","/" que permiten sumar, restar, multiplicar y dividir dos valores del tipo `fixedpt` y devuelven un valor del tipo `fixedpt`. Además, el operador "=" compara valores `fixedpt` y devuelve un valor `Boolean`: verdadero, cuando ambos operandos representan el mismo número o falso cuando ambos operandos representan números distintos.

Para convertir un valor `fixedpt` en un valor `Real`, existe el método `toReal`. Es posible construir un valor `fixedpt` a partir de un valor `Real` por medio del operador `Make`.

C.5.7 Constantes, literales de tipo de datos y expresiones constantes

C.5.7.1 Constantes

Las constantes en eODL se corresponden con los sinónimos en SDL.

C.5.7.2 Literales de tipo de datos

En esta cláusula se enumeran los literales de tipos de datos que no se corresponden con géneros SDL predefinidos.

C.5.7.2.1 Literales de tipos enteros

Los literales enteros decimal y hexadecimal se soportan en la correspondencia SDL: los literales en decimal se soportan mediante el género `Integer` de SDL mientras que los literales enteros hexadecimales pueden escribirse como literales del género `Bitstring` de SDL y convertirse a continuación en `Integer`. Los literales enteros en octal han de convertirse a literales decimales o bien a literales hexadecimales.

C.5.7.2.2 Literales de tipo carácter

Para un literal del tipo `char` de eODL se establece una correspondencia tal que forme un literal legal del género SDL predefinido `Character`. Si el literal eODL es un carácter cuyo valor es mayor que 127, no es posible establecer esta correspondencia. En este caso se recomienda utilizar el género `wchar`.

Para un literal del tipo `wchar` de eODL se establece una correspondencia tal que forme un literal `Natural` legal.

C.5.7.2.3 Literales de cadenas de caracteres

Para un literal del tipo `string` de eODL se establece una correspondencia tal que forme un literal legal del género SDL predefinido `Charstring`. Si el literal eODL contiene un carácter con un valor

superior a 127, no es posible establecer esta correspondencia. En tal caso se recomienda utilizar el género `wstring`.

Para un literal del tipo `wstring` de eODL se establece una correspondencia tal que forme un literal legal del género SDL `wchar`. Esto incluye la posibilidad de convertir un literal `Charstring` en un valor SDL `wstring`.

C.5.7.2.4 Literales de tipo de número de coma fija

Los valores de coma fija sólo pueden crearse convirtiéndolos en números reales.

C.5.7.3 Expresiones constantes

Las expresiones constantes en eODL se corresponden con expresiones constantes en SDL. Ambas expresiones tienen que representar los mismos valores.

C.5.8 Tipos de señal

Los tipos de señal se corresponden con tipos de datos de valor.

C.5.9 Excepciones

Tanto el UIT-eODL como el SDL soportan **excepciones**. La única diferencia es que las **excepciones** SDL no nombran miembros de **excepción**. Las **excepciones** se definen en el lote de interfaz.

C.5.10 Interfaces y elementos de interacción

C.5.10.1 Interfaces

Una interfaz de eODL agrupa:

- **operaciones**,
- flujos de **señal**,
- interacciones de **medios continuos** (trenes),
- atributos.

Las interacciones de **medios continuos** no tienen correspondencia en SDL.

Una interfaz `I` de eODL se corresponde con las interfaces SDL `exported_I` e `imported_I`. Estas interfaces se definen en el paquete SDL `I` del paquete interfaz. La interfaz `exported_I` contiene:

- todas las **operaciones**; y
- las **señales** consumidas.

mientras que la interfaz `imported_I` contiene las **señales** producidas.

Esta correspondencia es tal que el tipo de interfaz `exported_<interface-name>` contiene todo lo que necesita el cliente de la interfaz para invocar un servicio representado por la interfaz.

En las siguientes cláusulas se detalla esta correspondencia.

C.5.10.2 Elementos de interacción operacionales

Una **operación** eODL se corresponde con un procedimiento remoto que se declara en SDL en la interfaz `exported_<interface-name>`. Ambos conceptos soportan como máximo un tipo de retorno, parámetros que pueden ser de entrada, salida y de entrada-salida así como **excepciones**. No se establece correspondencia alguna para los ejemplares del concepto "context".

C.5.10.3 Elementos de interacción de señal

Si una interfaz eODL declara un **elemento de interacción** para el consumo de una **señal** `A`, la interfaz SDL correspondiente a `exported_<interface-name>` declara el uso de la señal `A`.

Si una interfaz eODL declara un **elemento de interacción** para la producción de una **señal** `A`, la interfaz SDL correspondiente `imported_<interface-name>` declara el uso de la señal `A`.

C.5.10.4 Elementos de interacción de atributos

Los atributos en eODL se corresponden con un par de procedimientos distantes set/get de la interfaz SDL `exported_<interface-name>`. Si el atributo es de sólo lectura no se genera el procedimiento distante set.

C.5.10.5 Referencias de interfaz

Un tipo de interfaz SDL define implícitamente un tipo especial de `PId`. Un ejemplar de este tipo de `PId` sirve de referencia de la interfaz. Por otra parte, este concepto proporciona seguridad de tipo: un cliente que tenga dicho `PId` puede enviar señales a un agente que implemente esta interfaz e invocar procedimientos distantes en el mismo. El tipo de interfaz SDL que sirve como tipo de referencia de interfaz es `exported_<interface-name>`.

C.5.10.6 Herencia

Tanto el UIT-eODL como el SDL soportan varias herencias de interfaz. Por este motivo la correspondencia es canónica.

C.5.11 Objetos computacionales

En eODL, los **CO** llevan encapsulado el estado y el comportamiento. Proporcionan interfaces (tales como referencias de interfaz) al entorno y pueden utilizar, a su vez, interfaces (referencias de interfaz) de otros **CO**.

En SDL, un **CO** es un agente de tipo de proceso. Los **CO** tienen tres puertas: una de entrada denominada `initial`, otra denominada `provides` y una tercera denominada `uses`. Estas puertas soportan varias interfaces como se verá en las siguientes cláusulas. Las tres puertas se conectan al entorno del proceso porque constituyen la interfaz entre el **CO** y el entorno.

El propio tipo de proceso **CO** se define en el lote de definición.

Para crear procesos de **CO**, se define una fábrica de **CO**. La fábrica de **CO** se representa como tipo de proceso definido en el mismo ámbito que el tipo de proceso del **CO**. Tanto el propio **tipo de CO** como su fábrica son tipos de procesos definidos dentro de un tipo de agente del género bloque. Este bloque se denomina `SDL component`. Obsérvese que el `SDL component` definido en esta correspondencia es un concepto de la perspectiva computacional mientras que el *component* definido en esta Recomendación es un concepto de la perspectiva **despliegue**. Un componente SDL representa un tipo de bloque con una interfaz perfectamente definida.

C.5.11.1 La interfaz de un CO

Además de proporcionar y utilizar interfaces, un **tipo de CO** expone su propia interfaz. Esta interfaz está constituida por tres componentes: la interfaz definida por el usuario, el componente implícito que identifica a la interfaz y la interfaz de configuración.

El **metamodelo** define un **tipo de CO** como género de interfaz y restringe los **elementos de interacción** contenidos a ejemplares de *AttributeDef*. Cuando el **tipo de CO** se considera únicamente como interfaz, se denomina "interfaz definida por el usuario".

Para identificar **CO**, cada **tipo de CO** tiene una clave de atributo implícito de sólo lectura del tipo predefinido `ComponentKey`. Este atributo es el único **elemento de interacción** en la interfaz predefinida `ComponentBase`.

La interfaz `ComponentBase` define para el procedimiento de acceso, la clave de un **CO**. La clave de un **CO** implementa su identidad y necesita ser única, al menos con respecto al componente SDL en el que está contenido el **CO**. En el lote `eodl` está declarado el procedimiento externo

`compute_co_key` que devuelve dicha clave única. Una implementación de esta correspondencia tiene que proporcionar una implementación de dicho procedimiento o bien generar código que calcule la clave única de otro modo.

La interfaz de configuración define procedimientos para soportar las operaciones del **puerto**. Esta interfaz se explica con más detalle en la cláusula relativa a la correspondencia de puertos. La interfaz de configuración se hereda de la interfaz predefinida `ConfigBase`. `ConfigBase` declara operaciones para **puertos** genéricos.

Todas estas interfaces se definen en el lote de interfaz y están contenidas en un lote que tiene el mismo nombre que el **CO**. La interfaz definida por el usuario se denomina `<CO-name>_attributes` y se hereda de la interfaz predefinida `ComponentBase`. La interfaz de configuración se denomina `<CO-name>_config` y se hereda de la interfaz predefinida `ComponentBase`. Por último, se define la interfaz `<CO-name>` que se hereda sencillamente de `<CO-name>_attributes` y de `<CO-name>_config`.

La interfaz `<CO-name>` es soportada por la puerta `initial`. Se define un canal que va del entorno al tipo de proceso del **CO**.

C.5.11.2 Interfaces soportadas

La puerta `provides` referencia todas las interfaces que **soporta** el **CO**. A través de esta puerta, los clientes pueden utilizar el **CO**. La interfaz SDL `exported_<name>` se soporta en el sentido del entorno al tipo de proceso, mientras que la interfaz SDL `imported_<name>` se soporta en el sentido del tipo de proceso al entorno.

C.5.11.3 Interfaces requeridas

La puerta `uses` referencia todas las interfaces que **requiere** y utiliza el **CO**. A través de esta puerta, el **CO** (actuando como cliente) puede utilizar otros **CO**. La interfaz SDL `imported_<name>` se soporta en el sentido del entorno al tipo de proceso mientras que la interfaz SDL `exported_<name>` se soporta en el sentido del tipo de proceso al entorno.

C.5.11.4 Herencia

Dado que el SDL no soporta varias herencias, la herencia se realiza por delegación. Esto se lleva a cabo del siguiente modo:

1) La interfaz definida por el usuario del **tipo de CO** y la interfaz de configuración del **tipo de CO** se heredan de las interfaces correspondientes de los súper **CO**.

2) La puerta `provides` soporta todas las interfaces que están soportadas por los súper **CO**. Esta puerta soporta asimismo todas las interfaces soportadas por el propio **tipo de CO**.

3) La puerta `uses` soporta todas las interfaces requeridas por los súper **CO**. Soporta asimismo todas las interfaces requeridas por el propio **tipo de CO**.

Sobre los aspectos comportamentales de la herencia múltiple, véase C.7.2.5.

C.5.11.5 Fábricas de CO

Para cada **tipo de CO** hay una fábrica de **CO** asociada `<CO-name>_factory` que implementa la interfaz de fábrica. Esta interfaz se define en un lote `<CO-name>_factory` que definido a su vez en el mismo lote en el que se definen las interfaces del **CO**. La interfaz de fábrica hereda la interfaz predefinida `CoFactoryBase`.

La interfaz `CoFactoryBase` contiene los siguientes procedimientos. El procedimiento `get_co_type` devuelve el nombre completamente calificado del **tipo de CO**. El carácter de calificación es el punto decimal. El procedimiento `generic_create` ejemplifica el **tipo de CO** asociado. El procedimiento `list_cos` devuelve una lista de valores de `ComponentKey` de los **CO**

ejemplificados. El procedimiento `resolve_co` toma un valor `ComponentKey` y devuelve el **CO** asociado.

La interfaz de fábrica declara un procedimiento `create_<CO-name>` que crea el **tipo de CO** asociado. El agente de fábrica tiene una puerta `factory` que soporta la interfaz `CoFactoryBase` en sentido entrante.

C.5.11.6 Encapsulación del tipo de CO y de la fábrica del CO – componente SDL

Tanto el **tipo de CO** como el tipo de fábrica de **CO** se definen en un tipo de bloque `<CO-name>_CO`. Este tipo de bloque se denomina componente SDL. En cada componente SDL hay un conjunto de ejemplares `factory` del tipo `<CO-name>_factory` así como un conjunto de ejemplares `cos` del tipo `<CO-name>`. El conjunto de ejemplares `factory` contiene exactamente un ejemplar. El conjunto de ejemplares `cos` no contiene en principio ningún ejemplar no habiendo restricción alguna sobre el número máximo de ejemplares. Las puertas `initial`, `provides` y `uses` del **tipo de CO** se duplican por el componente SDL así como la puerta `factory` del tipo de fábrica y estás puertas se conectan mediante canales.

El componente SDL se define en el lote de definición y representa un **tipo de CO**.

C.6 Correlación de los conceptos de la perspectiva de configuración

C.6.1 Puertos suministrados

El concepto de **puertos suministrados** en eODL es un mecanismo que permite distribuir las referencias de interfaz que son proporcionadas por un **CO** a los clientes de dicho **CO**.

Este concepto se corresponde con un conjunto de procedimientos remotos declarados en la interfaz de configuración del **CO**.

Un **puerto suministrado** `foo` del tipo `bar` se corresponde con un procedimiento distante `provide_foo` que devuelve una referencia (`PId`) a `bar`. Si `foo` es del atributo **sencillo**, se requiere que cada llamada a `provide_foo` devuelva el mismo `PId`. Si `foo` es del atributo **múltiple**, el propio usuario tiene que implementar la semántica (véase C.7.4.3 sobre gestión de **puertos**).

La interfaz de configuración se hereda de la interfaz predefinida `ConfigBase`. Esta interfaz declara el procedimiento `provide`. Este procedimiento tiene como argumento una cadena de caracteres. El **parámetro** real en la llamada de procedimiento designa un **puerto**. Si este **puerto** existe, se devuelve una referencia en forma de `PId`. Si el **puerto** no existe, se provoca una **excepción** `NoSuchPort`. La **excepción** `NoSuchPort` es predefinida.

C.6.2 Puertos utilizados

El concepto de **puerto utilizado** en eODL es un mecanismo que permite a un **CO** almacenar referencias de interfaz de otros **CO**.

Este concepto se corresponde con un conjunto de procedimientos distantes declarados en la interfaz de configuración del **CO**.

Un **puerto utilizado** `foo` del tipo `bar` se corresponde con un procedimiento distante `link_foo` que toma una referencia a `bar` como **parámetro**. Si el **puerto** es del atributo **sencillo** es que ya hay una referencia almacenada en dicho **puerto** y se provoca la **excepción** predefinida `AlreadyConnected`. Por otra parte, se declara un procedimiento distante `unlink_foo` que suprime la referencia almacenada del **puerto** `foo`. De no haber ninguna referencia almacenada en `foo`, se provoca la **excepción** predefinida `NotConnected`. Si el **puerto** es del atributo **múltiple**, se almacena una secuencia de referencias. La **excepción** `AlreadyConnected` no se provoca nunca.

La interfaz predefinida `ConfigBase` de la que se hereda la interfaz de configuración, declara un procedimiento `link` y un procedimiento `unlink`. Estos procedimientos pueden utilizarse de una

manera genérica para almacenar y suprimir una referencia en un **puerto utilizado**. Como sus homólogos en los **puertos suministrados**, toman un argumento de cadena de caracteres que designa el nombre del **puerto**. Una vez más, si el **puerto** del nombre designado no existe, se provoca la **excepción** `NoSuchPort`. El procedimiento de conexión genérico toma un `PId` como segundo argumento y lo almacena en el **puerto** designado si es posible, o bien provoca un `AlreadyConnected` en caso contrario. Se utiliza la misma semántica que en el procedimiento de conexión específico del **puerto** para decidir si es posible almacenar la referencia. El procedimiento genérico de desconexión provoca la **excepción** `NotConnected` cuando no hay referencia almacenada en el **puerto** designado.

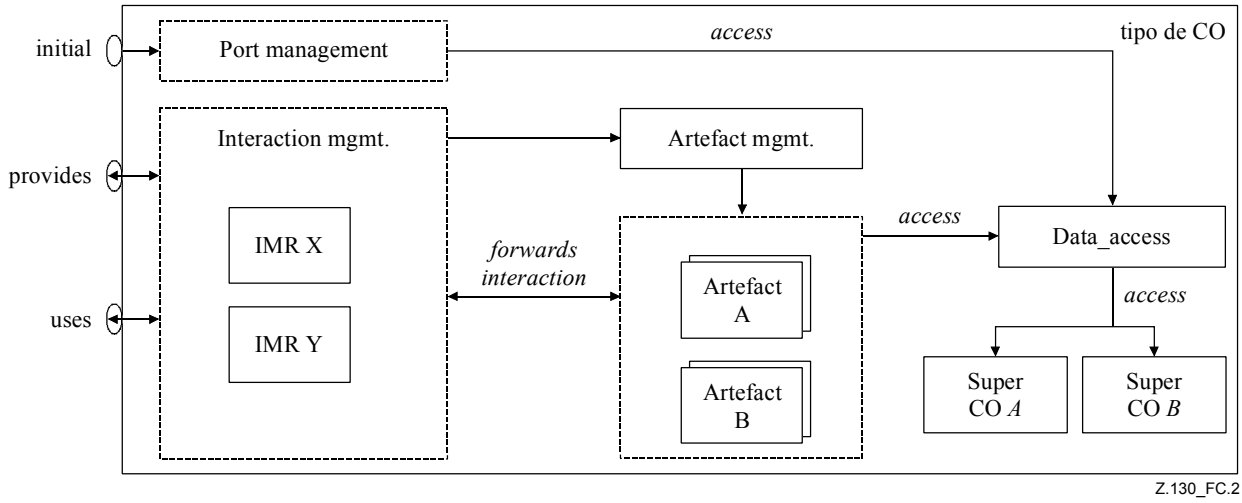
C.6.3 Servicio de denominación de SDL

Para que un **CO** pueda descubrir dinámicamente otros componentes SDL, se define un servicio de denominación. El servicio de denominación se implementa mediante el tipo de proceso `SDL_Component_Register_Type` en el lote `eodl`. Se requiere que cada sistema SDL derivado de un modelo eODL tenga un conjunto de ejemplares `SDL_Component_Registry` que contenga exactamente un ejemplar del tipo `SDL_Component_Register_Type`.

Tan pronto como se ejemplifica un componente SDL, se registra asimismo por medio del procedimiento exportado `register_SDLComponent`. Cualquier **CO** puede consultar el servicio de denominación utilizando `query_SDLComponent`. La clave para consultar un componente SDL es el nombre completamente calificado utilizando el carácter punto como carácter de calificación. El procedimiento de consulta devuelve una referencia a un ejemplar de componente SDL del tipo solicitado. Utilizando esta referencia, cualquier cliente puede solicitar la fábrica del componente SDL correspondiente a los **CO**.

C.7 Correspondencia de los conceptos de implementación

La figura C.2 representa la estructura interna de un proceso SDL que muestra un **tipo de CO** en forma esquemática con distintas representaciones SDL opcionales.



IMR Representación de gestión (*interaction management representation*)

Figura C.2/Z.130 – Proceso SDL que representa el tipo de CO

Un rectángulo representa un proceso SDL. Un rectángulo discontinuo representa en cambio un concepto. Por ejemplo "gestión de interacción" representa el concepto de gestión de interacción y contiene los procesos concretos de gestión de interacción como procesos SDL.

En la figura C.3, se presenta un ejemplo concreto de tipo de proceso CO en notación gráfica SDL. Los procedimientos `get_key`, `provide_SamplePort` y `provide` integran la gestión del puerto. Los procesos `interaction_interfaceA` e `interaction_interfaceB` se corresponden con "IMR X" e "IMR Y" de la figura C.2. Los conjuntos de ejemplares del proceso `A` y `B` son ejemplares de **artefactos**. El conjunto de ejemplares `base` corresponde a la casilla "súper CO A" de la figura C.2.

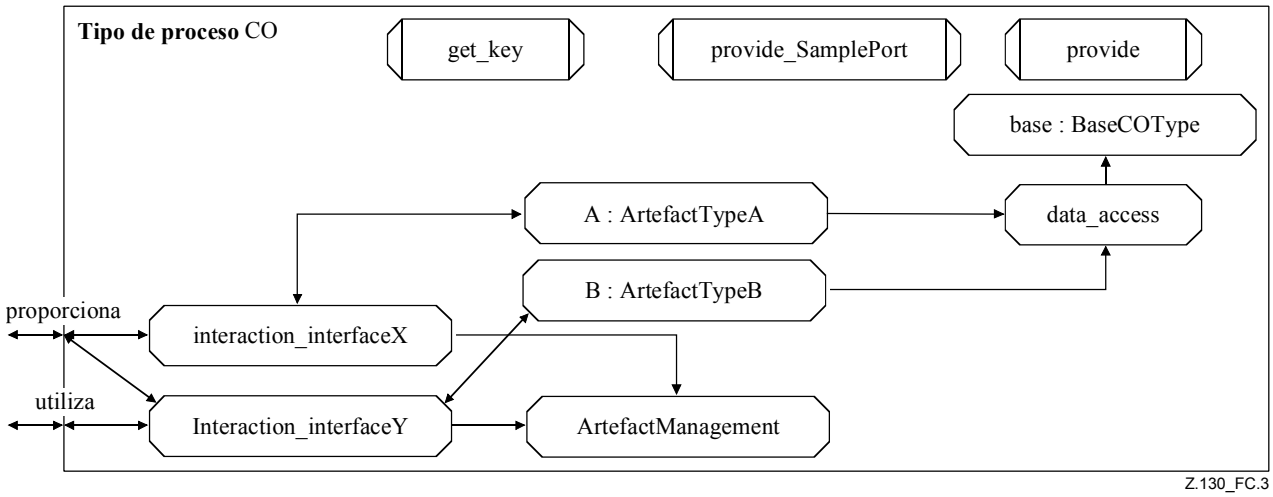


Figura C.3/Z1.30 – Ejemplo del tipo de proceso CO

C.7.1 Acceso de datos

Los datos de un CO se almacenan en un proceso `data_access` que implementa procedimientos get/set para que los artefactos puedan acceder a dichos datos. Los datos consisten en referencias utilizadas por la gestión del puerto y por la gestión de la interacción.

Para proporcionar un acceso con tipo a dichos datos, se declara un lote `<CO-Name>_data` en el lote de definición. Este lote de datos contiene una interfaz `internal_data` que declara todos los procedimientos get/set.

C.7.2 Artefactos y elementos de implementación

Los **artefactos** son elementos constructivos del lenguaje de programación que contienen **elementos de implementación**. En SDL-2000 se corresponden con tipos de procesos referenciados que se definen en los lotes de definición. Los **artefactos** se ejemplifican como un conjunto de ejemplares dentro del tipo de proceso **CO** que implementan.

Para acceder a los datos de un **CO**, hay un canal definido del conjunto de ejemplares **artefacto** al proceso `data_access` que transporta todas las llamadas de procedimiento al proceso de acceso a los datos.

Los **elementos de implementación** asocian los **artefactos** y los **elementos de interacción** que el **artefacto** implementa. No tienen representación alguna en SDL.

La implementación de un **elemento de interacción** depende del caso correspondiente. Existen dos casos de implementación

- caso suministro;
- caso utilización.

El cuadro C.1 proporciona la semántica de los géneros de **elementos de interacción** y de los casos de implementación

Cuadro C.1/Z.130 – Semántica de los géneros de elementos de interacción y de los casos de implementación

Tipo de elemento de interacción en el modelo de diseño	Definición del caso del elemento de implementación en el modelo de diseño	Semántica de los elementos de implementación
operación/atributo	suministro	Implementación del comportamiento de la **operación**/**operaciones** de acceso de suministro al atributo
operación/atributo	utilización	Llamada a operación explícita posible
consumir	suministro	Implementación del consumo de señal
consumir	utilización	Implementación del envío de señal
producir	suministro	Implementación del envío de señal
producir	utilización	Implementación del consumo de señal

Al igual que la correspondencia de las interfaces descritas en C.5.10 sobre interfaces y elementos de interacción, cada interfaz eODL se corresponde adicionalmente en el lote de definición, con un lote del mismo nombre que la interfaz que contiene dos interfaces SDL: `exported_<Interface-name>` e `imported_<Interface-name>`. La correspondencia de los **elementos de interacción** es exactamente la descrita en C.5.10 excepto por el procedimiento y la señal. Cada procedimiento y cada señal llevan un **parámetro** formal adicional del tipo `PId`. Este **parámetro** se utiliza para transportar información del emisor y del receptor, respectivamente. Véase C.7.4 para mayor información. Los tipos de señal a los que se refieren estas interfaces (que llevan un **parámetro** formal adicional del tipo `PId`) se definen en el lote de definición.

Las cláusulas siguientes presentan más detalles relativos a los **elementos de interacción** de la implementación.

C.7.2.1 Operación implementación

Para implementar un **elemento de interacción operacional** de una interfaz, el **artefacto** debe contener un procedimiento exportado que implemente el procedimiento definido en la interfaz SDL `exported_<Interface-Name>` de la correspondiente interfaz del lote de definición (véase la cláusula anterior). Por consiguiente implementa un procedimiento que contiene un **parámetro** adicional del tipo `PId`. El **artefacto** puede utilizar este **parámetro** para obtener información acerca del remitente original de la llamada de procedimiento.

C.7.2.2 Llamada a una operación desde un artefacto

Las llamadas de operación a otros **CO** se realizan del siguiente modo: el **artefacto** envía una llamada de procedimiento a dicha representación de gestión de interacción que implementa el importe de `imported_<interface-name>` de la interfaz que contiene la operación a llamar. Este procedimiento se define en el lote de definición y contiene un **parámetro** formal adicional del tipo `PId`. El **parámetro** `PId` real se utiliza para designar el receptor de la llamada de procedimiento. La representación de gestión de interacción se encarga de entregar la llamada de procedimiento a su receptor.

C.7.2.3 Envío de una señal

Como en la llamada de un procedimiento, un **artefacto** no envía una señal directamente a su receptor, sino a la representación de gestión de interacción. La señal contiene un **parámetro** formal adicional que designa al receptor de la señal y se define en el lote de definición. La representación de gestión de interacción se encarga de entregar la señal a su receptor.

C.7.2.4 Consumo de una señal

Para implementar el consumo de una señal, el **artefacto** necesita implementar un manejador de señal que acepte la correspondiente señal definida en el lote de definición.

C.7.2.5 Herencia de artefactos

En los **artefactos** se permite la herencia múltiple. Puesto que en SDL no se permiten las herencias múltiples para los tipos de agente, esto se realiza por delegación. Los **artefactos** de base están contenidos en el **artefacto** derivado y las correspondientes puertas de cada **artefacto** base y del **artefacto** derivado están conectadas por un canal. Todas las llamadas de procedimiento y las señales que van a los **elementos de implementación** que no se hayan redefinido se pasan directamente del entorno del **artefacto** derivado al correspondiente **artefacto** base. Si un cierto **elemento de implementación** se redefine en el **artefacto** derivado, el **elemento de implementación** redefinido se define en el **artefacto** derivado.

C.7.3 Gestión de artefactos y diagrama de ejemplificación

La gestión de **artefactos** es responsable de la creación y gestión de ejemplares de **artefactos**. En un **CO** hay un conjunto de ejemplares `artefact_<CO-name>` para cada **artefacto** que implementa el **CO**. No obstante, el diagrama de ejemplificación que se anota en el modelo define qué ejemplar de **artefacto** se utiliza en una interacción. La gestión de **artefactos** implementa el diagrama de ejemplificación.

La gestión de **artefactos** se realiza como proceso `artefactmanagement` contenido en el tipo de proceso del **CO**. El conjunto de ejemplares de dicho proceso contiene exactamente un ejemplar. Para cada tipo de **artefacto** que implementa el **CO** en cuestión, hay un procedimiento distante exportado `get_artefact_<artefact-name>` que devuelve una referencia a un ejemplar de dicho tipo de **artefacto**. Estos procedimientos se implementan automáticamente. Su implementación depende del diagrama de ejemplificación a utilizar:

- *Artefacto a petición*: Durante cada llamada de procedimiento, se crea un nuevo ejemplar y se devuelve una referencia al mismo.
- *Grupo de artefactos*: Se crea un número limitado de ejemplares [un "pool" (grupo)] y se devuelve una referencia a uno de estos ejemplares.
- *Singular*: Sólo hay un ejemplar del **artefacto** y se devuelve una referencia al mismo.
- *Definido por el usuario*: Dado que es el usuario quien define la semántica, el procedimiento no puede implementarse automáticamente. Por este motivo, se declara un procedimiento referenciado siendo el propio usuario quien tiene que implementarlo.

C.7.4 Representación de la gestión de interacción

En la correspondencia con SDL, la representación de la gestión de interacción se comporta como un intermediario entre los **elementos de implementación** y el entorno de un **CO**. Cada representación de **elemento de interacción** maneja tanto las interacciones de entrada como las de salida (referidas al **CO**).

Cada interfaz requerida o soportada por un **CO** se representa mediante un proceso del ámbito del tipo de proceso **CO**. Este proceso implementa cada **elemento de interacción** ya sea entregando la petición de información:

- desde el entorno del **CO** al **elemento de implementación** correspondiente, respetando por consiguiente el diagrama de ejemplificación del **artefacto**, utilizando la representación de la gestión del **artefacto**; o
- desde el **elemento de implementación** al entorno del **CO**.

Sólo hay un único ejemplar de proceso por cada interfaz. No obstante, si hay un **puerto múltiple** de este tipo se plantea una excepción a esta regla. En tal caso, hay un cierto número de ejemplares. El

número concreto depende de la implementación. Por otra parte, para cada proceso de gestión de interacción hay una variable implícita interna del **CO** `<process-name>_reference` que contiene el `PId` de dicho proceso. Si hay más de una referencia de ejemplar a guardar, se utiliza una cadena de `PId`.

C.7.4.1 Representación de la gestión de interacción implementando la interacción desde el CO al entorno

Para las interacciones con el entorno, la representación de la gestión de interacción implementa:

- todas las llamadas de **operación** (sólo para las interfaces requeridas); y
- todas las señales que pudieran enviarse.

Soporta la interfaz `imported_<interface-name>` del lote de definición en sentido entrante (desde los **artefactos**) e `imported_<interface-name>` de la interfaz de definición en sentido saliente (hacia el entorno).

Una **operación** (más concretamente la utilización de una **operación**) se implementa del siguiente modo:

1) El valor de `sender` se guarda en una variable temporal.
2) La referencia a un ejemplar de **artefacto** se adquiere mediante llamada al oportuno procedimiento de la representación de gestión del **artefacto**.
3) Se llama al procedimiento (se añade el valor de `sender` guardado a la lista de parámetros) con la referencia al ejemplar de **artefacto** como destino.
4) Si se hubieran declarado **excepciones** a la **operación**, dichas **excepciones** deben capturarse y plantearse de nuevo.

El envío de una señal se implementa del siguiente modo: cuando la representación de un **elemento de interacción** recibe la señal especificada (con el **parámetro** adicional que especifica el destino) envía la señal especificada (sin **parámetro** adicional) al destino.

C.7.4.2 Representación de la gestión de interacción desde el entorno al CO

Para la interacción en este sentido, la representación de gestión de la interacción implementa:

- todas las **operaciones** declaradas en la interfaz, y
- todas las **señales** producidas y consumidas.

Soporta la interfaz `exported_<interface-name>` del lote de interfaz en sentido entrante (desde el entorno) y `exported_<interface-name>` del lote de definición en sentido saliente (hacia los **artefactos**).

Una **operación** (más concretamente: el suministro de una **operación** al entorno) se implementa del siguiente modo. Cuando se recibe la llamada de procedimiento, se adquiere una referencia al ejemplar de **artefacto** llamando a la gestión de **artefactos**. A continuación, se efectúa una llamada de procedimiento a dicho ejemplar, proporcionando el valor de la variable `sender` como **parámetro** adicional. Si la **operación** tiene un **parámetro** de retorno, el valor de retorno se devuelve al remitente. Si se plantea una **operación**, la representación de gestión de interacciones se plantea de nuevo.

El consumo de una **señal** se implementa del siguiente modo: cuando la representación del **elemento de interacción** recibe la señal procedente del entorno, envía la señal correspondiente (con un **parámetro** adicional que contiene el valor de la variable `sender`) al **artefacto**.

C.7.4.3 Gestión de puerto

La gestión de puerto se encarga de:

- crear las representaciones del **elemento de interacción**;

- gestionar las referencias a las interfaces;
- implementar **operaciones** de acceso específicas del **puerto**; e
- implementar operaciones de acceso genéricas del **puerto**.

Todas estas tareas se implementan automáticamente.

C.7.4.4 Representación de la gestión del puerto

La gestión del puerto se implementa mediante varios procedimientos exportados y por la transición inicial de la máquina de estados del proceso **tipo de CO**.

C.7.4.5 Creación de las representaciones de la gestión de interacciones

Corresponde a la gestión de puertos la creación de los procesos que representan la gestión de interacciones. Esto se realiza en la transición inicial del proceso **tipo de CO**. Tras la creación del proceso de gestión de interacciones, la gestión de puertos almacena las referencias a cada proceso en el proceso de acceso de datos por medio de la variable `interaction_<interface-name>`.

C.7.4.6 Gestión de las referencias a las interfaces

Para cada **puerto**, hay una variable implícita local del **CO** `port_<port-name>`. El tipo de esta variable es o bien el tipo de referencia de la interfaz que tipifica el puerto (cuando el atributo del **puerto** es **sencillo**) o una cadena de PId (cuando el atributo del **puerto** es **múltiple**). Las operaciones del **puerto** pueden manipular directamente estas variables internas.

C.7.4.7 Implementación de operaciones de acceso específicas del puerto

Estas operaciones particulares se implementan del siguiente modo.

Un solo puerto suministrado: Esta operación devuelve la referencia almacenada en la variable interna.

Varios puertos suministrados: Ha de elegirse una referencia de un conjunto de referencias. Se llama al procedimiento `choose_provide_<port-name>` (definido en el proceso de gestión de puertos) para tomar esta decisión. Este procedimiento debe ser implementado por el usuario y por consiguiente es referenciado.

Un solo puerto utilizado: La operación `link` almacena una referencia dada en el proceso de acceso de datos utilizando la variable `port_<port-name>`. Si ya hubiera una referencia almacenada en éste, se plantearía la **excepción** `AlreadyConnected`. La operación `unlink` asigna el valor `Null` a la variable local **CO**. Si ya se le hubiera asignado el valor `Null`, la operación provocaría la **excepción** `NotConnected`.

Varios puertos utilizados: La operación `link` almacena una referencia dada en una secuencia de referencias. Para determinar exactamente dónde colocar la referencia dentro de la secuencia, el usuario ha de implementar el procedimiento `choose_link_<port-name>` (definido en el proceso de gestión de puertos) que toma la referencia y la almacena en algún punto de la secuencia. Análogamente, la operación de desconexión llama a `choose_link_<port-name>` para suprimir la referencia correspondiente.

C.8 Omisión de un comportamiento generado automáticamente

La correspondencia eODL-SDL presentada hasta el momento responde a la idea de que el usuario sólo desea implementar la lógica de la empresa. No obstante, si el usuario desease generar código de producción a partir del SDL, podría desear evitar el código generado automáticamente e implementar el propio.

Para que esto sea posible, la correspondencia proporciona la opción de omitir el código generado automáticamente. Específicamente, esto significa que no se generarán las siguientes entidades:

- gestión de **artefactos** y conjuntos de ejemplares de **artefactos**;
- representaciones de la gestión de interacciones; y
- gestión de puertos.

En resumidas cuentas, el tipo de proceso **CO** no contiene entidades. El usuario puede implementar el tipo de proceso **CO** como prefiera.

C.9 Conceptos de eODL sin correspondencia

Los siguientes conceptos de eODL no tienen correspondencia con SDL-2000:

- la identificación en tiempo de **ejecución** de any-type (cualquier tipo),
- la interacción de **medios continuos**,
- los **componentes de soporte lógico**,
- los **ensamblajes**,
- las restricciones y propiedades,
- los conceptos del entorno objetivo,
- el plan de despliegue.

C.10 Lote eODL predefinido

A continuación se indica el contenido completo del lote `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;
```

Anexo D

Representación en XML del metamodelo eODL

El **metamodelo** se ha definido utilizando UML. Su representación en XML conforme a OMG XMI [6] tiene por objeto poder ser leído por herramientas e integra el anexo D. Los datos reales están disponibles en el lote de soporte lógico "Z.130 anexo D.xml".

NOTA – Este lote de soporte lógico Z.130 anexo D.xml está disponible en la base de datos del UIT-T de lenguajes formales en la dirección http://www.itu.int/ITU-T/formal-language/xml/database/itu-t/z/z130/2003/.

Apéndice I

Ejemplo: la Cena de los Filósofos

I.1 Introducción

El objeto del presente apéndice es presentar un ejemplo de cómo puede utilizarse el eODL para el diseño, implementación y **despliegue** de los sistemas distribuidos.

El problema de la Cena de los Filósofos lo describió por primera vez Edsger W. Dijkstra en 1965. Se trata de un modelo y un método universal para comprobar y comparar teorías de asignación de recursos. Dijkstra pretendía utilizarlo como ayuda a la creación de un sistema operativo por capas, creando una máquina que pudiera considerarse autómata totalmente determinista.

Un número configurable de filósofos (procesos) están sentados a una mesa redonda en la que hay un número finito de tenedores (recursos). Los filósofos ejecutan acciones: piensan, comen y duermen. No necesitan ningún recurso para pensar o dormir, pero cada uno de ellos necesita dos tenedores para comer, uno para la mano izquierda y otro para la derecha. Por consiguiente, antes de empezar a comer, cada filósofo intenta coger los dos tenedores próximos a él. Esto significa que dos filósofos vecinos no pueden comer simultáneamente.

Cuando se produzca un cambio de actividad, es decir siempre que comiencen a comer, a pensar o a dormir, los filósofos notificarán a un observador. Se notificará asimismo al observador el estado crítico de tener hambre.

I.2 Descripción

El problema consiste en que un conjunto finito de procesos comparte un conjunto finito de recursos, cada uno de los cuales sólo puede utilizarse en un proceso a la vez, pudiendo provocando por consiguiente situaciones de bloqueo o círculos viciosos.

El conjunto finito de procesos, recursos y las interacciones dinámicas entre éstos constituye un sistema distribuido. El objetivo es distribuir las implementaciones de los recursos y procesos por la red objetivo. Además, los recursos han de conectarse con los procesos.

En el caso del ejemplo se incluyen tres **tipos de CO** diferentes:

- Filósofo.
- Tenedor.
- Observador.

Para diseñar, implementar y desplegar el ejemplo, hay que ejecutar los pasos siguientes:

Fase de diseño

- Definir un modelo con los elementos del ejemplo que comprenda los **tipos de CO**, los puertos y las interfaces.
- Definir un modelo de la estructura de implementación.

Fase de implementación

- Implementar los **artefactos** con arreglo al modelo (suministrar la lógica de la empresa).
- Generar los **componentes de soporte lógico** con arreglo al modelo.
- Definir un modelo de la estructura inicial del sistema (**configuración inicial**) mediante la definición de un ensamblaje.
- Empaquetar los **componentes de soporte lógico** y la información del modelo relacionado para poder enviar la implementación a los clientes.

Fase de integración

- Entregar el paquete al cliente.
- Modelar el entorno objetivo de las instalaciones del cliente.
- Determinar una asignación adecuada de los **componentes de soporte lógico** del lote al entorno objetivo.
- Instalar los **componentes de soporte lógico** asignados en los nodos objetivo identificados.
- Establecer la **configuración inicial** mediante la interconexión de los **CO iniciales** con arreglo a la **configuración inicial**.

En I.3 se especifica el ejemplo de la "Cena de los Filósofos" en eODL. En esta especificación, se definen tres **tipos de CO**:

- El tipo de objeto o_Philosopher representa al filósofo.
- El tipo de objeto o_Fork representa al tenedor.
- El tipo de objeto o_Observer representa al observador.

La cláusula I.4 contiene la correspondencia del modelo eODL con arreglo a las reglas de correspondencia presentadas en el anexo C. Sólo se establecen correspondencias para los conceptos de las perspectivas computacional, de configuración y de implementación, pero no hay correspondencia para los conceptos de la perspectiva de despliegue. El modelo SDL consta de dos lotes principales:

- el lote de interfaz SDL phil_interface; y
- el lote de definición de SDL phil_definition.

I.3 El ejemplo en 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 El ejemplo en SDL-2000

```
use eODL;
/* /--------------------------------\ */
/*     data types and interface       */
/*     needed by clients              */
package phil_interface;

package DiningPhilosophers;

  /* exceptions */
  exception ForkNotAvailable;
  exception NotTheEater;

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

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

  /* interface i_Fork */
  package i_Fork;
    /* declaration of exported procedures */
    interface exported_i_Fork;
      procedure obtain_fork(in o_Philosopher)
```

```
            raise ForkNotAvailable;
      procedure release_fork(in o_Philosopher)
                raise NoTheEater;
    endinterface;

     /* declaration of consumed signals */
    interface imported_i_Fork;
         /* no consumed signals declared */
    endinterface;
  endpackage;

  /* definition of CO type o_Fork */
  use i_Fork;
  package o_Fork;

    /* contains attributes defined in CO type o_Fork */
    interface o_Fork_attributes
       inherits <<package eODL/package ComponentModel>>ComponentBase;
    endinterface;

    /* port operations */
    interface o_Fork_config
       inherits <<package eODL/package ComponentModel>>ConfigBase adding;
      /* provided port "fork" */
      procedure provide_fork -> exported_i_Fork;
    endinterface;

    /* combine config and attributes interfaces */
    interface o_Fork
      inherits o_Fork_attributes, o_Fork_config;
    endinterface;

    /* declaration of interface of CO factory */
    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; /* state -> eodl_state ! */
      public name string;
       public philosoph exported_i_Philosopher;
    operators
       /* create -> 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;

    /* contains attributes defined in CO type 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;
/* \--------------------------------/ */

/* /--------------------------------\ */
/*     implementation package         */
use eODL;
package phil_definition;

package DiningPhilosophers;


  /* used to define operations implemented or */
  /* used by artefacts */
  package i_Fork;
    /* operations implemented by artefacts */
    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;

    /* operations used by artefacts */
    interface imported_i_Fork;
    endinterface;
  endpackage;


  /* state attributes */
  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;

/* artefact: referenced definition */
use a_PhilosopherImpl;
use o_Fork_state;
process type a_ForkImpl with
   use (a_ForkImpl_in);;
  referenced;

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

  /* gate definitions */
  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;

  /* defines the factory process */
  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;

    /* stores component keys here */
    dcl keys ComponentKeysSeq;

    /* creates a CO and returns a reference to it */
    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;

    /* creates a CO and returns a reference to it */
    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;

  /* returns name of CO type */
  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;

  /* returns reference to a 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;

  /* returns a list of CO keys */
   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;
/* \--------------------------------/ */

/* defines the CO type itself */
process type o_Fork;

  /* gates used for comm. with environment */
  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;

  /* /--------------------------------\ */
  /*     encapsules state variables     */
  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;
/* \--------------------------------/ */

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


/* in case of CO type inheritance an instance of this type */
/* is contained in an "eODL-inherited" type. The following */
/* gate is used to interface with the state_access process */
/* of the inherited type. */
gate data_accessor
  in with internal_data;
channel route_state_access nodelay
  from
    env via data_accessor to data_access;
endchannel;


/* /--------------------------------\ */
/*     manages artefact instances     */
process artefactmanagement(1,1);
  /* signals for delegation of artefact creations */
  signal create_a_ForkImpl_req;
  signal create_a_ForkImpl_res(Pid);
  /* Artefact-Pool and pointer in pool */
  dcl a_ForkImpl_seq PIdSeq := (. .);
  dcl a_ForkImpl_ptr Integer := 0;
  /* returns an artefact instance reference */
  /* implements a pool of size POOLSIZE */
  /* creates a new artefact if pool is not yet of */
  /* size POOLSIZE, otherwise returns the "next" */
  /* artefact. */
  exported procedure get_artefact_a_ForkImpl -> PId;
    dcl new_pid PId;
    start;
     decision length(a_ForkImpl_seq);
      (0:1):
        /* delegate creation of artefact */
         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;
    /* delegation continued ... */
    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;
  /* create artefact instance for procedure */
  state wait_for_signal;
  input create_a_ForkImpl_req;
  create artefact_a_ForkImpl;
   output create_a_ForkImpl_res(offspring) to sender;
  nextstate -;
endprocess;
/* \-------------------------------/ */

/* this is the interactionmanagementrepresentation */
/* for interface i_fork. handles procedure calls   */
/* from the environment */
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;

/* portmanagement */
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 for internal variables */
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_key
  procedure get_key -> ComponentKey;
    start;
    return the_key;
  endprocedure;


/* computes key and instantiates        */
/* interactionmanagementrepresentations */
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;

/* process instance set of artefact a_ForkImpl */
process  artefact_a_ForkImpl(0,2): a_ForkImpl;

/* connects artefact and 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;
  /* \--------------------------------/ */

  /* instance sets for factory and co 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 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;

    /* returns reference to a 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;

  /* returns a list of CO keys */
   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;


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


/* in case of CO type inheritance an instance of this type */
/* is contained in an "eODL-inherited" type. The following */
/* gate is used to interface with the state_access process */
/* of the inherited type. */
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;

  /* artefact: referenced definition */
  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;
```

Apéndice II

Procesamiento de la información y soporte de herramientas

II.1 Introducción

La existencia de un **metamodelo** sencillo y completo proporciona una base estable para el desarrollo de soporte lógico, incluso en áreas de aplicación complejas. Para que la técnica sea utilizable y en particular para que sea de fácil uso para los desarrolladores, se necesita el adecuado soporte de herramientas. En general, estas herramientas pueden estar destinadas al propio proceso de modelado, como es el caso de los editores y simuladores, o pueden cubrir más fases, tales como las de la implementación y **despliegue** en **plataformas objetivo**.

Debido a la gran diversidad de tareas que han de soportar las herramientas que procesan un único modelo desde el comienzo de la especificación hasta el **despliegue y la ejemplificación**, se prevé la utilización de diferentes herramientas formando una cadena de herramientas. De este modo, se plantea la cuestión de disponer de un formato de intercambio entre herramientas. Para ello, se utiliza por defecto una notación normalizada como mínimo. Debido a la aplicación de OMG MOF, se sobreentiende para el **metamodelo** la representación basada en XML que, por consiguiente, puede utilizarse como formato de intercambio entre las distintas herramientas.

La definición concreta de las herramientas y de su funcionalidad no puede someterse a normalización. Aunque en la práctica, pueden diseñarse arbitrariamente herramientas sencillas que en realidad están formadas por cadenas de herramientas, las cláusulas siguientes agrupan los temas sobre herramientas en base a determinadas cuestiones. Las herramientas reales pueden abarcar varios de estas cuestiones.

II.2 Cuestiones sobre las herramientas de modelado

Las herramientas para la manipulación de la información del modelo pueden utilizar una representación arbitraria del modelo con la única restricción de que cada representación tenga una correspondencia apropiada en el **metamodelo**. Estas representaciones pueden variar entre lenguajes de programación y notaciones gráficas, tal como el UML. Puede utilizarse cualquier herramienta que soporte el procesamiento de esta información sin restricciones.

Dado que el **metamodelo** cubre prácticamente todo el ciclo vital del soporte lógico, hay una diversidad de cuestiones de modelado posibles que se resolverán, por supuesto, con soporte de herramientas. Un modelo puede ampliarse gradualmente por la adición de información suplementaria en varias iteraciones de modelado en diferentes momentos. Por consiguiente, puede utilizarse un conjunto de **tipos de CO** existentes para especificar más adelante un **ensamblaje**, pudiendo darse posteriormente una asignación concreta de este **ensamblaje** a una **plataforma objetivo**. En general, las cuestiones de modelado, ordenadas por instante de aplicación, son las siguientes:

- especificación del **tipo de CO**,
- especificación del **ensamblaje**,
- paquete de implementación (de los **componentes de soporte lógico**),
- modelado del entorno,
- asignación de los **componentes de soporte lógico** a una **plataforma objetivo**,
- ejemplificación de los **ensamblajes** respectivos de los **tipos de CO**.

El primer paso a ejecutar es la especificación de los **tipos de CO** con arreglo al **metamodelo**. Cuando se dispone de dichos tipos ya pueden definirse los ensamblajes. Cada tipo puede utilizarse en un número arbitrario de ensamblajes. El paso siguiente consiste en proporcionar la implementación correspondiente a todo el **ensamblaje** que contiene el código de implementación de los **tipos de CO** utilizados agrupado en **componentes de soporte lógico**. El empaquetado de las implementaciones puede llevarse a cabo mediante herramientas de archivo, tales como el compresor de archivos (zip). Una vez realizado lo anterior, puede enviarse el modelo junto con el lote de implementación del **ensamblaje** para desplegarlos en las plataformas de los clientes. Para poder desplegar un **ensamblaje**, hay que determinar antes la distribución de los **CO**. Durante este proceso, el modelo se enriquece con información suplementaria relacionada fundamentalmente con el entorno concreto y la actividad empresarial específica del cliente. El modelo del entorno objetivo se compara con el modelo del **ensamblaje** para establecer una asignación adecuada de cada **CO** a cada nodo de la plataforma. Esto puede hacerse manualmente, encargándose de ello si es posible el administrador del sistema, o semiautomáticamente mediante una función automática que proporcione una solución de **configuración inicial** del **ensamblaje**. En ambos casos, hay que cumplir los requisitos de cada **CO** del **ensamblaje** en el entorno objetivo. No se especifica dónde hay que obtener el modelo de entorno. Podría obtenerse directamente del entorno objetivo, en cuyo caso se requeriría una arquitectura especial. Como resultado del paso de asignación, el modelo contiene información sobre dónde instalar cada **componente de soporte lógico**. La ejemplificación real de los **tipos de CO** o ensamblajes puede formar parte del modelo. Una herramienta adecuada tendría que mantener el modelo actualizado durante el tiempo de **ejecución**.

II.3 Cuestiones sobre la herramienta generatriz

Las herramientas que dan soporte a la implementación de las entidades del modelo pueden simplificar enormemente las cosas en comparación con la implementación manual. Puede generarse código para todos los lenguajes de implementación que tengan una correspondencia con el **metamodelo**. En general, a partir del modelo puede generarse la siguiente información:

- esqueletos para **tipos de CO**,

- código para las especificaciones de la calidad de servicio.

El código generado puede ofrecer al implementador un marco que sirva como base a las implementaciones de **tipo de CO**. Mientras el **metamodelo** no contenga aspectos comportamentales, no se puede generar automáticamente la implementación de la lógica de la empresa y habría que generarla por tanto manualmente.

II.4 Cuestiones sobre las herramientas de despliegue

El despliegue, incluida la ejemplificación de ensamblajes en la **plataforma objetivo** de los clientes, requiere ser soportado por herramientas adecuadas, pero depende asimismo del adecuado soporte en la propia plataforma. Por consiguiente, las herramientas reales para el **despliegue** están estrechamente vinculadas a arquitecturas de plataforma concretas con las que tienen que interaccionar. Las herramientas no pueden ser independientes mientras no haya una arquitectura de plataforma normalizada con la que se pueda colaborar.

En términos generales, el proceso de **despliegue** comprende varias tareas que comienzan con la determinación de una distribución adecuada y terminan en la instalación y ejemplificación del soporte lógico. Las herramientas pueden encargarse de las siguientes tareas comunes de **despliegue**:

- modelado del entorno,
- asignación de **componentes de soporte lógico** a la **plataforma objetivo**,
- instalación de **componentes de soporte lógico** en la **plataforma objetivo**,
- ejemplificación del **tipo de CO** correspondiente a cada **ensamblaje**,
- procesamiento de restricciones y acciones.

Las tareas de modelado del entorno y las de asignación de **componentes de soporte lógico** a las **plataformas objetivo** ya se mencionaron en el contexto de las cuestiones sobre las herramientas de modelado. De hecho, en estas tareas se amplían los modelos, razón por la cual se utilizaron en dicho contexto. En realidad, se prevé que estas tareas se ejecuten, en la mayor parte de los casos, como parte de **despliegue** del **ensamblaje**. Como ya se ha expuesto, una plataforma puede soportar herramientas para obtener información de modelado del entorno. Una vez realizadas estas tareas y determinada la oportuna asignación de **componentes de soporte lógico** a una **plataforma objetivo**, el paso siguiente consiste en cargar e instalar el soporte lógico en el nodo especificado. Tras esto, puede ejemplificarse el **ensamblaje** de **tipos de CO** con la ayuda de otra herramienta o capacidad de la plataforma. Finalmente, hay que procesar en tiempo de **ejecución**, de una manera u otra, las restricciones y acciones contenidas en el modelo.

En la figura II.1, se representan las cuestiones de la cadena de herramientas desde el modelado hasta el **despliegue**.

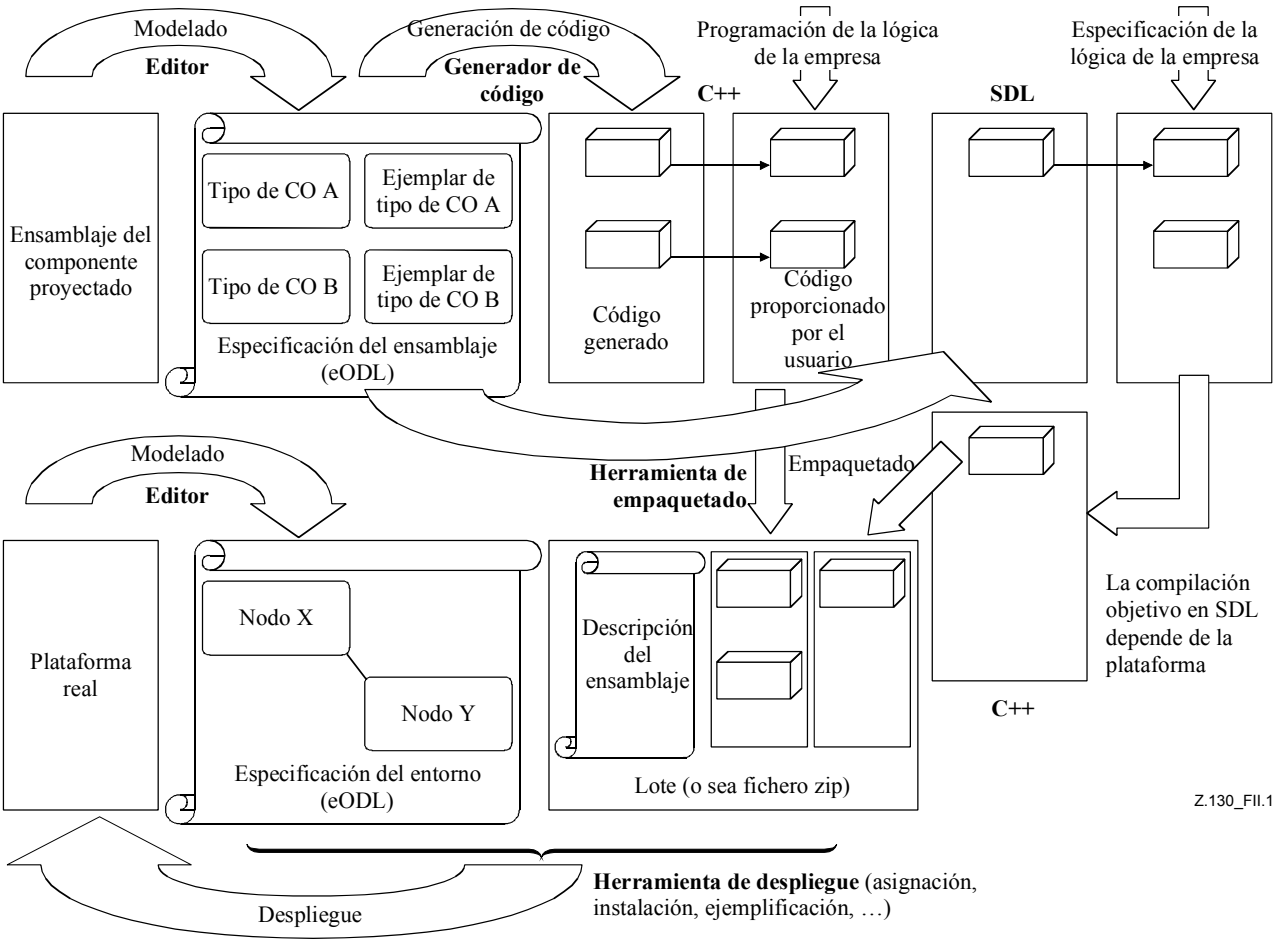


Figura II.1/Z.130 – Procesamiento de la información

SERIES DE RECOMENDACIONES DEL UIT-T

Serie A	Organización del trabajo del UIT-T
Serie B	Medios de expresión: definiciones, símbolos, clasificación
Serie C	Estadísticas generales de telecomunicaciones
Serie D	Principios generales de tarificación
Serie E	Explotación general de la red, servicio telefónico, explotación del servicio y factores humanos
Serie F	Servicios de telecomunicación no telefónicos
Serie G	Sistemas y medios de transmisión, sistemas y redes digitales
Serie H	Sistemas audiovisuales y multimedios
Serie I	Red digital de servicios integrados
Serie J	Redes de cable y transmisión de programas radiofónicos y televisivos, y de otras señales multimedios
Serie K	Protección contra las interferencias
Serie L	Construcción, instalación y protección de los cables y otros elementos de planta exterior
Serie M	RGT y mantenimiento de redes: sistemas de transmisión, circuitos telefónicos, telegrafía, facsímil y circuitos arrendados internacionales
Serie N	Mantenimiento: circuitos internacionales para transmisiones radiofónicas y de televisión
Serie O	Especificaciones de los aparatos de medida
Serie P	Calidad de transmisión telefónica, instalaciones telefónicas y redes locales
Serie Q	Conmutación y señalización
Serie R	Transmisión telegráfica
Serie S	Equipos terminales para servicios de telegrafía
Serie T	Terminales para servicios de telemática
Serie U	Conmutación telegráfica
Serie V	Comunicación de datos por la red telefónica
Serie X	Redes de datos y comunicación entre sistemas abiertos
Serie Y	Infraestructura mundial de la información, aspectos del protocolo Internet y Redes de la próxima generación
Serie Z	**Lenguajes y aspectos generales de soporte lógico para sistemas de telecomunicación**

Impreso en Suiza
Ginebra, 2004