UNIVERSIDAD POLITÉCNICA DE CARTAGENA
E. T. S. Ingeniería de Telecomunicaciones

Desarrollo de un editor textual para la creación de modelos de componentes V3CMM

Alejandro Illán Guillén

Cristina Vicente Chicote
Diego Alonso Cáceres

Título del Proyecto

Desarrollo de un editor textual para la creación de modelos de componentes V3CMM

Autor

Alejandro Illán Guillén

Titulación

Ingeniería Técnica de Telecomunicación, Especialidad Telemática

Directores

Cristina Vicente Chicote
Diego Alonso Cáceres

Defensa

Septiembre 2012

ÍNDICE

1. *INTRODUCCIÓN* 6

1.1. Objetivos 7

1.2. Fases de desarrollo del Proyecto...7

1.3. Organización de la memoria 8

2. *TRABAJOS RELACIONADOS* 9

2.1. Los primeros editores...9

2.2. Editores textuales: Xtext...10

2.3. Editores gráficos: GMF..12

3. *TRABAJOS PREVIOS: V3CMM* 14

3.1. V3CMM 14

3.2. ¿Tres vistas? ¿Por qué?...14

3.3. Meta-modelo V3CMM...16

3.4. Restricciones OCL...20

4. *DESARROLLO DE LOS EDITORES* 21

4.1. Retos del desarrollo..21

4.2. Esquemas sintácticos sugeridos para cada una de las vistas.............................22

4.3. Descripción del procedimiento para la creación de los editores 26

4.4. Funcionalidades añadidas a los editores 32

4.4.1. Coloración de sintaxis... 32

4.4.2. Auto-compleción del esquema.. 35

4.4.3. Adaptación de puertos a su ámbito 38

4.4.4. Modificación del ContentAssist.. 40

5. *GUÍA DE INSTALACIÓN Y USO DE LOS EDITORES* 45

5.1. Consideraciones iniciales...45

5.2. Instalación para desarrolladores..45

5.3. Instalación para usuarios...47

5.4. Guía de usuario para desarrolladores..47

5.5. Guía de usuario 48

6. *CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURAS* 64
6.1. Conclusiones 64
6.2. Líneas de Trabajo Futuras 65
7. *REFERENCIAS* 66
8. *APENDICE 1: VISTAS* 67
1.1. Vista de Interfaces y tipos de datos 67
1.2. Vista de componentes 69
1.3. Vista de máquina de estados 70
1.4. Vista de aplicación 71
9. *APENDICE 2: RESTRICCIONES OCL ADICIONALES* 73

Capítulo 1º

Introducción

V3CMM (3-View Component Meta-Model) [1] es un lenguaje de modelado basado en componentes que permite el diseño de aplicaciones de forma independiente de la plataforma y el dominio concreto de aplicación. La adopción de un enfoque de Desarrollo de Software Dirigido por Modelos (DSDM) [2] facilita la transformación de los diseños modelados con V3CMM en otros modelos (por ejemplo, para análisis o simulación) o incluso en código. V3CMM define tres vistas complementarias, altamente cohesionadas y débilmente acopladas: (1) una vista estructural, para describir la estructura estática de los componentes, tanto simples como compuestos, que integran la arquitectura de la aplicación, (2) una vista de coordinación, basada en máquinas de estados, para modelar el comportamiento de los componentes frente a los mensajes que reciben del exterior y (3) una vista algorítmica, basada en diagramas de actividad, que permite describir los algoritmos que ejecuta cada componente dependiendo del estado en que se encuentre. Existe una cuarta vista en la que se definen los elementos comunes a las tres anteriores como, por ejemplo, los tipos de datos o las interfaces.

Este Proyecto Final de Carrera tiene como objetivo abordar el desarrollo de un conjunto de editores textuales de modelos para el lenguaje V3CMM, utilizando para ello el framework Xtext [3]. En concreto, se crearán cuatro editores que permitirán modelar la arquitectura de una aplicación en base a la definición de componentes, máquinas de estado e interfaces de servicio. Con ello se pretende ofrecer a los diseñadores un conjunto de herramientas que faciliten el uso de la vista estructural y de coordinación del lenguaje V3CMM, proporcionándoles una sintaxis textual sencilla y expresiva, así como algunas características avanzadas de compleción y coloración de sintaxis y de validación sintáctica de los modelos.

1.1. Objetivos

El objetivo principal de este Proyecto Final de Carrera es desarrollar una herramienta que proporcione una sintaxis textual concreta para el lenguaje basado en componentes V3CMM. Este objetivo se subdivide en los siguientes:

1) Aprender los fundamentos del DSDM y las herramientas que dan soporte a este paradigma en el entorno de desarrollo Eclipse, en particular, el framework Xtext.

2) Conocer y comprender el lenguaje de modelado V3CMM, cuyo meta-modelo se definió previamente en base a Eclipse Modeling Framework Project (EMF) [4].

3) Implementar editores textuales de modelos para el lenguaje V3CMM utilizando el framework Xtext.

4) Desarrollar modelos de prueba para comprobar el correcto funcionamiento de la herramienta.

Por último, nos proponemos analizar los resultados obtenidos y extraer conclusiones sobre la herramienta desarrollada, así como su utilidad dentro del campo de la robótica y el modelado de software basado en componentes. Concluimos con algunas líneas de trabajos futuros de la herramienta desarrollada y sus posibles aplicaciones.

1.2. Fases de desarrollo del Proyecto

El presente Proyecto se ha llevado a cabo siguiendo las etapas que a continuación se enumeran y explican brevemente:

a) Estudio y análisis de los fundamentos del DSDM y de las herramientas Eclipse asociadas: Estudio desarrollado en el curso de "Introducción Práctica al Desarrollo de Software Dirigido por Modelos" impartido en la Universidad Politécnica de Cartagena [5].

b) Definición y formalización del problema y su contexto: Planteamiento del problema en base a la experiencia de los directores del Proyecto. Decisión del esquema de cada uno de los editores a desarrollar en base al meta-modelo.

c) Estudio y análisis del meta-modelo de V3CMM: Revisión bibliográfica de contenidos acerca de V3CMM y comprensión del funcionamiento y finalidad del meta-modelo.

d) Implementación de las herramientas:

1. **Inclusión de restricciones OCL en el meta-modelo para completar la sintaxis abstracta del lenguaje V3CMM y poder incorporar capacidades de validación en las herramientas que se construyan a partir de él.**

2. **Definición de la sintaxis textual concreta para el lenguaje V3CMM y creación de editores textuales de modelos con Xtext para darle soporte.**

3. **Implementación de funcionalidades avanzadas que faciliten el uso de los editores textuales:**

    - Coloración de sintaxis [6] distintiva para cada uno de los modelos/vistas soportados.

    - Auto-compleción del esquema textual asociado a cada vista.

    - Identificación del ámbito de definición [7] de los puertos de cada componente para facilitar su conexionado.

    - Modificación del "Content Assist" [8] para que el asistente de contenido del editor sólo mostre al diseñador puertos compatibles al que quiere conectar.

e) Evaluación de las herramientas desarrolladas, extracción de conclusiones y redacción de la memoria.

1.3. Organización de la memoria

El resto de la memoria se organiza como se indica a continuación:

- **Capítulo 2**: Breve revisión a los distintos tipos de editores y a las herramientas basadas en MDE (Model Driven Engineering) que les dan soporte en la plataforma Eclipse Modeling Project.
- **Capítulo 3:** Revisión de trabajos previos: ¿Cómo funciona V3CMM?
- **Capítulo 4:** Desarrollo de los editores textuales.
- **Capítulo 5:** Guía de uso de la herramienta.
- **Capítulo 6:** Exposición de resultados, conclusiones y trabajos futuros.
- **Referencias bibliográficas**
- **Apéndice 1:** Explicación detallada de cada una de las vistas de V3CMM.
- **Apéndice 2:** Listado detallado de las restricciones OCL añadidas.

CAPÍTULO 2º

Trabajos relacionados

En este capítulo se tratará la evolución de los editores, tanto textuales como gráficos, hasta ahora. Además se hablará de las herramientas Eclipse para la creación de editores personalizados.

2.1. Los primeros editores

Los programas que ejecutaban los primeros computadores eran almacenados en una gran cantidad de tarjetas perforadas, que debían ser entregadas a la máquina mediante un lector de tarjetas. Este método era como poco ineficiente y costoso.

Con la aparición del monitor de tubos catódicos nacieron los primeros editores de texto. Su aparición como herramienta de desarrollo supuso un gran avance y aumento en la productividad del software, a pesar de requerir un aprendizaje por parte del desarrollador que pudiera ser más o menos costoso.

La evolución seguida por estos editores desde entonces ha sido enorme, y aunque siguen existiendo editores extremadamente sencillos, otros han alcanzado un gran nivel de complejidad e incorporación de funciones. Algunas de estas funciones que facilitan la labor del usuario pueden ser: la coloración de palabras reservadas, compleción automática de palabras iniciadas, plegado y desplegado de ciertas regiones, etc.

Eclipse es una plataforma de desarrollo de software extensible a través de un mecanismo basado en plug-ins. Una de estas extensiones es la herramienta Xtext que, siguiendo los

principios del DSDM, permite crear editores textuales complejos, con muchas funcionalidades y específicos a un lenguaje previamente modelado a través de un meta-modelo.

Otro tipo de editores muy interesante son los gráficos, que utilizan una notación visual para especificar un modelo. Este tipo de editores son muy fáciles de usar para el desarrollador, ya que no requieren prácticamente aprendizaje. Eclipse también proporciona una herramienta para la creación de estos editores basada en el DSDM llamada *Graphical Modeling Framework* (GMF) [9].

2.2. Editores textuales: Xtext

"*Un editor textual es un programa que permite crear y modificar archivos digitales compuestos* únicamente *por texto sin formato, conocidos comúnmente como archivos de texto o texto plano. El programa lee el archivo e interpreta los bytes leídos según el código de caracteres que usa el editor.*" [10]

La herramienta Xtext disponible en la plataforma Eclipse permite crear un editor textual de un lenguaje, previamente modelado en EMF mediante un meta-modelo, en un tiempo relativamente corto. Además proporciona un marco de trabajo amigable y sencillo para la definición de la sintaxis concreta que deberá seguir nuestro lenguaje en el editor.

Una vez definida dicha sintaxis, la herramienta crea un editor de texto para nuestro lenguaje basada en Java, lo que la hace independiente del sistema operativo y nos permite ejecutarla en cualquier Máquina Virtual de Java compatible. Además, implementa un *parser* de texto que permite analizar y serializar el texto en base a la gramática de nuestro lenguaje, un editor en árbol, un marco de alcance y vinculación (scoping) y un validador gramatical. Todos estos componentes se integran en tiempo de ejecución y, como se basan en EMF, pueden ser usados conjuntamente con otro tipo de editores EMF, como un editor gráfico GMF.

Xtext utiliza el framework de *Google Guice* [11], inyectando las dependencias para crear nuestro lenguaje así como la infraestructura del mismo. A pesar de utilizar un módulo central y externo para ello, Xtext proporciona una implementación predeterminada, siendo lo más destacable de la misma la capacidad de personalización que *Google Guice* otorga al desarrollador, el cual puede modificar todas las clases de forma libre y poco invasiva. Un claro ejemplo de esta capacidad de configuración son las herramientas desarrolladas en este Proyecto, habiéndose implementado coloraciones de sintaxis alternativas para cada editor, inyección de código, etc.

A modo de ejemplo, el código que se muestra en el Bloque 1 define la sintaxis de la vista para la definición de interfaces y tipos de datos de V3CMM. Primero se declara la gramática del nuevo lenguaje con un identificador unívoco (*v3cmm.Idt*) y se importa el paquete Ecore de donde se deriva la gramática. Los conceptos del lenguaje, que se corresponden con elementos EClass (clases) del meta-modelo, son identificadores seguidos por ':', se puede observar que se definen los siguientes conceptos: *ROOT_IDT*, *Import*, *Datatype*, Interface, *Service* y *Parameter*. El primero de ellos constituye el elemento raíz de esta vista, ya que EMF requiere que exista una relación de composición que albergue todos los elementos. Por tanto, la única función del elemento es la de actuar como Root (raíz). Para efectuar esta relación de composición aparecen varios atributos, a saber: *imports*, *datatypes* e *interfaces*, que crean la contención de *Import*, *Datatype* e *Interface* respectivamente. El elemento *Import* define una sentencia para importar otros archivos desde el que se está creando. El elemento *Datatype* por su parte, define un tipo de dato según un atributo *name* de la clase, que será común a cualquier archivo

que lo importe haciendo uso de la sentencia anterior. *Interface* define una interfaz que contiene *Service*, que define un servicio de la interfaz y que a su vez contiene un *Parameter*.

```
grammar v3cmm.Idt with org.eclipse.Xtext.common.Terminals
import "platform:/resource/es.upct.dsie.v3cmm/Meta-
Model/es.upct.dsie.v3cmm.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

ROOT_IDT returns ROOT_IDT:
      (imports+=Import)*
      (datatypes+=Datatype)*
      (interfaces+=Interface)*;

Import returns Import:
      'import' importURI=EString ';';

Datatype returns Datatype:
      'datatype' name=EString '"'documentation=STRING'"';
Interface returns Interface:
      'interface' name=EString
       'extends' extends=[Interface]
      'version' version=EInt '"'documentation=STRING'"'
      '{' (services+=Service)* '}';

EString returns ecore::EString:
      STRING | ID;

Service returns Service:
      name=EString '"'documentation=STRING'"'
      '(' (parameters+=Parameter)* ')';

EInt returns ecore::EInt:
      '-'? INT;

Parameter returns Parameter:
      name=EString ':' direction=ParameterKind type=[Datatype]
      '"'documentation=STRING'"'(',')+;

enum ParameterKind returns ParameterKind:

      inCopy = 'inCopy' | inRef = 'inRef' | out = 'out' |
inOut = 'inOut';
```

Bloque 1: Código que define la gramática de la vista de interfaces y tipos de datos

En la Figura 1 se puede observar un modelo realizado con el editor generado con la gramática anteriormente descrita. Se puede observar que las palabras que en el bloque de código aparecen como String, esto es, las que aparecen de color azul entre comillas simples, en tiempo de ejecución se convierten en palabras reservadas del editor. Se ha introducido además una errata en una palabra reservada para mostrar la validación de sintaxis del editor. También se comprueba que proporciona coloración en las palabras reservadas, una de las mejoras en los editores que ya se indicaban con anterioridad.

```
import name;
datatype name2 "Documentation";
datatipe name3 "Documentation2";
interface name4 extends name4 version 1 "Documentation3"
{
    name5 "Documentation4"
        (
            name6:inOut name2 "Documentation5",
            name7:inCopy name3 "Documentation6"
        )
}
```

Figura 1: Aspecto general de un fichero de definición de Interfaces y Tipos de Datos

2.3. Editores gráficos: GMF

GMF es la herramienta basada en EMF que proporciona la plataforma Eclipse para la creación de editores gráficos para el desarrollo de lenguajes de modelado de dominio específico, de forma relativamente sencilla y rápida.

Una característica destacable de GMF es que permite reutilizar las definiciones gráficas de los elementos en diferentes dominios y aplicaciones. Esto es debido a que tanto las componentes gráficas como la paleta de herramientas para la creación de las mismas en el editor se definen por separado, asociándolas más tarde mediante un mapeado o mapping. A continuación, se explican brevemente los pasos en los que se basa el funcionamiento de GMF:

a) Definición del meta-modelo: Al igual que Xtext, GMF necesita partir de una base EMF, esto es, un meta-modelo que especifique la sintaxis abstracta del editor. Hecho esto, se genera el fichero con extensión *.genmodel*, que permite generar el código del plug-in correspondiente al editor en árbol.

b) Definición de la sintaxis concreta gráfica: Para ello, primero se debe decidir qué primitivas del meta-modelo harán la función de nodo en el editor, cuales harán la función de link o conector entre nodos y por último, las propiedades o atributos que jugarán el papel de etiquetas. Una vez hecho esto, se puede modificar el aspecto visual de los elementos otorgándoles diferentes formas y colores. Todo esto se lleva a cabo en un fichero con extensión *.gmfgraph*.

c) Definición de la paleta de herramientas: La paleta de herramientas muestra las primitivas que el desarrollador definió como sintaxis gráfica y proporciona la funcionalidad “drag&drop” para hacer uso de las mismas. Todo ello queda recogido en un fichero con extensión *.gmftool*.

d) Correspondencias y mapping: Todo lo anteriormente definido cobra sentido en este paso mediante el mapping o mapeado. Este proceso consiste en relacionar los elementos del meta-modelo con su definición sintáctica gráfica y ésta a su vez con un elemento de la paleta de herramientas. La extensión del fichero que guarda estas relaciones es *.gmfmap*.

A partir del fichero anterior se obtiene el fichero *.gmfgen,* que permite generar el código que, una vez ejecutado, contiene el editor gráfico.

En la Figura 2 se observan las distintas partes que tiene un editor GMF una vez ejecutado. El espacio central es el denominado "Canvas". En él se pueden arrastrar y depositar los elementos de la paleta de herramientas, que se encuentra en la parte derecha de la imagen. Así, en este ejemplo, un usuario puede, a través de la creación de un diagrama, especificar una coreografía de movimientos para un robot humanoide. Se aprecia la total personalización de los elementos a los que se les ha dado colores, formas e iconos distintivos.

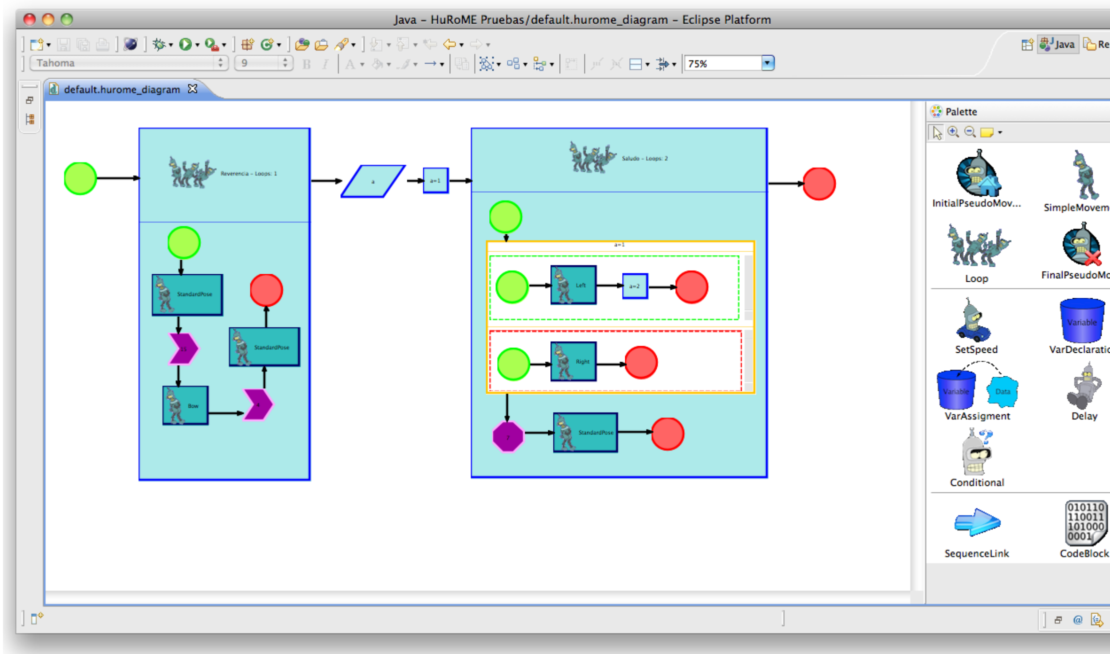

Figura 2: Editor gráfico "HuRoME" para la definición de coreografías de robots humanoides

CAPÍTULO 3º

Trabajos previos: V3CMM

En este capítulo se aborda de forma breve qué es V3CMM [1], para qué sirve y cómo funciona. Los apéndices adjuntos al final de esta memoria permiten profundizar en los detalles del lenguaje V3CMM.

3.1. V3CMM

V3CMM (3-View Component Meta-Model) es un lenguaje de modelado basado en componentes que permite el diseño de aplicaciones de forma independiente de la plataforma y el dominio concreto de la misma. La adopción de un enfoque de Desarrollo de Software Dirigido por Modelos (DSDM) facilita la transformación de los diseños modelados con V3CMM en otros modelos (por ejemplo, para análisis o simulación) o incluso en código.

Para ello, hace uso de tres vistas más una, que sirve de cohesión entre las anteriores, dichas vistas se hacen necesarias a la hora de definir la aplicación a diferentes niveles.

3.2. ¿Tres vistas? ¿Por qué?

V3CMM define tres vistas complementarias, altamente cohesionadas y débilmente acopladas:

1) una vista estructural, para describir la estructura estática de los componentes, tanto simples como compuestos, que integran la arquitectura de la aplicación.

2) una vista de coordinación, basada en máquinas de estados, para modelar el comportamiento de los componentes frente a los mensajes que reciben del exterior.

3) una vista algorítmica, basada en diagramas de actividad, que permite describir los algoritmos que ejecuta cada componente dependiendo del estado en que se encuentre. En la versión de V3CMM empleada en el presente Proyecto no se hizo uso de ésta tercera vista.

Existe una cuarta vista en la que se definen los elementos comunes a las tres anteriores como son los tipos de datos o las interfaces.

En la Figura 3 se resume la base de funcionamiento de V3CMM. Se observa cómo se relacionan cada una de las vistas entre sí de forma esquemática y por qué se hace necesario el uso de cada una de ellas.

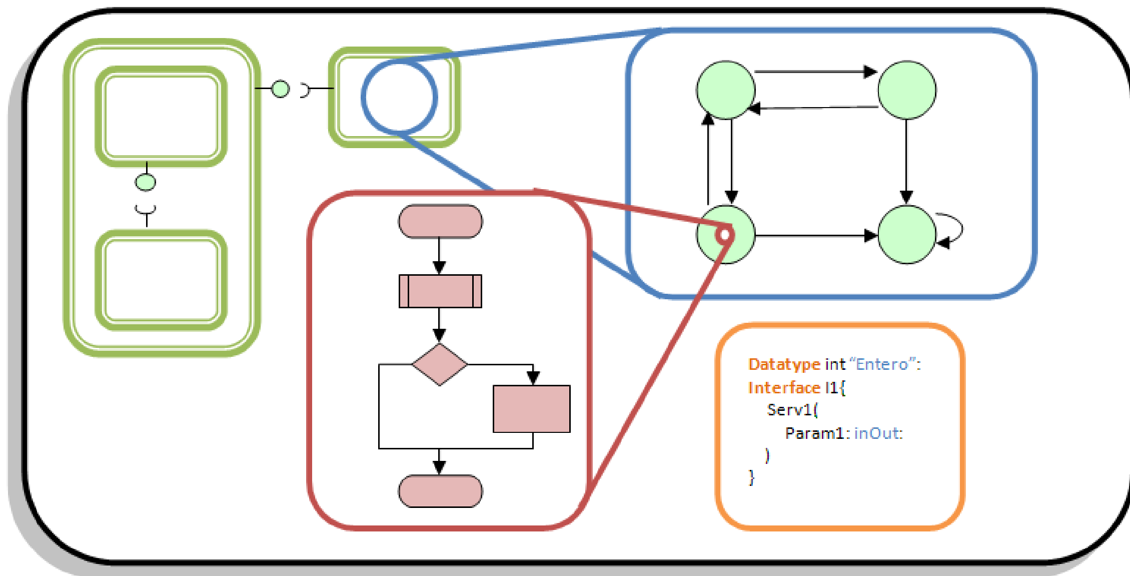

Figura 3: Funcionamiento esquemático de V3CMM

Los elementos en verde son componentes que se definen desde la vista estructural (vista 1), dicha vista permite también definir los puertos y conectores de cada componente. "Haciendo zoom" en uno de estos componentes, llegamos a la vista de coordinación (vista 2), marcada en color azul, que modela el comportamiento del componente mediante una máquina de estados, lo que permite ejecutar algoritmos adecuados a cada mensaje recibido desde el exterior. Adentrándose de nuevo en cada estado de la máquina, encontramos la vista algorítmica (vista 3), en color rojo, que, como su propio nombre indica, define el algoritmo a seguir mediante diagramas de actividad. El recuadro naranja simboliza la cuarta vista, en ella se definen tipos de datos e interfaces comunes a las vistas anteriores, que acceden a ellos haciendo uso de la sentencia *import*.

Además de las mencionadas, también se puede encontrar en el meta-modelo una nueva vista, la de "aplicación". En dicha vista se recoge, como su nombre indica, la aplicación. En el esquema de la página anterior es representada simbólicamente como el cuadro negro que enmarca todo.

Así pues, lo que en un principio parecían ser 3 vistas en realidad son 5, de las cuáles en el presente Proyecto se han implementado 4: la vista de componentes, la de interfaces y tipos de datos, la de máquinas de estado y la de aplicación.

3.3. Meta-modelo V3CMM

El meta-modelo de V3CMM es un gran conglomerado de clases relacionadas entre sí. En dicho meta-modelo se encuentran todas las vistas que componen el lenguaje, caracterizadas por un elemento contenedor, o root, único para cada una, que hereda de un elemento raíz general denominado ROOT, de forma que todas puedan tener acceso a una clase común que luego se convertirá en una sentencia indispensable a la hora de generar aplicaciones, el import. El nombre que recibe cada uno de los contenedores de las diferentes vistas es ROOT_XXX, donde XXX es la extensión del fichero que se genera con esa vista.

1) ROOT_IDT es el elemento raíz de la vista de tipos de datos e interfaces, mostrada en la Figura 4. Como es de esperar, contiene DataTypes e Interfaces. Estas últimas contienen a su vez servicios, que son contenedores de parámetros.

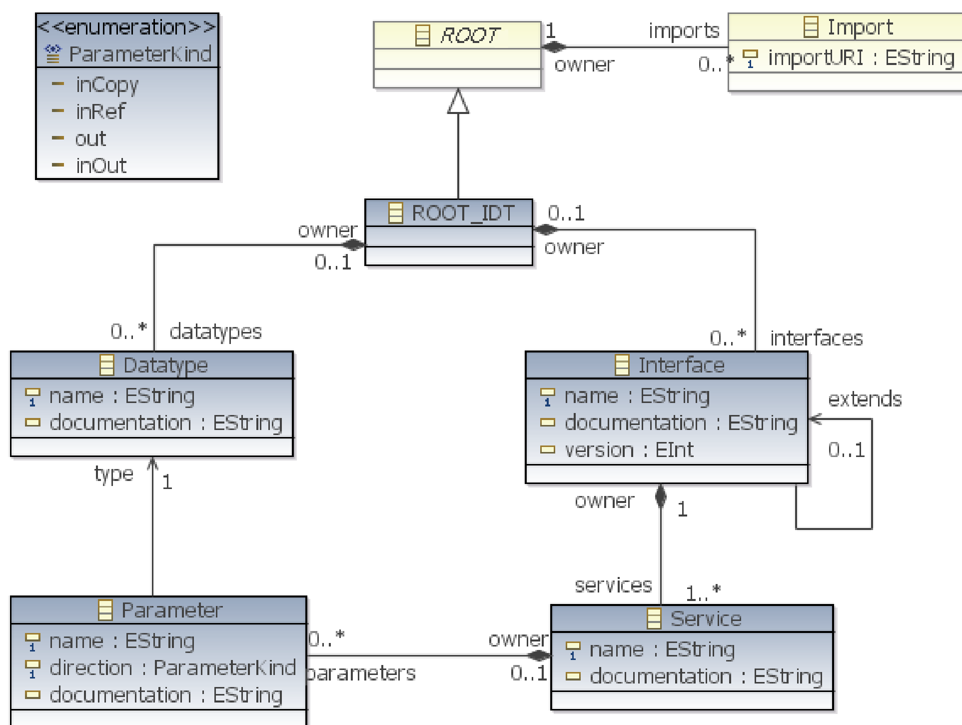

Figura 4: Parte del meta-modelo destinada a la vista de interfaces

2) ROOT_CDR es el elemento principal de la vista de componentes o estructural, modelada con la parte del meta-modelo que se muestra en la Figura 5. Este elemento contiene la definición e instancia de los componentes, que pueden ser de tipo Simple, Complex o BlackBox. La definición de un componente contiene la definición de sus puertos, y a su vez la instancia de componente contiene instancias de puertos. Si el componente definido es complejo, además contendrá Portlinks, que se utilizan para interconectar componentes internos. Dichos links pueden ser de tipo Assembly, si conectan dos puertos en el mismo nivel de composición, o de tipo Delegation, si conectan un puerto con otro definido en su componente padre. Esta vista se relaciona con la vista de máquinas de estado mediante la referencia de una máquina con el componente que la contendrá y con la vista de tipos de datos mediante la referencia de la definición de puertos con las interfaces que requerirá/proveerá dicho puerto.

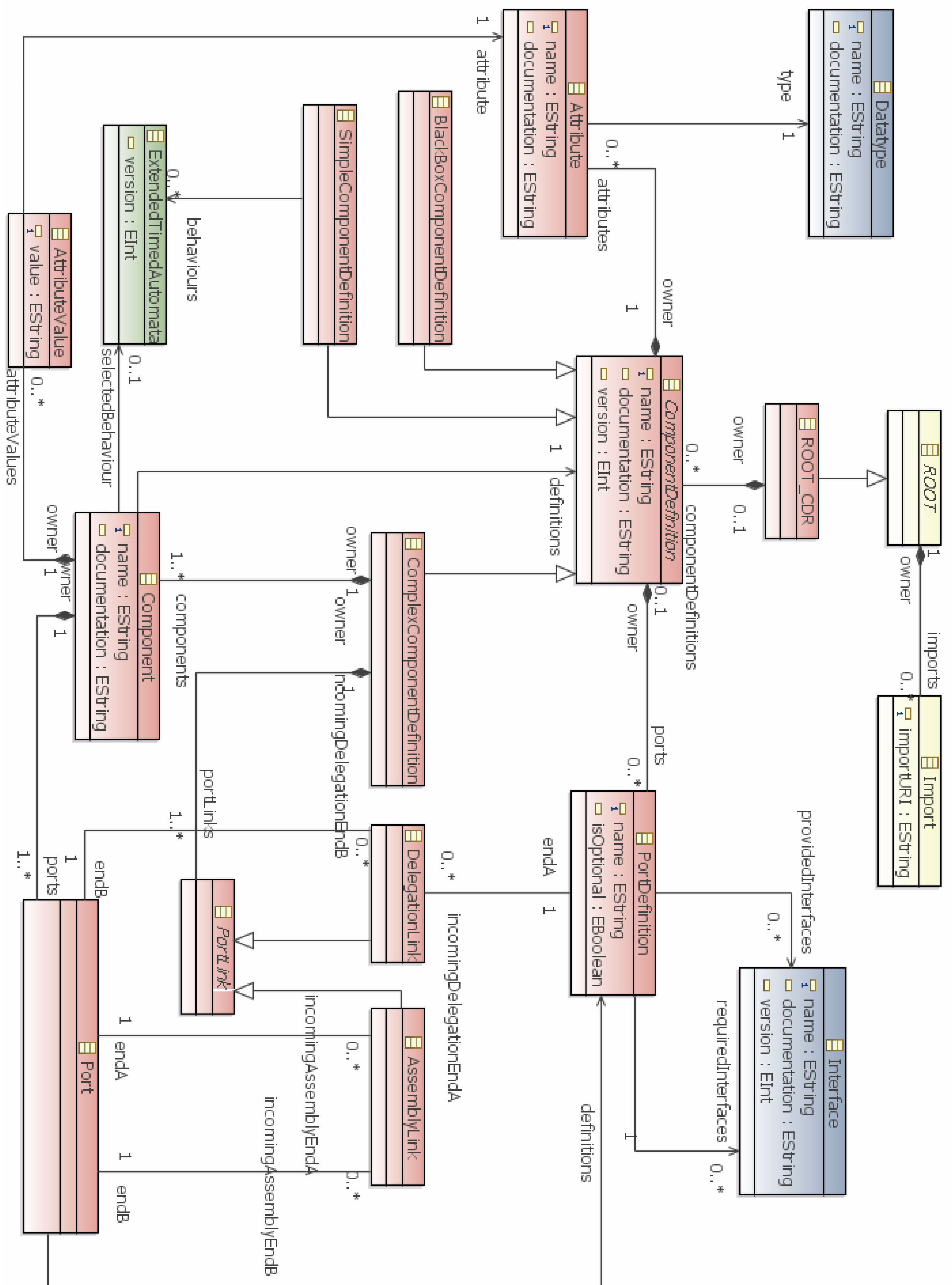

Figura 5: Parte del meta-modelo destinada a la vista de componentes

3) ROOT_APP es el elemento raíz de la vista de aplicación, cuyo meta-modelo se puede observar en la Figura 6. Dicho root sólo contiene la definición de un componente compuesto sin puertos. Desde esta vista se crea la aplicación a través de la especificación del sistema que estará conformado por un conjunto de componentes convenientemente interconectados para llevar a cabo la funcionalidad requerida por la aplicación. En este sentido, por lo general, los ComplexComponentDefinition de las aplicaciones no son unidades reutilizables, a diferencia de los definidos desde la vista estructural.

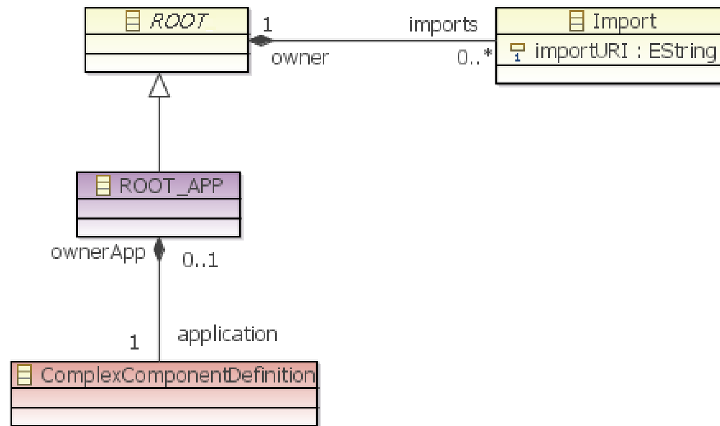

Figura 6: Parte del meta-modelo destinada a la vista de aplicación

4) ROOT_XTA es el elemento contenedor de la vista de coordinación o máquinas de estado, cuyo modelado se corresponde con el de la Figura 7. Contiene un autómata temporizado que a su vez puede contener regiones. Dentro de dichas regiones se pueden encontrar diferentes tipos de estados (State, InitialPseudoState, FinalState,…) y transiciones, que son disparadas por los eventos a los que referencian. Esta vista se relaciona con la de tipos de datos mediante una referencia desde un evento de carácter externo con un servicio que lo active.

Como se observa, en el meta-modelo descrito no se ha modelado la vista algorítmica basada en diagramas de actividad, ya que los directores consideraron oportuno dejarla a un lado y que el presente Proyecto se centrase en las demás vistas.

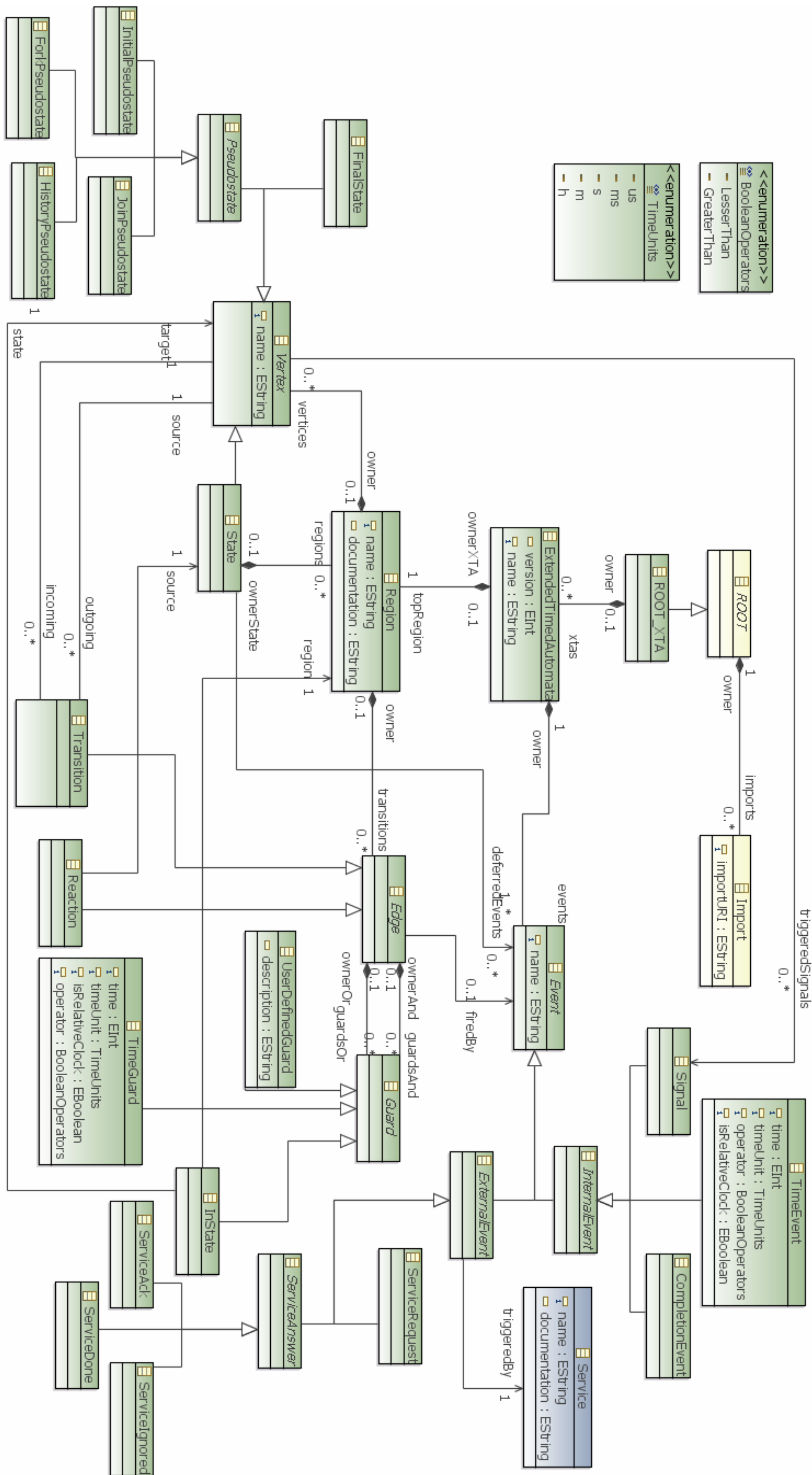

Figura 7: Parte del meta-modelo destinada a la vista de máquinas de estado

3.4. Restricciones OCL

El OCL (Object Constraint Language, lenguaje de restricciones de objetos) [12] es un lenguaje que permite a los desarrolladores de software escribir restricciones sobre modelos de objetos. Las posibilidades son muchas, OCL permite introducir condiciones a la relación entre clases del meta-modelo, valores previos, restricciones sobre operaciones, etc. Gracias a estas restricciones en lenguaje OCL el desarrollador es capaz de crear un amplio conjunto de invariantes que completan la sintaxis abstracta del lenguaje definida en el meta-modelo.

En el presente Proyecto se ha completado el meta-modelo de V3CMM con una serie de restricciones OCL que impiden el uso incorrecto del lenguaje y que, por lo tanto, facilitan la labor del desarrollador, que se beneficia del validador de modelos (compuesto, en parte, por las restricciones OCL) para crear especificaciones correctas.

La definición de restricciones es simple. Se comienza con la palabra reservada invariant y a continuación el nombre que se le desea dar a la restricción, dicho nombre será el mensaje que imprima Eclipse cuando se incumpla. La sintaxis de OCL es de muy alto nivel, y resulta sencillo escribir restricciones siguiendo el propio lenguaje.

A continuación, se presenta una restricción significativa a modo de ejemplo. Las restantes restricciones implementadas se encuentran detalladas al final del documento, en un apéndice destinado a tal fin.

Restricción: *Una interfaz no puede tener servicios repetidos. Dos servicios son iguales si se llaman igual y tienen los mismos parámetros.*

invariant *repeatedService:*

self*.services -> forAll (s1, s2 : Service | s1 <> s2* ***implies*** *s1.name <> s2.name* ***and*** *s1.parameters -> forAll (j : Parameter | s2.parameters -> forAll (k : Parameter | k.name<> j.name* ***or*** *k.direction <> j.direction)));*

Todas las restricciones se implementan desde el editor OCL, que presenta el siguiente aspecto:

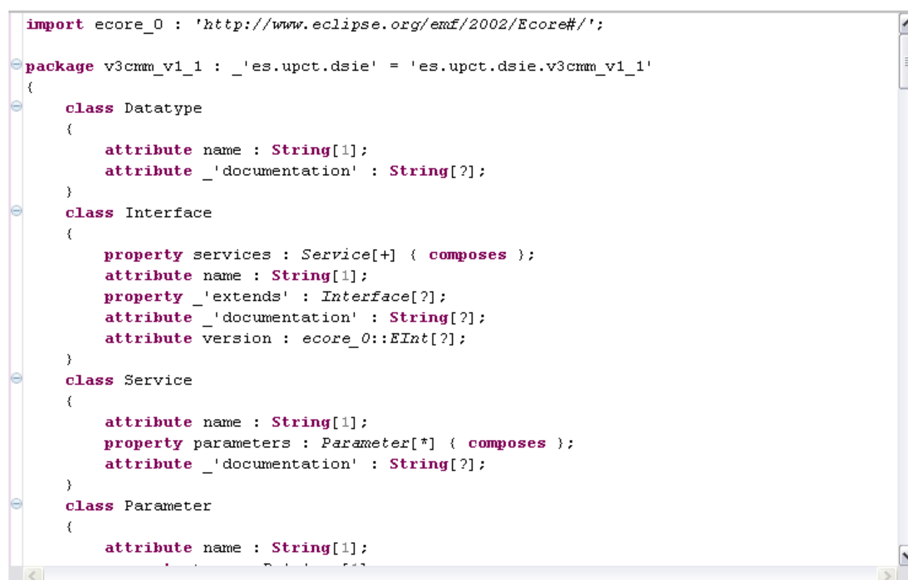

Figura 8: Editor OCL de Eclipse

CAPÍTULO 4º

Desarrollo de los editores

En el siguiente capítulo se detallan todos los pasos seguidos para conseguir la implementación de los editores, así como los pasos para introducir las mejoras en los mismos, como son la coloración de la sintaxis o la auto-compleción del esquema. La primera sección del capítulo justifica los retos del trabajo desarrollado en el presente Proyecto y se expone cómo Eclipse y sus herramientas se involucran en el desarrollo realizado.

4.1. Retos del desarrollo

Actualmente, existen líneas de investigación que trabajan en mecanismos para elevar el nivel de abstracción del proceso de desarrollo de software basado en componentes. En este sentido, el objetivo principal de éstos editores es el de acercar los beneficios del DSDM a este paradigma.

La posibilidad de tener unos editores especializados a un lenguaje concreto hace que la labor de trabajar con los mismos sea sencilla y ventajosa. No obstante, tradicionalmente el desarrollo de las herramientas de soporte de un lenguaje ha supuesto un alto coste. Eclipse y los plug-ins para DSDM ofrecen los mecanismos para reducir dicho coste y generar de forma automática los editores básicos del lenguaje. Así, surge un beneficio en dos sentidos, por un lado, se facilita el desarrollo de nuevos lenguajes y, por otro, los usuarios reciben los medios para realizar su trabajo de forma eficiente.

A continuación, se exponen las principales capacidades que deberán tener los editores, y cómo las herramientas de Eclipse hacen frente a los retos que ellas suponen.

a) Uno de los objetivos es diseñar una herramienta útil para la definición de aplicaciones conformes al lenguaje V3CMM. Para ello se hace necesaria la especificación de la sintaxis abstracta del lenguaje y una serie de reglas dirigidas a restringir los modelos definidos en base al lenguaje, ya que no todos tienen por qué ser válidos.

Como ya se ha visto, la herramienta EMF permite definir meta-modelos que describen la sistaxis abstracta del lenguaje. En base a dicha descripción se pueden añadir restricciones OCL que, como se vio en el apartado 3.4, restrinjan y complementen el uso del lenguaje para obtener modelos validados.

b) Creación de los editores en una plataforma portable como Java, además de poder introducir mejoras en los mismos de la forma más rápida, cómoda y eficaz. Se hacen entonces necesarias herramientas optimizadas para tal fin, que quiten complejidad al trabajo del desarrollador, ya que la implementación de un editor desde cero es siempre costosa.

Eclipse proporciona la herramienta Xtext que, como se ha dicho en el apartado 2.2, permite crear editores textuales a partir de meta-modelos EMF en un tiempo relativamente corto y con una facilidad en la labor aceptable. Y no sólo eso, gracias a la gran capacidad de personalización de la herramienta se puede añadir mejoras al editor creado, como son la coloración de sintaxis, la auto-compleción del esquema, la adaptación de los componentes al ámbito de definición, etc. Además, Xtext genera la herramienta en código Java, lo que permitirá al desarrollador ejecutar el editor en cualquier entorno de trabajo con Java disponible.

4.2. Esquemas sintácticos sugeridos para cada una de las vistas

Se indicarán en primer lugar las características generales de cada uno de los editores según la vista con la que se corresponde, más tarde se describirá cómo es el esqueleto o plantilla de la sintaxis de cada una de las vistas.

1) Vista de Tipos de Datos e Interfaces (*Interfaces & DataTypes*):

- La extensión que tendrán los ficheros de texto que modelen ésta vista será **.idt.**

- El elemento raíz de ésta vista será la clase del meta-modelo llamada *ROOT_IDT*.

- Un fichero de éste tipo podrá contener:

  - Cero o más cláusulas *import*

  - Cero o más declaraciones de tipos de datos (*Datatype*) e interfaces (*Interface*)

2) Vista de Componentes o Estructural (Component Definition Repository):

- La extensión de los modelos textuales de ésta vista será **.cdr.**

- El elemento principal será la clase *ROOT_*CDR del meta-modelo.

- Los ficheros con ésta extensión podrán contener:

  - Cero o más cláusulas *import*
  - Cero o más definiciones de componentes de tipo blackbox (*BlackboxComponentDefinition*),
  - Cero o más definiciones de componentes simples (*SimpleComponentDefinition*)
  - Cero o más definiciones de componentes compuestos (*ComplexComponentDefinition*)

3) Vista de Máquinas de Estado o de Coordinación (Extended Timed Automata):

- La extensión de los ficheros asociados es **.xta.**

- El elemento contenedor será ROOT_XTA.

- Los ficheros de la vista de coordinación contienen:

  - Cero o más cláusulas *import*
  - Las definiciones de los eventos que es capaz de procesar la máquina
  - Una región ortogonal (*Region*) que contendrá estados, pseudo-estados, transiciones y reacciones. Estos estados podrán contener, a su vez, 0 o más regiones con estados, pseudo-estados, transiciones y reacciones.

4) Vista de Aplicación (Application):

- Los ficheros para modelar ésta vista tendrán extensión **.app.**

- El elemento root será *ROOT_APP*.

- Este tipo de ficheros contienen:

  - Cero o más cláusulas *import*
  - Una única definición de un componente compuesto, cuya estructura es idéntica a la descrita en la vista estructural, pero con la salvedad de que no tendrá definiciones de puertos.

Se muestran a continuación los esquemas que definen la sintaxis concreta textual planteada para cada una de las vistas. A través del color de las palabras reservadas pueden identificarse cada una de las vistas del lenguaje.

1) Vista de Tipos de Datos e Interfaces:

```
import importURI_1;
…
import importURI_N;

datatype name_dt1 "documentation";
…
datatype name_dtN "documentation";

interface name_i1 extends refInterface  version v  "documentation"
{
      name _i1s1 "documentation" (
            name_i1s1p1 : direction refDataType "documentation",
            …
            name_i1s1pN  : direction refDataType "documentation"
      )
      …
      name _i1sN "documentation" (
            name_i1sNp1 : direction refDataType "documentation",
            …
            name_i1sNpN  : direction refDataType "documentation"
      )
}
…
```

Bloque 2: Esqueleto del editor de la vista de interfaces y tipos de datos

2) Vista de Componentes o Estructural

```
import importURI_1;
…
import importURI_N;

blackboxCD name_bbcd1 version v  "documentation" {
      ports {
      name _bbcd1p1
            -c  refInterface_req1, …, refInterface_reqN;
            -o  refInterface_prov, …, refInterface_provN;
      } …
} …
simpleCD name_scd1 version v  "documentation" behaviours
refExtendedTimedAutomata_xta1 , … ,
refExtendedTimedAutomata_xtaN  {
      ports {
      name _bbcd1p1
            -c  refInterface_req1, …, refInterface_reqN;
            -o  refInterface_prov, …, refInterface_provN;
      }
      …
      attributes {
       name _scd1a1 : refDatatype "documentation";
      …}
}
```

Bloque 3.1: Esqueleto del editor de la vista de componentes

```
ComplexCD name_ccd1 version v  "documentation" {
      ports {
      name _bbcd1p1
            -c  refInterface_req1, …, refInterface_reqN;
            -o  refInterface_prov, …, refInterface_provN;
      }
      …
      attributes {
       name _scd1a1 : refDatatype "documentation";
      …
      }
      component name_ccd1ic1 : refComponentDefinition
"documentation" selectedBehaviour refExtendedTimedAutomata_xta1  {
            ports {
                  name_ccd1icp1: refPortDefinition;
                  …
            }
         attributeValues {
            refAtribute = value;
            …
         }
       }
       …
      portLinks {
         compX.portA <-> compY.portB;
         …
         this.portA <-> compY.portB;
    }
}
…
```

Bloque 3.2: Esqueleto del editor de la vista de componentes

3) Vista de Máquinas de Estado o de Coordinación

```
import importURI_1;
…
import importURI_N;

timedAutomata name_xta1 version v  {
      events {
            providedService_Requests sr1,…,srN;
            internalSignals s1, …, sN;
            completionEvent ce1, …, ceN;
            timeEvents te1 (o  t  tU  r), …, teN (o  t  tU r);
            requiredService_Acks refServ1, …, refServN;
            requiredService_Ignored refServ1, …, refServN;
            requiedService_Done refServ1, …, refServN;
      }
```

Bloque 4.1: Esqueleto del editor de la vista máquinas de estado

```
        region name_region "documentation" {
                initial name_initial; | history name_history;
                state name_s1   {
                        region name_s1region1 "documentation" {
                                initial name_initial; | history
            name_history;
                                state name_s1region1s1 {
                                …
                                }
                        }
                        …
                        deferredEvent defEve1, …, defEveN;
                }
                forks f1, …, fN;
                joins j1, …, jN;
                finalStates fs1, …, fsN;
                transitions refSourceVertex -> refTargetVertex
                @   refEvent   [ AND a1, …, aN OR o1,…, oN ];
                …
                reactions refSourceState
                @   refEvent   [ AND a1, …, aN OR o1,…, oN ];
                …
        }
}
…
```

Bloque 4.2: Esqueleto del editor de la vista máquinas de estado

4) Vista de Aplicación

```
import importURI_1;
…
import importURI_N;

application name_app version v  "documentation"
{
        //esqueleto del ComplexCD
}
```

Bloque 5: Esqueleto del editor de la vista de aplicación

4.3. Descripción del procedimiento para la creación de los editores

A continuación, se describen detalladamente los pasos a seguir para la realización de los editores haciendo uso de las herramientas Eclipse.

1) Antes de empezar con los editores, se debe crear un proyecto de tipo EMF para albergar el meta-modelo del lenguaje. Este primer proyecto es la base de los posteriores, de tipo Xtext, que contendrán, ahora sí, los editores.

Estando en el espacio de trabajo de Eclipse se va a *File* > *New* > *Other*, o bien se usa el atajo desde teclado *Control* + *n*. Se abrirá una ventana donde Eclipse recoge todos los tipos de proyecto y fichero que se pueden crear. Como se ha dicho, en primer lugar se debe crear un *Empty EMF Project,* que se encuentra dentro de la carpeta *Eclipse Modeling Framework*. En la

Figura 9 se muestra dónde encontrar esta opción. Click en *Next* y llega el momento de dar nombre al proyecto.

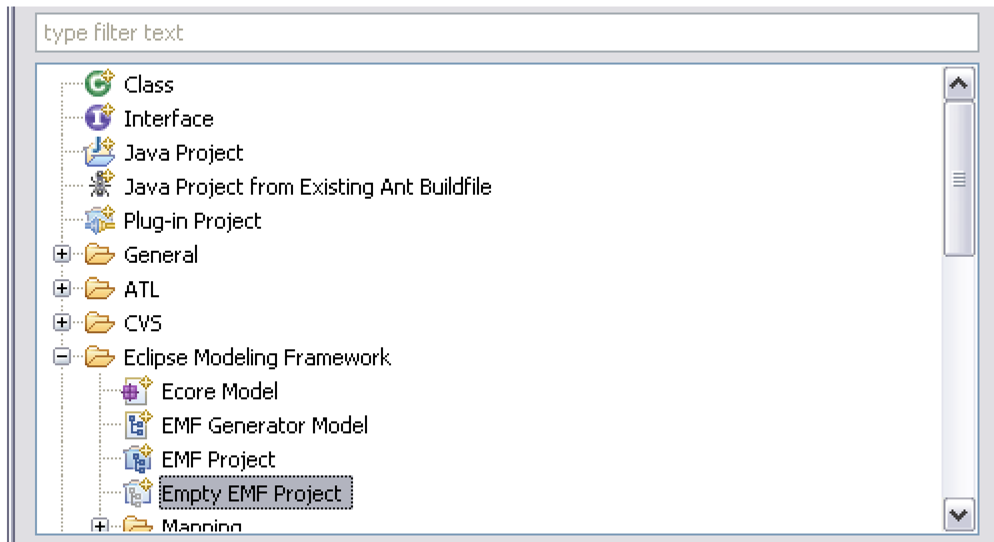

Figura 9: Captura de la ventana que contiene todos los tipos de proyecto

2) Este paso es opcional, pero aconsejable: Dentro del nuevo proyecto se crea una carpeta (*File* > *New* > *Folder*) con el nombre MetaModel. En esta carpeta se guardará el meta-modelo y todos los archivos relacionados con él, de esta forma la tarea de localizarlo siempre será sencilla, cómoda y rápida.

Ahora se debe crear el meta-modelo que sirve de base a los editores. El empleado en el presente Proyecto ya estaba implementado con anterioridad, por tanto sólo ha sido necesario copiarlo en la carpeta MetaModel. Si no estuviera ya definido, se debe crear un fichero *Ecore Model* desde cero. Este fichero puede editarse de tres formas diferentes: con un editor en árbol (*tree editor*), un editor de texto o un editor gráfico. En la Figura 10 se refleja un ejemplo de cada uno de los editores anteriores. Para meta-modelos de reducido tamaño es aconsejable utilizar el editor gráfico, para meta-modelos del tamaño de V3CMM lo más apropiado es utilizar el editor en árbol. El editor textual debería ser sólo para usuarios avanzados.

Para editar/visualizar el meta-modelo en el editor en árbol, se debe desplegar el menú contextual sobre el fichero *.ecore* y seleccionar la opción *Open With* > *Sample Ecore Model Editor*. Para editarlo en modo textual hay dos opciones:

a) Seleccionando la opción *Text Editor*, que abrirá un editor basado en modelos UML.

b) Seleccionando la opción *OCLinEcore,* que abrirá el editor OCL, que, como es de suponer, no sólo sirve para introducir restricciones al meta-modelo, sino que también se puede usar como editor para definirlo.

Si lo que se quiere es utilizar el editor gráfico, también existen dos opciones:

a) Crear un fichero con la extensión *.ecorediag,* que forma parte de las herramientas de EMF Tools.

b) Crear un fichero con la extensión *.ecore_diagram*, que hace uso de un editor GMF simplificado.

Sendos archivos se crean de la misma forma: menú contextual en el fichero *.ecore* y seleccionar *Initialize Ecore Diagram File* o *Initialize ecore_diagram diagram file* respectivamente.

Antes de trabajar con el meta-modelo debe ser validado por EMF para comprobar que, por ejemplo, ninguna relación ha quedado "en el aire" o que todos los atributos han sido nombrados correctamente, etc. Para ello, se abre el menú contextual en el primer desplegable del meta-modelo y se hace click en "Validate". Si todo ha ido bien debería aparecer un mensaje de aprobación.

En este momento se es capaz de crear modelos en base al meta-modelo desde el editor en árbol. Para ello basta con abrir el meta-modelo y seleccionar la opción *Create Dynamic Instance* del menú contextual que aparece al hacer click en un elemento raíz.

3) Una vez el meta-modelo está descrito, se pueden introducir las restricciones OCL que harán que el lenguaje sea más robusto y útil. Como se mencionó en el apartado 3.4, deben introducirse desde el *OCLinEcore.* En el Apéndice 2: "Restricciones OCL Adicionales" se encuentran detalladas y explicadas todas las restricciones añadidas al meta-modelo de V3CMM.

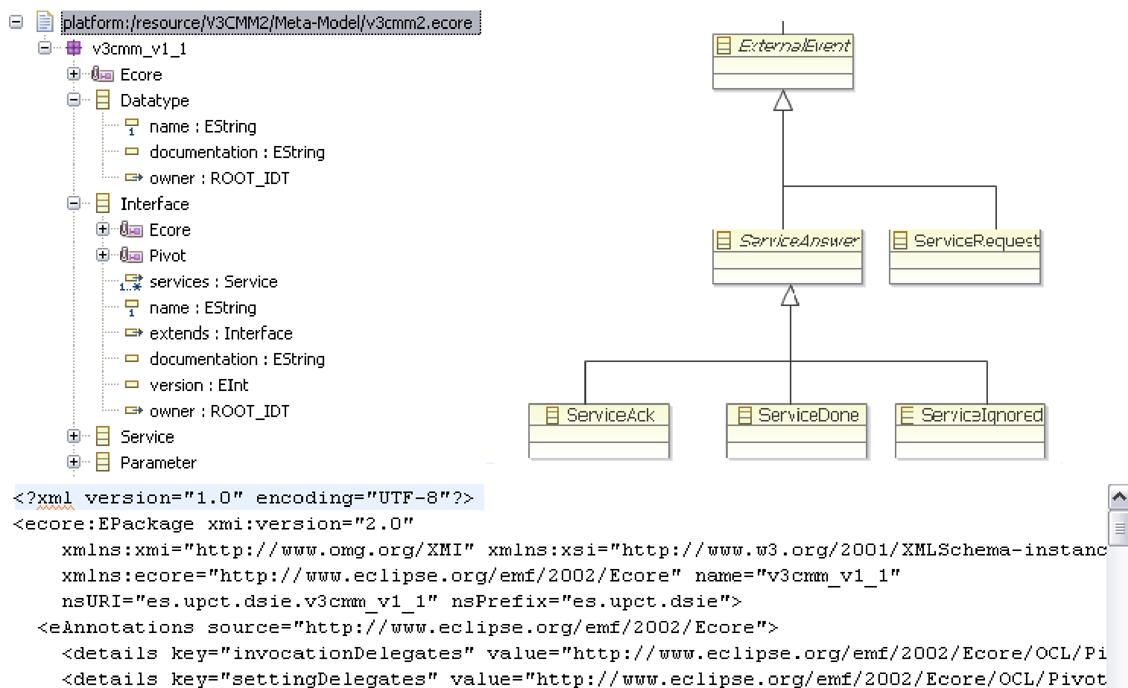

Figura 10: Tree Editor (arriba izda). Editor gráfico (arriba dcha). Editor textual (abajo)

4) Es el momento de crear el archivo *.genmodel,* a partir del cual se generarán los ficheros necesarios para la creación de los editores en base al meta-modelo. *Control + n*, se abre la ya

conocida ventana de los diferentes tipos de proyecto y archivo y se selecciona la opción *EMF Generator Model,* dentro de la carpeta *Eclipse Modeling Framework.* El siguiente paso es darle nombre al archivo y seleccionar la carpeta donde se creará. Es recomendable crearlo en la misma carpeta que el meta-modelo y nombrarlo igual. Eclipse muestra ahora una lista de tipos de fichero, en ella se selecciona el tipo *Ecore Model*, ya que se quiere generar a partir de un meta-modelo definido así. El último paso es seleccionar el meta-modelo desde el espacio de trabajo.

5) Para generar los ficheros Java necesarios se abre el *.genmodel* que acaba de crearse. Se hace click derecho sobre el primer elemento que aparece y se selecciona la opción *Generate All.*

6) El siguiente paso es crear los editores. Se debe crear entonces un proyecto de tipo Xtext para cada uno. *File* > *New* > *Other* y se selecciona la opción *Xtext Project From Existing Ecore Models*, que se encuentra en la carpeta *Xtext*. La siguiente ventana es muy importante, se debe seleccionar en primer lugar el fichero *.genmodel* generado a partir del meta-modelo y luego seleccionar el elemento raíz del editor, en este caso, tenemos cinco roots: el general y cuatro que heredan de él, uno por cada editor. Seleccionamos el que se corresponde con el editor que se esté creando. El siguiente paso es opcional a la hora de crear un editor, pero en el presente Proyecto se hace totalmente necesario y obligatorio: modificar la extensión que tendrán los ficheros que se generarán con este editor. La extensión por defecto es *.mydsl*, si se quiere modificar se debe escribir la extensión deseada en los campos necesarios. Además, también es posible modificar el nombre completo del proyecto tal y como muestra la Figura 11.

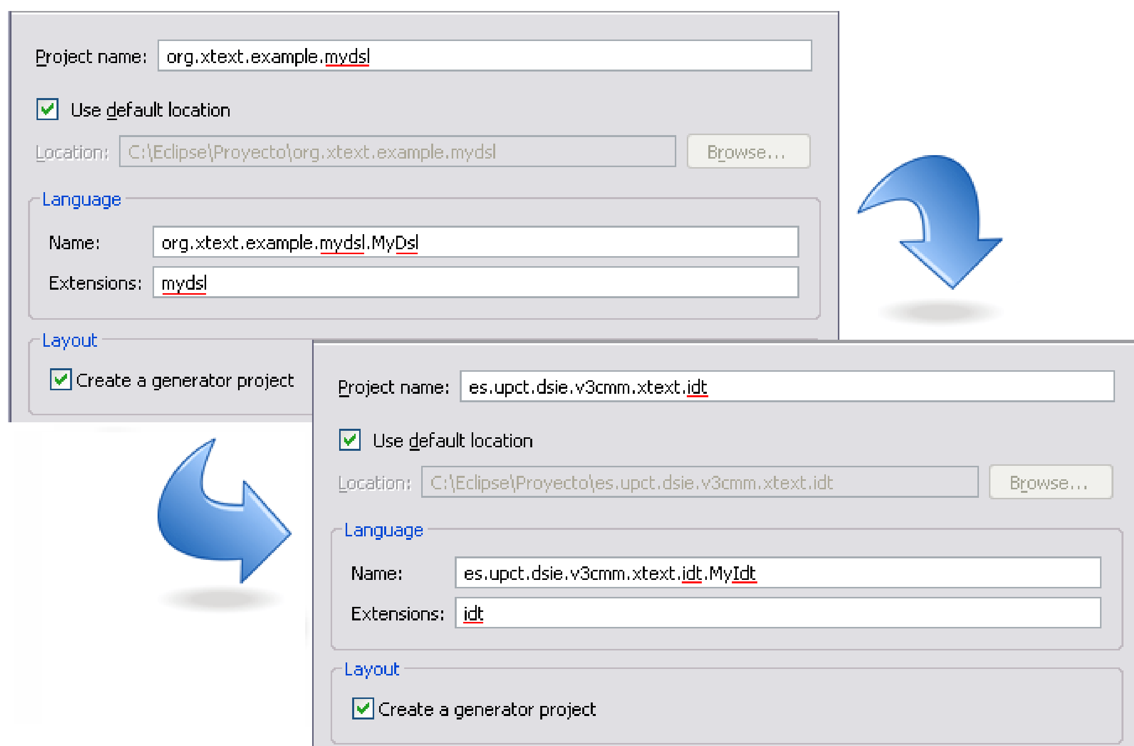

Figura 11: Nueva extensión .idt y nombre de proyecto para la vista de tipos de datos e interfaces

7) Hecho esto, Xtext genera tres plug-ins distribuidos en tres carpetas diferentes pero que pertenecen al mismo proyecto, tal y como se aprecia en la Figura 12:

a) La primera de las tres carpetas (*es.upct.dsie.v3cmm.Xtext.idt*) es el plug-in principal. Es de tipo Xtext y es en la que se guarda la gramática del editor, el generador de código Java y todas las clases importantes para ejecutarlo.

b) La segunda es el plug-in del editor textual (*es.upct.dsie.v3cmm.Xtext.idt.ui*). Es de tipo EMF y contiene las preferencias del interfaz de usuario, en las clases que contiene se introducirán más adelante las modificaciones oportunas para cambiar, por ejemplo, la coloración de palabras reservadas en la sintaxis.

c) La última carpeta (*es.upct.dsie.v3cmm.Xtext.idt.generator*) es el plug-in generador M2T (*Model to Text*) también de tipo EMF. Es la que sirve para generar el editor textual a partir de modelos.

Se despliega el plug-in principal, se entra en *src* y luego en el paquete con icono blanco y cuyo nombre coincide con el del proyecto. Ahí se guarda la sintaxis concreta del editor (fichero con extensión *.Xtext*) y el Workflow (con extensión *.mwe2*), que es el encargado de generar el código Java del editor en base a la gramática definida.

Xtext habrá generado automáticamente el archivo MyXXX.Xtext, donde XXX es la extensión definida en el paso 6, con una sintaxis acorde al meta-modelo. Dicho fichero debe ser modificado para ajustar la sintaxis del editor a la requerida. En el apartado 2.2 se puede ver el aspecto del fichero *MyIdt.Xtext*.

8) Cuando se hace uso de un meta-modelo con restricciones OCL, ha de modificarse el parámetro *registerForImportedPackages*, que por defecto está a **true,** en el Workflow. Esto evita que Xtext registre un meta-modelo ya registrado por EMF, de lo contrario, aparecería el siguiente mensaje de error al lanzar el editor: "*An object may not circulary contain itself*", que significa que el Root intenta validarse así mismo más de una vez.

9) Ya se puede generar el código Java para el editor. Se despliega el menú sobre el Workflow y se ejecuta con la opción *Run As > MWE2 Workflow*. Eclipse pide permiso para acceder a Google Guice. Una vez se le da, crea los ficheros de código Java.

Para crear nuevos editores cuya base es el mismo meta-modelo se repite el proceso desde el paso número 6, si se desea crear un editor con un meta-modelo distinto se debe hacer todo el proceso desde cero.

Siguiendo los pasos descritos, se obtienen unos editores por defecto, es decir, no poseen coloración de sintaxis distintiva, ni auto-compleción del esquema, etc. La forma de añadir todas éstas funcionalidades se trata en el siguiente punto de la memoria.

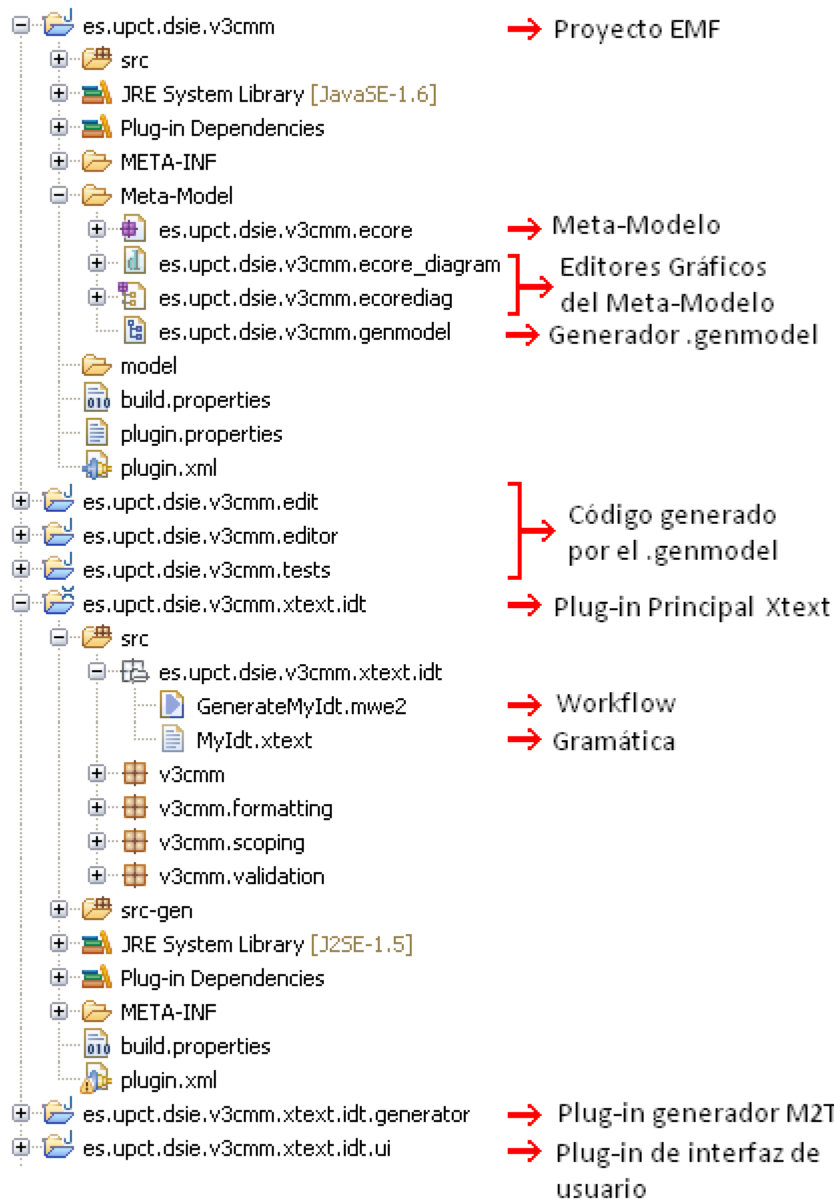

Figura 12: Aspecto del espacio de trabajo tras generar el primer proyecto Xtext

4.4. Funcionalidades añadidas a los editores

En este punto se va a explicar cómo se introducen las mejoras en los editores textuales Xtext.

4.4.1. Coloración de sintaxis

Para poder discriminar con facilidad el tipo de editor que estamos usando se ha implementado coloraciones diferentes en las palabras reservadas de cada uno de los editores. En el punto 4.1 puede observarse que el esqueleto o plantilla de cada editor presenta un color diferente.

Para modificar el color de la sintaxis en un editor Xtext se debe crear una clase que implementará una interfaz ya definida en Eclipse, llamada *IHighlightingConfiguration*. Todas las clases que se deben modificar para llevar a cabo esta mejora pueden verse en la Figura 13.

1) Se va a la carpeta *src* del plugin de interfaz de usuario y allí se crea un nuevo paquete. En este caso se ha nombrado *v3cmm.syntaxcoloring*, siguiendo el esquema de nombres que usa Eclipse en el proyecto.

2) Se crea una nueva clase Java. El nombre de esta clase podría ser, por ejemplo, *MyHighlightingConfiguration.java*. La clase implementada se detalla en los Bloques 6.1 y 6.2.

```
public class MyHighlightingConfiguration implements IHighlightingConfiguration
{
	public static final String KEYWORD_ID = "keyword";
	public static final String PUNCTUATION_ID = "punctuation";
	public static final String COMMENT_ID = "comment";
	public static final String STRING_ID = "string";
	public static final String NUMBER_ID = "number";
	public static final String DEFAULT_ID = "default";
	public static final String INVALID_TOKEN_ID = "error";

	public void configure(IHighlightingConfigurationAcceptor acceptor) {
		acceptor.acceptDefaultHighlighting(KEYWORD_ID, "Keyword",
		keywordTextStyle());

		acceptor.acceptDefaultHighlighting(PUNCTUATION_ID, "Punctuation
		character", punctuationTextStyle());

		acceptor.acceptDefaultHighlighting(COMMENT_ID, "Comment",
		commentTextStyle());

		acceptor.acceptDefaultHighlighting(STRING_ID, "String",
		stringTextStyle());

		acceptor.acceptDefaultHighlighting(NUMBER_ID, "Number",
		numberTextStyle());

		acceptor.acceptDefaultHighlighting(DEFAULT_ID, "Default",
		defaultTextStyle());

		acceptor.acceptDefaultHighlighting(INVALID_TOKEN_ID, "Invalid
		Symbol", errorTextStyle());
		}
```

Bloque 6.1: Clase MyHighlightingConfiguration.java

```java
	public TextStyle defaultTextStyle() {
		TextStyle textStyle = new TextStyle();
		//textStyle.setBackgroundColor(new RGB(255, 255, 255));
		textStyle.setColor(new RGB(0, 0, 0));
		return textStyle;
	}

	public TextStyle errorTextStyle() {
		TextStyle textStyle = defaultTextStyle().copy();
		//textStyle.setColor(new RGB(255, 0, 0));
		return textStyle;
	}

	public TextStyle numberTextStyle() {
		TextStyle textStyle = defaultTextStyle().copy();
		textStyle.setColor(new RGB(125, 125, 125));
		return textStyle;
	}

	public TextStyle stringTextStyle() {
		TextStyle textStyle = defaultTextStyle().copy();
		textStyle.setColor(new RGB(42, 0, 255));
		return textStyle;
	}

	public TextStyle commentTextStyle() {
		TextStyle textStyle = defaultTextStyle().copy();
		textStyle.setColor(new RGB(63, 127, 95));
		return textStyle;
	}

	public TextStyle keywordTextStyle() {
		TextStyle textStyle = defaultTextStyle().copy();
		textStyle.setColor(new RGB(0, 0, 255));
		textStyle.setStyle(SWT.BOLD);
		return textStyle;
	}

	public TextStyle punctuationTextStyle() {
		TextStyle textStyle = defaultTextStyle().copy();
		return textStyle;
	}
}
```

Bloque 6.2: Clase MyHighlightingConfiguration.java

Para modificar el color se deben modificar los valores que hay en el paréntesis de “RGB”, siguiendo el código de colores de éste formato, dentro del método *keywordStyle*. En el ejemplo, se ha puesto como color el (0, 0, 255) que, según este código de colores, es el azul.

Existe la posibilidad, como se ve en el fichero, de modificar otros colores del editor como por ejemplo el de los comentarios (*commentTextStyle*), los números (*numberTextStyle*) o incluso el subrayado de las palabras erróneas (*errorTextStyle*).

Lo anteriormente explicado debe seguirse en cada uno los proyectos que conforman cada editor, escogiendo para cada uno un color.

Una vez se ha modificado dicha clase debemos registrarla para que Eclipse la tenga en cuenta a la hora de ejecutar el editor. Dicho registro debe hacerse en el módulo de interfaz de usuario, que se encuentra en el paquete con extensión ui de la carpeta *src*, dentro, a su vez, del plug-in de interfaz de usuario. Este módulo es una clase Java llamada *XxxUiModule.java*, donde Xxx es la extensión que usará el editor.

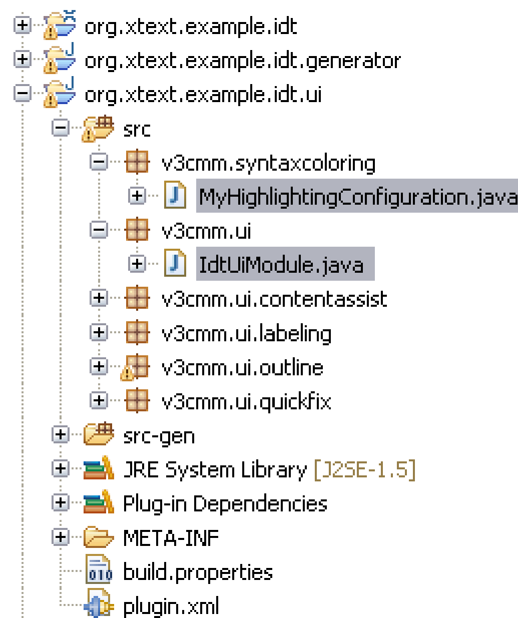

Figura 13: Clases que intervienen en la coloración de la sintaxis de los editores

El aspecto del fichero de registro es el del Bloque 7.

```
public class IdtUiModule extends v3cmm.ui.AbstractIdtUiModule {
	public IdtUiModule(AbstractUIPlugin plugin) {
		super(plugin);
	}

//Registro de la clase de coloración

	public Class<? extends IHighlightingConfiguration>
	bindIHighlightingConfiguration(){
		return MyHighlightingConfiguration.class;
	}
}
```

Bloque 7: Clase IdtUiModule.java para el registro de modificaciones

4.4.2. Auto-compleción del esquema

Ya que se conoce de antemano el esquema o esqueleto de la sintaxis de los editores, ¿por qué no generarlos automáticamente a medida que el usuario escribe? En esto consiste la siguiente mejora, completar de forma automática lo que el usuario escribe.

Para la implementación de ésta funcionalidad se debe crear un paquete para contener la clase al igual que con la coloración de sintaxis. Además, debe crearse en el mismo sitio, en la carpeta *src* del plug-in de interfaz de usuario. El nombre que se le ha dado al paquete ha sido *v3cmm.autoedit*. En él se ha creado la clase *MyAutoEdit.java* que debe implementar la interfaz *IAutoEditStrategy* para desempeñar la función requerida y que se muestra en el Bloque 8. En la Figura 14 pueden observarse todas las clases que intervienen en la auto-compleción del esquema.

```
public class MyAutoEdit implements IAutoEditStrategy {
      public void customizeDocumentCommand(IDocument document,
      DocumentCommand command) {

            try {
            // Patron 1: Si se escribe 'interface' seguido de
            //un identificador y se pone ' ' introduce 'extends'
                  if(command.text.equals(" ") &&
            isTokenEqualTo(document,command,"interface",1,true)){
                        command.text= " extends ";
                  }

            // Patron 2: La combinacion 'interface [nombre]
            //extends[referencia]' y se introduce ' ' añade 'version'
                  else if(command.text.equals(" ") &&
      isTokenEqualTo(document, command, "interface", 3, true) &&
            isTokenEqualTo(document, command, "extends", 1, true)) {
                        command.text = " version ";
                  }
            //Patron 3: 'interface [nombre] extends
            //[referencia] version [v]' añade las comillas
                  else if(command.text.equals(" ") &&
      isTokenEqualTo(document, command, "interface", 5, true) &&
      isTokenEqualTo(document, command, "extends", 3, true) &&
      isTokenEqualTo(document,command, "version",1,true)) {
                        command.text = " \"\"";
                  }

            //Patron 4: 'datatype [nombre] ' introduce comillas
                  else if(command.text.equals(" ") &&
      isTokenEqualTo(document,command,"datatype",1,true)){
                        command.text= " \"\";";
                  }

            }catch (BadLocationException e) {
                  e.printStackTrace(); }
      }
```

Bloque 8: Clase MyAutoEdit.java para la compleción del esquema

Como se ve en el código del Bloque 8, el método que completa el esquema se basa en la búsqueda de tokens de referencia haciendo uso del método *isTokenEqualTo*, cuya implementación se describe en el Bloque 10, y así decidir qué introducir. Por ejemplo, el esquema de un fichero .idt tiene un aspecto parecido al mostrado en el Bloque 9:

```
interface name_i1 extends refInterface  version v  "documentation"
{
      name _i1s1 "documentation" (
            name_i1s1p1 : direction refDataType "documentation",
            …
            name_i1s1pN  : direction refDataType "documentation"
      )
      …
      name _i1sN "documentation" (
            name_i1sNp1 : direction refDataType "documentation",
            …
            name_i1sNpN  : direction refDataType "documentation"
      )
}
…
```

Bloque 9: Esquema general de un fichero .idt

El primer patrón de búsqueda del método tiene como referente la palabra "interface" y cuando el usuario la escribe seguida de un nombre, automáticamente se introduce la palabra "extends". De la misma forma funcionan todos los otros patrones, pero con diferentes referencias.

```
private boolean isTokenEqualTo(IDocument document, DocumentCommand
command, String key, int n, boolean ignoreCase) throws
BadLocationException
      {
            boolean result = false;
            int line = document.getLineOfOffset(command.offset);
            int lineStart = document.getLineOffset(line);
            String[] tokens = document.get(lineStart, command.offset
            - lineStart).split(" ");

            if(n>=0 && n<tokens.length)
            {
                  if(ignoreCase)
                        result = tokens[tokens.length-n-
                        1].equalsIgnoreCase(key);
                  else
                        result = tokens[tokens.length-n-
                        1].equals(key);
            }
            return result;
      }
}
```

Bloque 10: Método IsTokenEqualTo

Considerando la última línea de texto en un documento, este método devuelve true si la palabra “key” aparece n posiciones previas del final de la línea. El booleano *ignoreCase* permite seleccionar si se quiere que el método distinga de mayúsculas y minúsculas. De forma que, por ejemplo, si se pone el booleano a “true” y la palabra clave que se usa como referencia es “interface”, el método reconocerá como igual la palabra “INTERFACE” o cualquier otra combinación de mayúsculas-minúsculas que el usuario utilice para escribir la palabra. Si *ignoreCase* estuviera a false en el ejemplo anterior, para el método, esas palabras no serían iguales.

Una vez se ha definido esta clase debe ser registrada en el módulo de interfaz de usuario, es decir, en el mismo en el que registrábamos la coloración de sintaxis, que quedaría ahora de la siguiente forma:

```
public class IdtUiModule extends v3cmm.ui.AbstractIdtUiModule {
	public IdtUiModule(AbstractUIPlugin plugin) {
		super(plugin);
	}
	//Coloración de sintaxis
	public Class<? extends IHighlightingConfiguration>
	bindIHighlightingConfiguration(){
		return MyHighlightingConfiguration.class;
	}
	//Auto-compleción
	public Class<? extends IAutoEditStrategy>bindIAutoEditStrategy(){
		return MyAutoEdit.class;
	}
}
```

Bloque 11: Clase IdtUiModule.java para el registro de modificaciones

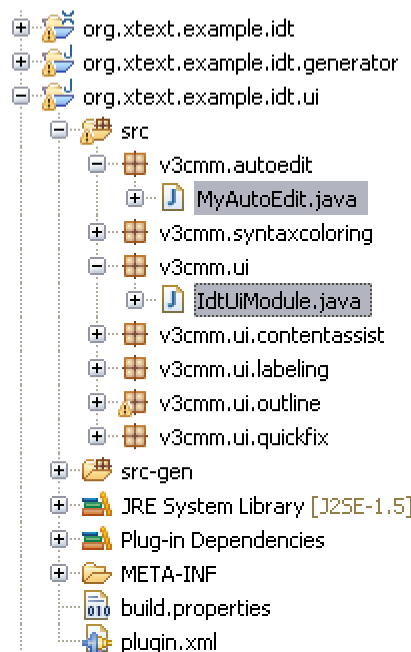

Figura 14: Clases que intervienen en la auto-compleción de la sintaxis

4.4.3. Adaptación de puertos a su ámbito

Si en la vista estructural se denotaran los puertos sólo por sus nombres, y varios componentes tuvieran puertos que se llamasen igual, sería lógico que al conectarlos el usuario no supiera exactamente el puerto de qué componente está conectando. Para evitar este problema se ha implementado esta mejora, que permite hacer declaraciones como las que se muestran en el Bloque 12.

```
ComplexCD CompComplejo {

      ports {
      puertoComplejo
            -c  interfaz_requerida;
            -o  interfaz_proveida;
      }

      component comp1 : comp1_definición{
            ports {
                  puerto1: puerto1_definición;
            }
       }

         component comp2 : comp2_definición{
            ports {
                  puerto1: puerto1_definición;
            }
       }

         component comp3 : comp3_definición{
            ports {
                  puerto1: puerto1_definición;
            }
       }

         portLinks {
          comp1.port1 <-> comp2.port1;   ← Esto es lo que se buscaba
    }
}
```

Bloque 12: Caso de uso para la adaptación de puertos a su ámbito

La implementación de esta mejora se hace en la clase Java denominada *NameProvider.java*, definida en el Bloque 13, que debe crearse desde cero en el paquete con extensión *.scopping* que se encuentra en la carpeta *src* del plug-in Xtext principal. Puede verse el emplazamiento de la clase en la Figura 15. Dado que la finalidad de esta mejora la de identificar puertos de componentes, es obvio que sólo se incluirá en el editor de la vista de componentes.

Primero se importan ciertos elementos del meta-modelo para poder trabajar con ellos desde la clase Java. Lo que hace esta clase, básicamente, es analizar todos los elementos que se crean y son susceptibles de ser referenciados. Cuando uno de estos elementos es un puerto, o una definición de puerto, recoge su nombre y el del componente que lo contiene, los coloca en un string siguiendo el esquema “componente.puerto” y lo devuelve.

```
import v3cmm.Component;
import v3cmm.ComponentDefinition;
import v3cmm.Port;
import v3cmm.PortDefinition;

public class NameProvider extends IQualifiedNameProvider.AbstractImpl{

	@override
	public String getQualifiedName(EObject obj) {
		String result=null;
		if(obj.eClass().getName()=="Port" ||
		obj.eClass().getName()=="PortDefinition")
		{
			StringBuilder sb=new StringBuilder();
			if(obj.eClass().getName()=="Port"){

			sb.append(((Component)obj.eContainer()).getName());
				sb.append(".");
				sb.append(((Port)obj).getName());
		}
		else{

	sb.append(((ComponentDefinition)obj.eContainer()).getName());
				sb.append(".");
				sb.append(((PortDefinition)obj).getName());
			}

			result=sb.toString();
		}
		return result;
	}

}
```

Bloque 13: Implementación de la clase NameProvider.java

Esta clase trabaja en tiempo de ejecución, ya que debe seguir los pasos del usuario porque no puede conocer de antemano las clases que se crearán, pero no es una modificación de una característica de la interfaz de usuario del editor. Por ello debe registrarse en una clase distinta y no en el de interfaz de usuario como se hacía en las otras mejoras.

La clase de registro se llama XxxRuntimeModule.java, donde Xxx es la extensión del editor. Se encuentra en el paquete sin extensión dentro de la carpeta *src*, en el plug-in principal del proyecto Xtext.

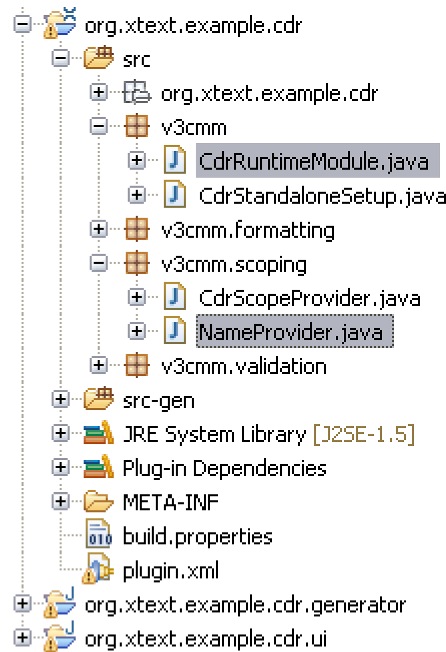

Figura 15: Clases que intervienen en la adaptación de puertos a su ámbito

La clase de registro CdrRuntimeModule quedaría como se muestra en el Bloque 14.

```
public class CdrRuntimeModule extends v3cmm.AbstractCdrRuntimeModule
{

	public Class<? extends IQualifiedNameProvider>
	bindIQualifiedNameProvider()
	{
		return NameProvider.class;
	}
}
```

Bloque 14: Clase de registro CdrRuntimeModule

4.4.4. Modificación del ContentAssist

El ContentAssist es el asistente que ayuda al usuario sugiriendo opciones a medida que éste escribe el modelo, por ejemplo, mostrando las posibles definiciones que pueden ser utilizadas al instanciar un componente.

Cuando se definen componentes en la vista estructural, se deben interconectar mediante puertos, pero dichos puertos deben ser compatibles, esto es, que las interfaces, con sus correspondientes servicios, que requiere un puerto deben ser proveídas por el otro. Para facilitar la labor al usuario, se ha modificado el ContentAssist de forma que sólo muestre los puertos compatibles con el que se desea conectar.

Al igual que en el apartado anterior, esta mejora sólo se incluye en la vista de componentes por la misma razón.

El ContentAssist se modela con la clase Java que se encuentra en el paquete con extensión *.contentassist* de la carpeta src del plug-in de interfaz de usuario. Recibe el nombre de *XxxProposalProvider.java*, donde Xxx es la extensión empleada en el editor. Se puede comprobar la ubicación exacta de los ficheros que interviene en la Figura 16. El aspecto que presenta la clase implementada en el presente Proyecto se detalla en los Bloque 15.1, 15.2, 15.3, 15.4 y 15.5.

```
package v3cmm.ui.contentassist;
import java.util.List;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.Xtext.Assignment;
import org.eclipse.Xtext.ui.editor.contentassist.ContentAssistContext;
import org.eclipse.Xtext.ui.editor.contentassist.
       ICompletionProposalAcceptor;
import v3cmm.ui.contentassist.AbstractCdrProposalProvider;
import es.upct.dsie.v3cmm.AssemblyLink;
import es.upct.dsie.v3cmm.ComplexComponentDefinition;
import es.upct.dsie.v3cmm.Component;
import es.upct.dsie.v3cmm.DelegationLink;
import es.upct.dsie.v3cmm.Interface;
import es.upct.dsie.v3cmm.Port;
import es.upct.dsie.v3cmm.PortDefinition;
```

Bloque 15.1: Primer fragmento de la clase

La clase List de java facilitará la creación y manipulación de tablas. Se hace necesario importar elementos del meta-modelo para poder recorrerlos y almacenarlos desde la clase Java.

El método del Bloque 15.2 es el encargado de los AssemblyLink. En el Bloque 15.4 se encuentra el de los DelegationLink.

```
public class CdrProposalProvider extends AbstractCdrProposalProvider {
      public void completeAssemblyLink_EndB (EObject obj, Assignment
      assigment, ContentAssistContext context,
      ICompletionProposalAcceptor acceptor)
      {

            List<Port> ports;
            List<Component>
      components=((ComplexComponentDefinition)obj.eContainer()).
      getComponents();
            Port portA=((AssemblyLink)obj).getEndA();
            for(int i=0; i<components.size(); i++){
                  ports=components.get(i).getPorts();
                  for(int j=0; i<ports.size(); j++){
                  if(comparaPuertosAssembly(portA,ports.get(j)))
                        {
                        acceptor.accept(createCompletionProposal(
                  ports.get(j).getName(),context));
                        }
                  }
            }
      `
```

Bloque 15.2: Segundo fragmento de la clase. AssemblyLinks

Lo que hace el método, paso a paso, es: (1) crea una lista de puertos y otra de componentes. (2) Toma el terminal que queremos conectar, busca todos los componentes que pueden ser conectados y los introduce en la lista de componentes para después recorrer sus puertos comparándolos con el que se quiere conectar. (3) Si son compatibles los añade a la lista de puertos.

En el código del Bloque 15.2 se observa una llamada al método *comparaPuertosAssembly*. Este método determina si dos puertos son compatibles según sus interfaces. Como los puertos son Assembly, son compatibles si uno ofrece las interfaces que el otro requiere. La implementación el método se describe en el Bloque 15.3.

```
public boolean comparaPuertosAssembly(Port a, Port b){
      if(comparaInterfaces(a.getDefinitions().getProvidedInterfaces()
      ,b.getDefinitions().getRequiredInterfaces())&
      comparaInterfaces(b.getDefinitions().getProvidedInterfaces(),a.
      getDefinitions().getRequiredInterfaces()))
      {
            return true;
      }
      else return false;
}
```

Bloque 15.3: Tercer fragmento de la clase. ComparaPuertosAssembly

La conexión de los enlaces Delegation tiene un funcionamiento similar al de los Assembly: un primer método busca y compara los puertos de los componentes haciendo uso del segundo.

```
public void completeDelegationLink_EndB (EObject obj, Assignment
assigment, ContentAssistContext context, ICompletionProposalAcceptor
acceptor){
            List<Port> ports;
            List<Component>
      components=((ComplexComponentDefinition)obj.
      eContainer()).getComponents();
            PortDefinition defA=((DelegationLink)obj).getEndA();
            for(int i=0; i<components.size(); i++){
                  ports=components.get(i).getPorts();
                  for(int j=0; j<ports.size();j++){
                        if(comparaPuertosDelegation(defA,ports.get(j))){
                              acceptor.accept(createCompletionProposal(
                              ports.get(j).getName(), context));
                        }
                  }
            }
      }
public boolean comparaPuertosDelegation(PortDefinition a, Port b){
      if(comparaInterfaces(a.getProvidedInterfaces(),
      b.getDefinitions().getProvidedInterfaces())&
      comparaInterfaces(a.getRequiredInterfaces(),b.getDefinitions()
      .getRequiredInterfaces())) {
            return true;
      }else return false;
}
```

Bloque 15.4: Cuarto fragmento de la clase. DelegationLinks

Como se ve en el Bloque 15.4, en los Delegation lo que ofrece y requiere un puerto debe ser igual a lo que ofrece y requiere su definición, sin cruzarlos.

Del método reflejado en el Bloque 15.5 se valen los de comparación de puertos descritos en los Bloque 15.3 y 15.4 para saber si dos interfaces son iguales.

```
public boolean comparaInterfaces(List<Interface> l1, List<Interface> l2){
            if(l1.size()!=l2.size()){
                  return false;
            }
            else{
                  for(int i=0;i<l1.size();i++){
                        for(int j=0;j<l2.size();j++){
                              if(l1.get(i)!=l2.get(j)) return false;
                        }
                  }
                  return true;
            }
      }
}
```

Bloque 15.5: Último fragmento de la clase. ComparaInterfaces

Una vez se ha implementado la clase que realice la función deseada, debe ser registrada en la misma clase que se registró la coloración de sintaxis y la auto-compleción del esquema que, con todo ello, quedaría tal y como se muestra en el Bloque 16.

```
public class CdrUiModule extends v3cmm.ui.AbstractCdrUiModule {
      public CdrUiModule(AbstractUIPlugin plugin) {
            super(plugin);
      }
      //SyntaxColoring
      public Class<? extends
      IHighlightingConfiguration>bindIHighlightingConfiguration()
            {
            return MyHighlightingConfiguration.class;
            }
      //Auto-Complecion
      public Class<? extends IAutoEditStrategy> bindIAutoEditStrategy(){
            return MyAutoEdit.class;
      }
      //ContentAssist
      public Class<? extends
      org.eclipse.Xtext.ui.editor.contentassist.IContentProposalProvider>
      bindIContentProposalProvider(){
            return CdrProposalProvider.class;
      }

}
```

Bloque 16: Clase CdrUiModule completa, esto es, con todas las mejoras registradas.

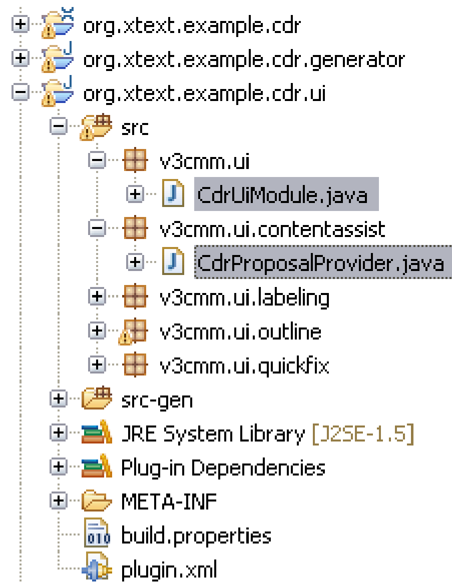

Figura 16: Clases que intervienen en la modificación del ContentAssist

Cualquiera de las anteriores mejoras pueden "desactivarse" a placer del desarrollador. Para ello simplemente se debe comentar el método correspondiente en el módulo de registro, de ésta forma Eclipse no la incluirá al ser ejecutados los editores.

CAPÍTULO 5º

Guía de instalación y uso de los editores

En este capítulo se detallará cómo utilizar las herramientas desarrolladas para V3CMM en el presente Proyecto.

5.1. Consideraciones iniciales

Antes de abordar con más profundidad la instalación de las herramientas debemos tener en cuenta las siguientes consideraciones iniciales:

- La versión de Eclipse que se ha empleado es Helios con el bundle *Eclipse Modeling Tools* v.1.3.2 [13]. Como mínimo debe instalarse la herramienta para Xtext y la herramienta para OCL.

- La máquina virtual de Java instalada en el computador en el que se llevó a cabo el Proyecto es la versión 6 Update 26. Se recomienda una versión igual o superior.

5.2. Instalación para desarrolladores

Se brinda al desarrollador la opción de instalar el workspace completo tal y como ha quedado tras la realización del Proyecto. Para ello, en primer lugar, se debe instalar el paquete *Eclipse Modeling Tools* mencionado en el apartado anterior y a continuación importar el directorio que se adjunta en el CD con este Proyecto.

Se detalla a continuación cómo se haría la instalación del paquete:

Una vez se ha descargado el archivo con extensión .zip se descomprime y se ejecuta Eclipse. Desde el workspace creado se despliega el menú *Help* y se selecciona la opción *Install Modeling Components* tal y como se muestra en la Figura 17.

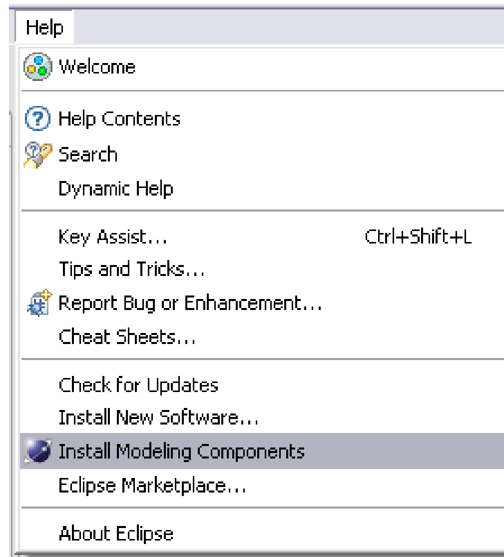

Figura 17: Menú Help del WorkSpace

Al clickar esta opción, aparecerá una nueva ventana (puede tardar un rato en cargar) en la que se seleccionaran las herramientas que se desea instalar. Como ya se mencionó antes, deben instalarse como mínimo las herramientas Xtext y OCL para poder trabajar con el presente Proyecto. Acto seguido se mostrará una nueva ventana con el resumen de las herramientas que se van a instalar. Si se está de acuerdo se pulsa *Next* y en la siguiente ventana se aceptan los términos de licencia de Eclipse. Cuando el proceso de instalación haya finalizado debemos reiniciar Eclipse para que las herramientas estén disponibles.

Una vez se disponga de las herramientas necesarias se debe seguir el siguiente procedimiento para importar el workspace:

En el menú *File* se selecciona la opción *Import* → *General* → *Existing Projects into Workspace*. En la siguiente pantalla que aparece se debe seleccionar la opción *Select Archive File* y se busca el archivo .zip que contiene el workspace. Una vez hecho esto, se seleccionan todos los proyectos que aparecen en la lista tal y como muestra la Figura 18 y se cargan.

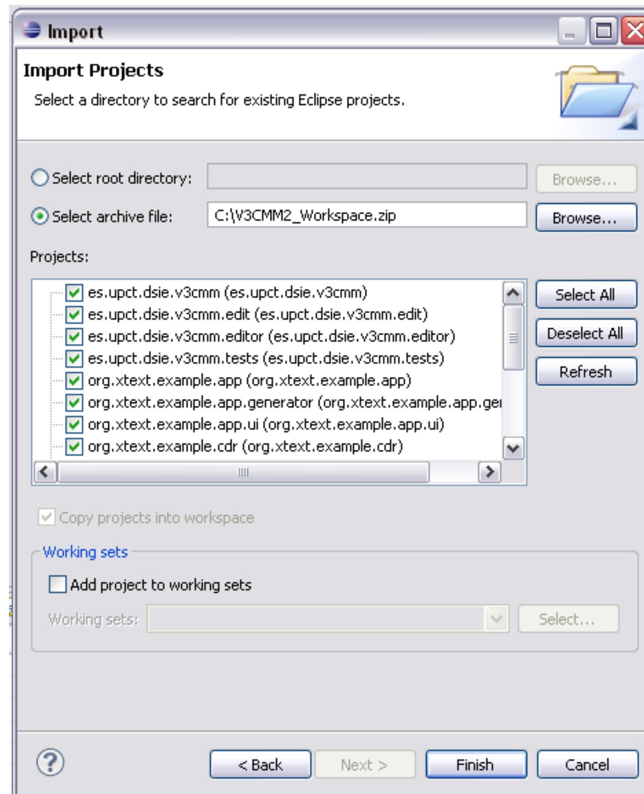

Figura 18: Menú Import

5.3. Instalación para usuarios

Se proponen dos opciones para la instalación de los editores: la primera basada en ficheros JAR y la segunda basada en un Producto Eclipse.

Para llevar a cabo la instalación mediante el primer método, se deben copiar todos los plugins que se adjuntan en el .zip en la carpeta “plugins”, que se encuentra en el directorio principal de Eclipse. Hecho esto se reinicia Eclipse y ya se dispone de los plug-in.

Si se desea hacer uso del Producto Eclipse simplemente debe ejecutarse el mismo, sin necesidad de instalar Eclipse, las herramientas o cualquier plug-in. Esta es la opción más cómoda e intuitiva para el usuario.

5.4. Guía de usuario para desarrolladores

Si se han seguido todos los pasos descritos hasta ahora, se tendrá una serie de proyectos con la implementación completa de los editores. Para poder hacer uso de los mismos, se deben lanzar las aplicaciones Java creadas en una nueva configuración de Eclipse. Esto puede hacerse de varias formas. Una de ellas es hacer click en la opción *Run* de la barra de herramientas y seleccionar *Run Configurations.* En la nueva ventana que se abre se selecciona *Eclipse Application*, se crea una *New_Configuration* como se muestra la Figura 19 y pulsamos en *Run*. No es necesario crear una nueva configuración cada vez que se quiera lanzar Eclipse, en las veces posteriores basta con hacer click en el icono verde de *run* que se observa en la Figura 20.

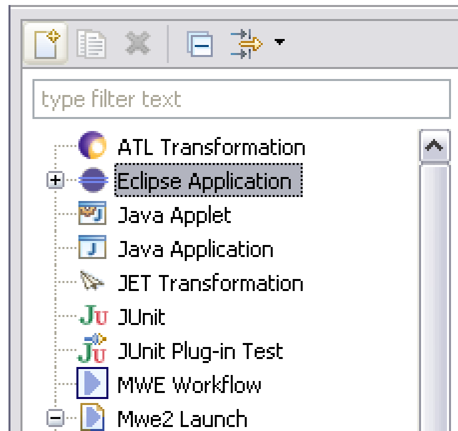

Figura 19: Haciendo clic en el icono resaltado, se crea la nueva configuración

Una vez se ha creado la nueva configuración se abre un segundo Eclipse donde ya es posible definir ficheros con las extensiones creadas en el Proyecto. Al crear un fichero con una de las extensiones, Xtext le aplicará su propia sintaxis y todas las mejoras introducidas.

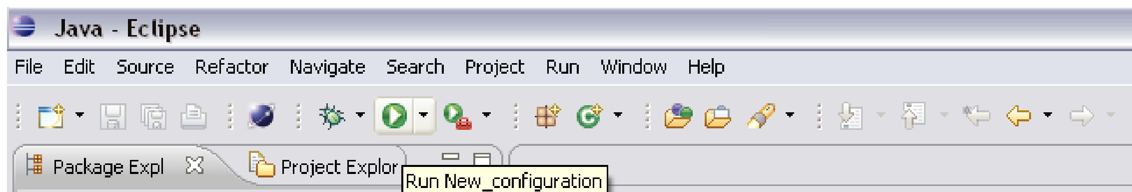

Figura 20: Botón de Run *resaltado para lanzar nuevas configuraciones*

5.5. Guía de usuario

A continuación se muestra cómo se utilizan los editores definiendo un ejemplo típico en la literatura sobre Ingeniería del Software: un cajero automático [14]. El cajero implementado consta de 6 componentes independientes, a saber: una pantalla, un teclado, una impresora de tickets, un tarjetero, un billetero y un controlador o unidad central. La estructura por componentes que se plantea se muestra en la Figura 21.

Se implementará un controlador con cinco puertos:

- Tres de ellos requerirán una interfaz para conectar la pantalla, el teclado y la impresora. Dichos componentes por tanto deberán proveer a la unidad central de control dichas interfaces.
- Los otros dos puertos poseerán dos interfaces y se conectarán con el billetero y el tarjetero cada uno. Deben tener dos interfaces ya que por una se requerirán servicios de los componentes, como por ejemplo leer el número de la tarjeta, y por la otra se recibirán señales de control de los componentes, como por ejemplo cuando un cliente introduce su tarjeta.

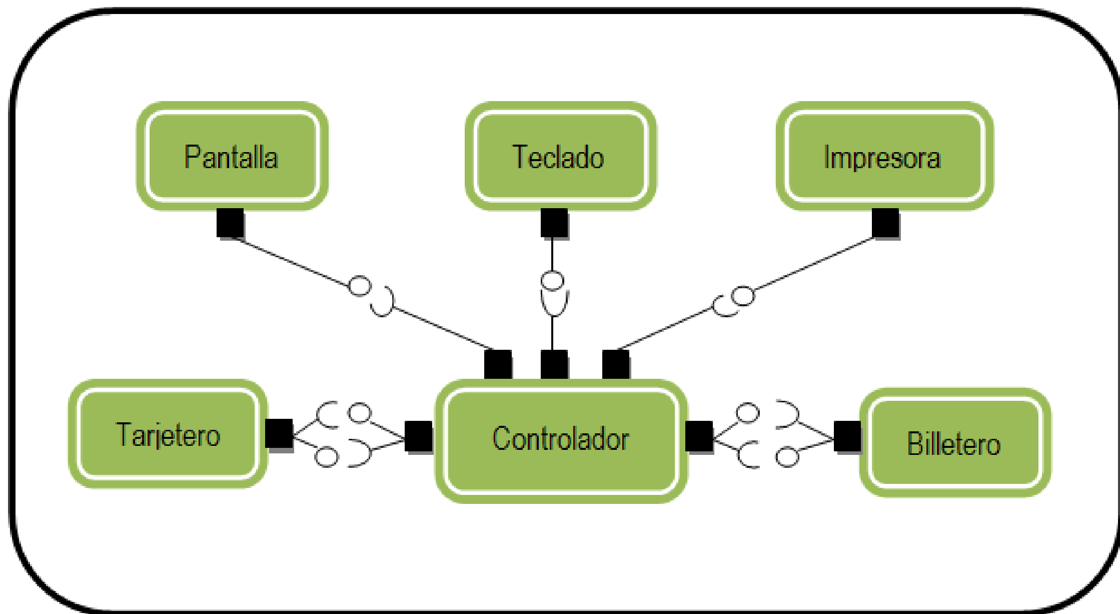

Figura 21: Esquema de componentes del cajero implementado

Dicho esto, la definición de los componentes será:

- Pantalla:
  - La interfaz que ofrece la pantalla por su puerto se define en un fichero .idt, tal y como muestra el Bloque 17.

```
import "Tipos.idt";

interface I_Display{
	print(linea: inCopy String);
	cfg(linea: inCopy String);
	reconfig();
	reset();
}
```

Bloque 17: Interfaz ofrecida por la pantalla

Se puede observar que se importa un fichero llamado "*Tipos.idt*". En él se han definido los tipos de datos String, int, boolean y char, haciendo uso de la sentencia *datatype*.

Tal y como muestra su definición, la interfaz *I_Display* contiene cuatro servicios (*print*, *cfg*, *reconfig* y *reset*) y que algunos de ellos necesitan parámetros de entrada (*linea*).

  - Una vez se tiene la interfaz, se define el componente en un fichero .cdr y se le asocia dicha interfaz al puerto que la proveerá. En el Bloque 18 se proporciona la declaración del componente.

```
import "I_Display.idt";
import "StateM_Display.xta";

simpleCD Display behaviours StateM_Display {
	ports {
		Display_Port -o I_Display;
	}
}
```

Bloque 18: Componente "Pantalla"

Llama la atención de inmediato que se importa un fichero de extensión .xta. Dicho fichero contiene la máquina de estados que modela el comportamiento del componente. En cuanto a la asignación de la interfaz al puerto, se ha utilizado el símbolo reservado "`-o`", que significa que será ofrecida. Si se quisiera requerir la interfaz se utilizaría el símbolo "`-c`".

- La máquina de estados que se ha definido se muestra en los Bloques 19.1 y 19.2.

```
import "I_Display.idt";
timedAutomata StateM_Display {
	events {
	providedService_Requests e_cfg<-cfg, e_reconfig<-reconfig, e_print<-
print,	e_reset<-reset;
	internalSignals i_checked, i_error, i_wrongConfig, i_config;
	timeEvents one_seg(>1s);
	}
	region Principal {
		state Rutina{
			region Rutina{
				initial iniA;
				state Hardware_checkA;
				state Error;
				state Work{
					region Work{
						initial iniW;
						state IdleW;
						state Check_config;
						state Working;
						transitions{
							iniW -> IdleW;
							IdleW -> Check_config @ e_cfg;
							Check_config -> IdleW @ i_wrongConfig;
							Check_config -> Working @ i_config;
							Working -> IdleW @ e_reconfig;
							Working -> Working @ e_print;}
					}
				}
				transitions{
					iniA -> Hardware_checkA;
					Hardware_checkA -> Work @ i_checked;
					Work -> Error @ i_error;}
			}
		}
```

Bloque 19.1: Primer fragmento de la máquina de estados de la pantalla

```
state Check{
                  region Check{
                        initial iniB;
                        state IdleB;
                        state Hardware_checkB;
                        transitions{
                              iniB -> IdleB;
                              IdleB -> Hardware_checkB @ i_checked;
                              Hardware_checkB -> Hardware_checkB @ one_seg;
                              Hardware_checkB -> Error @ i_error;
                        }
                  }
            }
      }
}
```

Bloque 19.2: Segundo fragmento de la máquina de estados de la pantalla

Se puede apreciar que se importa la interfaz del componente para poder así realizar llamadas a los servicios que se requieren y referencias a los que se ofrecen.

La máquina modelada, que puede verse en la Figura 22, está compuesta por dos “sub-máquinas” que trabajan en paralelo. Para poder implementarlas se han definido dos estados independientes con una región dentro de cada uno:

- La primera se denomina “Rutina” y es la máquina principal que define el comportamiento de la pantalla. Se observa que tiene varios estados “simples” y uno (Work) que está compuesto por otra máquina de estados dentro. El objetivo de hacerlo así es para simplificar, ya que los estados que conforman *Work* tienen un objetivo común y siempre se realizan cuando el componente realiza su función.

- La segunda se denomina “Check” y se encarga de revisar periódicamente el estado del hardware del componente.

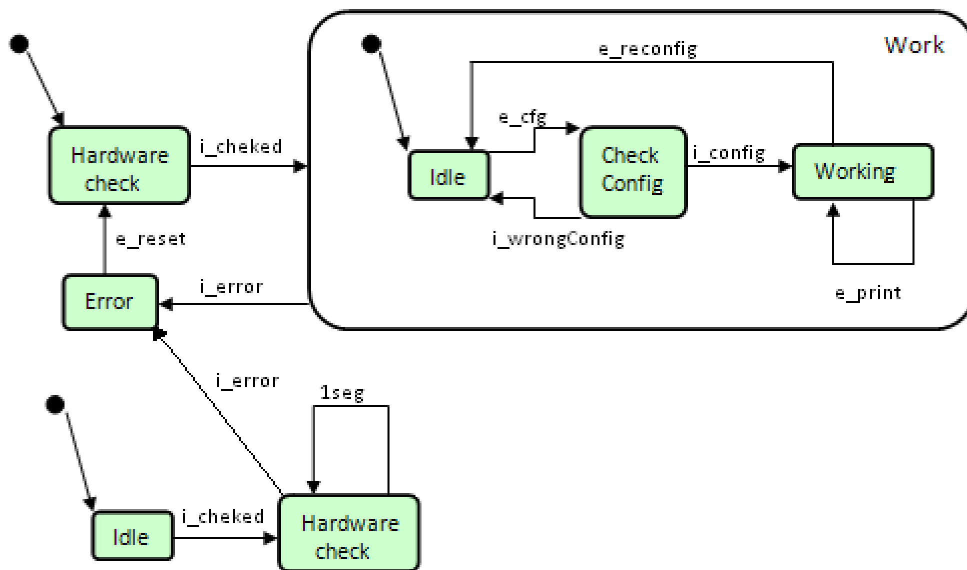

Figura 22: Máquina de estados del componente pantalla

- Teclado:
    - La interfaz que ofrece el teclado es la implementada en el Bloque 20.

```
import "Tipos.idt";

interface I_Keyboard{
readMenu(return:out char);
readKey(return: out char);
readStr(return: out String);
reset();
}
```

Bloque 20: Interfaz del teclado

Algo que se no vio en la interfaz de la pantalla es que los servicios pueden retornar parámetros. Todos los servicios para leer del teclado retornan a quien realizó la llamada un dato.

- El componente quedaría definido como en el Bloque 21.

```
import "I_Keyboard.idt";
import "StateM_Keyboard.xta";

simpleCD Keyboard behaviours StateM_Keyboard
{
	ports {
		Keyboard_Port -o I_Keyboard;
	}
}
```

Bloque 21: Componente "teclado"

- La máquina de estados que modela el teclado es muy similar a la anterior. Se detalla en el Bloque 22.

```
import "I_Keyboard.idt";
timedAutomata StateM_Keyboard {
      events {
      providedService_Requests e_readMenu<-readMenu, e_readKey<-readKey,
      e_readStr<-readStr, e_reset<-reset;
      internalSignals i_checked, i_error, i_wrongConfig, i_config;
      timeEvents one_seg(>1s);
      }
      region Principal {
            state Rutina{
                  region Rutina{
                        initial iniA;
                        state Hardware_checkA;
                        state Error;
                        state IdleA;
                        transitions{
                              iniA -> Hardware_checkA;
                              Hardware_checkA -> IdleA @ i_checked;
                              IdleA -> Error @ i_error;
                              IdleA -> IdleA @ e_readMenu;
                              IdleA -> IdleA @ e_readKey;
                              IdleA -> IdleA @ e_readStr;
                        }
                  }
            }
            state Check{
                  region Check{
                        initial iniB;
                        state IdleB;
                        state Hardware_checkB;
                        transitions{
                              iniB -> IdleB;
                              IdleB -> Hardware_checkB @ i_checked;
                              Hardware_checkB -> Hardware_checkB @ one_seg;
                              Hardware_checkB -> Error @ i_error;
                        }
                  }
            }
      }
}
```

Bloque 22: Máquina de estados del teclado

- La interfaz que ofrece la impresora es la definida en el Bloque 23.

```
import "Tipos.idt";

interface I_Print{
	print(linea: inCopy String);
	startPage();
	endPage();
	newLine();
	reset();
}
```

Bloque 23: Interfaz de la impresora

- El componente se define como muestra el Bloque 24.

```
import "StateM_Printer.xta";
import "I_Print.idt";

simpleCD Printer behaviours StateM_Printer{
	ports {
		Printer_Port -o I_Print;
	}
}
```

Bloque 24: Componente impresora

- La máquina de estados correspondiente se define en los Bloque 25.1 y 25.2.

```
import "I_Print.idt";
timedAutomata StateM_Printer {
	events {
	providedService_Requests e_startPage<-startPage, e_newLine<-newLine,
	e_print<-print, e_endPage<-endPage, e_reset<-reset;
	internalSignals i_checked, i_error;
	timeEvents one_seg(>1s);}
	region Principal {
		state Rutina{
			region Rutina{
				initial iniA;
				state Hardware_checkA;
				state IdleA;
				state Error;
				state Printing{
					region Printing{
						initial iniPrinting;
						state Wait;
						state Print;
						transitions{
							iniPrinting -> Wait;
							Wait -> Print @ e_print;
							Print -> Wait @ e_newLine;}
					}
				}
```

Bloque 25.1: Primer fragmento de la máquina de estados de la impresora

```
                        transitions{
                              iniA -> Hardware_checkA;
                              Hardware_checkA -> IdleA @ i_checked;
                              IdleA -> Printing @ e_startPage;
                              IdleA -> Error @ i_error;
                              Printing -> IdleA @ e_endPage;
                              Printing -> Error @ i_error;
                              Printing -> Hardware_checkA @ e_reset;
                        }
                  }
            }
            state Check{
                  region Check{
                        initial iniB;
                        state IdleB;
                        state Hardware_checkB;
                        transitions{
                              iniB -> IdleB;
                              IdleB -> Hardware_checkB @ i_checked;
                              Hardware_checkB->Hardware_checkB @one_seg;
                              Hardware_checkB -> Error @ i_error;
                        }
                  }
            }
      }
```

Bloque 25.2: Segundo fragmento de la máquina de estados de la impresora

- Tarjetero:
  - Este componente contiene dos interfaces en el mismo puerto, como muestra el Bloque 26.

```
import "Tipos.idt";
interface I_Card_P{
      confiscar();
      expulsar();
      recuperarT();
      leerNum(return: out int);
      reset();
}
interface I_Card_R{
      hay_tarjeta(return: out boolean);
      puertaT_abierta();
      puertaT_cerrada();
}
```

Bloque 26: Interfaces del componente “tarjetero”

- El componente por tanto quedaría definido como en el Bloque 27.

```
import "I_Card.idt";
import "StateM_Card.xta";

simpleCD Card behaviours StateM_Card {
        ports {
                Card_Port -c I_Card_R -o I_Card_P;
        }
}
```

Bloque 27: Componente "tarjetero"

Como se puede apreciar, se han asignado las dos interfaces al puerto: una es ofrecida y la otra requerida.

- La máquina de estados de este componente (Figura 23) es un tanto especial, ya que debe controlar, además de su rutina y el checkeo del hardware, el sensor que indica si alguien introduce una tarjeta y el estado de la puerta trasera. Se puede observar su definición en los Bloques 28.1 y 28.2.

La rutina principal del tarjetero espera en el estado *Idle* hasta que un cliente introduce su tarjeta y lleva a la máquina al estado *Operando*. Desde ese estado se lee el número de la tarjeta, se expulsa o se confisca en función de las órdenes de la unidad central. Desde el estado *Idle* también se controla que si algún empleado abre la puerta trasera, el tarjetero quede fuera de servicio durante ese período de tiempo.

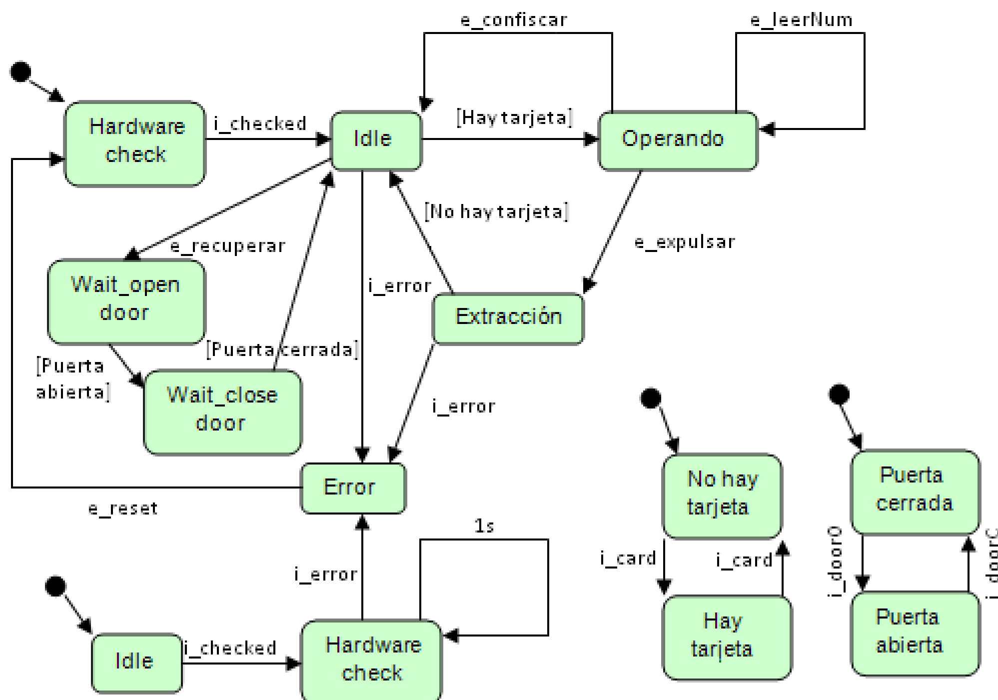

Figura 23: Máquina de estados del tarjetero

```
import "I_Card.idt";

timedAutomata StateM_Card {
      events {
      providedService_Requests e_confiscar<-confiscar, e_expulsar<-expulsar,
e_recuperar<-recuperarT, e_reset<-reset, e_leerNum<-leerNum;
      internalSignals i_checked, i_error, i_card, i_doorO,i_doorC;
      timeEvents one_seg(>1s), fifteen_seg(>15s);
      }
      region Principal {
            state Rutina{
                  region Rutina{
                        initial iniA;
                        state Hardware_checkA;
                        state Error;
                        state IdleA;
                        state Operando;
                        state Extraccion;
                        state Wait_open_door;
                        state Wait_close_door;
                        transitions{
                              iniA -> Hardware_checkA;
                              Hardware_checkA -> IdleA @ i_checked;
                              IdleA -> Error @ i_error;
                              IdleA -> Operando [OR InState Hay_tarjeta
                              in region Sensor_tarjeta];
                              IdleA -> Wait_open_door @ e_recuperar;
                              Wait_open_door -> Wait_close_door [OR InState
                              Puerta_abierta in region Puerta];
                              Wait_close_door -> IdleA [OR InState
                              Puerta_cerrada in region Puerta];
                              Error -> Hardware_checkA @ e_reset;
                              Operando -> IdleA @ e_confiscar;
                              Operando -> Operando @ e_leerNum;
                              Operando -> Extraccion @ e_expulsar;
                              Extraccion -> IdleA [OR InState No_hay_tarjeta
                              in region Sensor_tarjeta];
                              Extraccion -> Error @ fifteen_seg;
                        }
                  }
            }
            state Check{
                  region Check{
                        initial iniB;
                        state IdleB;
                        state Hardware_checkB;
                        transitions{
                              iniB -> IdleB;
                              IdleB -> Hardware_checkB @ i_checked;
                              Hardware_checkB -> Hardware_checkB @ one_seg;
                              Hardware_checkB -> Error @ i_error;
                        }
                  }
            }
```

Bloque 28.1: Máquina de estados del tarjetero

```
            state Sensor_tarjeta{
                  region Sensor_tarjeta{
                        initial iniC;
                        state Hay_tarjeta;
                        state No_hay_tarjeta;
                        transitions{
                              iniC -> No_hay_tarjeta;
                              No_hay_tarjeta -> Hay_tarjeta @ i_card;
                              Hay_tarjeta -> No_hay_tarjeta @ i_card;
                        }
                  }
            }
            state Puerta{
                  region Puerta{
                        initial iniD;
                        state Puerta_abierta;
                        state Puerta_cerrada;
                        transitions{
                              iniD -> Puerta_cerrada;
                              Puerta_cerrada -> Puerta_abierta @ i_doorO;
                              Puerta_abierta -> Puerta_cerrada @ i_doorC;
                        }
                  }
            }
      }
}
```

Bloque 28.2: Máquina de estados del tarjetero

- Billetero:
    - Al igual que el tarjetero, el billetero también tiene dos interfaces: una para proveer y otra para requerir. Su interfaz queda definida en el Bloque 29.

```
import "Tipos.idt";
interface I_Cash_P{
      recuperarC();
      retirar(cantidad: inCopy int);
      reset();
}
interface I_Cash_R{
      hay_dinero (return: out boolean);
      puertaC_abierta();
      puertaC_cerrada();
}
```

Bloque 29: Interfaz del billetero

- La definición del componente “*Billetero*” es muy similar a la del tarjetero, tal y como puede apreciarse en el Bloque 30.

```
import "I_Cash.idt";
import "StateM_Cash.xta";

simpleCD Cash behaviours StateM_Cash {
	ports {
		Cash_Port -c I_Cash_R -o I_Cash_P;
	}
}
```

Bloque 30: Componente “Billetero”

- Al igual que el componente anterior, este debe controlar un sensor para saber si hay dinero en el cajón y otro para la puerta además de su rutina principal y la de checkeo. Su máquina de estados queda definida en los Bloques 31.1 y 31.2.

```
import "I_Cash.idt";

timedAutomata StateM_Cash {
	events {
	providedService_Requests e_retirar<-retirar,
	e_recuperar<-recuperarC, e_reset<-reset;
	internalSignals i_checked, i_error, i_cash, i_doorO,i_doorC;
	timeEvents one_seg(>1s), fifteen_seg(>15s);
	}
	region Principal {
		state Rutina{
			region Rutina{
				initial iniA;
				state Hardware_checkA;
				state Error;
				state IdleA;
				state Extraccion;
				state Wait_open_door;
				state Wait_close_door;
				transitions{
					iniA -> Hardware_checkA;
					Hardware_checkA -> IdleA @ i_checked;
					IdleA -> Error @ i_error;
					IdleA -> Wait_open_door@ e_recuperar;
					IdleA -> Extraccion @ e_retirar;
					Extraccion -> IdleA [OR InState
					No_hay_dinero in región Sensor_dinero];
					Extraccion -> Error @ fifteen_seg;
					Wait_open_door -> Wait_close_door [OR
					InState Puerta_abierta in region Puerta];
					Wait_close_door -> IdleA [OR InState
					Puerta_cerrada in region Puerta];
					Error -> Hardware_checkA @ e_reset;
				}
			}
		}
```

Bloque 31.1: Máquina de estados del tarjetero

```
		state Check{
			region Check{
				initial iniB;
				state IdleB;
				state Hardware_checkB;
				transitions{
					iniB -> IdleB;
					IdleB -> Hardware_checkB @ i_checked;
					Hardware_checkB->Hardware_checkB @ one_seg;
					Hardware_checkB -> Error @ i_error;
				}
			}
		}
		state Sensor_dinero{
			region Sensor_dinero{
				initial iniC;
				state Hay_dinero;
				state No_hay_dinero;
				transitions{
					iniC -> No_hay_dinero;
					No_hay_dinero -> Hay_dinero @ i_cash;
					Hay_dinero -> No_hay_dinero @ i_cash;
				}
			}
		}
		state Puerta{
			region Puerta{
				initial iniD;
				state Puerta_abierta;
				state Puerta_cerrada;
				transitions{
					iniD -> Puerta_cerrada;
					Puerta_cerrada -> Puerta_abierta @ i_doorO;
					Puerta_abierta -> Puerta_cerrada @ i_doorC;
				}
			}
		}
	}
}
```

Bloque 31.2: Máquina de estados del tarjetero

La rutina principal del billetero comienza con el estado *Idle*, el cual abandona cuando la unidad central ordena una extracción de dinero por parte del cliente o una recuperación por parte de un empleado a través de la puerta trasera, en cuyo caso el billetero debe permanecer a la espera hasta finalizar dicha acción.

- Unidad Central:
    - Para la unidad de control no se han definido interfaces ya que todas las que requiere y provee se han definido anteriormente para los demás componentes. Por tanto solo será necesario importarlas desde los ficheros en los que sea necesario.
    - El componente de la unidad central se ha definido con el código que se muestra en el Bloque 32.

```
import "I_Keyboard.idt";
import "I_Display.idt";
import "I_Print.idt";
import "I_Card.idt";
import "I_Cash.idt";
import "StateM_UC.xta";

simpleCD Unidad_Central behaviours StateM_UC {
      ports {
            KeyBoard_PortUC -c I_Keyboard;
            Display_PortUC -c I_Display;
            Printer_PortUC -c I_Print;
            Card_PortUC -c I_Card_P -o I_Card_R;
            Cash_PortUC -c I_Cash_P -o I_Cash_R;
            }
}
```

Bloque 32: Componente "Unidad de control"

Como se dijo anteriormente, el componente importa todas las interfaces para poder asociarlas a sus puertos.

- En esta máquina de estados (Figura 24) se han obviado muchos estados y se ha centrado el diseño en un funcionamiento básico del cajero. Se muestra la implementación de la máquina en los Bloques 33.1 y 33.2.

```
import "I_Keyboard.idt";
import "I_Display.idt";
import "I_Print.idt";
import "I_Card.idt";
import "I_Cash.idt";

timedAutomata StateM_UC {
      events {
      providedService_Requests e_card<-hay_tarjeta, e_door_cashO<-
      puertaC_abierta, e_door_cashC<-puertaC_abierta,e_door_cardO<-
      puertaT_abierta, e_door_cardC<-puertaT_abierta;
      internalSignals i_checked, i_error, i_wrong, i_ok;
      timeEvents one_seg(>1s);
      requiredService_Done d_reset->reset, d_leerNum->leerNum,
      d_readStr-  >readStr, d_readKey->readKey, d_recu_card->recuperarT,
      d_recu_cash->recuperarC, d_confiscar->confiscar,
      d_expulsar>expulsar,d_retirar->retirar;
      }
      region Principal {
            state Rutina{
                  region Rutina{
                        initial ini;
                        state Hardware_check;
                        state Error;
                        state Idle;
                        state Leer_Tarjeta;
                        state Pedir_PIN;
```

Bloque 33.1: Máquina de estados de la unidad central

```
                                        state Comprobar_PIN;
                                        state Contador;
                                        state Cantidad;
                                        state Entregar_Efectivo;
                                        transitions{
                                              ini -> Hardware_check;
                                              Hardware_check -> Idle @ i_checked;
                                              Idle -> Error @ i_error;
                                              Idle -> Leer_Tarjeta @ e_card;
                                              Idle -> Idle @ d_recu_card;
                                              Idle -> Idle @ d_recu_cash;
                                              Leer_Tarjeta -> Pedir_PIN @ d_leerNum;
                                              Leer_Tarjeta -> Error @ i_error;
                                              Pedir_PIN -> Comprobar_PIN @ d_readStr;
                                              Comprobar_PIN -> Contador @ i_wrong;
                                              Contador -> Pedir_PIN;
                                              Contador -> Idle @ d_confiscar;
                                              Comprobar_PIN -> Cantidad @ i_ok;
                                              Cantidad -> Entregar_Efectivo @ d_readKey;
                                              Entregar_Efectivo -> Idle @ d_expulsar;
                                              Entregar_Efectivo -> Error @ i_error;
                                              Error -> Hardware_check @ d_reset;
                                        }
                                  }
                            }
                      }
}
```

Bloque 33.2: Máquina de estados de la unidad central

Dentro de los estados de esta máquina se producen actividades, esto es, que en el estado *Leer Tarjeta* se invoca el servicio *leerNum()* del tarjetero, en *Pedir PIN* se llama a los servicios que permiten mostrar por pantalla mensajes y el que permite leer del teclado, en *Contador* se controla el número de veces que se introduce un PIN erróneo y se confisca la tarjeta si es necesario, en *Cantidad* se lee la pulsación del cliente que determina el dinero a sacar llamando al servicio oportuno y en *Entregar Efectivo* se llama a *retirar()* para que el billetero ponga el dinero a disposición del cliente y luego a *expulsar()* para que éste pueda recoger su tarjeta.

Figura 24: Máquina de estados de la unidad de central de control

Lo que se ha implementado anteriormente son en realidad las definiciones de los componentes. Es ahora cuando se instancian, relacionándolos con su definición, y se acoplan unos con otros tal y como se muestra en la Figura 21 mediante el fichero .app del Bloque 34.

```
import "Card.cdr";
import "Cash.cdr";
import "Display.cdr";
import "Keyboard.cdr";
import "Printer.cdr";
import "Unidad_Central.cdr";
import "StateM_Card.xta";
import "StateM_Cash.xta";
import "StateM_Display.xta";
import "StateM_Keyboard.xta";
import "StateM_Printer.xta";
import "StateM_UC.xta";

application Cajero {
	component Tarjetero: Card selectedBehaviour StateM_Card {
		ports{
			P_Tarjetero: Card_Port;}
	}
	component Billetero: Cash selectedBehaviour StateM_Cash {
		ports{
			P_Billetero: Cash_Port;}
	}
	component Pantalla: Display selectedBehaviour StateM_Display {
		ports{
			P_Pantalla: Display_Port;}
	}
	component Teclado: Keyboard selectedBehaviour StateM_Keyboard {
		ports{
			P_Teclado: Keyboard_Port;}
	}
	component Impresora: Printer selectedBehaviour StateM_Printer {
		ports{
			P_Impresora: Printer_Port;}
	}
	component Unidad_Control: Unidad_Central selectedBehaviour StateM_UC
{
		ports{
			UC_Pantalla: Display_PortUC;
			UC_Teclado: Keyboard_PortUC;
			UC_Impresora: Printer_PortUC;
			UC_Tarjetero: Card_PortUC;
			UC_Billetero: Cash_PortUC;}
	}
	portLinks {
		P_Tarjetero <-> UC_Tarjetero;
		P_Billetero <-> UC_Billetero;
		P_Impresora <-> UC_Impresora;
		P_Pantalla <-> UC_Pantalla;
		P_Teclado <-> UC_Teclado;}
}
```

Bloque 34: Fichero de aplicación del caso de uso

CAPÍTULO 6º

Conclusiones y Líneas De Trabajo Futuras

En este capítulo se exponen las conclusiones más relevantes del Proyecto alcanzadas durante su realización. Para terminar, se introducen algunas posibles líneas de trabajo futuro.

6.1. Conclusiones

A lo largo del desarrollo de este Proyecto se han alcanzado varias conclusiones, entre las que cabe destacar las siguientes:

- Se han desarrollado los editores para un lenguaje dirigido al modelado de aplicaciones basadas en componentes.

- El desarrollo de estas herramientas repercute en trabajos que se llevan en paralelo en el mismo departamento, como por ejemplo, la implementación de un framework de cohesión y traducción de código de componentes para la generación automática de aplicaciones. Además, se demuestra la aplicabilidad de un paradigma software de gran potencia y actualmente en plena ebullición, como es el DSDM. Esta visión podría repercutir positivamente sobre los procesos productivos actuales causando un gran impacto.

- Se eleva el nivel de abstracción de desarrollo basado en componentes, de forma que el usuario ha de ser solamente experto en el ámbito de aplicación y no tiene por qué poseer conocimientos técnicos específicos de la plataforma de desarrollo.

- El lenguaje creado es amigable y fácil de aprender. Además, los editores poseen una serie de mejoras y ayudas al usuario que hacen que la labor sea aún más sencilla,

permitiendo incluso a alguien sin conocimientos previos del lenguaje crear aplicaciones sencillas.

- Según la experiencia obtenida al realizar el Proyecto, con el uso del DSDM los tiempos de desarrollo de software se acortan notablemente y la complejidad disminuye. Esto es debido, básicamente, a que se coloca al desarrollador en un nivel más alto de abstracción, ya que solo se le exponen las partes relevantes de la aplicación y se ocultan los aspectos de implementación más básicos. Aún así el desarrollador tiene la libertad suficiente como para ir a un nivel más bajo y conseguir mayor detalle.

- Cabe destacar la potencia del entorno Eclipse con los plug-ins DSDM. Eclipse proporciona un marco de trabajo en el que el desarrollo de aplicaciones se hace notablemente más rápido y llevadero. Gracias a ello apenas se ha escrito código Java en la creación de las herramientas ya que se genera automáticamente a partir del meta-modelo suministrado.

- Como punto negativo, cabe mencionar la poca documentación existente sobre el uso de las herramientas de Eclipse, muchas veces, ésta es ciertamente confusa e incompleta.

6.2. Líneas de Trabajo Futuras

En este punto se incluyen algunas posibles extensiones y mejoras que pueden ser interesantes para los editores.

- Definición de librerías de componentes específicos para aplicaciones de interés.

- Depuración del meta-modelo para su simplificación y posible inclusión de mejoras, nuevas vistas o funcionalidades.

- Implementación de transformaciones para la generación de código ejecutable a partir de modelos definidos con los editores.

- Implementación de los editores en una versión más actual de las herramientas, para comprobar su compatibilidad y hacer uso de las posibles mejoras que proporciona al desarrollador.

- Mejoras en los asistentes de contenido de los editores, para hacer la labor del usuario aún más ágil y cómoda.

- Adición de nuevas restricciones OCL para guiar mejor al usuario en la relación de las diferentes clases del meta-modelo y facilitar así el uso de los editores.

Referencias bibliográficas

[1] Alonso, D., Vicente-Chicote, C., Ortíz, F., Pástor, J.A., Álvarez, B.: "V3CMM: a 3-view component meta-model for model-driven robotic software development". Journal of Software Engineering for Robotics. Enero 2010, vol.1, nº1

[2] T. Stahl, M. Voelter, and K. Czarnecki, Model- Driven Software Development: Technology, Engineering, Management, ed. Wiley, 2006, ISBN: 978-0-470-02570-3.

[3] Xtext, http://www.eclipse.org/Xtext/

[4] EMF, http://eclipse.org/modeling/emf/

[5] C. Vicente-Chicote, D. Alonso y J. F. Inglés-Romero (2011). *Introducción Práctica al Desarrollo de Software Dirigido por Modelos.* Universidad Politécnica de Cartagena.

[6] Coloración de sintaxis, http://www.eclipse.org/Xtext/documentation/2_1_0/195-highlighting.php

[7] Identificación del ámbito, http://www.eclipse.org/Xtext/documentation/2_1_0/140-labelprovider.php

[8] ContentAssist, http://www.eclipse.org/Xtext/documentation/2_1_0/150-contentassist.php

[9] GMF, http://www.eclipse.org/modeling/gmp/

[10] Wikipedia: Editor textual, http://es.wikipedia.org/wiki/Editor_de_texto

[11] Google Guice, http://es.wikipedia.org/wiki/Google_Guice

[12] "Object Constraint Language (OCL) Specification v2.0, The Object Management Group. 2006."

[13] Descarga Eclipse Modeling Tools, http://www.eclipse.org/downloads/

[14] I. Jacobson, G. Booch y J. Rumbaugh (2007). *El proceso unificado de desarrollo de software*. Pearson Addison-Wesley.

APÉNDICE 1

Vistas

En este apéndice se explica más detalladamente el meta-modelo de V3CMM. Para facilitar dicha labor se ha dividido en 4 partes, que se corresponden con cada unas de las vistas ya mencionadas. Se puede observar que todas las vistas heredan de la meta-clase ROOT y que por tanto tienen acceso a la clase import que más tarde servirá para acoplar los ficheros entre ellos.

1.1. Vista de Interfaces y tipos de datos

Esta vista es “común” a las demás, ya que todas pueden hacer uso de tipos de datos definidos en ella mediante la sentencia *import*, tal y como se mencionó más arriba. Además, los puertos de la vista de componentes hacen uso de las interfaces definidas también desde estos ficheros. Se muestra en la Figura 25 las clases del meta-modelo que sirven para dar soporte a esta vista.

Como ya se ha dicho anteriormente, todas las vistas tienen una clase raíz que hereda de *ROOT*. En este caso, *ROOT_IDT* es el elemento raíz de la vista. En él se contienen cero o más (cardinalidad) interfaces y tipos de datos. A su vez, estos pueden saber quién los contiene mediante la relación “*EOpposite*” que se ha denominado “*owner*”.

Las interfaces se identifican, como casi todas las clases del meta-modelo, mediante un nombre obligatorio (cardinalidad 1) y una documentación y versión opcionales. La clase *Interface* tiene una referencia a ella misma, denominada “*extends*”. Esta referencia significa que una interfaz puede heredar de otra.

Los servicios que ofrece un puerto están ocultos en la interfaz que los contiene. Una interfaz debe tener al menos un servicio, que se denotará con un nombre obligatorio.

Los servicios necesitan hacer uso de parámetros. Estos parámetros son los que define la clase *Parameter*. Para reconocer un parámetro se le da un nombre obligatorio y se indica "de dónde viene", es decir, si se toma de una salida, si se pasa por referencia, etc.

Los parámetros pueden ser de diferentes tipos, es por esto que necesita una referencia a *Datatype*.

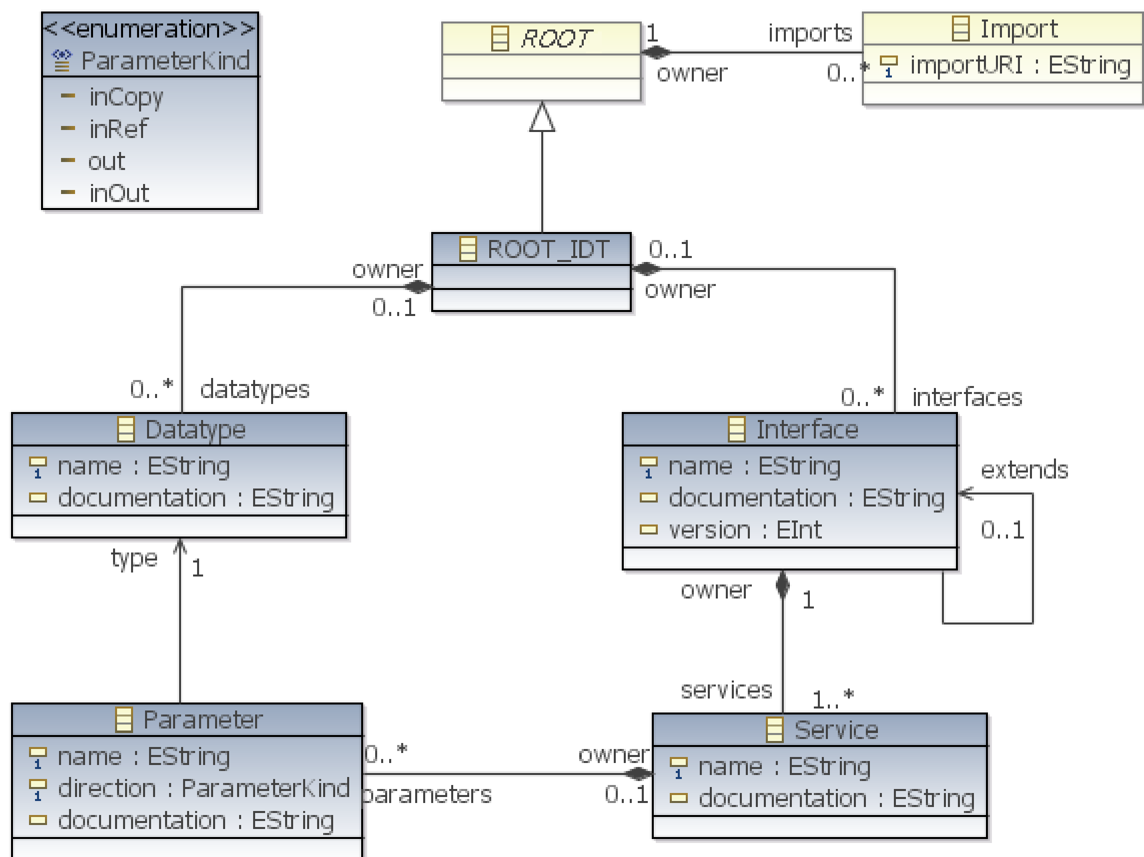

Figura 25: Parte del meta-modelo correspondiente a la vista de Interfaces y Tipos de Datos

1.2. Vista de componentes

La vista de componentes o estructural es la que se utiliza para definir el esqueleto de la aplicación. En ella se definen todos los componentes que la componen además de sus puertos. En la Figura 26 se muestra el conglomerado de clases que modela la vista de componentes.

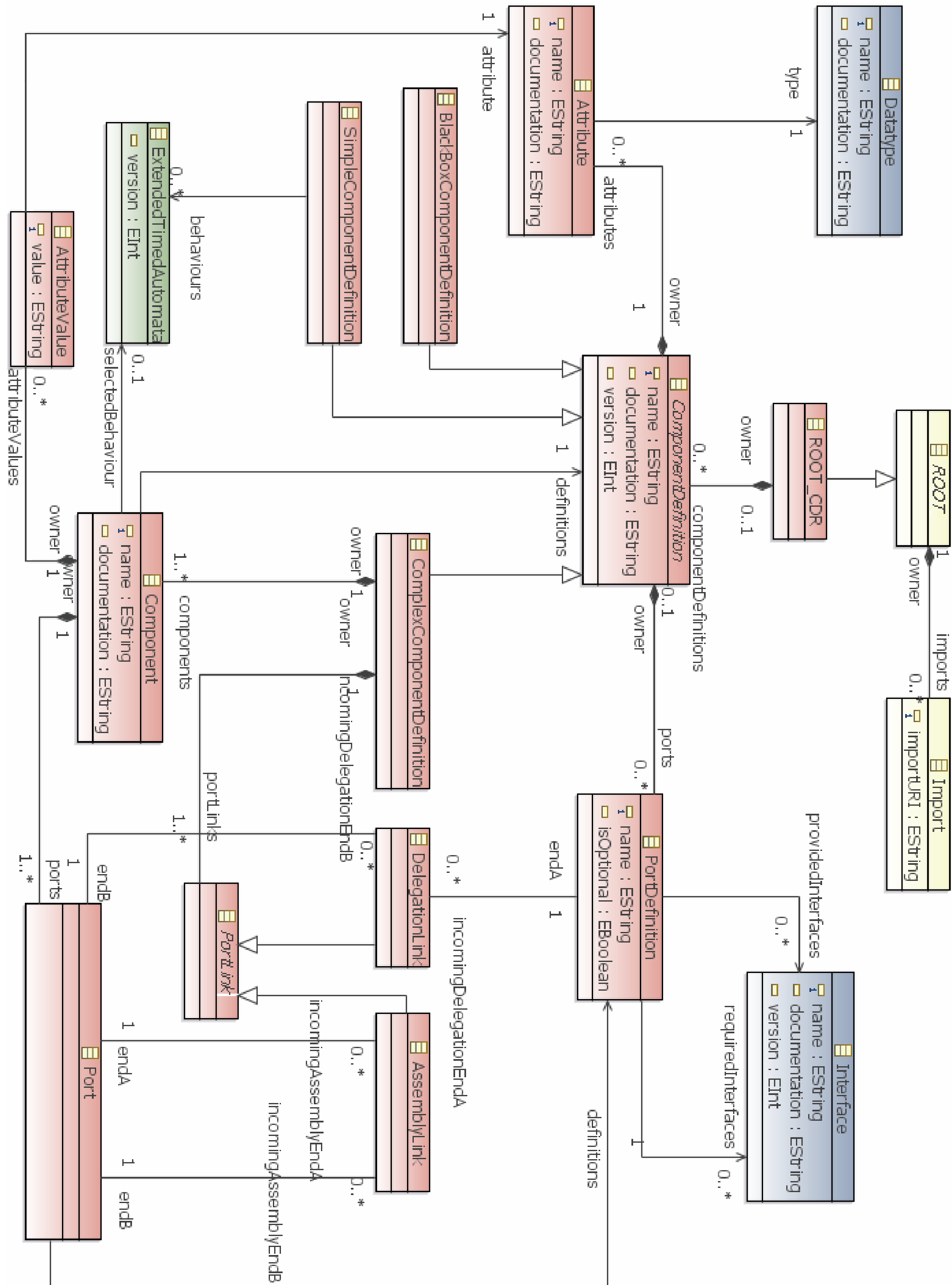

Figura 26: Parte del meta-modelo correspondiente a la vista de Componentes

El primer elemento que contiene el *ROOT* de esta vista es una o más definiciones de componente caracterizadas por la meta-clase *ComponentDefinition*. Estas definiciones de componente pueden ser de diferentes tipos debido a que de ella heredan 3 subclases tal y como se observa en el meta-modelo: *ComplexComponentDefintion*, *SimpleComponentDefinition* o *BlackBoxComponentDefinition*.

Las definiciones de componente contienen a su vez definiciones de puertos y atributos. Las definiciones de puertos tienen dos referencias a cero o más interfaces (de la vista de interfaces y tipos de datos), una referencia para interfaces requeridas (*requiredInterfaces*) y otra para proveídas (*providedInterfaces*). Los atributos, al igual que los parámetros en la vista anterior, deben tener una referencia a *Datatype* para saber de qué tipo son.

Una definición de componente simple solo tiene una referencia a la máquina de estados que modela su comportamiento, cosa de la que carece la caja negra. Ambos contienen definiciones de puertos y atributos, ya que heredan esa posibilidad.

El componente complejo tiene las mismas características que los anteriores pero, por su propia naturaleza, debe contener además componentes y enlaces para interconectarlos, representados por las clases *Component* y *PortLink* y contenidos mediante las relaciones de composición *components* y *portlinks* respectivamente. Como se observa en la cardinalidad de sendas relaciones, el componente complejo debe contener al menos una instanciación de cada clase.

Cada componente inmerso en un componente complejo contendrá sus propios puertos y asignará valores a los atributos que contiene su definición.

De la clase *PortLink* heredan los dos tipos de enlaces que pueden ser utilizados: los de tipo *Assembly*, que sirven para interconectar puertos de diferentes componentes; y los de tipo *Delegation*, que sirven para conectar componentes internos con el componente contenedor o, lo que es lo mismo, con la definición del puerto.

1.3. Vista de máquina de estados

Esta vista, como su nombre indica, es la que se utiliza para definir la máquina de estados que "está dentro" de su componente correspondiente. Se modela con la parte del meta-modelo que se ve en la Figura 28.

En primer lugar, y como en las anteriores vistas, está la clase contenedora, *ROOT_IDT*, que contiene inmediatamente un autómata temporizado, que sería la representación de la máquina de estados. Dicho autómata contiene regiones ortogonales y eventos. Las regiones ortogonales son las que contienen los estados de la máquina. El distinguir entre autómatas y regiones, permite tener varias máquinas de estado ejecutándose en paralelo en un mismo componente. Los estados posibles heredan todos de un mismo vértice y pueden ser de tipo: estado final; pseudoestado, dentro del cual se encuentran otros tipos como son el estado de bifurcación (fork) o el estado de unión (join) ó estado, el cual puede contener otra region con mas estados dentro de la misma. Los eventos sirven para disparar las transiciones de un estado a otro o para ir directamente a un estado sin pasar por ningún otro (clases *Transition* y *Reaction* respectivamente). Los eventos pueden ser de tipo externo ó interno y cada uno de ellos tiene otros tipos de eventos según servicios, límites de tiempo o señales procedentes de otros estados. Estos eventos pueden ir condicionados por las guardas, que son condiciones que deben cumplirse a la vez que los eventos para que pueda producirse la transición entre

estados. Estas guardas pueden ser de temporizadas o pueden ser definidas por el propio usuario. Un ejemplo para que se comprenda con facilidad el funcionamiento de las guardas sería ver la posibilidad de que nuestra máquina de estados cumpla que: “la transición del estado 1 al estado 2 debe realizarse sí y solo sí llega un evento procedente del estado 3 y han pasado más de 5 segundos desde la última vez que se estuvo en el estado 2”.

1.4. Vista de aplicación

Por último, la vista de aplicación es la que nos permite “compilar” todo lo implementado en un ejecutable. Su sintaxis esta implementada en la sección del meta-modelo de la Figura 27.

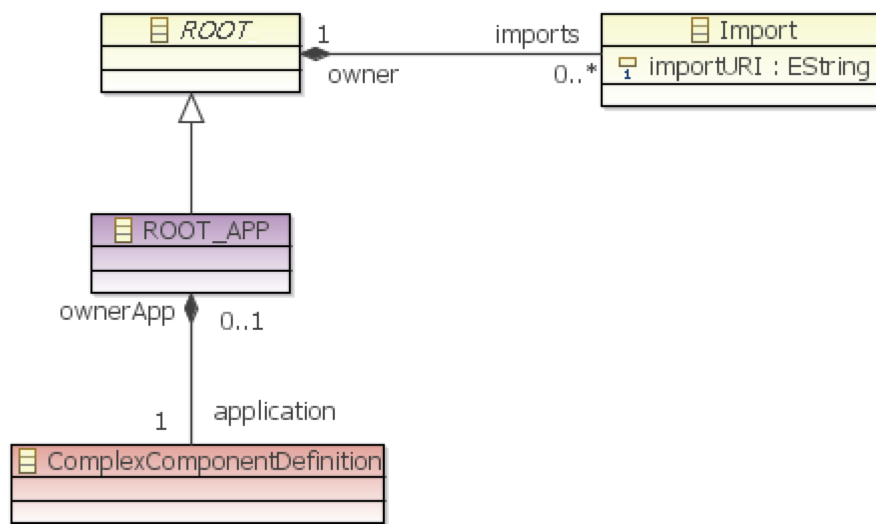

Figura 27: Parte del meta-modelo correspondiente a la vista de aplicación

Como se puede observar, esta vista es tremendamente sencilla, ya que solo está compuesta por su clase raíz haciendo de contenedor para una definición de componente complejo, ya que toda la aplicación es un conjunto de componentes que pueden ser enmarcados en ella. Este hecho quedaba reflejado en el punto 3.2, donde se muestra un esquema simbólico del funcionamiento de V3CMM. La vista de aplicación quedaba representada por el cuadro negro que contenía a todo lo demás.

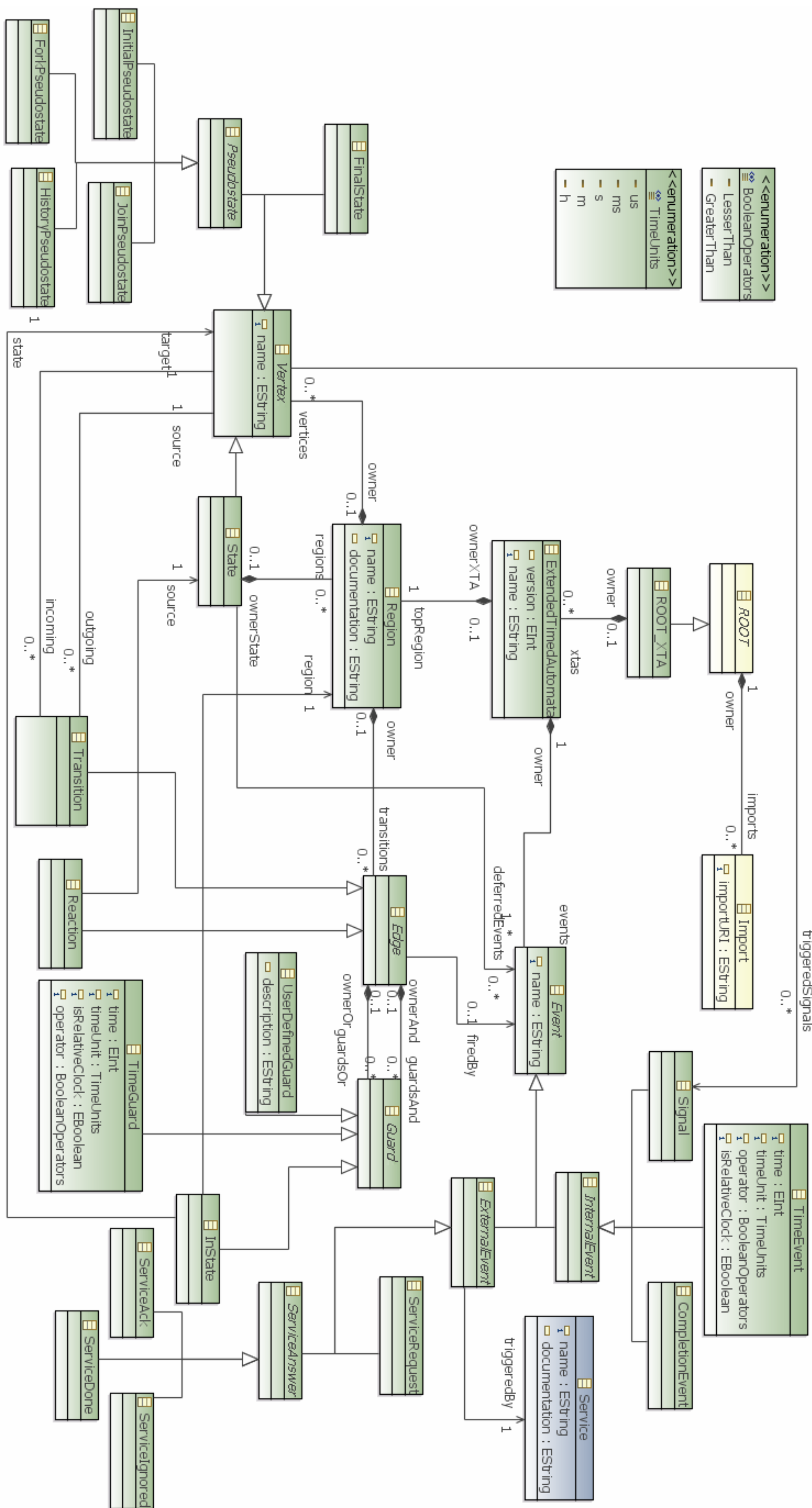

Figura 28: Parte del meta-modelo correspondiente a la vista de máquinas de estado

APÉNDICE 2

Restricciones OCL adicionales

En éste segundo apéndice se detallan todas y cada una de las restricciones OCL añadidas al meta-modelo que hacen posible la corrección de errores a la hora de definir el código. Se irán describiendo las restricciones según aparecen en cada uno de los elementos del meta-modelo.

- Restricciones en *Interface*:

Restricción *repeatedService*: Una interfaz no puede tener servicios repetidos. Dos servicios son iguales si se llaman igual y tienen los mismos parámetros.

```
class Interface{
	invariant repeatedService:
		self.services -> forAll (s1, s2 : Service | s1 <> s2
		implies s1.name <> s2.name or s1.parameters ->
	forAll (j : Parameter | s2.parameters -> forAll (k :
	Parameter | k.name <> j.name or k.direction <>
	j.direction)));
	property services#owner : Service[+] { composes };
	attribute name : String[1];
	property _'extends' : Interface[?];
	attribute _'documentation' : String[?];
	attribute version : ecore_0::EInt[?];
	property owner#interfaces : ROOT_IDT[?];
}
```

- Restricciones en *PortDefinition*:

Restricción *repeatedRequiredInterface*: Un *PortDefinition* no puede requerir la misma interfaz más de una vez. Dos interfaces son iguales si tienen los mismos servicios.

Restricción *requiringAndProvidingSameInterface*: Un *PortDefinition* no puede requerir y proveer a la vez la misma interfaz.

Restricción *repeatedProvidedInterface*: Un *PortDefinition* no puede proveer la misma interfaz más de una vez.

```
class PortDefinition{
      invariant repeatedRequiredInterface:
                  self.requiredInterfaces -> forAll (i1, i2 :
                  Interface | i1 <> i2 implies i1.name <> i2.name);
      invariant requiringAndProvidingSameInterface:
                  self.providedInterfaces -> forAll (i : Interface |
                  not self.requiredInterfaces -> exists (j :
                  Interface | j=i));
      invariant repeatedProvidedInterface:
                  self.providedInterfaces -> forAll (i1, i2 :
                  Interface | i1 <> i2 implies i1.name <> i2.name);
      attribute name : String[1];
      property providedInterfaces : Interface[*];
      property requiredInterfaces : Interface[*];
      property owner#ports : ComponentDefinition[?];
      property incomingDelegationEndA#endA : DelegationLink[*];
}
```

- Restricciones en *ComplexComponentDefinition*:

Restricción *allPortsConnected*: Todos los puertos de un *ComplexComponentDefinition* deben estar conectados.

Restricción *IfOwnerIsRootAppNoAttributes*: Si el *ComplexComponentDefinition* está contenido en un padre de tipo aplicación (*ROOT_APP*) no puede contener atributos (*Attribute*).

```
class ComplexComponentDefinition extends ComponentDefinition{
      invariant allPortsConnected:
            self.components->forAll ( c : Component | c.ports->
            forAll ( p  : Port | (p.incomingAssemblyEndA->size()=1)
            and ( p.incomingAssemblyEndB->size()=1) and
            (p.incomingDelegationEndB->size()=1) ) ) and self.ports
            ->forAll( d : PortDefinition | d.incomingDelegationEndA
            ->size()=1);
      invariant IfOwnerIsRootAppNoAttributes:
            self.ownerApp->size()=1 implies self.attributes
            ->size()=0;
      property components#owner : Component[+] { composes };
      property portLinks#owner : PortLink[+] { composes };
      property ownerApp#application : ROOT_APP[1];
}
```

- Restricciones en *AssemblyLink*:

Restricción *selfComponentPortConnectionNotAllowed_AssemblyLink*: Un *AssemblyLink* no puede crear conexiones reflexivas de un componente.

Restricción *componentsContainedInSameComplexComponent_AssemblyLink*: Un link de tipo *AssemblyLink* no puede conectar componentes que no pertenezcan al mismo componente complejo.

Restricciones *Ass_compatiblePorts_required_endA*, *Ass_compatiblePorts_provided_endA*, *Ass_compatiblePorts_provided_endB* y *Ass_compatiblePorts_provided_endB*: Un link de tipo *AssemblyLink* debe conectar solamente puertos compatibles, esto es, que uno requiera lo que el otro provee y viceversa.

```
class AssemblyLink extends PortLink{
      invariant selfComponentPortConnectionNotAllowed_AssemblyLink:
            self .endA.owner <> self.endB.owner;
      invariant componentsContainedInSameComplexComponent_AssemblyLink:
            self .endA.owner.owner = self.endB.owner.owner;
      invariant Ass_compatiblePorts_required_endA:
            self.endA.definitions.requiredInterfaces -> forAll (i:
            Interface | self.endB.definitions.requiredInterfaces ->
            exists (j : Interface | j = i));
      invariant Ass_compatiblePorts_provided_endA:
            self.endA.definitions.providedInterfaces -> forAll (i:
            Interface | self.endB.definitions.providedInterfaces ->
            exists (j : Interface | j = i));
      invariant Ass_compatiblePorts_required_endB:
            self.endB.definitions.requiredInterfaces -> forAll (i:
            Interface | self.endA.definitions.requiredInterfaces ->
            exists (j : Interface | j = i));
      invariant Ass_compatiblePorts_provided_endB:
            self.endB.definitions.providedInterfaces -> forAll (i:
            Interface | self.endA.definitions.providedInterfaces ->
            exists (j : Interface | j = i));
      property endA#incomingAssemblyEndA : Port[1];
      property endB#incomingAssemblyEndB : Port[1];
}
```

- Restricciones en *DelegationLink*:

Restricciones *Del_compatiblePorts_required_endA*, *Del_compatiblePorts_provided_endA*, *Del_compatiblePorts_required_endB* y *Del_compatiblePorts_provided_endB*: Un link de tipo *DelegationLink* debe conectar un *Port* con su *PortDefinition*, por tanto debe asegurarse que ambos proveen y requieren las mismas interfaces. Cada una de las restricciones añadidas se asegura por separado de que las interfaces requeridas y proveídas de cada puerto sea compatible.

```
class DelegationLink extends PortLink{
	invariant Del_compatiblePorts_required_endA:
		self.endA.requiredInterfaces -> forAll (i: Interface |
		self.endB.definitions.requiredInterfaces -> exists (j :
		Interface | j = i));
	invariant Del_compatiblePorts_provided_endA:
		self.endA.providedInterfaces -> forAll (i: Interface |
		self.endB.definitions.providedInterfaces -> exists (j :
		Interface | j = i));
	invariant Del_compatiblePorts_required_endB:
		self.endB.definitions.requiredInterfaces -> forAll (i:
		Interface | self.endA.requiredInterfaces -> exists (j :
		Interface | j = i));
	invariant Del_compatiblePorts_provided_endB:
		self.endB.definitions.providedInterfaces -> forAll (i:
		Interface | self.endA.providedInterfaces -> exists (j :
		Interface | j = i));
}
```

- Restricciones en *Component*:

Restricción *NoAtributesValue*: Un *Component* debe dar valor a todos los atributos que contenga su definición.

Restricción *EqualNumberPort*: Un *Component* debe tener tantos puertos como definiciones de puertos tenga su definición.

Restricción *NoServiceRequest*: Tiene que haber un *ServiceRequest* por cada uno de los servicios ofrecidos por el componente cuyo comportamiento describe este autómata temporizado.

Restricción *SelectedBehaviourOnlyIfDefinitionIsSimpleComponent*: Sólo cuando un *Component* tiene por definición un *SimpleComponentDefinition* tiene que estar fijada la referencia *selectedBehaviour*.

```
class Component{
	invariant NoAtributesValue:
		self.attributeValues->size()=
		self.definitions.attributes->size();
	invariant EqualNumberPort:
		self.ports->size()=self.definitions.ports->size();
	invariant NoServiceRequest:
		self.definitions.ports.providedInterfaces.services->
		size() =ServiceRequest->allInstances()->size();
	invariant SelectedBehaviourOnlyIfDefinitionIsSimpleComponent:
		not(self.definitions.oclIsTypeOf
		(SimpleComponentDefinition)) implies selectedBehaviour->
		size()=0;
	attribute name : String[1];
	property ports#owner : Port[+] { composes };
	property attributeValues#owner : AttributeValue[*]{ composes };
	property definitions : ComponentDefinition[1];
	property selectedBehaviour : ExtendedTimedAutomata[?];
	property owner#components : ComplexComponentDefinition[1];
}
```

- Restricciones en *TimeEvent*:

Restricción *TimeMoreThan0*: El valor de tiempo fijado en un *TimeEvent* debe ser mayor que cero.

```
class TimeEvent extends InternalEvent{
      invariant TimeMoreThan0:
            self.time>0;
      attribute time : ecore_0::EInt[1];
      attribute timeUnit : TimeUnits[1];
      attribute isRelativeClock : Boolean[1];
      attribute operator : BooleanOperators[1];
}
```

- Restricciones en *Region*:

Restricción *VertexConnected*: Todos los *Vertex* contenidos en una *Region* deben estar conectados.

Restricción *noInitialOrHistoricPseudostateInRegion*: Sólo tiene que haber un pseudo-estado de tipo inicial o histórico en una misma *Region*.

```
class Region{
      invariant VertexConnected:
            self.vertices->forAll(v : Vertex | self.outgoing ->
            size()=1 and self.incoming->size()=1);
      invariant noInitialOrHistoricPseudostateInRegion:
            self.vertices ->select(p: Pseudostate | p.oclIsTypeOf
            (InitialPseudostate) or
            p.oclIsTypeOf(HistoryPseudostate))->size=1;
      attribute name : String[1];
      property transitions#owner : Edge[*] { composes };
      property vertices#owner : Vertex[*] { composes };
      attribute _'documentation' : String[?];
      property ownerXTA#topRegion : ExtendedTimedAutomata[?];
      property ownerState#regions : State[?];
}
```

- Restricciones en *FinalState*:

Restricción *finalStateWithOutgoingTransition*: Un *FinalState* no tiene transiciones de salida ya que marca el final de una máquina de estados.

```
class FinalState extends Vertex{
      invariant finalStateWithOutgoingTransition:
            self.outgoing->size()<>0;
}
```

- Restricciones en *TimeGuard*:

Restricción *TimeMoreThan0*: El valor de tiempo fijado en un *TimeGuard* debe ser mayor que cero.

```
class TimeGuard extends Guard{
      invariant TimeMoreThan0:
            self.time>0;
      attribute time : ecore_0::EInt[1];
      attribute timeUnit : TimeUnits[1];
      attribute isRelativeClock : Boolean[1];
      attribute operator : BooleanOperators[1];
}
```

- Restricciones en *InState*:

Restricción *NoStateInRegion*: Cuando se utiliza una guarda del tipo *InState*, el *State* al que apunta debe pertenecer a la *Region* a la que apunta.

```
class InState extends Guard{
      invariant NoStateInRegion:
            self.state->select(s:State|self.region.vertices)->size()=1;
      property region : Region[1];
      property state : Vertex[1];
}
```

- Restricciones en *InitialPseudoState*:

Restricción *initialPseudostateWithoutIncomingTransition*: Un *InitialPseudostate* no tiene transiciones de entrada ya que marca el inicio de una máquina de estados.

Restricción *initialPseudostateWithOutgoingTransition* : Además sólo tiene una transición de salida que no debe tener guardas.

```
class InitialPseudostate extends Pseudostate{
      invariant initialPseudostateWithoutIncomingTransition:
            self.incoming->size()=0;
      invariant initialPseudostateWithOutgoingTransition:
            (self.outgoing->size()=1) and (self.outgoing.guardsAnd->
            size()=0 and self.outgoing.guardsOr->size()=0);
}
```

- Restricciones en *HistoryPseudostate*:

Restricción *HistoryPseudostateWithoutIncomingTransition*: Un *HistoryPseudostate* sólo tienen una transición saliente y sin guarda.

Restricción *HistoryPseudostateWithOutgoingTransition* : Además de no tener transiciones entrantes.

```
class HistoryPseudostate extends Pseudostate{
      invariant HistoryPseudostateWithoutIncomingTransition:
            self.incoming->size()=0;
      invariant HistoryPseudostateWithOutgoingTransition:
            (self.outgoing->size()=1)and(self.outgoing.guardsAnd->
            size()=0 and self.outgoing.guardsOr->size()=0);
}
```

- Restricciones en *JoinPseudostate*:

Restricción *noGuardInJoinIncomingTransitions:* Las transiciones que llegan a un *JoinPseudostate* no tienen guarda.

Restricción *noMoreThanOneTransitionPerRegionToJoin* : Además no puede llegar más de una transición por *Region* a un *JoinPseudostate*.

```
class JoinPseudostate extends Pseudostate{
      invariant noGuardInJoinIncomingTransitions:
            self.incoming->forAll(t: Transition | t.guardsAnd->
            size()=0 and t.guardsOr->size()=0);
      invariant noMoreThanOneTransitionPerRegionToJoin:
            self.incoming->forAll(t1,t2 : Transition | t1<>t2
            implies t1.source.owner<>t2.source.owner);
}
```

- Restricciones en *ForkPseudostate*:

Restricción *forkTransitionsGoToOrthogonalRegionsInSameMacroState:* No pueden salir más transiciones de *ForkPseudostate* que regiones ortogonales tiene el macro-estado que se alcanza.

Restricción *noMoreCannotBeTargetedMoreThanOnceByFork*: además ninguna *Region* puede ser alcanzada más de una vez.

Restricción noGuardInForkOutgoingTransitions: Las transiciones de salida de un *ForkPseudostate* no tienen guarda

Restricción *forkStateWith1IncomingTransition*: Un *ForkPseudostate* sólo puede tener una transición entrante.

```
class ForkPseudostate extends Pseudostate{
      invariant forkTransitionsGoToOrthogonalRegionsInSameMacroState:
            self.outgoing->forAll(s1,s2: Transition | s1 <> s2
            implies s1.owner <> s2.owner and
            s1.owner.ownerState=s2.owner.ownerState);
      invariant noMoreCannotBeTargetedMoreThanOnceByFork:
            self.outgoing->forAll(t1,t2 : Transition | t1<>t2 implies
            t1.target.owner<>t2.target.owner);
      invariant noGuardInForkOutgoingTransitions:
            self.outgoing->forAll(t: Transition | t.guardsAnd->
            size()=0 and t.guardsOr->size()=0);
      invariant forkStateWith1IncomingTransition:
            self.incoming->size()=1;
}
```

- Restricciones en *ROOT_IDT*:

<u>Restricción *repeatedDataType*:</u> Un *Datatype* contenido en el *ROOT_IDT* debe ser único e irrepetible.

<u>Restricción *repeatedInterface*:</u> Una *Interface* contenida en el *ROOT_IDT* debe ser única e irrepetible.

```
class ROOT_IDT extends ROOT{
      invariant repeatedDataType:
            self.datatypes -> forAll (d1,d2: Datatype | d1<>d2
            implies d1.name <> d2.name);
      invariant repeatedInterface:
            self.interfaces -> forAll (i1, i2 : Interface | i1 <> i2
            implies i1.name <> i2.name and i1.services-> forAll (j:
            Service | i2.services -> forAll(k: Service |
            k.name<>j.name)));
      property datatypes#owner : Datatype[*] { composes };
      property interfaces#owner : Interface[*] { composes };
}
```

- Restricciones en *ROOT_APP*:

<u>Restricción *NoPortDefinition*:</u> El *ROOT_APP* no tiene *PortDefinition*.

```
class ROOT_APP extends ROOT{
      invariant NoPortDefinition:
            self.application.ports->size()=0;
      property application#ownerApp:ComplexComponentDefinition[1]
      {composes};
}
```