Universidad a distancia de Madrid
Escuela de Ciencias Técnicas e Ingeniería

Máster Universitario en Arquitectura del Software

Trabajo fin de master

Diseño de un DSL orientado a dispositivos conectados a internet mediante arquitectura MDE

Autor: César Laso Laso

Directores:
Dr. Javier Bravo Agapito
Dr. Javier Fernandez Salgado

Junio 2018

Índice general

1. Introducción **13**

1.1. Contextualización 13

1.2. Motivación 14

1.2.1. ¿Como vamos a mejorarlo? 14

1.2.2. ¿Por que debe ser mejorado? 15

1.3. Objetivos 15

1.3.1. Introducción 15

1.3.2. Objetivos generales 16

1.3.3. Objetivos específicos 16

2. Estado de la cuestión **19**

2.1. MDA - Arquitectura dirigida por modelos 20

2.1.1. Modelos 21

2.1.2. Meta-Modelos 21

2.1.3. Plataformas 21

2.2. Lenguajes específicos de dominio 22

2.2.1. Eclipse EMF 22

2.2.2. Xtext 24

2.2.3. MPS 25

2.3. Trabajos relacionados 26

2.3.1. DSL for arduino 26

2.3.2. OpenHab 28

3. Desarrollo **29**

3.1. Requisitos del proyecto 29

3.2. Modelos 30

3.2.1. Modelos relativos al dominio del dispositivo IOT 30

3.2.1.1. Modelo Programa 30

3.2.1.2. Modelo Operating System Actuators 32

3.2.1.3. Modelo GPIO 32

3.2.2. Modelo estado de salud 34

3.2.3. Modelos relativos al protocolo de comunicación 37

3.2.3.1. Modelo: Servicios de comunicación base 37

3.2.3.2. Modelo: Servicios de comunicación de estado 38

3.2.3.3. Modelo communication health services 40

3.2.3.4. Modelo: Dispositivos 41

3.2.4. Modelos relativos al servidor y usuarios 41

3.2.5. Modelos comunes al dominio del dispositivo IOT 41

3.2.5.1. Modelo Eventos/Transacciones 41

3.2.5.1.1. Modelo evento 41

3.2.5.1.2. Modelo actuador 44

3.2.5.2. Modelo Timers 45

3.2.5.3. Modelo Proyecto IOT 45

3.2.5.4. Modelo Programa 47

3.3. Transformación de modelo ECore Eclipse Modeling framework (ECore) a modelo genmodel. Generación de proyectos editor Java a partir de modelos ECore 49

3.3.1. Código del modelo 49

3.3.2. Edición del modelo 51

3.3.3. Editor del modelo 56

3.3.4. Test unitarios 57

3.3.5. Pasos a realizar en la transformación 59

3.3.6. Generadores Ecore a genmodel para otros lenguajes 65

3.4. Editor de modelos gráfico generado a partir de los proyectos edit,editor 66

3.4.1. Ejecución de EMF en modo runtime 67

3.4.2. GMP Graphical model proyect 69

3.5. DSL 70

3.5.1. Generación de gramática a partir de modelos 71

3.5.2. Gramática 71

3.5.3. Ejemplos de programas 74

3.6. Ejemplos 76

3.7. Generación de código 77

3.7.1. Objetivos 77

3.7.2. Xtend 77

3.8. Recursos utilizados 78

4. Conclusiones y líneas futuras **80**

4.1. Conclusiones 80

4.1.1. Sobre Eclipse Modeling Framework 82

4.1.2. Sobre DSL 84

4.2. Líneas futuras 85

5. Referencias bibliográficas y recursos electrónicos **86**

6. Agradecimientos **90**

7. Autorización de difusión **91**

A. Modelos adicionales Ecore **92**

B. DSL **95**

B.1. Ejemplo de programa ejemplo - Versión conceptual 95

B.2. Gramática con el DSL del proyecto 96

B.3. Xtend lectura de la gramática DSL mediante ecore y posterior conversión del modulo de comunicaciones, apartado telemetría a su implementación en Java 106

C. Casos de uso - Ejemplos de programas escritos en el lenguaje creado 112

C.1. Programa adaptación maquinas con monedero electrónico a su versión conectada a internet 112

C.2. Programa modernización futbolín 114

C.3. Programa modernización ascensor de hotel sin pantalla gráfica 118

Índice de figuras

2.1. Definición de un modelo ECore en formato tree 23

2.2. Edición de una instancia generada a partir de un modelo ECore en su formato de visualización tree 23

2.3. Representación Extensible Markup Language (XML) del modelo ECore Events 24

2.4. IDE Eclipse con Xtext editando DSL 25

2.5. IDE MPS editando un DSL 26

2.6. IDE MBEDDR 27

2.7. DSL for Arduino ejemplo fragmento código 27

2.8. Openhab reglas ejemplo 28

3.1. Modelo Programa IOT. Diagrama de clases 31

3.2. Modelo OperatingSystemActuators. Diagrama de clases 32

3.3. Modelo GPIO. Diagrama de clases 34

3.4. Modelo GPIO - Eventos. Diagrama de clases 35

3.5. Modelo GPIO - Actuadores. Diagrama de clases 35

3.6. Modelo Health. Representación Ecore tree 36

3.7. Modelo Servicios de comunicación. Diagrama de clases 38

3.8. Modelo Servicios de comunicación - Estado de conexión.. Diagrama de clases 39

3.9. Modelo Servicios comunicación - Housekeeping (Estado de salud).. Diagrama de clases 40

3.10. Modelo Dispositivos. Diagrama de clases 42

3.11. Modelo Servidor. Diagrama de clases 43

3.12. Modelo Usuarios. Diagrama de clases 43

3.13. Modelo Eventos / Transacciones. Diagrama de clases 44

3.14. Modelo Timers. Representación Ecore tree 46

3.15. Modelo Proyecto IOT. Diagrama de clases 46

3.16. Modelo Programa IOT. Diagrama de clases 48

3.17. Especificación de propiedades del paquete y clases en su conversión a Java 50

3.18. Genmodel - Especificación de propiedades del atributo de clase en su conversión a Java 50

3.19. Fragmento de código con la interfaz en lenguaje Java generada a partir de modelo ECore Pin 52

3.20. Interfaz de la factoría Java generada a partir de modelo ECore. Con esta factoria instanciaremos objetos 52

3.21. Propiedades del paquete con metadatos generados a partir de ECore . 53

3.22. Resultado de la implementación de un interfaz generado por ECore . 54

3.23. Resultado de la implementación del interfaz de una factoría ECore . . 55

3.24. Proyecto genmodel Edit, visualización de paquetes 56

3.25. Proyecto genmodel Editor, visualización de paquetes 57

3.26. Proyecto genmodel Editor, código generado 58

3.27. Proyecto genmodel Tests, visualización de paquetes 58

3.28. Proyecto genmodel Tests, código generado 59

3.29. Creación de un archivo genmodel, paso 1 61

3.30. Creación de un archivo genmodel desde ECore, paso 2 61

3.31. Creación de un archivo genmodel desde ECore, paso 3 62

3.32. Genmodel desde otros tipos de modelado 62

3.33. Genmodel ruta del proyecto 63

3.34. Genmodel selección de raíz y dependencias 64

3.35. Genmodel selección de proyectos a generar 64

3.36. Genmodel generación de proyectos desde varios modelos simultáneamente 65

3.37. Genmodel selección de tipo de proyecto multiple 66

3.38. Ejecutar EMF en modo runtime desde proyecto existente - Paso 1 . . 67

3.39. Ejecutar EMF en modo runtime desde proyecto existente - Paso 2 . . 68

3.40. EMF Wizard - Creación de un objeto a partir de un modelo de forma gráfica 68

3.41. Editor gráfico del modelo a partir de metamodelo ecore 69

3.42. El editor gráfico nos ayuda en la selección de subelementos 69

3.43. GMF permite personalizar el editor de objetos modelo a un nivel mayor. 70

3.44. Creación de un proyecto Xtext desde un proyecto EMF ecore. 72

3.45. Xtext en su importación desde ECore. Selección de modelos que serán generados en la gramática y selección de elemento raíz. 72

3.46. Nombre de nuestro DSL y extensión de archivo para el editor Eclipse EMF. 73

3.47. DSL - Ejemplo de gramática para un evento 73

3.48. Editor DSL. Autocompletado con modelos que implementen eventos. 74

3.49. Editor DSL - Restricción en variables 75

3.50. Editor DSL - Tipado en atributos 75

3.51. Editor DSL - Autocompletado de variables 76

3.52. Código Xtend generando plantilla java 78

A.1. Proyecto IOT - Representación de las dependencias entre los modelos 93

A.2. Modelo Estado de salud. Diagrama de clases 93

A.3. Modelo Timers. Diagrama de clases 94

Resumen

El mercado tecnológico esta muy fragmentado respecto a soluciones Internet of things (IOT). Necesitamos que el desarrollo de productos basado en estos sistemas sea compatible con la mayoría de plataformas del mercado.

En este proyecto, pretendemos crear un Domain specific language (DSL) donde ortogonalizamos el diseño de elección tecnológica. De esta manera podemos especificar las relaciones entre los diferentes elementos que conforman el diseño en el propio vocabulario del área de negocio en cuestión. Este modelo, es formal y permite definir proyecciones a diferentes soluciones tecnológica sobre un lenguaje destino de propósito general.

Utilizando Model Driven Enginering (MDE) en su variante Model Driven Software Development (MDSD), mediante el entorno de desarrollo Eclipse Model Framework (EMF) podemos utilizar estas definiciones e implementarlas directamente en software mediante la transformación de modelos ECore Eclipse Modeling framework (ECore), evitando la descoordinación entre requisitos, documentación e implementación dentro de un proyecto software, así como la automatización en la validación y verificación del programa generado.

Palabras clave— MDE MDSD DSL IOT Ecore UML Xtext Xtend

Abstract

The technological market is very fragmented with respect to Internet of things (IOT) solutions. We need the product development based on these systems to be compatible with most market platforms.

In this project, we intend to create a Domain specific language (DSL) where we orthogonalize the design of technological choice. This way, we can specify the relationships between the different elements that make up the design in the vocabulary of the business area in question. The model, is formal and allows defining projections to different technological solutions on a general purpose target language.

Using Model Driven Enginering (MDE) in Model Driven Software Development (MDSD) variant, design and developt with the Eclipse Model Framework (EMF) development environment we can use these definitions and implement them directly in software through the transformation of ECore Eclipse Modeling framework (ECore) models, avoiding the lack of coordination between requirements, documentation and implementation within a software project, as automation in the validation and verification of the generated program.

Keywords— MDE MDSD DSL IOT Ecore UML Xtext Xtend

Capítulo 1

Introducción

1.1. Contextualización

Actualmente existen multitud de productos hardware orientados al Internet of things (IOT). Dispositivos de bajo coste, los cuales combinan hardware y software para conectar entradas y salidas físicas o digitales a internet.

IOT esta compuesto por diferentes soluciones tecnológicas. Esto implica tomar decisiones en función de las tecnologías presentes.

La naturaleza de este proyecto fin de máster esta condicionada por la asignatura *Diseño avanzado en arquitecturas software* [48], la cual trata las áreas Model Driven Enginering (MDE), Domain specific language (DSL) y patrones de arquitectura de software.

El mercado tecnológico esta muy fragmentado respecto a soluciones IOT. Contamos con plataformas basadas en hardware abierto tales como Raspberry Pi[17], BeagleBone [28] o Arduino [29].

Cada una de estas plataformas hardware cuenta con su entorno de desarrollo propio, siendo necesario código en un lenguaje específico para cada plataforma.

1.2. Motivación

Necesitamos que el desarrollo de productos basado en hardware sea compatible con la mayoría de plataformas del mercado, pudiendo de esta manera no depender de la plataforma específica seleccionada, ni del software requerido por esta.

Para ello, se hace necesario implementar nuestro desarrollo utilizando una abstracción de la plataforma de forma que sea totalmente independiente de esta, no quedando atado a ningún sistema hardware/software en concreto, incluyendo las partes especificas a la plataforma de forma independiente al desarrollo del producto.

Esto supone una ventaja competitiva al poder definir nuestro producto sin depender de una implementación concreta.

Que vamos a mejorar con la solución basada en MDE/Model Driven Software Development (MDSD):

- Alineación del modelo de negocio con el desarrollo software.
- Documentación actualizada entre modelos y código.
- Independencia de la plataforma hardware y software destino mediante la generación de código.

1.2.1. ¿Como vamos a mejorarlo?

- Mediante la creación de un lenguaje específico de dominio orientado a este tipo de producto.
- Con la creación de modelos que definan los elementos de este lenguaje y de los diferentes aspectos que integran este dominio, los cuales definan las cualidades y relaciones entre los diferentes elementos del diseño.
- Utilizar un DSL donde ortogonalizamos el diseño de elección tecnológica. De esta manera podemos especificar las relaciones entre los diferentes elementos que conforman el diseño en el propio vocabulario del área de negocio en cuestión. Este modelo, es formal y permite definir proyecciones a diferentes soluciones tecnológicas, normalmente sobre lenguajes de propósito general. Esto propuesta tiene otras ventajas, a continuación enumeramos algunas de ellas
    - Se vertebra el proceso desde un lenguaje cercano a los requisitos del área de negocio hasta la propuesta tecnológica,

  - Se pueden definir patrones arquitectónicos usados transversalmente en diferentes soluciones tecnológicos
  - El proceso de ingeniería software no esta conducido por documentación, sino esta conducido por modelos.

- Despliegue del DSL a la plataforma destino, con su implementación concreta tanto en plataforma de desarrollo, lenguaje de programación, como en implementación de software con hardware.
- Automatización de la documentación, validación y verificación de lo definido junto al modelo de negocio.

Utilizando MDE y MDSD podemos utilizar estas definiciones e implementarlas directamente en software mediante la transformación de modelos, evitando así la descoordinación entre requisitos, documentación e implementación dentro de un proyecto software.

En la sección anexa B.1 podemos ver un primer ejemplo de un código no final en un DSL simulado.

1.2.2. ¿Por que debe ser mejorado?

El desarrollo de este tipo de productos requiere una alta integración con la plataforma destino, lo que supone duplicar el desarrollo para cada una de las diferentes plataformas. Esto normalmente implica tener diferentes equipos y soluciones para cada una de ellas, con los costes que supone.

1.3. Objetivos

1.3.1. Introducción

El objetivo de este proyecto es la creación de un DSL de carácter declarativo utilizando la metodología MDE aplicado al desarrollo de aplicaciones orientadas al IOT.

Mediante este lenguaje propuesto en este trabajo, o mediante un o simular, un creador de productos IOT podría desarrollar un programa completo y funcional sin

conocer los aspectos internos de funcionamiento de la plataforma, siendo suficiente con que este describa las necesidades que quiere que realice su producto en un lenguaje alineado con este dominio.

Al utilizar DSL simplificamos la documentación, reducimos el código necesario reduciendo posibles errores, evitamos la creación de tests innecesarios ya que el propio lenguaje específico nos impide realizar tareas no permitidas. En resumen, el programa generado es válido y verificado en si mismo en el dominio.

Uno de los lenguajes específicos declarativos más conocidos es el lenguaje de cuarta generación Structured Query Language (SQL). Este lenguaje permite mediante una serie de declaraciones, especificar como insertar y recuperar información de una base de datos. El programador no tiene por que conocer como internamente el programa accede a memoria RAM, accede a páginas de memoria, accesos a disco, simplemente tiene que ocuparse de especificar que datos quiere leer o procesar.

El lenguaje modela un sistema empotrado mediante transacciones que reacciona a eventos externos. Los eventos son los definidos por (Liu and Layldan) Sprunt, Sha y Lehoczky [40] , donde tenemos eventos de tipo periódico, aperiódico y de tipo ráfaga. Las transacciones definen operaciones de computación o generación de nuevos eventos.

1.3.2. Objetivos generales

Mediante MDE generaremos una plataforma de desarrollo para dispositivos orientados al IOT, creando un Platform-Independent Model (PIM) textual con un DSL y generar varios Platform-Specific Model (PSM) para las plataformas IOT, servidor y simuladores de ambos.

1.3.3. Objetivos específicos

Estudiar el impacto de MDE en el desarrollo de un proyecto IOT para poder ver de primera mano sus ventajas y desventajas. En 2015 realice un proyecto completo de IOT similar sin utilizar la metodología MDE, de esta forma puedo comparar plazos de desarrollo y su utilidad en un posible proyecto no académico (profesional).

Mediante modelos podremos definir los requisitos del proyecto, simplificar y abstraer las tareas de:

- Gestión de interrupciones.
- Gestión de plataformas de despliegue.
- Gestión de tareas en microcontroladores, tareas cooperativas generadas en el hilo principal.
- Comunicación entre dispositivos y servidor.
- Monitorización y captura del estado de salud (housekeeping) de los dispositivos (uso de cpu, ram, particiones, red), y de a ser posible algunos elementos observables (voltajes, consumos).

De esta forma el creador de producto, puede expresar todas estas necesidades a muy alto nivel, para el desarrollo de prototipos.

Como objetivos específicos contamos con:

- Conocer y realizar un proyecto con las tecnologías: ECore Eclipse Modeling framework (ECore), Eclipse EMF, Query/View/Transformation (QVT), Xtext, Xtend
- Realizar todo el proceso utilizando la metodología MDE, creando modelos y realizando transformaciones entre modelos y de modelos a código.
- A través de un PIM escrito en un lenguaje descriptivo, generar varios PSM tanto para los dispositivos IOT como para servidor
- Creación de un lenguaje específico de dominio DSL, de tipo declarativo y basado en eventos. Estos eventos podrán ser generados por:
  - Entradas de propósito general General purpose input output (GPIO)
  - Temporizadores y calendarios, periódicos y aperiódicos.
  - Llamadas remotas de servidor al dispositivo y del dispositivo a servidor mediante un Remote Procedure Call (RPC).
- Creación de un protocolo de comunicación en red donde:
  - Notificar los eventos generados por la red de dispositivos.
  - Recibir eventos generados por el servidor.
  - housekeeping Notificar el estado de salud del dispositivo.
- Procesador y compilador del lenguaje a las siguientes plataformas IOT

  - ESP8266 versión Arduino. Estudio de la plataforma.
  - Simulación PC. Compilación a lenguaje Java donde poder realizar comprobaciones del programa.
  - Raspberry. Compilación a lenguaje Java. Reutilizando parte de la transformación de modelos a código de la versión Simulación PC.

- Generación de un esqueleto para servidor donde:
  - Poder comunicar con los dispositivos de la red de dispositivos.
  - Visualizar estado de salud de los dispositivos.
  - Visualizar eventos de los dispositivos.
  - Enviar eventos a los dispositivos.

Capítulo 2

Estado de la cuestión

Hoy en día contamos con microcontroladores y ordenadores embebidos de muy bajo coste. Soluciones como Raspberry PI [17], Beaglebone [28], o Arduino [29] son claros ejemplos de hardware de bajo precio y alta calidad, la cual nos permite realizar proyectos de integración hardware entre dispositivos de una forma sencilla.

Uno de los problemas que nos podemos encontrar con el desarrollo de cualquier solución tecnológica son los problemas derivados de trasladar una idea de negocio o proyecto a un producto finalizado, que funcione de forma adecuada, sea estable y que realice exactamente lo definido.

Un problema común en la mayoría de desarrollo software, debido a que no existe una comunicación entre el diseño de la aplicación y su implementación, desarrollándola utilizando diseños estáticos, donde una modificación no pasa a las siguientes fases de desarrollo.

Esto es debido a que el desarrollo de software no se realiza de la misma forma que se realiza otro producto, donde existe una fase de diseño y otra de desarrollo totalmente diferenciadas. El software es completamente manipulable, por lo que el propio desarrollo del software es en si mismo el diseño de este, pudiendo separar estos conceptos, más a bajo nivel, y definir una separación entre interfaces e implementaciones.

Como solución a todos estos problemas, surgen varias metodologías, entre ellas MDE, y dentro de esta la metodología MDSD.

Mediante esta metodología, podemos definir unos modelos que especifiquen de forma abstracta cual va a ser la funcionalidad a cubrir, y mediante transformaciones

Capa	Nombre	Ejemplo
M3	Meta metamodelado	MOF
M2	Metamodelo	UML / Ecore
M1	Modelo	Clases, estructuras de datos, funciones
M0	Instancia	Datos instanciados

Fuente: Elaboración propia

Tabla 2.1: Diferentes capas de la arquitectura MDA

de ellos, podremos especificar detalles cada vez de más bajo nivel o niveles de implementación como por ejemplo, el lenguaje a utilizar, el framework de este lenguaje, la plataforma de despliegue, o incluso los scripts de despliegue.

Una ventaja respecto a otros sistemas, es que no es un proceso en el que desarrollamos un único modelo y a partir de el realizamos una implementación, sino que la propia implementación la realizamos nosotros mismos en base a transformaciones, con lo que siempre vamos a tener concordancia con el modelo y la transformación, contando así con ventajas como:

- Una documentación fiable, exacta al modelo.
- Separación del dominio e implementación.
- Test unitarios y de integración derivados del modelo.

2.1. MDA - Arquitectura dirigida por modelos

La arquitectura dirigida por modelos Model Driven Architecture (MDA), es una iniciativa originariamente propuesta por el Object Management Group (OMG).

MDA [34] define un sistema de arquitectura software basado en cuatro capas (detalle en tabla 2.1) cada una de ellas en un nivel de abstracción diferente.

MDA utiliza en parte los siguientes estándares:

- Unified Modeling Language (UML) [25]
- Meta-Object Facility (MOF) [23]

- XML Metadata Interchange (XMI) [26]
- Enterprise Distributed Object Computing (EDOC) [24]
- Software Process Engineering Metamodel (SPEM) [22]
- Common Warehouse Metamodel (CWM) [20]

2.1.1. Modelos

Dentro de MDA, un modelo es un conjunto de elementos utilizados para representar una entidad en un nivel de abstracción determinado.

Un modelo puede estar representado de forma textual o de forma gráfica mediante una serie de diagramas interrelacionados.

2.1.2. Meta-Modelos

Un meta modelo [33], es un modelo del lenguaje, el cual define las relaciones, semántica, usos, restricciones de una serie de modelos.

2.1.3. Plataformas

MDA utiliza los conceptos Computation-Independent Model (CIM), PIM y PSM. [5]

CIM es una representación del sistema sin especificar como estará implementada en un sistema informático. Podríamos tomar como ejemplo de un CIM, el lenguaje de modelado Business Process Model and Notation (BPMN).

Mediante el PIM, normalmente definido en UML con anotaciones, representamos un modelo el cual es independiente de la plataforma despliegue. La idea principal con este concepto es la transformación de esta representación a una o varias representaciones PSM, específicas para una plataforma determinada.

Uno de los estándares que define MDA para realizar la transformación se denomina QVT.

Por último contamos con PSM, el cual es un modelo ejecutable en una plataforma especifica. En el caso de MDA, puede estar definido por anotaciones específicas de UML, tales como UML executable [21], o de otras plataformas de generación de código tales como ECore / xtext.

2.2. Lenguajes específicos de dominio

Los lenguajes específicos de dominio DSL nos permiten abstraernos de las peculiaridades con la implementación y centrarnos en definir nuestro producto en un lenguaje más cercano al dominio de negocio utilizado, por ejemplo los sistemas conectados a internet basados en arquitecturas Raspberry o Arduinio.

Para ello, nos es muy útil contar con un lenguaje de definición en el que podamos asignar eventos a actuaciones, ya sean locales o remotas, sin tener que entrar en como son capturados estos eventos, o como son realizadas dichas actuaciones.

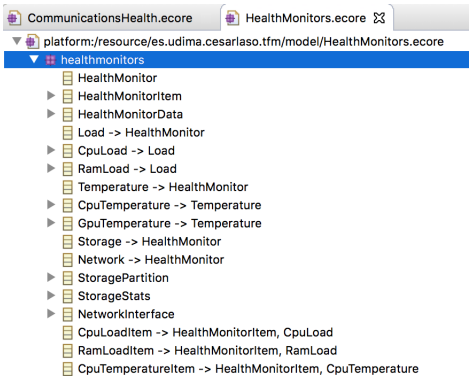
2.2.1. Eclipse EMF

Eclipse Model Framework (EMF) [9] es un framework de modelado y metamodelado que extiende de la funcionalidad proporcionada por el Integrated Development Environment (IDE) Eclipse. Este framework introduce el concepto de modelos ECore, modelos similares a clases de un lenguaje de programación común, con el añadido de la creación de editores y visualizadores de los datos generados a partir de dichos modelos.

Los modelos ECore, proporcionan funcionalidad básica en la generación de clases para el lenguaje Java, la edición de modelos mediante el propio IDE Eclipse, la creación de test unitarios, así como la personalización en la generación de todo este código.

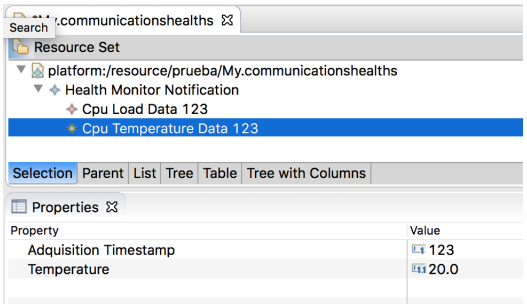
Mediante la utilización del propio editor integrado, aseguramos la correcta generación y construcción de los modelos. En la figura 2.1 podemos observar la definición de modelo y en la siguiente figura 2.2 mostramos la edición de una instancia de datos generados a partir de un modelo ECore.

ECore gestiona la persistencia de sus datos mediante archivos Extensible Markup Language (XML), con lo que podemos editarlos de forma textual si fuera necesario sin necesidad de contar con el propio IDE. La figura 2.3 muestra uno de los mode-



Fuente: Elaboración propia

Figura 2.1: Definición de un modelo ECore en formato tree. En este tipo de visualización podemos ver la herencia de los modelos, los interfaces que implementa, y en caso de desplegar los elementos del árbol, sus atributos.



Fuente: Elaboración propia

Figura 2.2: Edición de una instancia generada a partir de un modelo ECore en su formato de visualización tree. Esta visualización es personalizable en textos e iconos, permitiendo desarrollar editores muy alineados con la temática del negocio. El editor ejecuta las reglas de validación de los atributos, mostrando los tipos de datos permitidos para cada uno de ellos y mostrando error en caso de valores no permitidos tanto por el tipo de dato, como por la validación adicional que indiquemos.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI
   " xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="
   events" nsURI="http://tfm.cesarlaso.udima.es/events" nsPrefix="
   es.udima.cesarlaso.tfm.events">
  <eClassifiers xsi:type="ecore:EClass" name="Event" abstract="true
   ">
    <eStructuralFeatures xsi:type="ecore:EReference" name="
   actuators" lowerBound="1"
        upperBound="-1" eType="#//Actuator" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Actuator" abstract="
   true"/>
  <eClassifiers xsi:type="ecore:EClass" name="SequentialActuators"
   eSuperTypes="#//Actuator">
    <eStructuralFeatures xsi:type="ecore:EReference" name="
   actuators" lowerBound="1"
        upperBound="-1" eType="#//Actuator" containment="true"/>
  </eClassifiers>
</ecore:EPackage>
```

Fuente: Elaboración propia

Figura 2.3: Representación XML del modelo ECore Events

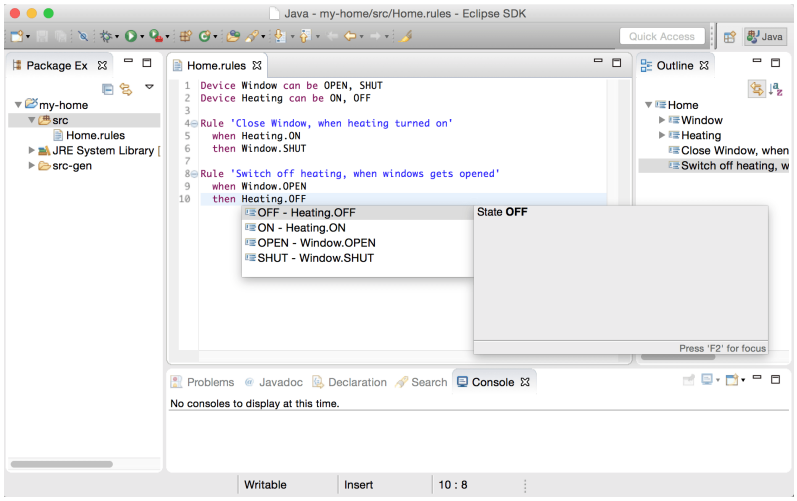
los en su formato XML, el cual posteriormente comentaremos en el desarrollo del proyecto.

2.2.2. Xtext

Para este trabajo, utilizaremos el kit de desarrollo Xtext framework for modeling languages (Xtext) [16].

Este producto, cuenta con una infraestructura completa de desarrollo de lenguajes tipados, incluyendo parser, linker, checkeador de tipos de datos, compilador y editor tanto de escritorio (basado en eclipse EMF), como online a través de un navegador web online. Ver figura 2.4.

El posterior desarrollo del DSL, lo realizaremos utilizando esta herramienta, ya que nos proporciona todo lo que vamos a necesitar para realizar un DSL que cumpla con nuestras necesidades.



Fuente: Web Xtext [16]

Figura 2.4: IDE Eclipse con Xtext editando DSL con soporte de tipado y autocompletado. El editor muestra el programa como si de un lenguaje de programación general se tratara, con todas las ventajas que aporta.

Esta herramienta es de gran calidad y es utilizada tanto en investigación como en desarrollo de proyectos industriales por empresas como la European Space Agency (ESA), BOSCH, Siemens o SAP [16]

2.2.3. MPS

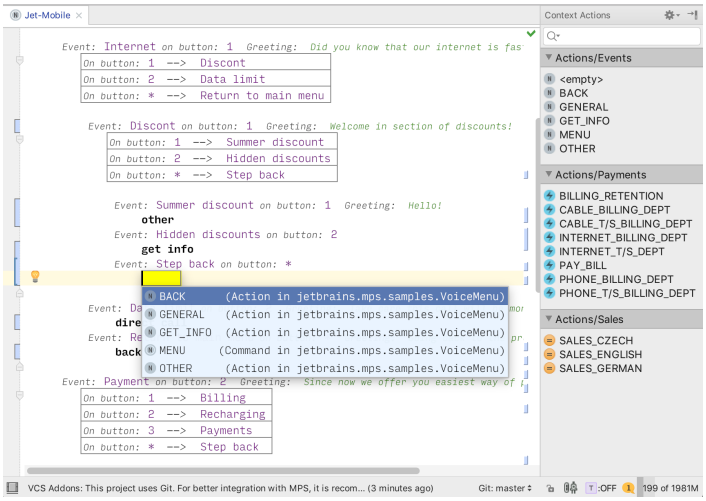
Otra herramienta de especial mención es MPS [43], de Jetbrains [42].

Esta herramienta, basada en el IDE IntelliJ [41], cuenta al igual que Xtext, con todas las herramientas necesarias (ver figura 2.5) para el desarrollo de lenguajes específicos de dominio, tales como compilador, parseador, linker, editor.

Los aspectos destacables de esta herramienta son una integración completa con el editor IntelliJ, soporte profesional por parte de la empresa Jetbrains e incluso formación en la herramienta y desarrollo del lenguaje.

El software cuenta con la licencia OpenSource Apache 2.0. [44].

Cuenta con soporte de generación de código basado en la Java Virtual Machine (JVM) y cuenta con soporte de terceros para C, .NET y javascript.



Fuente: Web MPS [43]

Figura 2.5: IDE MPS editando en un DSL con soporte de autocompletado. Soporta todas las opciones de refactorización que incluye el IDE en el que esta basado IntelliJ

Mención especial merece la aplicación mbeddr [14] basada en MPS, la cual incluye incluso herramientas de verificación y validación de software tal como podemos observar en la figura 2.6.

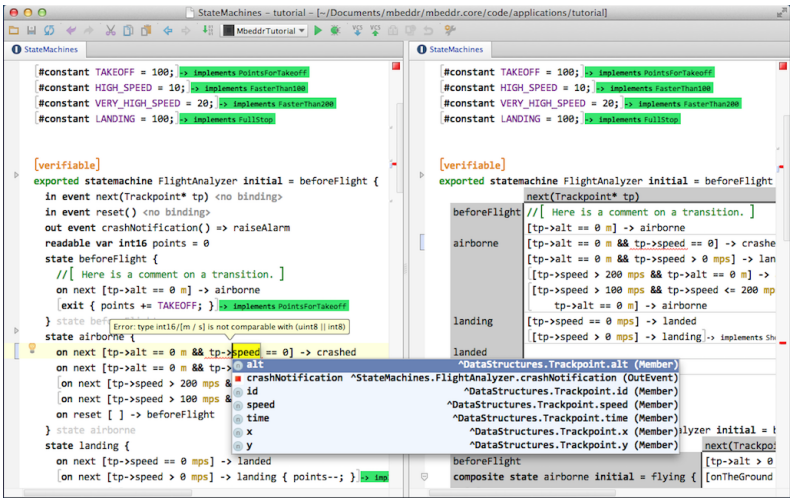
2.3. Trabajos relacionados

2.3.1. DSL for arduino

Koopman y Plasmeijer [30] crearon una plataforma similar a la de este trabajo, la cual define un DSL, el cual nombran como ARDSL que compila a código C++ para la plataforma Arduino UNO.

En este caso, el DSL está creado a partir del lenguaje de programación funcional Clean [38].

A continuación podemos ver en la figura 2.7 un ejemplo de código implementado en el DSL .



Fuente: Web mbeddr [14]

Figura 2.6: IDE MBEDDR editando en un DSL con soporte de verificación y validación.

```
helloworld :: ARDSL (a Bool Read) (a VOID Read) | ardsl a
helloworld =
    boolean ledOn.
    int lastTime.
    ardsl {
        setup =
        pinmode D13 OUTPUT :.
        digitalWrite D13 (lit False), loop =
        If (millis /. lit 500 >. lastTime) (
            ledOn =. . ledOn >== b.
            digitalWrite D13 b :.
            lastTime =. lastTime +. lit 1
        )
    }
```

Fuente: Elaboración propia

Figura 2.7: DSL for Arduino, ejemplo de código donde cada 500 ms se alterna el estado de un led. El lenguaje creado es muy cercano tanto al modelo de negocio como a un lenguaje de proposito general.

```
rule "Lights on when paused or stopped"
when
    Item PlexTVStatus changed to Paused or
    Item PlexTVStatus changed to Stopped
then
  logInfo("PlexLight", "Play to pause")
  dimLiving.apply(80)
end
```

Fuente: Openhab [6]

Figura 2.8: Fragmento de código en la definición de reglas Openhab. En el ejemplo se modifica un actuador, luces del salón, en función de un evento, estado de la reproducción del televisor

2.3.2. OpenHab

OpenHab [6] es un programa completo para el control de elementos conectados a internet, con soporte para múltiples plataformas y dispositivos IOT.

Este producto utiliza el mismo lenguaje de creación de DSL que este trabajo, el lenguaje Xtext para definir un DSL propio, en este caso utilizado para la gestión de máquinas de estado y reglas lógicas entre dispositivos.

Este producto, al ser un producto de consumidor final, tal como definen desde su propia web, cuenta con las siguientes ventajas:

- Diseñado para ser neutral respecto al fabricante, y frente al protocolo.
- Corre en cualquier dispositivo que sea capaz de ejecutar la JVM.
- Cuenta con un motor de reglas adaptado a la automatización.
- Cuenta con interfaces web y nativo iOS / Android.
- Código abierto, comunidad abierta.
- Proporciona APIs de integración.

Se muestra un ejemplo de configuración de este entorno en la figura 2.8.

Capítulo 3

Desarrollo

3.1. Requisitos del proyecto

Queremos crear un lenguaje específico para productos IOT, para ello tendremos que:

- Especificar los elementos del lenguaje mediante ECore.
- Especificar las transformaciones de estos elementos a lenguaje Java.
- Especificar la gramática del DSL utilizando los elementos del lenguaje anterior.
- Crear un proyecto que sea capaz de parsear este lenguaje y poder ejecutarlo.

Igualmente, para la parte hardware/software, necesitamos crear una plataforma con la que podamos abstraer el hardware/software. Este subproyecto cuenta con los siguientes requisitos:

- Especificar de una forma genérica eventos generados por dispositivos IOT, basados en GPIO.
- Crear una plataforma base de despliegue.
- Protocolo de comunicación entre dispositivos y servidor.
- Monitorización de salud de los dispositivos (uso de cpu, ram, particiones, red).
- Generación de código base para servidor.

3.2. Modelos

A continuación detallamos los modelos que han sido diseñados, mostrando el nombre del modelo, un diagrama de clases contenido para cada modelo, y una documentación generada a partir de las propias anotaciones del modelo.

El generador de aplicaciones IOT cuenta con los siguientes grupos de modelos:

- Modelos relativos al sistema de eventos. Todo el sistema depende de estos modelos, eventos y actuadores.
- Modelos relativos al protocolo de comunicación. Modela las diferentes partes dentro del modelo de comunicación. Eventos, seguridad, estado de salud.
- Modelos relativos al propio sistema IOT. Modela la composición básica del sistema, entradas/salidas, archivos, gráficos, sonidos.
- Modelos relativos al servidor, usuarios, seguridad.

Para todos los modelos analizados, mostraremos su diagrama de clases, generado a partir de la transformación creada con Eclipse Sirius (Sirius) [11] y en algunos de ellos, esquemas en formato ECore tree/doc... como anexo para un mayor entendimiento y fácil lectura de esta memoria.

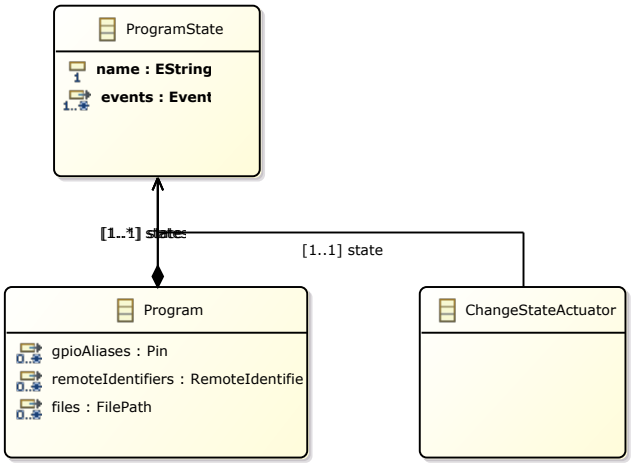
Debido a que el código fuente es muy extenso, no será incluido en la sección anexos y podrán ser descargado en formato zip en la siguiente URL: Laso [32]

3.2.1. Modelos relativos al dominio del dispositivo IOT

3.2.1.1. Modelo Programa

Este modelo conforma la base del programa que posteriormente definiremos en el DSL. Es la base del proyecto, a partir de este metamodelo, generaremos programas mediante un DSL creado con Xtext.

Este modelo representa un programa IOT en el que tenemos una serie de entradas, ya sean desde el propio GPIO, una serie de entradas remotas y acciones tanto en el propio GPIO como notificaciones remotas.



Fuente: Elaboración propia

Figura 3.1: Modelo Programa IOT. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

Como ejemplo, imaginemos un dispositivo IOT de lo más simple, con una sola entrada digital, utilizado para contabilizar monedas de una máquina de refrescos. Mediante el DSL que definimos en este proyecto podríamos generar un programa para una plataforma determinada que cumpla con las siguientes características: Gestión de la interrupción GPIO asociada a dicho evento de pulso de entrada de moneda, notificación a servidor de este evento, control de salud del propio dispositivo, etc.

El programa define una serie de GPIO pin, indicando estos con un alias, es decir, podemos asignar un nombre a una entrada, por ejemplo, Light Emitting Diode (LED) rojo →pin1. Cuenta con asignación de nombres de eventos remotos, y rutas de archivos a utilizar por el programa. Un programa cuenta con una serie de estados, necesario indicar el estado principal de arranque de este y posibilitando cambiar de estado. Un ejemplo de uso real es el testeo en el arranque de los diversos indicadores LED u otras entradas/salidas, y una vez terminado, pasar a un estado de error en caso de fallo, y en caso de funcionamiento normal, pasar al estado del programa principal.

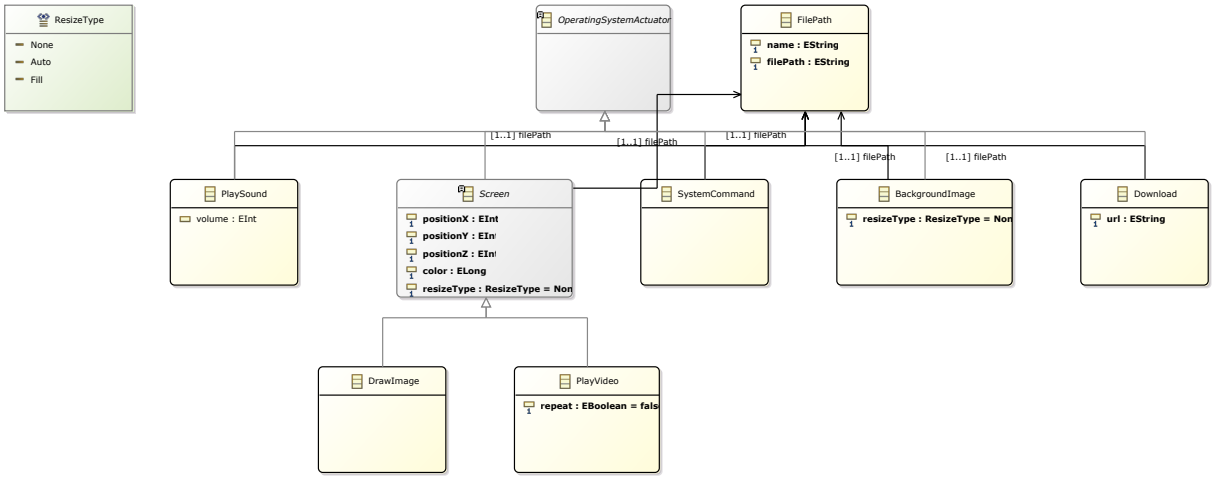
En la figura 3.16 podemos visualizar en formato diagrama de clases UML el modelo *Program*.

3.2.1.2. Modelo Operating System Actuators

Nuestra abstracción de un sistema IOT necesita contar con una serie de elementos que habitualmente encontramos proporcionados por los sistemas operativos.

Para ello necesitamos modelar estos elementos que utilizaremos desde nuestro DSL.

Este modelo, representado en la figura 3.2, representa las funcionalidades básicas que necesitamos por parte del sistema operativo para el funcionamiento de nuestros dispositivos IOT. Cuenta con una muestra de las tareas que podría implementar nuestro DSL, tareas de alto nivel tales como: descarga de archivos, ejecución de programas del sistema, visualización de imágenes en pantalla y reproducción de archivos de audio o vídeo.



Fuente: Elaboración propia

Figura 3.2: Modelo OperatingSystemActuators. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

3.2.1.3. Modelo GPIO

El modelado de las entradas y salidas en una placa hardware (o emulada) con soporte GPIO es una de las partes más importante del dispositivo IOT. Mediante este componente, podremos interactuar con otros dispositivos externos tales como sensores digitales, analógicos, actuadores, etc.

La representación de los modelos, se hará en tres grupos lógicos para su mejor modelado y separación de tareas: gpio, gpio events y gpio actuators.

En estos modelos definiremos de una forma abstracta estos conceptos para poder interactuar más adelante con ellos.

Los dispositivos GPIO cuentan con una serie de entradas analógicas o digitales. Muchos de ellos tienen entradas con múltiple funcionalidad pudiendo comportarse como entradas digitales o analógicas vía configuración en tiempo de ejecución. Así mismo, son capaces de funcionar en otras modalidades tales como Recommended Standard 232 (RS232) o Inter Integrated Circuit (I2C), compartiendo la misma entrega física entre las diversas opciones. Esto quiere decir, que podemos corromper hardware si por error cambiamos un valor de una variable en ejecución, algo que en un lenguaje general debería ser permitido, en nuestro DSL no deberíamos dejar, con lo que nuestro programa sería válido en compilación.

Esta funcionalidad tan específica, deberá quedar fuera del modelo, debiéndose soportar en el código no generado automáticamente a partir de los modelos.

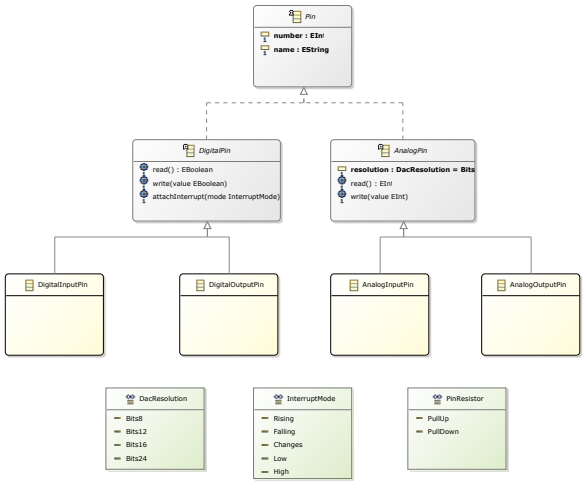
Debido a que posteriormente en la sección DSL crearemos un lenguaje de tipado fuerte, modelaremos las entradas / salidas especificando el tipo de estas (digital / analógico), evitando posibles fallos en hardware si por ejemplo activamos una salida en un dispositivo de entrada (pudiendo incluso quemar el dispositivo sino incluye algún tipo de resistencia previa).

Otra funcionalidad igualmente importante es la capacidad que tienen estos dispositivos de generar interrupciones vía hardware en el momento de cambio de estado de la señal digital, algo necesario para nuestra definición posterior de sistema de eventos, soportado en mayor medida por este tipo de dispositivos.

En el caso de los sistemas que no posean la característica de interrupción vía hardware, deberá ser emulada mediante software realizando polling sobre el estado de la entrada en cuestión. En el caso de dispositivos con sistema operativo de propósito general como Raspberry PI, se realizará en el hilo principal. En el caso de Arduino, la consulta se realiza dentro del Main loop.

Esta casuística de emulación de interrupción vía software no quedará definido en este meta-modelo sino que será definido en la implementación a código final de la plataforma en cuestión.

Nuestro modelo GPIO, cuenta con una serie de funcionalidades declarativas, permitiendo al desarrollador especificar una serie de tareas sin saber como en capas



Fuente: Elaboración propia

Figura 3.3: Modelo GPIO. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

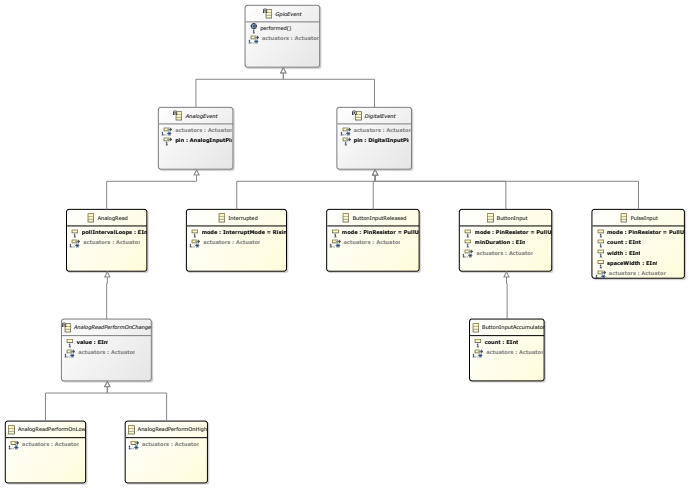
inferiores las realiza la plataforma. Podemos ver ejemplos como: ButtonInputAccumulator, evento por el cual cada vez que pulsamos un botón (cambiamos de estado durante un tiempo mínimo una entrada GPIO), lanzamos un evento con un contador de veces.

En la figura 3.3 podemos observar el diagrama de clases del modelo GPIO base, en la figura 3.4 mostramos los eventos generados por el dispositivo IOT. Por último, la figura 3.5 muestra los actuadores con los que contamos en el modelo.

3.2.2. Modelo estado de salud

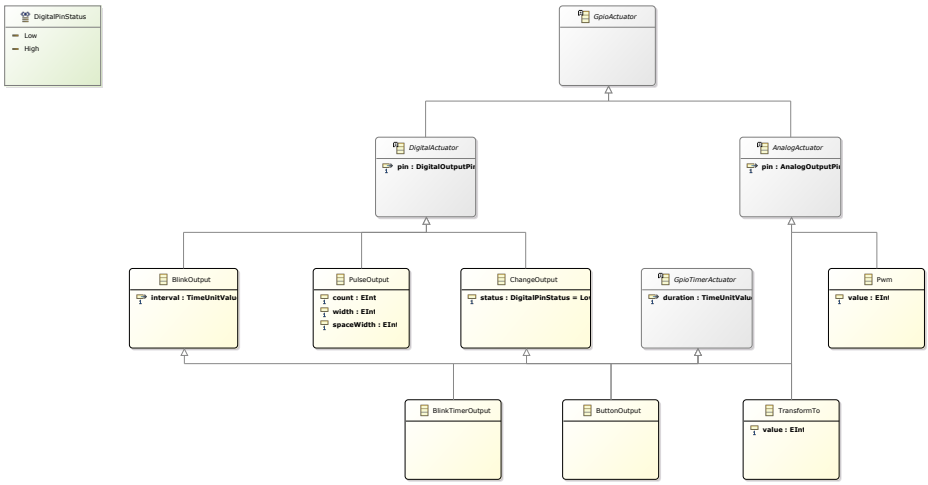
Representa el estado de salud de un dispositivo. La monitorización del estado de salud es una parte esencial del sistema ya que nos permitirá detectar comportamientos anómalos tales como una alta carga en CPU por un fallo seguramente en la programación, una carga anormal de memoria RAM, o detectar una temperatura de procesador demasiada alta para su correcto funcionamiento. En la siguiente figura 3.6 visualizamos el modelado en formato ECore tree. En la figura anexa A.2 podemos ver el modelo de clases.

Esta monitorización estará disponible en aquellos dispositivos que funcionen mediante un sistema operativo completo como Linux. Por este motivo, la comprobación



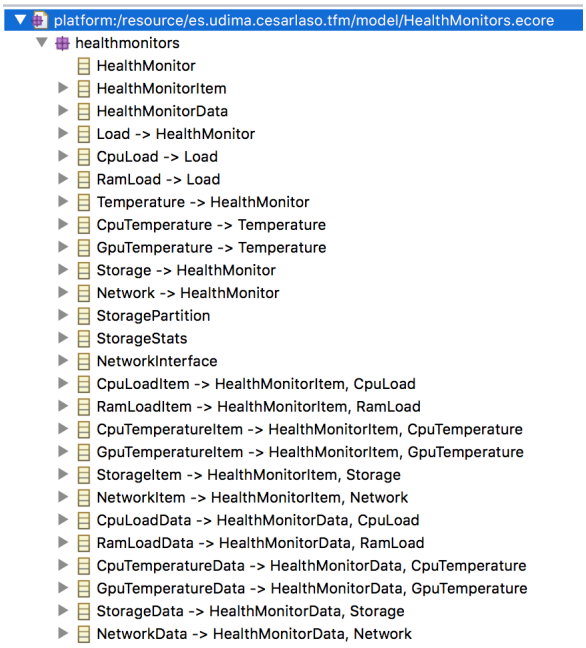
Fuente: Elaboración propia

Figura 3.4: Modelo GPIO - Eventos. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius



Fuente: Elaboración propia

Figura 3.5: Modelo GPIO - Actuadores. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius



Fuente: Elaboración propia

Figura 3.6: Modelo Health. Representación visual en formato Ecore tree. La representación es generada por el propio entorno EMF

del estado de salud no podrá ser aplicada a los dispositivos de tipo microcontrolador ya que al carecer de un sistema operativo completo, correr directamente en el hardware, o correr en un sistema operativo de tiempo real muy limitado no es posible en esta fase actual del desarrollo realizar esta implementación. En líneas futuras podríamos implementar dicha funcionalidad en el sistema operativo FreeRTOS [1], sistema operativo el cual corre el dispositivo ESP8266 [4].

Este modelo será utilizado por el modelo de comunicación en la sección 3.2.3.3.

3.2.3. Modelos relativos al protocolo de comunicación

3.2.3.1. Modelo: Servicios de comunicación base

El modelo servicios de comunicación, representa las bases del protocolo de comunicación entre dispositivos IOT y una serie de servidores centrales o distribuidos.

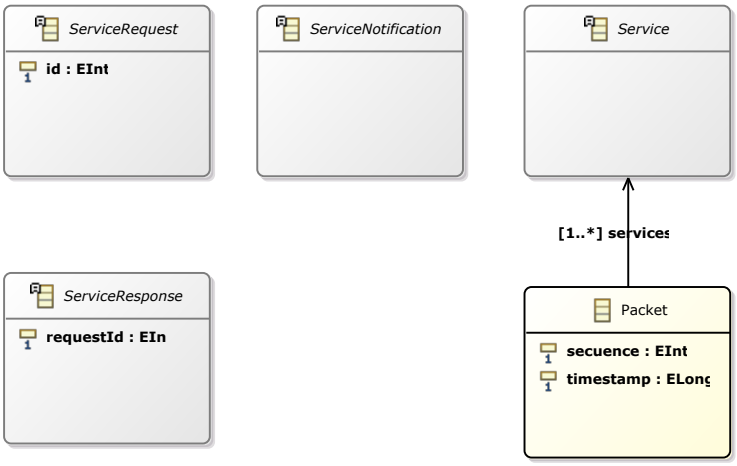
Como puede observarse en la figura 3.7, tenemos la clase principal *Packet*, que contiene los elementos base del protocolo de comunicación: número de secuencia, timestamp, servicios.

Un paquete contiene dentro de su payload uno o varios servicios, definidos posteriormente en los posteriores modelos.

Este diseño se ha realizado de esta forma ya que puede ser habitual que un dispositivo no tenga conexión con el servidor de forma continua, teniendo este que guardar toda la comunicación en su memoria a largo plazo para posteriormente cuando consigue conexión, reenviar toda esta información. Al poder utilizar un solo paquete, simplificamos la comunicación al poder reenviar toda esta comunicación en un solo paso. Igualmente reducimos la sobrecarga en el protocolo subyacente Transmission Control Protocol (TCP), acumulando varios servicios de información en un solo paquete.

También hemos definido los conceptos petición *ServiceRequest*, respuesta a una petición *ServiceResponse* y notificación sin respuesta previa *ServiceNotification* que se utilizarán en otros modelos de comunicación.

A continuación mostramos en la figura 3.7, como se visualiza este modelo en otro tipo de visualización en este caso una vista del diagrama de clases generado, con sus tipos de datos y junto a su documentación, todo ello partiendo siempre del mismo modelo ECore.



Fuente: Elaboración propia

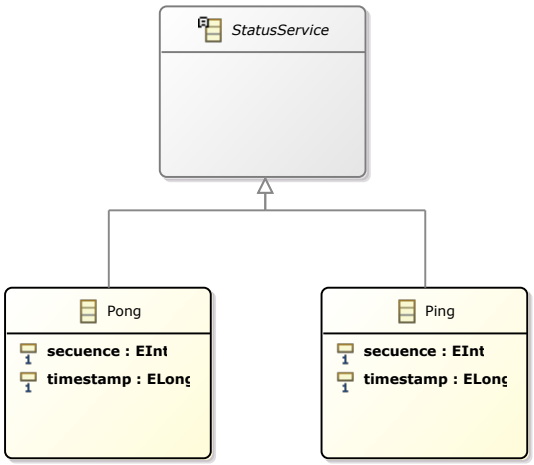
Figura 3.7: Modelo Servicios de comunicación. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

3.2.3.2. Modelo: Servicios de comunicación de estado

El sistema de comunicación necesita mantener la conexión TCP para poder tener control del dispositivo desde servidor. La mayoría de protocolos resuelven este problema enviando un paquete de información de estado cada cierto tiempo si el sistema detecta que no se está enviando información. El protocolo de nivel inferior utilizado, en el caso propio de este protocolo siendo TCP, incluye en su capa de control un modelo de establecer keepalive, este sistema cuenta con el problema de su poca configuración en determinados dispositivos y cuenta con un tiempo entre paquetes relativamente alto, del orden de dos horas por defecto en el kernel estandar de Linux. Algunas implementaciones, permiten configurar este parámetro no siendo una configuración sencilla de aplicar a nivel de aplicación. Por este motivo es necesario crear un servicio de control de estado.

Como puede observarse en la figura 3.8, contamos con la clase *StatusService* que a su vez implementa la clase del modelo anterior *Service*. A su vez, vemos las clases *Ping* y *Pong* que implementan la clase *StatusService*.

El funcionamiento de este modelo es relativamente sencillo. Si el sistema no detecta actividad en ninguna de las dos direcciones de comunicación, tanto de cliente a servidor como de servidor a cliente en un tiempo determinado, enviará un paquete *Ping*, respondiendo a su llegada el destinatario con un paquete *Pong*.



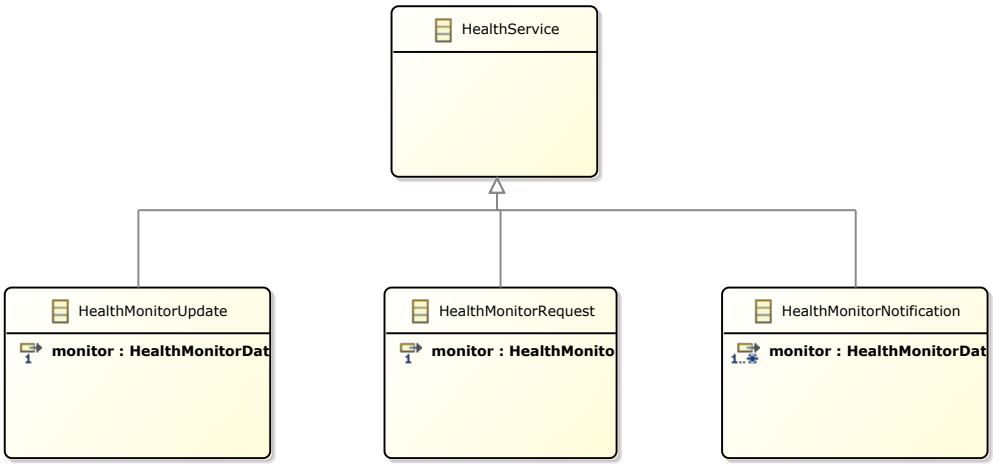
Fuente: Elaboración propia

Figura 3.8: Modelo Servicios de comunicación - Estado de conexión.. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

3.2.3.3. Modelo communication health services

Un sistema IOT necesita de un buen sistema de monitorización y comunicación de su estado actual de salud, comúnmente llamado housekeeping. Podemos evitar posibles problemas en el dispositivo si contamos con la información necesaria.

Para ello, utilizaremos una monitorización constante de variables del sistema tales como cargas de CPU, memoria, tráfico de red, estado de las particiones de disco. Modelamos el sistema utilizando el modelo *HealthMonitor* el cual define una entidad abstracta de monitorización de salud.



Fuente: Elaboración propia

Figura 3.9: Modelo Servicios comunicación - Housekeeping (Estado de salud).. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

Para la comunicación del estado de salud definimos el servicio *HealthService* y dos subservicios tal como puede verse en la figura 3.9: *MonitorItems* y *MonitorChangeItemInterval*, primer servicio utilizado para enviar el estado de todos los sensores activos desde el dispositivo IOT al servidor.

El segundo subservicio lo utilizaremos para modificar el intervalo de captura del estado de salud de un elemento en concreto, pudiendo especificar si se desea recuperar la información actual sin esperar al intervalo de tiempo del elemento a monitorizar.

Este modelo necesita de los datos proporcionados el modelo salud en la sección 3.2.2.

3.2.3.4. Modelo: Dispositivos

El modelo Dispositivos presenta las diferentes opciones que tenemos para el despliegue de la aplicación generada por el DSL en nuestro dispositivo final.

Como puede observarse en la figura 3.10, contamos con una serie de dispositivos base, los que tienen sistema operativo completo y la versión Arduino. Todos ellos, comparten el mismo tipo de configuración en este caso solamente tenemos como medio de conexión de red WIFI, compatible con todos los despliegues.

3.2.4. Modelos relativos al servidor y usuarios

El modelo servidor define un servidor base en el que poder recibir y gestionar los eventos generadores por nuestros dispositivos IOT. En el modelo, se crea un modelo básico *JavaServer*, el cual es la mínima implementación requerida. La figura 3.11 muestra un diagrama de clases.

El modelo usuarios, cuenta con las cualidades que suelen tener los objetos tipo usuario, tales como su nombre, a que compañía pertenece, que dispositivos tiene asignados, conocer si tiene acceso a estos en un determinado momento y en que lugar están situados. La figura 3.12 muestra un diagrama de clases.

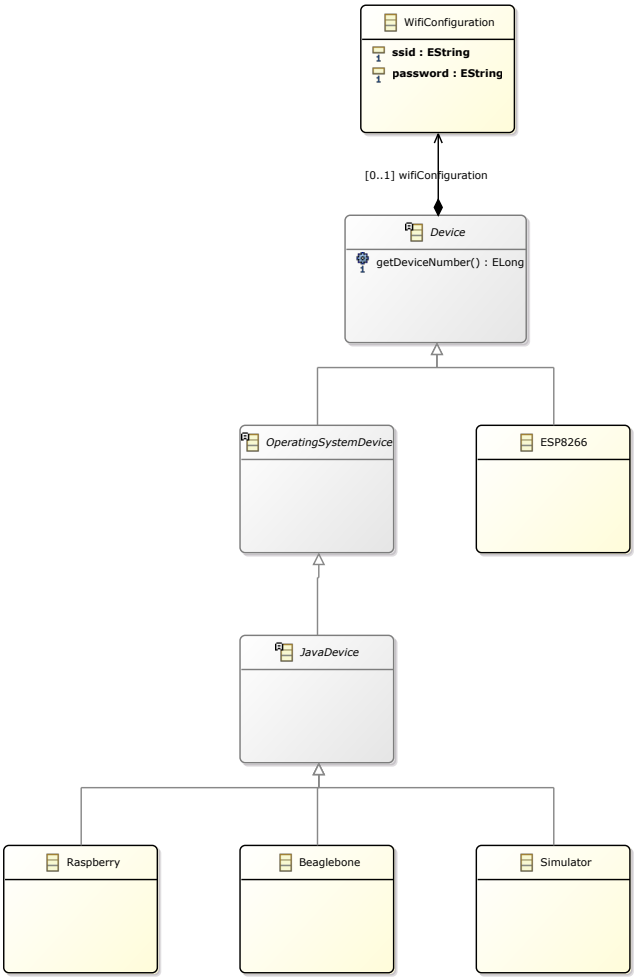
3.2.5. Modelos comunes al dominio del dispositivo IOT

Estos modelos aportan funcionalidades que son transversales a los modelos que componen este proyecto.

3.2.5.1. Modelo Eventos/Transacciones

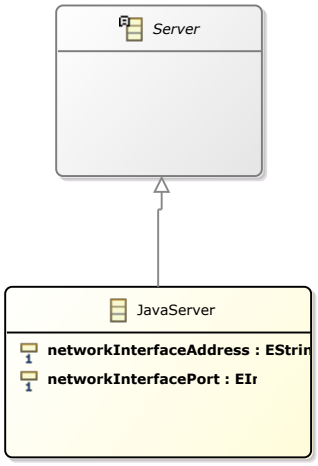
El modelo eventos representa una transacción base a todos los componentes que posteriormente modelaremos.

3.2.5.1.1. Modelo evento . El modelo *Evento* representa una entidad la cual es capaz de generar por sí misma acciones, permite definir los eventos del sistema. Estos pueden ser de diferente naturaleza: aperiódico y periódico.



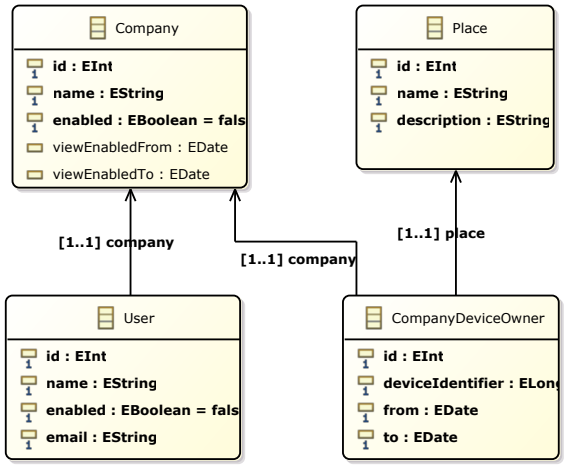
Fuente: Elaboración propia

Figura 3.10: Modelo Dispositivos. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius



Fuente: Elaboración propia

Figura 3.11: Modelo Servidor. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius



Fuente: Elaboración propia

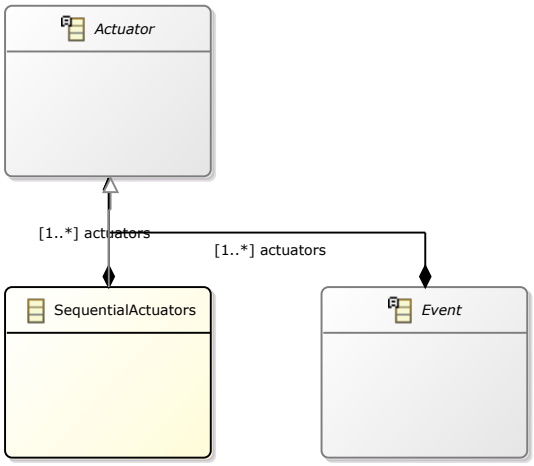
Figura 3.12: Modelo Usuarios. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

Como ejemplo de evento aperiódico contamos con un evento de entrada/salida, como puede ser un cambio de estado en una entrada digital. Como ejemplo de evento periódico, contamos con un temporizador el cual es ejecutado cada x unidades de tiempo.

Un evento siempre contendrá uno o varios actuadores. Un evento sin actuadores no tiene sentido con lo que no está permitido en el modelo.

3.2.5.1.2. Modelo actuador . El modelo actuador representa una acción a realizar por parte del sistema. Los actuadores dentro del evento se lanzarán de forma concurrente todos al mismo tiempo sin esperar a la finalización de cada uno de ellos. Si fuera necesario mantener un orden en los actuadores, es necesario utilizar el modelo contenedor *SequentialActuactor* el cual en su funcionamiento ejecuta el grupo de actuadores secuencialmente uno tras otro, esperando a la finalización de un actuador para invocar al siguiente.

A continuación en la figura 3.13 podemos visualizar esta representación mediante un diagrama de clases.



Fuente: Elaboración propia

Figura 3.13: Modelo Eventos / Transacciones. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

3.2.5.2. Modelo Timers

Este modelo representa la funcionalidad de un sistema de eventos por tiempo, ya sea mediante temporizadores periódicos a intervalos concretos o mediante periodos concretos [40].

Contamos con los siguientes eventos de tipo:

- *NowEvent*. Representa un evento para disparar en el mismo momento de su ejecución.
- *RepeatedEvent*. Representa un evento que disparará cada x unidades de tiempo.
- *ClockEvent*. Representa un evento que dispará cada x hora/minuto/segundo definidas.
- *CronEvent*. Representa un evento que dispará cada x unidades definidas en formato Unix CRON.

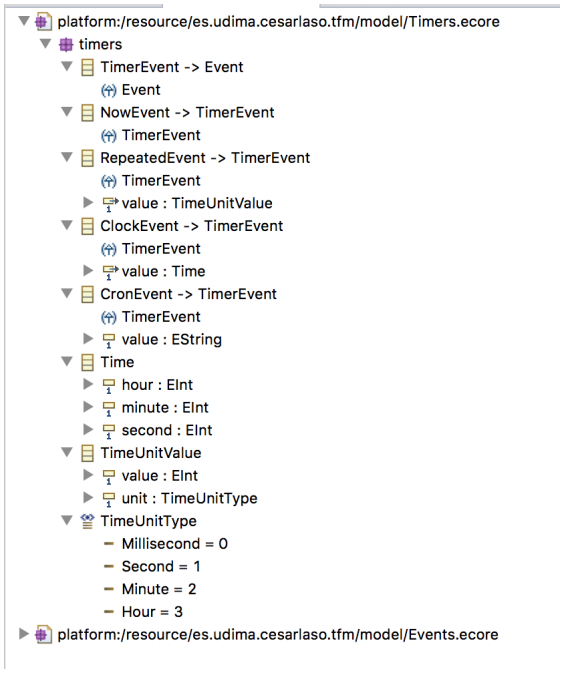
Estos eventos utilizan como tipo de dato los elementos definidos en *TimeUnit* y TimeUnitValue para así poder identificar sin ningún error la unidad de tiempo definida.

A continuación podemos visualizar la representación en formato ECore en la figura 3.14 y la representación como diagrama de clases en la figura anexa A.3 de este modelo de datos.

El modelo eventos representa la base de todos los componentes que posteriormente modelaremos. Podemos observar en la figura 3.14 la representación de este modelo en formato ECore tree, generado en el propio entorno EMF.

3.2.5.3. Modelo Proyecto IOT

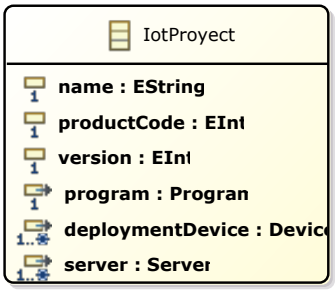
En el anexo A, podemos observar en la figura A.1 mostramos el árbol de dependencias entre los diferentes modelos tratados en este proyecto.



Fuente: Elaboración propia

Fuente: Elaboración propia

Figura 3.14: Modelo Timers. Representación visual en formato Ecore tree. La representación es generada por el propio entorno EMF



Fuente: Elaboración propia

Figura 3.15: Modelo Proyecto IOT. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

3.2.5.4. Modelo Programa

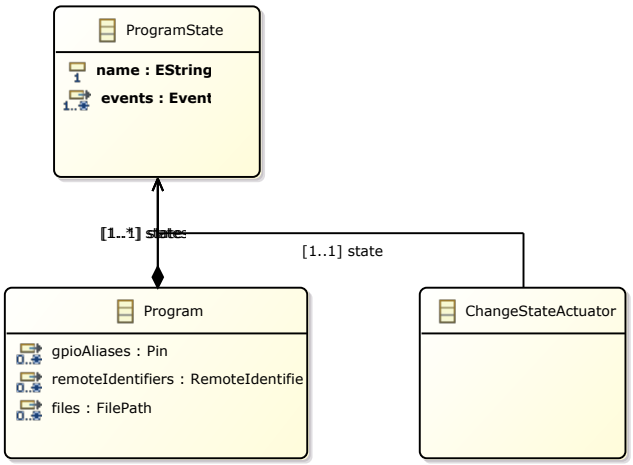
Este modelo conforma la base del programa que posteriormente definiremos en el DSL. Es la base del proyecto, a partir de este metamodelo, generaremos programas mediante un DSL creado con Xtext.

Este modelo representa un programa IOT en el que tenemos una serie de entradas, ya sean desde el propio GPIO, una serie de entradas remotas y acciones tanto en el propio GPIO como notificaciones remotas.

Como ejemplo, imaginemos un dispositivo IOT de lo más simple, con una sola entrada digital, utilizado para contabilizar monedas de una máquina de refrescos. Mediante el DSL que definimos en este proyecto podríamos generar un programa para una plataforma determinada que cumpla con las siguientes características: Gestión de la interrupción GPIO asociada a dicho evento de pulso de entrada de moneda, notificación a servidor de este evento, control de salud del propio dispositivo, etc.

El programa define una serie de GPIO pin, indicando estos con un alias, es decir, podemos asignar un nombre a una entrada, por ejemplo, LED rojo →pin1. Cuenta con asignación de nombres de eventos remotos, y rutas de archivos a utilizar por el programa. Un programa cuenta con una serie de estados, necesario indicar el estado principal de arranque de este y posibilitando cambiar de estado. Un ejemplo de uso real es el testeo en el arranque de los diversos indicadores LED u otras entradas/salidas, y una vez terminado, pasar a un estado de error en caso de fallo, y en caso de funcionamiento normal, pasar al estado del programa principal.

En la figura 3.16 podemos visualizar en formato diagrama de clases UML el modelo *Program*.



Fuente: Elaboración propia

Figura 3.16: Modelo Programa IOT. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

3.3. Transformación de modelo ECore a modelo genmodel. Generación de proyectos editor Java a partir de modelos ECore

Los modelos ECore proporcionan de forma abstracta una representación de metadatos. Para nuestro proyecto, necesitamos ser capaces de leer desde un programa todos estos datos desde un lenguaje de programación.

Esta transformación nos generará clases java a partir del formato ECore, con el ahorro de tiempo considerable que esto supone. Clases observables según el método ECore, y ampliables mediante interfaces y clases abstractas.

Esta transformación es personalizable en muchos aspectos, destacando a nivel de paquete y clase; ver figura 3.17 donde podemos especificar que tipo de generación, paquete destino, nombres... utilizaremos para la transformación, y también a nivel de atributo de cada una de las propiedades , ver figura 3.18 , donde podremos especificar con mucho detalle la generación de código, por ejemplo, indicando si el atributo utilizará las posibilidades de ECore de observar cambios en el modelo en runtime, especificar si la propiedad tendrá un icono en el editor ECore, como será representado, por resumir, todas las propiedades necesarias para las cuatro generaciones de código base que proporciona.

EMF proporciona una transformación de ECore a archivos genmodel, la cual generará cuatro aspectos resultado en formato Java:

- Código del modelo.
- Edición del modelo.
- Editor del modelo.
- Test unitarios.

3.3.1. Código del modelo

Primera transformación, base para todas las transformaciones posteriores. Nuestros metamodelos en formato ECore han de ser transformados al lenguaje de programación Java para que podamos interactuar con ellos.

Property	Value
▼All	
Base Package	es.udima.cesarlaso.tfm
Prefix	Gpiosactuators
▼Ecore	
▶Package	gpiosactuators
▼Edit	
Child Creation Extenders	false
Disposable Provider Factory	true
Extensible Provider Factory	false
▼Editor	
Generate Model Wizard	true
Multiple Editor Pages	true
▼Model	
Adapter Factory	true
Content Type Identifier	
Data Type Converters	false
File Extensions	gpiosactuators
Initialize by Loading	false
Literals Interface	true
Publication Location	
Resource Type	None

Fuente: Elaboración propia

Figura 3.17: Especificación de propiedades del paquete y clases en su conversión a Java

Problems | Javadoc | Declaration | Console | Properties | Error Log

Property	Value
▼Ecore	
▶Feature	duration : TimeUnitValue
▼Edit	
Children	true
Create Child	true
Notify	true
Property Category	
Property Description	
Property Filter Flags	
Property Multi-line	false
Property Sort Choices	false
Property Type	None

Fuente: Elaboración propia

Figura 3.18: Genmodel - Especificación de propiedades del atributo de clase en su conversión a Java

Esta transformación nos proporciona no solo código de las propiedades especificadas, sino que además nos proporciona código estático y tipado de los metadatos de estas propiedades, lo que nos evita tener que inspeccionar en runtime mediante reflection estos elementos.

Otra generación muy interesante es que las clases generadas son interfaces java, con lo que tendremos posteriormente todas las posibilidades para aumentar su funcionalidad.

Debido a esta peculiaridad, esta generación de código nos proporciona clases base ya implementadas con la funcionalidad, así como factorías para construir estas clases.

Una vez generado el código, tendremos tres paquetes resultado en nuestro proyecto:

- Modelo. Interfaces con getters/setter del propio modelo. Ver ejemplo generado en figura 3.19.
- Interface de la factoría. Utilizaremos para crear instancias. El nombre de la factoría, con la configuración por defecto será: ModeloFactory.eINSTANCE. Ver ejemplo generado en figura 3.20.
- Interface del paquete de clase. Con información relevante con los datos transformados, con nombres de constantes, de forma fuertemente tipada, así como los métodos para acceder a esta información. Ver ejemplo generado en figura 3.21.
- Implementaciones tanto de los modelos como de las factorías. Ver ejemplo generado en las figuras 3.22 y 3.23. Estas clases heredarán de sus interfaces base generadas y a su vez, de clases abstractas ECore, con toda la funcionalidad añadida que aporta el propio framework, y diposible de forma tipada, sin necesidad.

3.3.2. Edición del modelo

La generación de esta transformación [7] da como resultado un proyecto adicional con el mismo nombre que nuestro proyecto más la extensión .edit.
Este proyecto adicional, proporciona la base para el proyecto *Edit*, proporcionando toda la funcionalidad necesaria basada en el propio IDE eclipse como editor.

```
public interface Pin extends EObject {
  /**
   * Returns the value of the '<em><b>Number</b></em>' attribute.
   * <!-- begin-user-doc -->
   * <p>
   * If the meaning of the '<em>Number</em>' attribute isn't clear,
   * there really should be more of a description here...
   * </p>
   * <!-- end-user-doc -->
   * @return the value of the '<em>Number</em>' attribute.
   * @see #setNumber(int)
   * @see es.udima.cesarlaso.tfm.gpios.GpiosPackage#getPin_Number()
   * @model required="true"
   * @generated
   */
  int getNumber();
```

ejemplos/genmodel_gen_interface.java

Fuente: Elaboración propia

Figura 3.19: Fragmento de código con la interfaz en lenguaje Java generada a partir de modelo ECore Pin

```
public interface GpiosFactory extends EFactory {
  /**
   * The singleton instance of the factory.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  GpiosFactory eINSTANCE = es.udima.cesarlaso.tfm.gpios.impl.
   GpiosFactoryImpl.init();

  /**
   * Returns a new object of class '<em>Digital Input Pin</em>'.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @return a new object of class '<em>Digital Input Pin</em>'.
   * @generated
   */
  DigitalInputPin createDigitalInputPin();
```

ejemplos/genmodel_gen_factoria.java

Fuente: Elaboración propia

Figura 3.20: Interfaz de la factoría Java generada a partir de modelo ECore. Con esta factoria instanciaremos objetos

```
public interface GpiosPackage extends EPackage {
  /**
   * The package name.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  String eNAME = "gpios";

  /**
   * The package namespace URI.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  String eNS_URI = "http://tfm.cesarlaso.udima.es/gpios";

  /**
   * The package namespace name.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  String eNS_PREFIX = "es.udima.cesarlaso.tfm.gpios";

  /**
   * The singleton instance of the package.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  GpiosPackage eINSTANCE = es.udima.cesarlaso.tfm.gpios.impl.
   GpiosPackageImpl.init();
```

ejemplos/genmodel_gen_paquete.java

Fuente: Elaboración propia

Figura 3.21: Propiedades del paquete con metadatos generados a partir de ECore

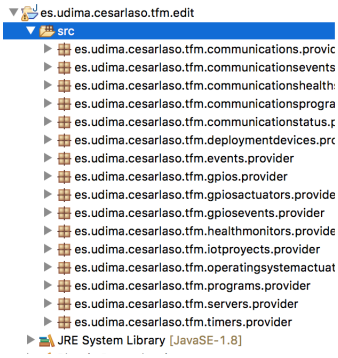
```
public abstract class DigitalPinImpl extends MinimalEObjectImpl.
    Container implements DigitalPin {
  /**
   * The default value of the '{@link #getNumber() <em>Number</em
   >}' attribute.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @see #getNumber()
   * @generated
   * @ordered
   */
  protected static final int NUMBER_EDEFAULT = 0;

  /**
   * The cached value of the '{@link #getNumber() <em>Number</em>}'
    attribute.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @see #getNumber()
   * @generated
   * @ordered
   */
  protected int number = NUMBER_EDEFAULT;

  /**
   * The default value of the '{@link #getName() <em>Name</em>}'
   attribute.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @see #getName()
   * @generated
   * @ordered
   */
  protected static final String NAME_EDEFAULT = null;

  /**
   * The cached value of the '{@link #getName() <em>Name</em>}'
   attribute.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @see #getName()
   * @generated
   * @ordered
   */
  protected String name = NAME_EDEFAULT;

  /**
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  protected DigitalPinImpl() {
    super();
  }

  /**
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
```

```
public class GpiosFactoryImpl extends EFactoryImpl implements
   GpiosFactory {
  /**
   * Creates the default factory implementation.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  public static GpiosFactory init() {
    try {
      GpiosFactory theGpiosFactory = (GpiosFactory)EPackage.
   Registry.INSTANCE.getEFactory(GpiosPackage.eNS_URI);
      if (theGpiosFactory != null) {
        return theGpiosFactory;
      }
    }
    catch (Exception exception) {
      EcorePlugin.INSTANCE.log(exception);
    }
    return new GpiosFactoryImpl();
  }
```

ejemplos/genmodel_impl_factoria.java

Fuente: Elaboración propia

Figura 3.23: Resultado de la implementación del interfaz de una factoría ECore



Fuente: Elaboración propia

Figura 3.24: Proyecto genmodel Edit, visualización de paquetes

En su parte de edición, proporciona soporte para la modificación de las propiedades de los modelos y la visualización estos mediante modelos EMF mediante el visor foundation [12].

Incluye un framework de comandos para la construcción de editores, dando soporte automático a operaciones complejas como deshacer y rehacer.

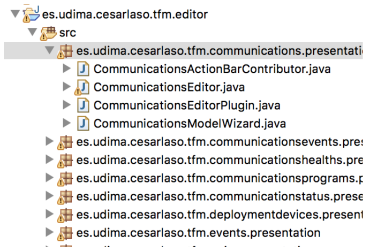
Incluye un generador de código capaz de generar todo lo necesario para construir un plugin editor, necesario para la siguiente transformación necesaria *Editor del modelo*.

Todo este código puede ser personalizado, tanto en la primera transformación de la generación de este código, como en la posterior transformación, de este código que a su vez generará código para el editor de modelo. La ventaja y a su vez desventaja del código generado en compilación, es su validación y verificación. El código generado es fuertemente tipado, con lo que no podemos esperar errores en ejecución sobre objetos con tipos incorrectos o no construidos adecuadamente, ya que podemos añadir reglas Constraint Language Specification (OCL) [19] en la validación de los modelos.

Se puede ver un ejemplo del proyecto generado en la figura 3.24 .

3.3.3. Editor del modelo

Este proyecto es generado a partir de la información obtenida de las dos primeras transformaciones, primero transformando los modelos ECore a código java mediante genmodel y posteriormente con las herramientas internas proporcionadas por Edit,



Fuente: Elaboración propia

Figura 3.25: Proyecto genmodel Editor, visualización de paquetes

a su vez obtenidas estas de los modelos ECore transformados.

Este proyecto proporciona la capa de presentación para la edición de modelos de forma gráfica en el IDE eclipse.

Para ello, nos proporciona las siguientes herramientas:

- Wizard para la creación de proyectos basados en nuestros objetos ECore.
- Editor gráfico de estos objetos generados.

La estructura de código generada, como podemos ver en la figura 3.25, al igual que el anterior proyecto *Edit*, sigue el mismo esquema, en este caso, nombre del proyecto más *Editor*

El código generado, igualmente modificable, respeta entre código generado y código propio; tal como vemos en la figura 3.26, podemos observar el añadido de etiquetas *generated* y como comentario html, con la etiqueta begin-user-doc , la documentación si existiera que escribimos en el modelo ECore.

3.3.4. Test unitarios

Este proyecto generado, ya el último de los generados por genmodel, es utilizado para realizar tests unitarios de nuestro código java generado.

La estructura de directorios, es al igual que los anteriores: nombre más tipo proyecto. En este caso, Tests, tal como vemos en la figura 3.27.

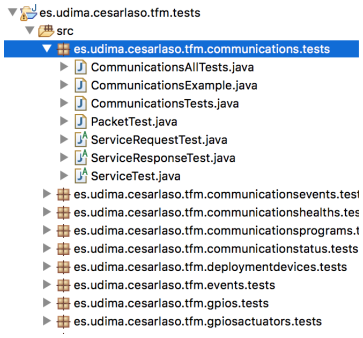
El código generado es un esqueleto de tests unitarios que utilizan la librería

```
public class CommunicationsEditor
  extends MultiPageEditorPart
  implements IEditingDomainProvider, ISelectionProvider,
   IMenuListener, IViewerProvider, IGotoMarker {
  /**
   * This keeps track of the editing domain that is used to track
   all changes to the model.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  protected AdapterFactoryEditingDomain editingDomain;

  /**
   * This is the one adapter factory used for providing views of
   the model.
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  protected ComposedAdapterFactory adapterFactory;
```

ejemplos/genmodel_editor_codigo.java

Fuente: Elaboración propia

Figura 3.26: Proyecto genmodel Editor, código generado



Fuente: Elaboración propia

Figura 3.27: Proyecto genmodel Tests, visualización de paquetes

```
public class CommunicationsTests extends TestSuite {

  /**
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  public static void main(String[] args) {
    TestRunner.run(suite());
  }

  /**
   * <!-- begin-user-doc -->
   * <!-- end-user-doc -->
   * @generated
   */
  public static Test suite() {
    TestSuite suite = new CommunicationsTests("communications Tests
   ");
    return suite;
  }
```

ejemplos/genmodel_tests_codigo.java

Fuente: Elaboración propia

Figura 3.28: Proyecto genmodel Tests, código generado

standard [18], los cuales implementan una base con la que posteriormente trabajar, incluendo las dependencias necesarias para trabajar con junit y ECore. Podemos ver un ejemplo de código generado en la figura 3.28.

3.3.5. Pasos a realizar en la transformación

Para cada meta modelo ECore es necesaria la generación de su archivo generador de código java mediante su correspondiente archivo descriptor con extensión genmodel.

Los metamodelos EMF están formados por dos componentes, los propios metamodelos ECore y los generadores. Este archivo descriptor nos ayuda a especificar como se ha de generar el código java respecto al modelo, pudiendo cambiar aspectos propios del lenguaje final de transformación (en el caso por defecto java).

A continuación, nos centramos en la plataforma de generación Java, y mostra-

mos como realizar la generación de un metamodelo, en este caso hemos elegido el metamodelo GpioActuators. como ejemplo.

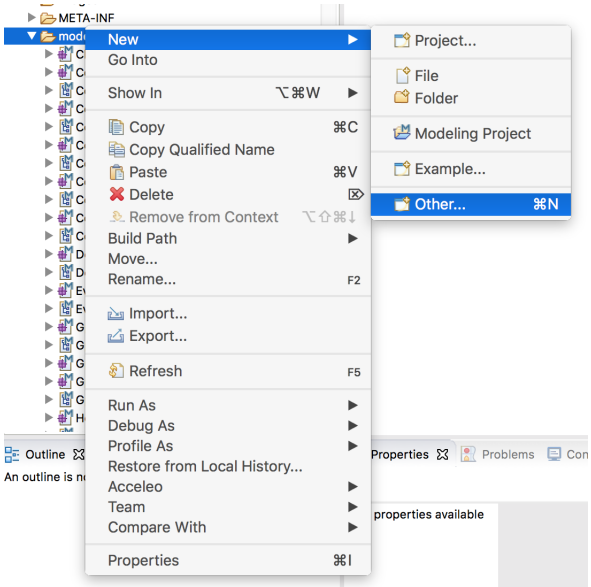
Los pasos a realizar para cada uno de los modelos ECore son los siguientes:

1. Crear un nuevo archivo de tipo EMF generator model, según podemos ver en las figuras 3.29 y 3.30.
2. Escribir como nombre de archivo el mismo nombre que el archivo ECore sin la extensión (figura 3.31). Guardarlo en la carpeta destino model.
3. Seleccionar modelo origen. El generador permite seleccionar varios tipos de archivo fuente, tales como UML, RationalRose, o en este caso ECore (figura 3.32). Seleccionamos ECore y seleccionamos desde el workspace el archivo GpioActuators.ecore (figura 3.33).
4. Seleccionamos el modelo principal GpioActuators y dejamos los modelos a los que se refiere como referencias ya que en los modelos referidos también vamos a generar código (figura 3.34). De esta forma podemos tener la generación de modelos separada.

Una vez terminada la creación de los generadores de los modelos podemos modificar las propiedades sobre que se generará con el modelo, en nuestro caso solo modificaremos el nombre del paquete base ya que por defecto queda vacío, usaremos el mismo nombre de paquete que para el proyecto: *es.udima.tfm.cesarlaso*.

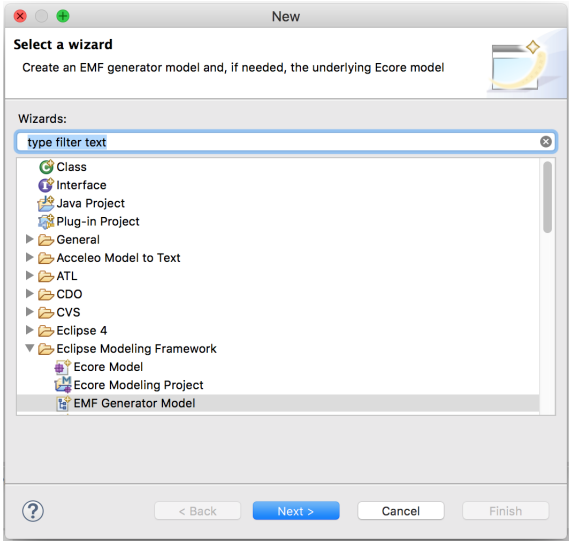
Cada vez que modifiquemos el modelo ECore, deberemos generar el código correspondiente mediante el genmodel. Es importante indicar que este proceso no se realiza automáticamente. Para ello es necesario mediante botón derecho o comando en menú, utilizar la opción de generar código: *Generate All*. Existe la posibilidad de especificar que proyectos necesitamos generar (modelo, edit, editor o tests), como podemos observar en la figura 3.35.

Una característica no resuelta en el generador de código de proyecto es que no es capaz de resolver inconsistencias de forma automática. Dependiendo del tipo de modificación que estemos realizando, cambio de tipo atributos, eliminar propiedades, será necesario eliminar todos los archivos generados por los generadores dentro de la carpeta *src* del propio proyecto contenedor de los modelos. También deberemos eliminar los proyectos adicionales generados de *edit, editor y test*. Dependiendo del tipo de modificación, como por ejemplo cambio del nombre del paquete, será necesario cambiar los archivos *build.properties, plugin.xml y plugin.properties* , eliminando



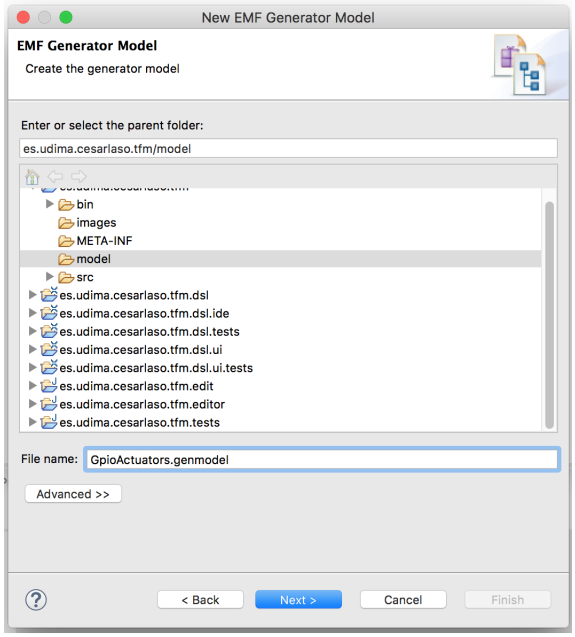
Fuente: Elaboración propia

Figura 3.29: Creación de un archivo genmodel. Si seleccionamos un archivo modelo, automáticamente en el *wizard* tendremos la selección premarcada en el próximo paso.



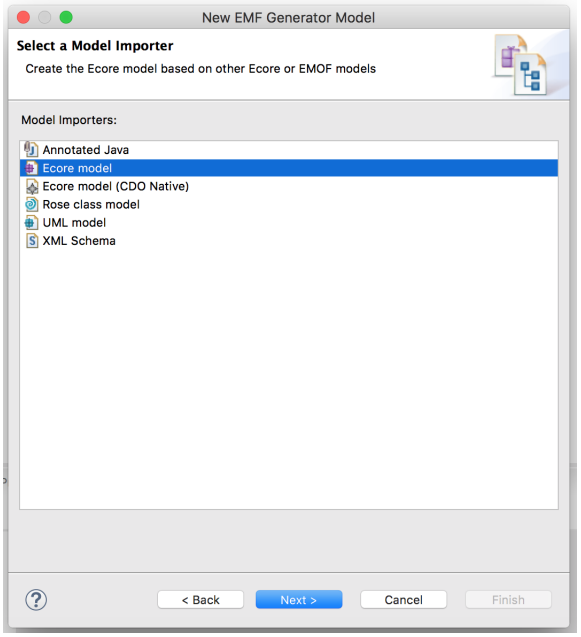
Fuente: Elaboración propia

Figura 3.30: EMF generator model permite la generación de proyectos de edición de código para java a partir de los modelos ECore



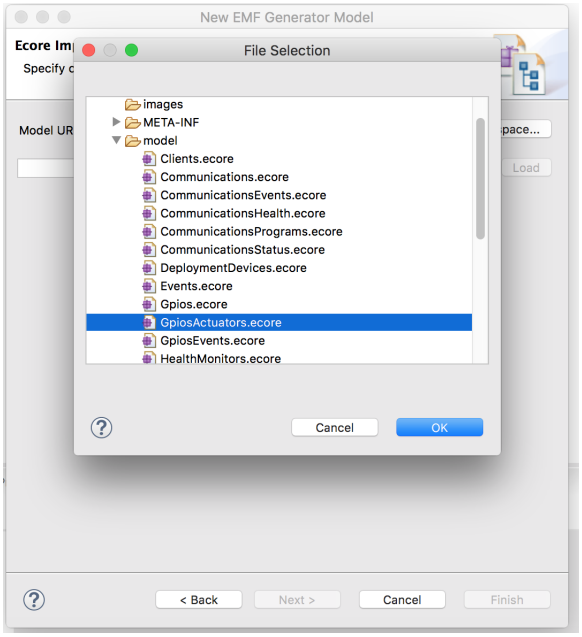
Fuente: Elaboración propia

Figura 3.31: Al generar el archivo genmodel se recomienda guardarlo en la misma ubicación que el archivo genmodel y con el mismo nombre que este.



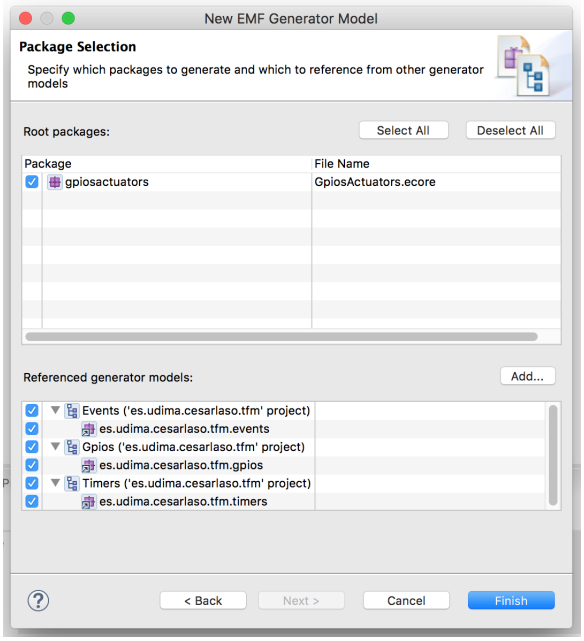
Fuente: Elaboración propia

Figura 3.32: EMF genmodel permite la transformación de varios tipos de modelos, no solamente ECore



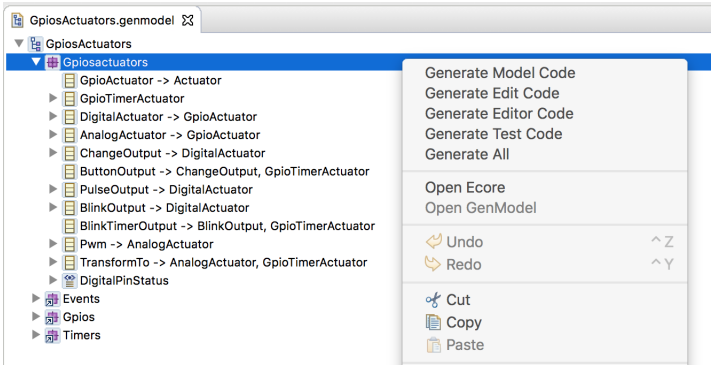
Fuente: Elaboración propia

Figura 3.33: EMF genmodel necesita conocer la ruta de los archivos modelo. En nuestro caso es recomendable utilizar la ruta relativa desde el workspace para mantener coherencia en el proyecto



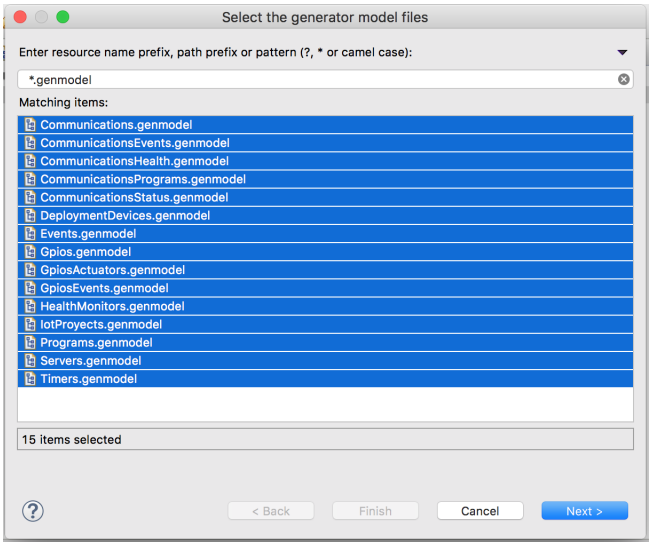
Fuente: Elaboración propia

Figura 3.34: EMF genmodel necesita conocer cual es el modelo raíz y cuales sus dependencias. En caso de que el modelo dependa de otros archivos modelos, será necesario importarlos como referencia para que no genere el código por duplicado



Fuente: Elaboración propia

Figura 3.35: Generación de código de proyecto desde genmodel. La generación permite seleccionar que proyecto que necesitamos generar. Incluye la posibilidad de generar todos a la vez.



Fuente: Elaboración propia

Figura 3.36: EMF generatormodel permite la generación desde varios modelos de forma simultanea, pulsando *ALT+SHIFT+G*. Permite seleccionar los modelos necesarios

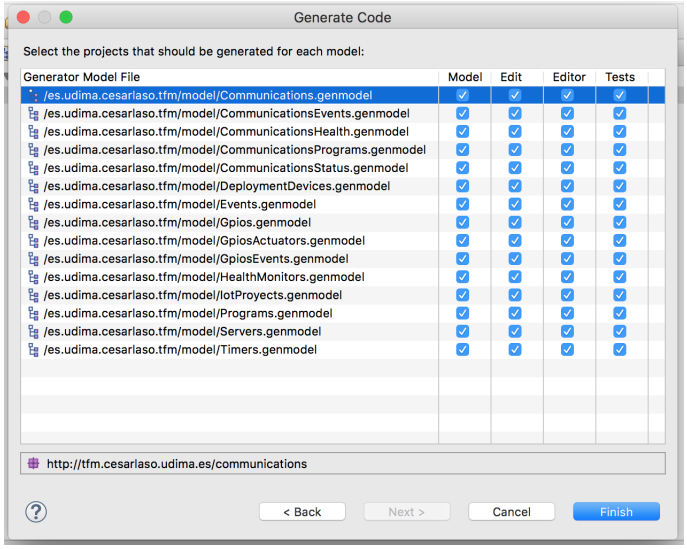
todas las referencia a modelos no existentes, hayan cambiado de nombre de paquete o de clase.

Una vez eliminados y realizadas las modificaciones correspondientes, procederemos a la generación de los archivos de proyecto java a partir de los modelos ECore. Este proceso lo podemos realizar de forma granular para un modelo en concreto, seleccionando el archivo genmodel correspondiente y generando el código necesario tal como vimos en la figura 3.33.

A su vez contamos con un método de generación de todo el proyecto de una sola vez, lo que nos ahorra tiempo, mediante la combinación de teclado *ALT+SHIFT+G* podemos generar todo el código asociado a los modelos ECore de una sola vez, tal como vemos en la figura 3.36. En el siguiente paso necesitaremos especificar qué tipos de proyectos generados necesitaremos (ver figura 3.37). Por defecto se seleccionarán todos.

3.3.6. Generadores Ecore a genmodel para otros lenguajes

Existen generadores a otros lenguajes como C++ mediante el proyecto [36] de la universidad de Murcia, podemos descargar el [35] código desde su propia web. Esta



Fuente: Elaboración propia

Figura 3.37: EMF generatormodel en su vista de generación múltiple permite en una sola vista seleccionar el tipo de proyectos a generar para cada modelo ECore. Permite seleccionar model,edit,editor,test

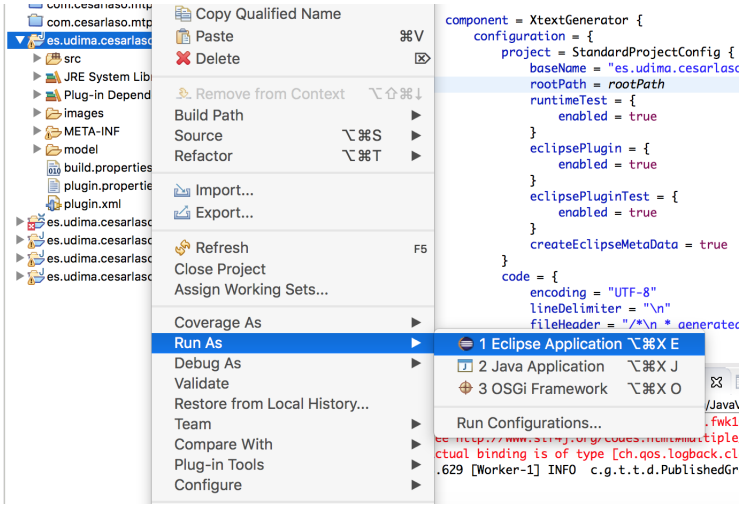
implementación actualmente incluye el parser de modelos desde C++ y generación de código a C++, el resto de funcionalidades tales como reflexión del propio metamodelo aún no están listas. Según el changelog, la última actualización es de 2010 por lo que podemos suponer que una vez terminada la cátedra se ha abandonado el proyecto.

3.4. Editor de modelos gráfico generado a partir de los proyectos edit,editor

Eclipse EMF permite generar objetos a partir de los modelos creados en ECore de forma gráfica.

Esto puede ser útil, ya que no tiene porque ser necesaria la creación de un DSL como veremos en la próxima sección.

Con el propio editor creado, alguien con los conocimientos suficientes podría generar un modelo ejecutable por un entorno EMF, siempre que contenemos con el *runtime* necesario para leer este modelo.



Fuente: Elaboración propia

Figura 3.38: Ejecutar EMF en modo runtime desde proyecto existente - Paso 1

El editor por defecto utilizará los iconos standard para los elementos de nuestro modelo. Esto si es modificado nos permite que el usuario de forma rápida sea capaz de situarse en el elemento necesario. Por ejemplo, podríamos asociar el icono de un diodo LED a los elementos salida GPIO, o un icono de un botón, a un elemento evento pulsador.

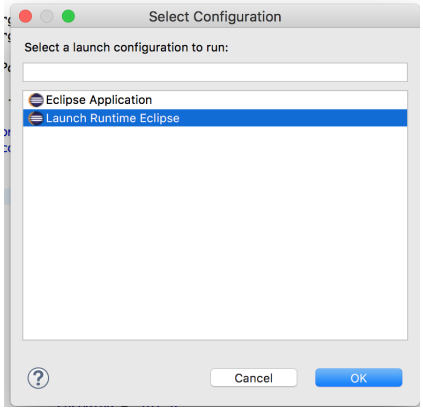
3.4.1. Ejecución de EMF en modo runtime

Para realizar esta tarea, a partir de nuestro proyecto ECore principal y una vez generados los proyectos necesarios edit y editor, ejecutaremos EMF en modo runtime, tal como vemos en la figura 3.38. Una vez seleccionado, seleccionaremos *Run as eclipse runtime* (Ver figura 3.39).

Una vez tenemos lanzado EMF en modo runtime, crearemos un nuevo proyecto vacío, no importa el tipo de proyecto a utilizar, podemos por ejemplo utilizar un proyecto base de java.

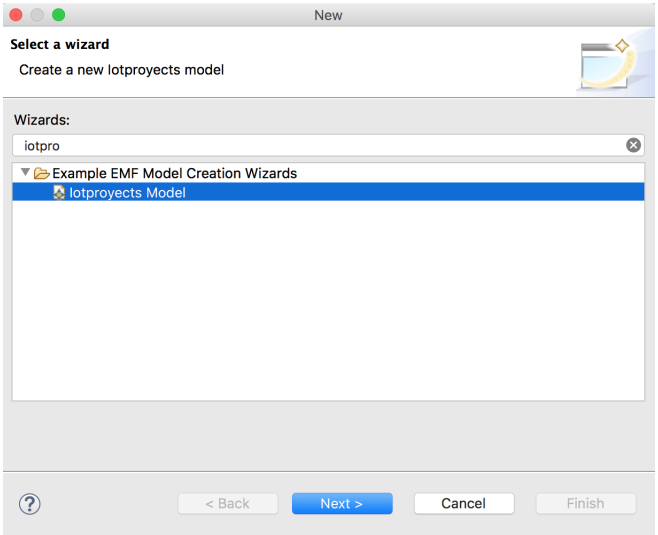
Una vez creado, vamos a utilizar los wizard generados automáticamente por los generadores de código anteriormente nombrados, para ello crearemos un nuevo archivo de tipo *EMF model wizard* seleccionando nuestro tipo de modelo *IotProyect*, tal como vemos en la figura 3.40.

Este editor nos permite crear objetos del modelo en forma de árbol, permitiéndo-



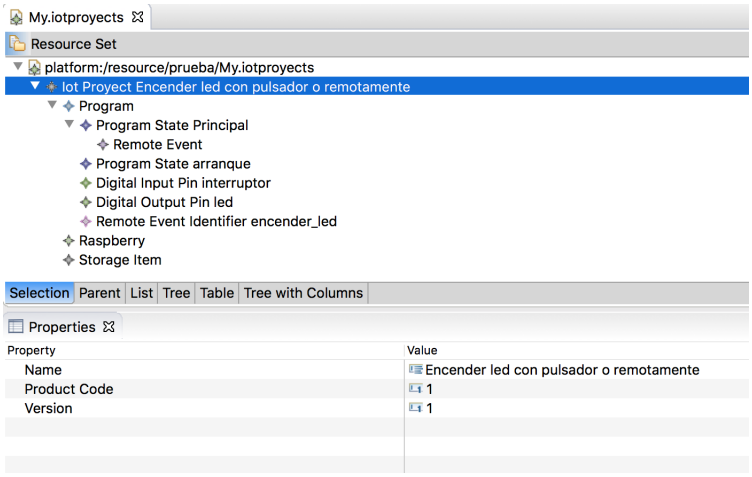
Fuente: Elaboración propia

Figura 3.39: Ejecutar EMF en modo runtime desde proyecto existente - Paso 2



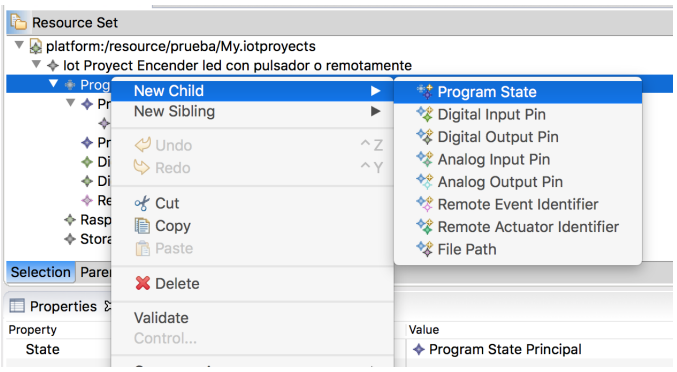
Fuente: Elaboración propia

Figura 3.40: EMF Wizard - Creación de un objeto a partir de un modelo de forma gráfica



Fuente: Elaboración propia

Figura 3.41: Editor gráfico del modelo a partir de metamodelo ecore



Fuente: Elaboración propia

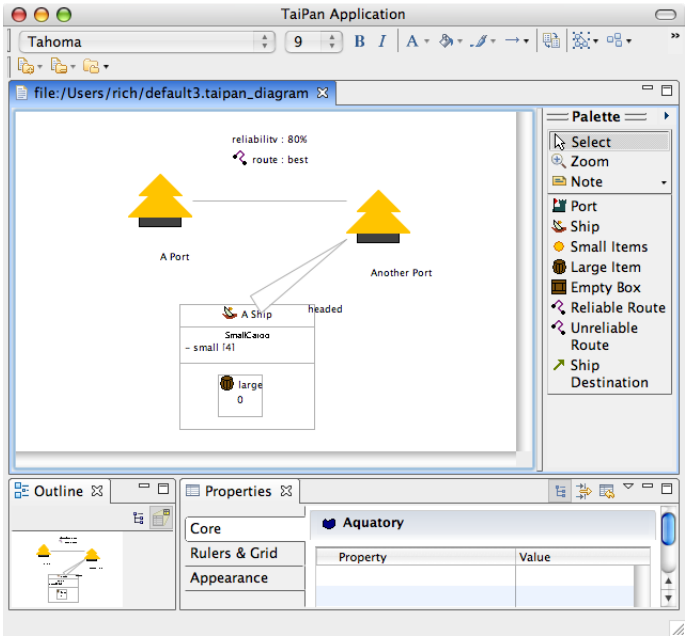
Figura 3.42: El editor gráfico nos ayuda en la selección de subelementos

nos desde el propio entorno gráfico modelar un programa a partir del meta-modelo ECore, tal como vemos en la figura 3.41.

El editor nos autocompletará (ver figura 3.42) con los elementos permitidos en cada sección del metamodelo.

3.4.2. GMP Graphical model proyect

Mediante [10] podemos ampliar el editor generado anteriormente y darle un aspecto totalmente personalizado para nuestro dominio de trabajo. Graphical mode-



Fuente: Web GMF [13]

Figura 3.43: GMF permite personalizar el editor de objetos modelo a un nivel mayor.

ling proyect (GMP) proporciona un kit completo de desarrollo, incluyendo el editor gráfico Graphical modeling framework (GMF).

Este tipo de editor requiere bastante esfuerzo adicional y queda fuera del alcance del proyecto, pero su mención es totalmente necesaria ya que nos permite crear editor gráficos basados en el editor EMF y totalmente en este caso adaptados al dominio.

En la figura 3.43 se muestra un ejemplo. Este ejemplo podemos verlo completo en el tutorial de GMF [13].

3.5. DSL

Los modelos ECore junto a sus transformaciones de modelo generados en Java, tienen la finalidad de servirnos como apoyo en la creación de un DSL. Este DSL no será más que un metamodelo de un programa final que utiliza las propiedades de los metadatos obtenidos por el parser del DSL.

Al crear un lenguaje propio podemos especificar que realiza nuestro proyecto, y no como tiene que realizar dichos procesos.

El lenguaje propuesto es un lenguaje fuertemente tipado donde el usuario no puede cometer errores de tipo, tanto en estructura, en tipo de variables o tipo de eventos. Los tipos de datos a definir por el usuario tendrán que ser los especificados en los modelos ECore. La posibilidad de interpretar datos en ejecución queda descartada por la naturaleza del DSL.

El formato de este lenguaje es una combinación de clave valor para identificar los elementos o funciones. Las listas de elementos las englobaremos entre llaves: {}.

3.5.1. Generación de gramática a partir de modelos

La gramática de nuestro lenguaje será definida en un archivo Xtext. Este puede ser creado desde cero, o mediante un *wizard* que creará una serie de reglas a partir de nuestros modelos ECore. Estas reglas base nos ayudarán a no tener que escribir este código previamente implementado en ECore.

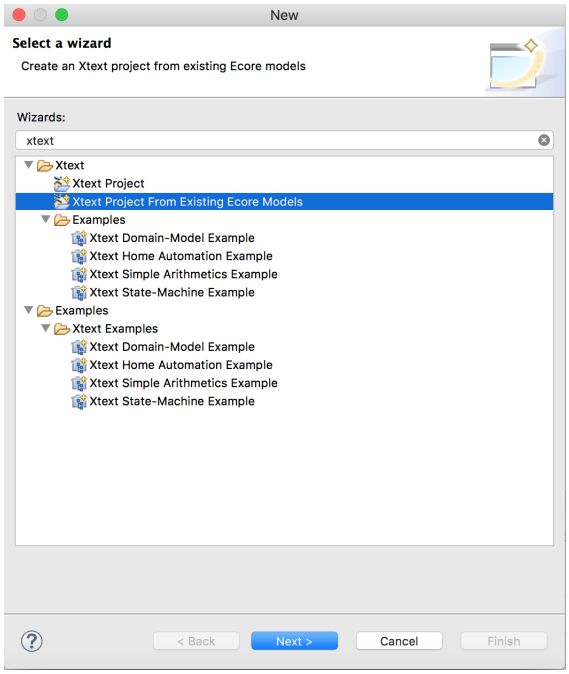
Solamente tendremos que realizar los retoques necesarios para que la gramática se adapte a nuestras necesidades.

Necesitaremos seguir los siguientes pasos:

1. Crearemos un nuevo proyecto de tipo *Xtext from ecore models* tal como podemos ver en la figura 3.44.
2. Una vez seleccionemos el proyecto anterior con los modelos (ver figura 3.45), tendremos que seleccionar cual será el elemento raíz de nuestra gramática, en nuestro caso el elementos es *IotProyect*.
3. Por último necesitaremos especificar el nombre del lenguaje y la extensión de archivo que tendrá (ver figura 3.46). Realizando estos pasos, el editor eclipse será capaz de aplicar las reglas gramaticales a este archivo.

3.5.2. Gramática

La figura 3.47 muestra un listado ejemplo de definición de un evento. En este caso el evento *analog read on high* que como el nombre indica, aplicara un evento cuando lea un valor desde un pin analógico y su valor sea mayor que un valor dado, consultando dicho estado cada x iteraciones o loops.



Fuente: Elaboración propia

Figura 3.44: Creación de un proyecto Xtext desde un proyecto EMF ecore.

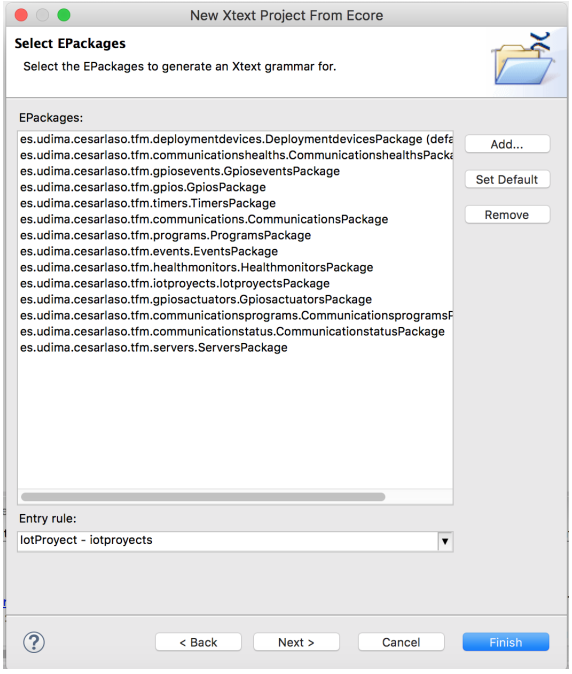
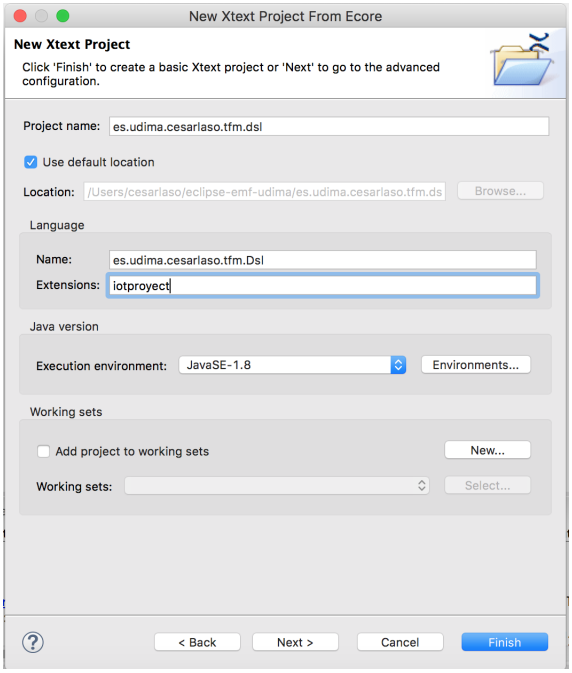


Figura 3.45: Xtext en su importación desde ECore. Selección de modelos que serán generados en la gramática y selección de elemento raíz.



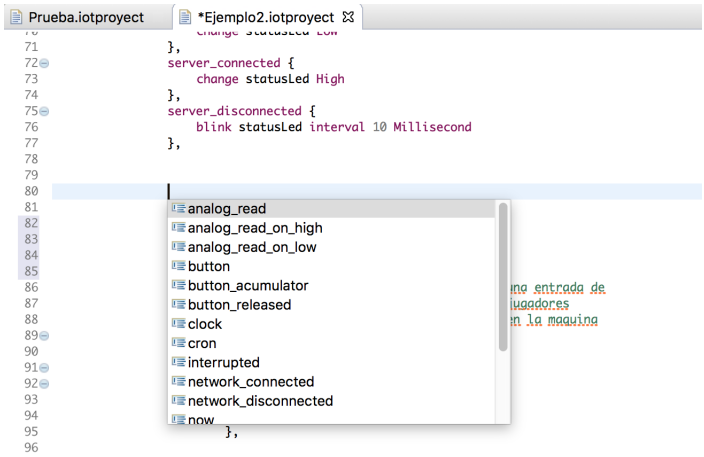
Fuente: Elaboración propia

Figura 3.46: Nombre de nuestro DSL y extensión de archivo para el editor Eclipse EMF.

```
GpioAnalogReadPerformOnHigh returns gpiosevents::
   AnalogReadPerformOnHigh:
  'analog_read_on_high'
  pin=[gpios::AnalogPin|EString]
  'every' pollIntervalLoops=EInt 'loops'
  'value_higher_than' value=EInt
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;
```

Fuente: Elaboración propia

Figura 3.47: DSL - Ejemplo de gramática para un evento



Fuente: Elaboración propia

Figura 3.48: Editor DSL. Autocompletado con modelos que implementen eventos.

Puede consultarse la definición completa del lenguaje creado en este proyecto en el anexo B.2.

3.5.3. Ejemplos de programas

Xtext proporciona al usuario final desarrollador del proyecto, un editor textual con soporte de auto completado. Al ser nuestro DSL fuertemente tipado y específico al dominio, nos evita cometer errores ya no solo en tipos sino en estructura.

La figura 3.48 muestra la sección de programa, el auto completado solo permite que se añadan secciones que hereden del modelo Evento, ya sean estas de tipo GPIO, eventos de red, o eventos de tiempo. Si introdujéramos otro elemento de la gramática, el editor mostraría un error de compilación.

A continuación podemos ver en la figura 3.49 como al especificar un elemento de tipo evento GPIO, el auto completado rellena la selección solamente con las variables anteriormente definidas, estas de tipo *GpioInput*, no permitiendo especificar variables de tipo salida en una entrada, o variables de tipo de digital en una analógica.

En la figura 3.50 vemos como el auto completado y restricción de la gramática permite no solo definir elementos globales sino incluso definir elementos dentro de elementos, en este caso, una entrada de pulso asociado a una variable ya existente, solo podrá contener dos modos válidos de interrupción.

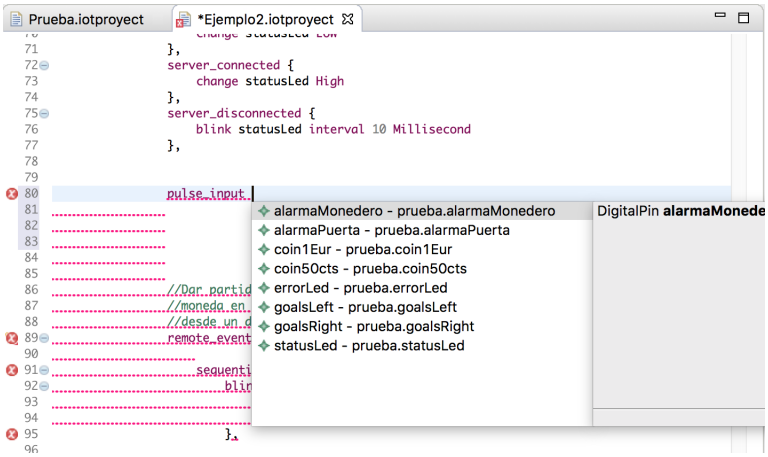
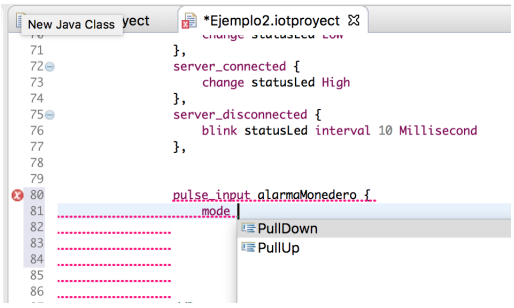
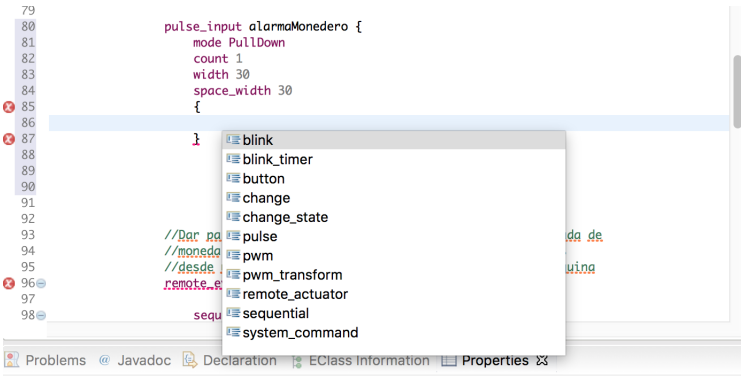


Figura 3.49: Editor DSL. El editor DSL solo permitirá utilizar las variables previamente definidas y permitidas para el tipo de evento definido en la gramática.



Fuente: Elaboración propia

Figura 3.50: Edición de programa DSL, los atributos están fuertemente tipados también en valor.



Fuente: Elaboración propia

Figura 3.51: Edición de programa DSL, al crear nuevos actuadores, nos muestra selección de los existentes.

Por último, podemos ver en la figura 3.51 como el editor nos auto completa con los elementos ya referenciados. En el caso de definir un actuador para un evento, el editor muestra todas las reglas gramaticales válidas para un actuador, del tipo que sea, incluso un grupo de actuadores secuenciales donde incluir a su vez otros actuadores.

3.6. Ejemplos

Con la gramática generada, podemos generar una amplia variedad de programas orientados al IOT.

Como ejemplo, pueden consultarse en la sección anexos los siguientes ejemplos de programas:

- Programa monedero. (Véase apéndice C.1).
- Programa futbolín. (Véase apéndice C.2).
- Programa ascensor. (Véase apéndice C.3).

3.7. Generación de código

3.7.1. Objetivos

Debido a los requisitos del proyecto, esta sección se realizará como experimentación de uno de los módulos del DSL , sin llegar a un código completo del DSL debido a que el tiempo necesario para completar esta parte queda fuera del alcance del proyecto.

3.7.2. Xtend

La generación de código a partir de la gramática anteriormente especificada en Xtext se realiza mediante el sistema de plantillas del lenguaje de programación [15].

Este lenguaje, compilado y ejecutado en la JVM cuenta con capacidades adicionales al lenguaje oficial Java.

apareció el año 2011, cuando Java no contaba con elementos como programación funcional, comprobación de tipos adicional, métodos adicionales en clases, inferencia automática de tipos, etc En su web muestran el lenguaje como Java10 hoy en día, ya que muchas de las opciones prometidas en java10 tampoco han sido post puestas para versiones posteriores.

Para este proyecto, la funcionalidad que no tiene java que tiene por ejemplo el lenguaje Scala (mediante la expansión de macros), es la posibilidad de tener plantillas con expresiones dentro del propio lenguaje.

Una característica adicional de es que no compila directamente a bytecode tal como lo hacen el resto de lenguajes alternativos de la JVM como (EPFL) [3] o s.r.o [45] , sino que transforma el lenguaje a código java, por lo que no requiere un runtime adicional como estos otros lenguajes.

Veamos un ejemplo en la figura 3.52 del código generado para convertir un atributo ECore a código java final.

Podemos ver un ejemplo completo de la generación de código para uno de los módulos de comunicaciones en el anexo B.3.

```
def generateReaderType(EAttribute attribute) {
  if (attribute.upperBound != 1) {
    //es un array
    '''
      final int  attribute  .name _size = in.readInt();
      final List<  attribute  .getEType.name >  attribute  .name  =
   new ArrayList<>(  attribute  .name _size);
      for (int i=0; i<  attribute  .name _size; i++) {
          readFuncName  (attribute.getEType.eClass)  ();
      }
    '''
  } else {
    '''
      final  attribute  .getEType.name  attribute  .name  = in.
     simpleTypeReader  (attribute.getEType)  ();
    '''
  }
}
```

Fuente: Elaboración propia

Figura 3.52: Código Xtend generando plantilla java. Mediante el uso del API ECore podemos acceder a los atributos de los modelos de forma fuertemente tipada. Interpolando estos en la plantilla, genemos código Java

3.8. Recursos utilizados

Para el desarrollo de este trabajo se han consultado los siguientes libros:

- Thomas Stahl, Markus Voelter y Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006. ISBN: 0470025700 Libro de referencia de la asignatura *Diseño avanzado en arquitecturas software*.
- Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend - Second Edition. Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices*. Packt Publishing, 2016. ISBN: 9781786464965.
- Leon Starr, George Mangogna y Stephen Mellor. *Models to Code. With No Mysterious Gaps*. Apress, 2017. ISBN: 978-1-4842-2217-1. DOI: 10.1007/978-1-4842-2217-1.
- Stephen J. Mellor, Kendall Scott y Dirk Weise. *MDA distilled: principles of model-driven architecture*. Addison-Wesley, 2004, pág. 150. ISBN: 0201788918.

Como material hardware se ha contado con el siguiente material para realizar pruebas:

- Raspberry PI [17] modelo B. Para el desarrollo de la funcionalidad IOT en java.
- ESPDuino (Modulo ESP8266 compatible Arduino). Para la comprobación de la funcionalidad en Arduino y freeRTOS.

Herramientas informáticas entornos IDE:

- Eclipse [8] EMF [9] para ECore, Xtext, Xtend.
- Netbeans para Java. Para la edición de código Java.
- IDE Arduino [29]. Para la realización de pruebas en código C y extraer la información necesaria para posteriormente generar modelos compatibles con los modelos Java.

Editores de texto y compilación de esta memoria en formato PDF:

- Latex [31], entorno completo para Linux y Mac.
- ShareLatex.com [37]. Edición y compilación online de archivos Latex.
- Adobe AcrobatReader. Para la visualización y comprobación de la memoria.

Máquinas virtuales:

- Java SE

El resto de recursos que puntualmente han sido utilizados serán anotados en sus correspondientes secciones y posteriormente al final del trabajo en la sección bibliografía.

Capítulo 4

Conclusiones y líneas futuras

4.1. Conclusiones

El desarrollo mediante MDE consigue centrar todo el proceso de diseño de la arquitectura del sistema en la realización de modelos y transformación entre estos modelos. Esto permite centralizar en una misma fuente de información tanto diagramas de clases, documentación y generación de código, tanto de , como de generación de código para distintos lenguajes de programación y plataformas mediante Xtext. Todo partiendo desde la misma fuente de información, los modelos ECore, realizando transformaciones y representaciones de estos modelos en sus respectivos formatos.

Un nivel tan alto de abstracción cuenta con una serie de inconvenientes, pues corremos el riesgo de alejarnos demasiado de nuestro producto o negocio y realizar un diseño demasiado genérico.

Empezar de cero utilizando esta metodología, sin contar con una amplia experiencia en esta, nos va a llevar a no poder cumplir unos plazos razonables. Solo deberíamos utilizar este tipo de metodologías, una vez dentro de nuestros productos necesitemos este grado de abstracción. Antes de ello siempre deberíamos definir nuestra linea de productos software siempre prefiriendo los siguientes pasos [27]:

1. Reglas de negocio dentro del propio código.
2. Reglas de negocio mediante variables en el propio código en compilación.
3. Configuración externalizada en runtime.

4. Motores de reglas dinámicos en runtime.
5. DSL y generación de código.

No podemos discutir la ventaja de usar un DSL, solamente tenemos que ver ejemplos de uso diario entre toda la comunidad dedicada a temas relativos a la informática, ejemplos como SQL, HyperText Markup Language (HTML), Cascading Style Sheets (CSS), etc; Lenguajes ampliamente conocidos y muy orientados a su dominio específico.

Realizar un DSL no es tarea fácil, solamente tenemos que ver la evolución de los lenguajes de programación generalistas para ver como en sus sucesivas iteraciones han ido evolucionando adaptándose a las necesidades del mercado, y en demasiadas ocasiones a tendencias o modas no alineadas con lo solicitado por el mundo empresarial. Pongamos como ejemplo el lenguaje de programación genérico , orientado a funcional Haskell [49], originado en el mundo académico y en el momento de escribir esta memoria, con poco recorrido.

Se debería utilizar un DSL cuando son sistemas no cambiantes y altamente definidos. Tomemos como ejemplo sistemas de soporte de vida, sistemas de tráfico, sistemas de aviación, satélites... Es altamente recomendable comenzar directamente con modelos y un lenguaje específico de dominio ya que las reglas del negocio es muy improbable que cambien a corto plazo.

En sistemas de otro tipo, según mi experiencia es muy complicado realizar un lenguaje específico del dominio ya que constantemente estamos modificando todas las capas del producto con cada cambio. En productos estables se puede llegar a utilizar motores de reglas, o simplemente simples archivos de configuración, fijos o modificables en runtime.

En este caso, con la realización de este desarrollo, hemos abstraído tal vez, demasiada información útil de los dispositivos a utilizar, dejando de lado posibles mejoras para un producto que seguramente estará siempre muy unido a una plataforma. Tomemos como referencia a Joel Spolsky en la entrada *The development abstraction layer* [39].

Hoy en día contamos con lenguajes donde es posible realizar meta programación desde el propio lenguaje, utilizando un DSL interno, en vez de externo mediante un lenguaje con sintaxis propia, o mediante una sintaxis en un formato de archivo conocido como XML o JavaScript Object Notation (JSON).

Igualmente, los propios IDE cuentan con herramientas de generación de código

integrado en el propio proceso de desarrollo, lo que simplifica mucho el desarrollo de producto.

Otro problema que igualmente tienen otros DSL es la dificultad de uso para alguien que no pertenezca al equipo de desarrollo. Esto supone un gran problema, ya que es un nuevo lenguaje a aprender y dominar, en este caso un lenguaje muy sencillo, en el caso de un lenguaje complejo, puede suponer un problema.

Al realizar este trabajo se ha constatado que se ha cometido el mismo error que otras plataformas de despliegue, es decir, crear un sistema propio para esta tarea. Los lenguajes de programación actuales son de muy alto nivel y permiten mucha mayor expresividad que la que podemos realizar mediante meta-modelado.

Este problema, muy actual en informática, es realmente importante, cada fabricante quiere que utilicemos su framework, dando una sensación de tener algo novedoso que nos soluciona los problemas, siendo todo lo contrario, nos ancla al uso de su producto.

Como solución, deberíamos intentar generar librerías para su utilización posterior en otros programas.

Otro problema que veo de difícil solución es la creación de código generado a partir de plantillas de texto. Esta transformación es poco útil, ya que perdemos la cantidad de utilidades que nos aportan los entornos de programación actuales, generando todo ese código extra que EMF genera. Código generable con lenguajes que permitan generar código dentro del propio lenguaje como C# o Scala, permitiendo meta-programación directamente en el propio lenguaje creando sintaxis propia utilizando la propia arquitectura base.

4.1.1. Sobre Eclipse Modeling Framework

Eclipse cuenta con una comunidad de usurarios grande y estable. Igualmente cuenta con multitud de desarrollos basados en este entorno, como por ejemplo EMF. Algunos de estos proyectos a su vez basados en EMF.

Tal multitud de proyectos nos llevan a tener muchos de ellos con características muy poco probadas con multitud de fallos. El desarrollo de este proyecto ha sido realmente tedioso principalmente por el mal funcionamiento de la plataforma por los siguientes motivos:

- Versiones no actualizadas con el entorno Eclipse actual ya que la última versión data de 2014.
- Multitud de funcionalidad discontinuada o no compatible con el entorno Eclipse actual.
- Incapacidad de renombrar elementos en EMF. Una vez renombrado un elemento, EMF no es capaz de realizar un refactoring adecuado, siendo necesaria la modificación del elemento en todas las capas de abstracción. Esto no debería ser necesario, ya que tanto Java como Xtext conocen estos elementos en compilación, y el propio IDE podría resolver el problema automáticamente.
- Cache no coherente en EMF. Con la modificación de cualquier elemento,EMF no es capaz de actualizar su representación interna. Solucionado eliminando todos los caches que genera en disco y reiniciando la aplicación. Esto implica lo siguiente: Se está generando un cache de datos que luego no sirven para nada, con la gran carga de Input / Output (IO) que supone. Para el desarrollo de este proyecto se ha solucionado estos tiempos de espera mediante la utilización de discos virtuales en RAM. Teniendo el *workspace* y la propia aplicación Eclipse montada directamente sobre un disco en RAM. Aún así no nos libramos de los tiempos de carga de todo el entorno que de por sí son siempre altos.
- Desarrollo prueba y error: La transformación de código a partir de los modelos, es realizada a partir de plantillas de texto con su propio lenguaje de programación (Xtext). Comprobar el código generado acaba convirtiéndose en una especie de lo que comúnmente llamamos printf debugging, algo que con los IDEs actuales debería estar más que solucionado. La integración mediante pruebas unitarias es realmente complicada en su integración y demasiado lenta en su ejecución.
- Mezcla de lenguajes: La generación de código en base a texto es una práctica no recomendable y que nos puede llevar a dejar fallos de seguridad. El paso transformación modelo a código ejecutable estaría mejor realizado si esta transformación fuera tipada, mediante una generación explícita para lenguaje.
- Huevo o gallina: ¿Que fue primero el modelo o la implementación? Esto es algo que creo es un error fundamental y un modelo no debería depender de una implementación concreta algo que en ECore ocurre por defecto.
- Simplicidad: Conseguimos hacer lo simple complicado y lo complicado igual de complicado. Mediante modelos no podemos expresar con la misma facilidad los requerimientos del dominio que con código. Sólo podremos realizarlo siempre que nuestro DSL cuente con todo lo necesario. Un DSL no puede ser un trabajo menor, es la base de todo el producto.

- Tendencia en modelado: No parece que el uso de MDA/MDE/MDSD/MDx sea algo que se este utilizando. Este tipo de metodología no termina de convencer a los usuarios, tan solo se mueve en entornos académicos de investigación y leyendo varios artículos de investigación, parece que todos terminan como este mismo TFM, hablando de la plataforma, realizando una mínima implementación de algo muy sencillo, pero sin ninguna implementación en proyectos reales, de varias dimensiones más complejos.

- Documentación: Por una parte genera de forma semi automática documentación en varios formatos, tales como: diagramas de clases, javadoc, excel... Documentación que solo sirve a personal técnico, documentación que al propio técnico le será mucho más útil ver directamente el código de alto nivel (no auto generado) como por ejemplo interfaces.

4.1.2. Sobre DSL

Los lenguajes específicos de dominio son útiles cuando el dominio del que tratan es especializado, pero no en un dominio de negocio sino técnico. Esto es algo que vemos en muchos DSL, lenguajes de bases de datos (SQL), de marcado de documentos (HTML), de estilo de documentos (CSS), de formateo de datos (XML).

Todos estos DSL siempre cuentan con que no son aplicaciones finales sino componentes que utilizaremos para una aplicación final.

En este caso hemos realizado un DSL para la creación de aplicaciones IOT, caso anteriormente descrito, en vez de realizar un DSL para una aplicación final. Ejemplo, DSL para la automatización de un contador de monedas.

En este caso sería útil la generación de código a partir de modelos, pero no parece práctica esta generación en la forma en la que se realiza con estas plataformas, siendo mucho más útil la generación de código en un DSL dentro del propio lenguaje, como por ejemplo mediante macros en lenguajes como Scala, o las multitudes de variantes de Lisp como por ejemplo Clojure (situándolo en la misma maquina virtual de java como scala).

4.2. Líneas futuras

El proyecto cuenta con un número limitado de componentes modelados tanto en eventos, actuadores, acceso a sistema operativo y hardware. A modo demostrativo para este trabajo nos podemos hacer una idea que si este DSL continuara en su desarrollo sería necesario integrar tanto librerías para estos dispositivos, sensores como actuadores (sensores de temperatura, humedad, presión, pantallas, motores...), librerías adicionales.

Por ello, tal vez el uso de debería ser utilizad de la siguiente forma, en vez de ser los propios modelos los que generen un código final, los metamodelos generen código intermedio que hagan de enlace entre kits de desarrollo o APIs ya existentes y el DSL.

Por otra parte, por lo limitado del alcance de este proyecto, no ha sido posible el desarrollo de la generación de código del DSL a su implementación para la JVM o para plataformas integradas como Arduino. Una tarea de continuación sería primero comenzar con una arquitectura con sistema operativo de proposito general como Raspberry PI o Beaglebone, y una vez desarrollado este entorno, generar código para Arduino en general, y código para el sistema en tiempo real FreeRTOS.

Capítulo 5

Referencias bibliográficas y recursos electrónicos

Referencias

[1] Richard Barry y Real Time Engineers Ltd. *FreeRTOS*. URL: https://www.freertos.org.

[2] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend - Second Edition. Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices*. Packt Publishing, 2016. ISBN: 9781786464965.

[3] École Polytechnique Fédérale Lausanne (EPFL). *The Scala Programming Language*. URL: https://www.scala-lang.org.

[4] Espressif. *ESP8266*. URL: https://www.espressif.com.

[5] Peter Fettke y Peter Loos. "Model driven architecture (MDA)". En: 45 (oct. de 2003), págs. 555-559.

[6] openHAB Foundation e.V. *openHab empowering the smart home*. 2018. URL: https://www.openhab.org.

[7] Eclipse foundation. *Ayuda referencia EMF Edit*. URL: http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.Edit.html (visitado 06-2018).

[8] Eclipse foundation. *Eclipse*. URL: https://www.eclipse.org.

[9] Eclipse foundation. *Eclipse entity model framework*. URL: https://www.eclipse.org/emf.

[10] Eclipse foundation. *Eclipse graphical modeling proyect*. URL: http://www.eclipse.org/modeling/gmp/ (visitado 06-2018).

[11] Eclipse foundation. *Eclipse Sirius*. URL: https://www.eclipse.org/sirius/.

[12] Eclipse foundation. *Framework JFace eclipse*. URL: https://wiki.eclipse.org/JFace#More_information.

[13] Eclipse foundation. *GMF tutorial*. URL: https://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1.

[14] Eclipse foundation. *MBEDDR*. URL: http://mbeddr.com/.

[15] Eclipse foundation. *Xtend is a flexible and expressive dialect of Java*. URL: http://www.eclipse.org/xtend/.

[16] Eclipse foundation. *Xtext*. URL: https://www.eclipse.org/Xtext/.

[17] Raspberry foundation. *Raspberry PI*. URL: https://www.raspberrypi.org.

[18] Erich Gamma y Kent Beck. *JUnit*. URL: https://junit.org.

[19] Object Management Group. *Constraint Language Specification*. 2014. URL: https://www.omg.org/spec/OCL.

[20] Object Management Group. *CWM Metadata Interchange Patterns Specification*. 2004. URL: https://www.omg.org/spec/MIPS.

[21] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models*. 2017. URL: https://www.omg.org/spec/FUML.

[22] Object Management Group. *Software Systems Process Engineering Metamodel Specification*. 2008. URL: https://www.omg.org/spec/SPEM.

[23] Object Management Group. *The MetaObject Facility Specification*. 2016. URL: https://www.omg.org/spec/MOF.

[24] Object Management Group. *UML Profile for Enterprise Distributed Object Computing Specification*. 2004. URL: https://www.omg.org/spec/EDOC.

[25] Object Management Group. *Unified Modeling Language Specification*. 2017. URL: https://www.omg.org/spec/UML.

[26] Object Management Group. *XML Metadata Interchange Specification*. 2015. URL: https://www.omg.org/spec/XMI.

[27] Mike Hadlow. *The configuration complexity clock*. URL: http://mikehadlow.blogspot.com/2012/05/configuration-complexity-clock.html.

[28] Texas Instruments. *Beaglebone*. URL: http://beagleboard.org.

[29] Design Institute Ivrea. *Arduino*. URL: https://www.arduino.cc.

[30] Pieter Koopman y Rinus Plasmeijer. "A Shallow Embedded Type Safe Extendable DSL for the Arduino". En: (2016). Ed. por Manuel Serrano y Jurriaan Hage, págs. 104-123.

[31] Leslie Lamport. *LaTeX – A document preparation system*. URL: https://www.latex-project.org.

[32] César Laso. *Código fuente del TFM*. URL: https://cesarlaso.com/udima/tfm/arquitectura-software-src.zip.

[33] Stephen J Mellor. "MDA Distilled". En: *Project Technology, Inc* (2003), págs. 2, 3. DOI: 10.1.1.453.6508. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.453.6508&rep=rep1&type=pdf.

[34] Stephen J. Mellor, Kendall Scott y Dirk Weise. *MDA distilled: principles of model-driven architecture*. Addison-Wesley, 2004, pág. 150. ISBN: 0201788918.

[35] Cátedra SAES - Universidad Murcia. *C++ EMF Implementation code*. URL: https://github.com/catedrasaes-umu/emf4cpp.

[36] Cátedra SAES - Universidad Murcia. *C++ implementation and type mapping for the Eclipse Modeling Framework (EMF) core*. URL: https://www.catedrasaes.org/html/projects/EMF4CPP/EMF4CPP.html.

[37] Overleaf. *Share latex*. URL: https://sharelatex.com/.

[38] Software Technology Research Group of Radboud University Nijmegen. *Clean programming language*. URL: https://clean.cs.ru.nl/Clean.

[39] Joel Spolsky. *The development abstraction layer*. URL: https://www.joelonsoftware.com/2006/04/11/the-development-abstraction-layer-2/.

[40] Brinkley Sprunt, Lui Sha y John Lehoczky. "Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System". En: *Software Engineering Institute, Carnegie Mellon University* (1989). DOI: 10.1.1.207.9431. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.207.9431&rep=rep1&type=pdf.

[41] Jetbrains s.r.o. *intelliJ Idea*. URL: https://www.jetbrains.com/idea/.

[42] Jetbrains s.r.o. *Jetbrains*. URL: https://www.jetbrains.com/.

[43] Jetbrains s.r.o. *MPS*. URL: https://www.jetbrains.com/mps/.

[44] Jetbrains s.r.o. *MPS FAQ*. URL: https://confluence.jetbrains.com/display/MPSD20181/FAQ.

[45] Jetbrains s.r.o. *Statically typed programming language for modern multiplatform applications*. URL: https://kotlinlang.org.

[46] Thomas Stahl, Markus Voelter y Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006. ISBN: 0470025700.

[47] Leon Starr, George Mangogna y Stephen Mellor. *Models to Code. With No Mysterious Gaps*. Apress, 2017. ISBN: 978-1-4842-2217-1. DOI: 10.1007/978-1-4842-2217-1.

[48] UDIMA. *Diseño avanzado en arquitecturas software*. URL: https://www.udima.es/es/diseno-avanzado-arquitecturas-software-master.html.

[49] Universidad de Yale y Universidad de Glasgow. *Haskell*. URL: https://www.haskell.org/.

Capítulo 6

Agradecimientos

A mi mujer y compañera de la vida Marianela.

Al Dr. Javier Fernandez Salgado por su ánimo, su ayuda prestada en tutorías y por la dirección de este trabajo fin de master.

Al Dr. Javier Bravo Agapito por la continuación en la dirección y posteriores tutorías y correcciones finales de este trabajo.

Capítulo 7

Autorización de difusión

El/la abajo firmante, matriculado/a en el Máster Universitario en Arquitectura del Software, autorizará a la Universidad a Distancia de Madrid (UDIMA) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el Trabajo Fin de Máster: "DSL orientado al IOT con arquitectura MDE", realizado durante el curso académico 2016-2017 bajo la dirección de Dr. Javier Fernandez Salgado y de Dr. Javier Bravo Agapito del Departamento de Ingeniería en Informática y Organización Industrial y a la Biblioteca de la UDIMA a depositarlo en el Archivo Institucional con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

César Laso Laso con DNI 72980225Y

cesar@cesarlaso.com

LASO LASO CESAR - 72980225Y

Firmado digitalmente por LASO LASO CESAR - 72980225Y
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-72980225Y, givenName=CESAR, sn=LASO LASO, cn=LASO LASO CESAR - 72980225Y
Fecha: 2018.06.04 23:02:42 +02'00'

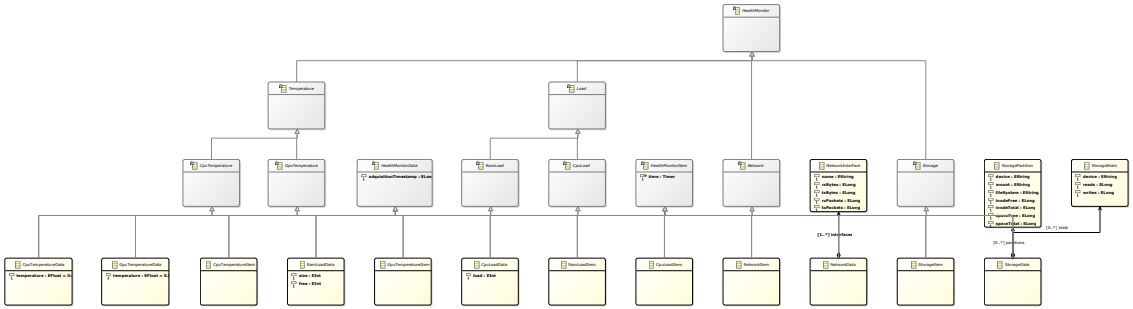
Apéndice A

Modelos adicionales Ecore

En esta sección anexa aparecen las figuras que por ocupar demasiado tamaño, no proceden aparecer en el contenido principal de la memoria.

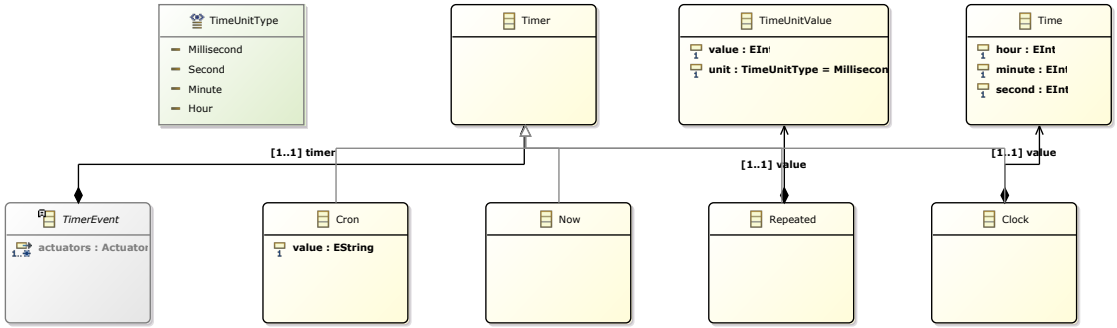
Fuente: Elaboración propia

Figura A.1: Proyecto IOT - Representación de las dependencias entre los modelos



Fuente: Elaboración propia

Figura A.2: Modelo Estado de salud. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius



Fuente: Elaboración propia

Figura A.3: Modelo Timers. Representación visual en formato diagrama de clases UML creada a partir del modelo Ecore mediante la transformación generada con Sirius

Apéndice B

DSL

B.1. Ejemplo de programa ejemplo - Versión conceptual

```
Programa "contador de monedas"

remote_event MONEDA_LEIDA 1
remote_event DAR_MONEDA 2

define actuator parpadear_led_accion
do
  send_pulses on GPIO_02 with 250 ms and 100 ms during 3 times
done

when button_pressed GPIO_01 with 100 ms
do
  notify_action MONEDA_LEIDA
  actuator parpadear_led_accion
done

when remote_event DAR_MONEDA_50
do
  send_pulses on GPIO_03 with 100 ms and 50 ms during 2 times
  actuator parpadear_led_accion
  remote_event_ack
done

when remote_event DAR_MONEDA_100
do
  //Simula dos monedas de 50
  send_pulses on GPIO_03 with 100 ms and 50 ms during 4 times
```

```
  actuator parpadear_led_accion
  remote_event_ack
done

when rising on GPIO_02 with debounce of 50ms
do
  notify_action CAJON_MONEDAS_ABIERTO
done

when falling on GPIO_02 with debounce of 50ms
do
  notify_action CAJON_MONEDAS_CERRADO
done

when button_pressed_and_released on GPIO_05 with debounce of 100ms
do
  reset
done

when button_count on GPIO_04 with debounce of 100ms and 3 counts
do

done
```

ejemplos/dsl_motivacion_ejemplo.txt

B.2. Gramática con el DSL del proyecto

```
// Gramatica para el trabajo fin de master sobre EMF e IOT version
   2
grammar es.udima.tfm.cesarlaso.IotDsl with org.eclipse.xtext.common
   .Terminals

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

import "platform:/resource/es.udima.cesarlaso.tfm/model/Events.
   ecore" as events

import "platform:/resource/es.udima.cesarlaso.tfm/model/
   HealthMonitors.ecore" as healthmonitors

import "platform:/resource/es.udima.cesarlaso.tfm/model/
   Communications.ecore" as communications
import "platform:/resource/es.udima.cesarlaso.tfm/model/
   CommunicationsHealth.ecore" as communicationshealt
import "platform:/resource/es.udima.cesarlaso.tfm/model/
   CommunicationsPrograms.ecore" as communicationsprograms
```

```
import "platform:/resource/es.udima.cesarlaso.tfm/model/
   CommunicationsStatus.ecore" as communicationstatus
import "platform:/resource/es.udima.cesarlaso.tfm/model/
   CommunicationsEvents.ecore" as communicationsevents

import "platform:/resource/es.udima.cesarlaso.tfm/model/
   OperatingSystemActuators.ecore" as operatingsystemactuators

import "platform:/resource/es.udima.cesarlaso.tfm/model/Gpios.ecore
   " as gpios
import "platform:/resource/es.udima.cesarlaso.tfm/model/
   GpiosActuators.ecore" as gpiosactuators
import "platform:/resource/es.udima.cesarlaso.tfm/model/GpiosEvents
   .ecore" as gpiosevents

import "platform:/resource/es.udima.cesarlaso.tfm/model/Programs.
   ecore" as programs
import "platform:/resource/es.udima.cesarlaso.tfm/model/Timers.
   ecore" as timers

import "platform:/resource/es.udima.cesarlaso.tfm/model/
   DeploymentDevices.ecore" as deploymentdevices
import "platform:/resource/es.udima.cesarlaso.tfm/model/Servers.
   ecore" as servers

import "platform:/resource/es.udima.cesarlaso.tfm/model/IotProyects
   .ecore"


IotProyect returns IotProyect:
  'IotProyect'
  name=EString
  '{'
    'product_code' productCode=EInt
    'version' version=EInt

    'devices' '{' deploymentDevice+=Device ( "," deploymentDevice+=
   Device)* '}'
    'servers' '{' server+=Server ( "," server+=Server)* '}'

    program=Program
  '}';


Device returns deploymentdevices::Device:
  Simulator | Raspberry | Beaglebone | ESP8266;

Server returns servers::Server:
  JavaServer;
```

```
Pin returns gpios::Pin:
  DigitalInputPin | DigitalOutputPin |
  AnalogInputPin | AnalogOutputPin;

Event returns events::Event:
  GpioInterrupted | GpioButtonInput | GpioButtonInputAccumulator |
   GpioButtonReleased |  GpioPulseInput |
  GpioAnalogRead | GpioAnalogReadPerformOnHigh |
   GpioAnalogReadPerformOnLow |
  NowEvent | RepeatedEvent | ClockEvent | CronEvent |
  RemoteEvent |
  NetworkConnected | NetworkDisconnected | ServerConnected |
   ServerDisconnected;

Actuator returns events::Actuator:
  SequentialActuators |
  ChangeOutput | ButtonOutput | PulseOutput | BlinkOutput |
   BlinkTimerOutput |
  Pwm | TransformTo |
  RemoteActuator |
  SystemCommand | Download | PlaySound | BackgroundImage |
   DrawImage | PlayVideo |
  ChangeStateActuator;

RemoteIdentifier returns communicationsprograms::RemoteIdentifier:
  RemoteEventIdentifier | RemoteActuatorIdentifier
;



EString returns ecore::EString:
  STRING | ID;

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

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


Program returns programs::Program:
  'program'
  '{'
    ( 'gpio_aliases' '{' gpioAliases+=Pin ( "," gpioAliases+=Pin)*
   '}' )?
    ( 'remote_identifiers' '{' remoteIdentifiers+=RemoteIdentifier
   ( "," remoteIdentifiers+=RemoteIdentifier)* '}' )?
    ( 'files' '{' files+=FilePath ( "," files+=FilePath)* '}' )?

    'start_state' state=[programs::ProgramState|EString]
```

```
    'states' '{' states+=ProgramState ( "," states+=ProgramState)*
    '}'
  '}';

ProgramState returns programs::ProgramState:
  'state'
  name=EString
  '{' events+=Event ( "," events+=Event)* '}';


NetworkConnected returns communicationsevents::NetworkConnected:
  'network_connected'
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;

NetworkDisconnected returns communicationsevents::
    NetworkDisconnected:
  'network_disconnected'
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;

ServerConnected returns communicationsevents::ServerConnected:
  'server_connected'
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;

ServerDisconnected returns communicationsevents::ServerDisconnected
    :
  'server_disconnected'
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;


NowEvent returns timers::NowEvent:
  'now'
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
  ;

RepeatedEvent returns timers::RepeatedEvent:
  'repeat_every'
  value=TimeUnitValue
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
  ;

ClockEvent returns timers::ClockEvent:
  'clock'
  value=Time
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
  ;

CronEvent returns timers::CronEvent:
```

```
  'cron'
  value=EString
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
  ;

GpioInterrupted returns gpiosevents::Interrupted:
  'interrupted'
  pin=[gpios::DigitalInputPin|EString]
  mode=GpioInterruptMode
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;

GpioButtonInput returns gpiosevents::ButtonInput:
  'button'
  pin=[gpios::DigitalInputPin|EString]
  '{'
  'mode' mode=PinResistor
  'min_duration' minDuration=EInt
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
  '}'
;

GpioButtonInputAccumulator returns gpiosevents::
   ButtonInputAccumulator:
  'button_acumulator'
  pin=[gpios::DigitalInputPin|EString]
  '{'
  'mode' mode=PinResistor
  'min_duration' minDuration=EInt
  'count' count=EInt
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
  '}'
;

GpioButtonReleased returns gpiosevents::ButtonInputReleased:
  'button_released'
  pin=[gpios::DigitalInputPin|EString]
  '{'
  'mode' mode=PinResistor
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
  '}'
;

GpioPulseInput returns gpiosevents::PulseInput:
  'pulse_input'
  pin=[gpios::DigitalInputPin|EString]
  '{'
  'mode' mode=PinResistor
  'count' count=EInt
  'width' width=EInt
  'space_width' spaceWidth=EInt
```

```
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
  '}'
;

GpioAnalogRead returns gpiosevents::AnalogRead:
  'analog_read'
  pin=[gpios::AnalogInputPin|EString]
  'every' pollIntervalLoops=EInt 'loops'
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;

GpioAnalogReadPerformOnHigh returns gpiosevents::
   AnalogReadPerformOnHigh:
  'analog_read_on_high'
  pin=[gpios::AnalogInputPin|EString]
  'every' pollIntervalLoops=EInt 'loops'
  'value_higher_than' value=EInt
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;

GpioAnalogReadPerformOnLow returns gpiosevents::
   AnalogReadPerformOnLow:
  'analog_read_on_low'
  pin=[gpios::AnalogInputPin|EString]
  'every' pollIntervalLoops=EInt 'loops'
  'value_lower_than' value=EInt
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;



Time returns timers::Time:
   hour=EInt 'h' minute=EInt 'm' second=EInt 's'
;

TimeUnitValue returns timers::TimeUnitValue:
  value=EInt
  unit=TimeUnitType
;

enum TimeUnitType returns timers::TimeUnitType:
  Millisecond = 'Millisecond' | Second = 'Second' | Minute = '
   Minute' | Hour = 'Hour';


FilePath returns operatingsystemactuators::FilePath:
  'file_path'
  name=EString
  'with_path'
  filePath=EString
;
```

```
RemoteEvent returns communicationsprograms::RemoteEvent:
  'remote_event'
  identifier=[communicationsprograms::RemoteEventIdentifier|EString
   ]
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
  ;



RemoteActuator returns communicationsprograms::RemoteActuator:
  'remote_actuator'
  identifier=[communicationsprograms::RemoteActuatorIdentifier|
   EString]
  ;


SystemCommand returns operatingsystemactuators::SystemCommand:
  'system_command'
  filePath=[operatingsystemactuators::FilePath|EString]
  ;

Download returns operatingsystemactuators::Download:
  'download'
  url=EString
  'to'
  filePath=[operatingsystemactuators::FilePath|EString]
  ;

PlaySound returns operatingsystemactuators::PlaySound:
  'play_sound'
  ( 'with_volume' volume=EInt )?
  filePath=[operatingsystemactuators::FilePath|EString]
  ;

BackgroundImage returns operatingsystemactuators::BackgroundImage:
  'background_image'
  filePath=[operatingsystemactuators::FilePath|EString]
  resizeType=ResizeType
  ;

DrawImage returns operatingsystemactuators::DrawImage:
  'draw_image'
  filePath=[operatingsystemactuators::FilePath|EString]
  '{'
  'x' positionX=EInt
  'y' positionY=EInt
  'z' positionZ=EInt
  'color' color=ELong
  'resize' resizeType=ResizeType
  '}'
```

```
  ;

PlayVideo returns operatingsystemactuators::PlayVideo:
  'play_video'
  filePath=[operatingsystemactuators::FilePath|EString]
  '{'
  'x' positionX=EInt
  'y' positionY=EInt
  'z' positionZ=EInt
  'color' color=ELong
  'resize' resizeType=ResizeType
  '}'
  ;


ChangeStateActuator returns programs::ChangeStateActuator:
  'change_state'
  state=[programs::ProgramState|EString]
  ;

enum ResizeType returns operatingsystemactuators::ResizeType:
  None = 'none' | Auto = 'auto' | Fill = 'fill';

enum PinResistor returns gpios::PinResistor:
  PullUp = 'pull_up' | PullDown = 'pull_down';

enum GpioInterruptMode returns gpios::InterruptMode:
  Rising = 'rising' | Falling = 'falling' | Changes = 'changes' |
   Low = 'low' | High = 'high';


enum DigitalPinStatus returns gpiosactuators::DigitalPinStatus:
  High = 'high' | Low = 'low';



DigitalInputPin returns gpios::DigitalInputPin:
  'gpio_input_digital'
  name=EString
  'connected_to_pin'
  number=EInt
  ;

DigitalOutputPin returns gpios::DigitalOutputPin:
  'gpio_output_digital'
  name=EString
  'connected_to_pin'
  number=EInt
  ;

AnalogInputPin returns gpios::AnalogInputPin:
```

```
  'gpio_input_analog'
  name=EString
  'connected_to_pin'
  number=EInt
  ;

AnalogOutputPin returns gpios::AnalogOutputPin:
  'gpio_output_analog'
  name=EString
  'connected_to_pin'
  number=EInt
  ;

RemoteEventIdentifier returns communicationsprograms::
   RemoteEventIdentifier:
  'remote_event_identifier'
  name=EString
;
RemoteActuatorIdentifier returns communicationsprograms::
   RemoteActuatorIdentifier:
  'remote_actuator_identifier'
  name=EString
;

SequentialActuators returns events::SequentialActuators:
  'sequential'
  '{' actuators+=Actuator ( "," actuators+=Actuator)* '}'
;


ChangeOutput returns gpiosactuators::ChangeOutput:
  'change'
  pin=[gpios::DigitalOutputPin|EString]
  status=DigitalPinStatus
  ;

ButtonOutput returns gpiosactuators::ButtonOutput:
  'button'
  pin=[gpios::DigitalOutputPin|EString]
  'duration' duration=TimeUnitValue
  ;


PulseOutput returns gpiosactuators::PulseOutput:
  'pulse'
  pin=[gpios::DigitalOutputPin|EString]
  '{'
  'times' count=EInt
  'width' width=EInt 'ms'
  'space_width' spaceWidth=EInt 'ms'
  '}';
```

```
BlinkOutput returns gpiosactuators::BlinkOutput:
  'blink'
  pin=[gpios::DigitalOutputPin|EString]
  'interval' interval=TimeUnitValue
  ;

BlinkTimerOutput returns gpiosactuators::BlinkTimerOutput:
  'blink_timer'
  pin=[gpios::DigitalOutputPin|EString]
  '{'
  'during' duration=TimeUnitValue
  'interval' interval=TimeUnitValue
  '}'
  ;

Pwm returns gpiosactuators::Pwm:
  'pwm'
  pin=[gpios::AnalogOutputPin|EString]
  '{'
  'value' value=EInt
  '}'
  ;


TransformTo returns gpiosactuators::TransformTo:
  'pwm_transform'
  pin=[gpios::AnalogOutputPin|EString]
  '{'
  'value' value=EInt
  'duration' duration=TimeUnitValue
  '}'
  ;

WifiConfiguration returns deploymentdevices::WifiConfiguration:
  'ssid'
  ssid=EString
  'with_password'
  password=EString
  ;

Simulator returns deploymentdevices::Simulator:
  {deploymentdevices::Simulator}
  'simulator'
  ;

Raspberry returns deploymentdevices::Raspberry:
  {deploymentdevices::Raspberry}
  'raspberry'
  ('{' wifiConfiguration=WifiConfiguration '}' )?
  ;
```

```
Beaglebone returns deploymentdevices::Beaglebone:
  {deploymentdevices::Beaglebone}
  'beaglebone'
  ('{' wifiConfiguration=WifiConfiguration '}' )?
  ;


ESP8266 returns deploymentdevices::ESP8266:
  {deploymentdevices::ESP8266}
  'esp8266'
  ('{' wifiConfiguration=WifiConfiguration '}' )?
  ;


JavaServer returns servers::JavaServer:
  'java_server'
  '{'
    'network_interface_address' networkInterfaceAddress=EString
    'network_interface_port' networkInterfacePort=EInt
  '}';
```

ejemplos/Dsl.xtext

B.3. Xtend lectura de la gramática DSL mediante ecore y posterior conversión del modulo de comunicaciones, apartado telemetría a su implementación en Java

```
package es.udima.tfm.cesarlaso.generator.java

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import java.util.ArrayList
import org.eclipse.emf.ecore.EAttribute
import org.eclipse.emf.ecore.EClass
import org.eclipse.emf.ecore.EClassifier
import java.util.List
import es.udima.cesarlaso.tfm.communications.impl.
   CommunicationsFactoryImpl
import es.udima.cesarlaso.tfm.communications.Packet
import es.udima.cesarlaso.tfm.communicationstatus.impl.
   StatusServiceImpl
```

```
import es.udima.cesarlaso.tfm.communicationstatus.impl.
   CommunicationstatusFactoryImpl
import es.udima.cesarlaso.tfm.communications.CommunicationsFactory
import es.udima.cesarlaso.tfm.communications.CommunicationsPackage
import es.udima.cesarlaso.tfm.communicationstatus.
   CommunicationstatusFactory
import es.udima.cesarlaso.tfm.communicationstatus.
   CommunicationstatusPackage

class TelemetryGenerator {

  private val Resource resource;
  private val IFileSystemAccess2 fsa;
  private val IGeneratorContext context;

  new(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext
   context) {
    this.resource = resource;
    this.fsa = fsa;
    this.context = context;
  }

  def build() {
    var f = CommunicationstatusFactory.eINSTANCE;
    //new CommunicationstatusFactoryImpl();
    val classes = new ArrayList<EClass>;

    classes.add(f.createPing().eClass);
    classes.add(f.createPong().eClass);

    val writer = buildWriter(classes);
    val reader = buildReader(classes);

    fsa.generateFile("writer.java", writer);
    fsa.generateFile("reader.java", reader);
  }

  def generateCommunications(EClass eclasz) {
    '''
      public void PacketWriter {

        public static void write(final Packet packet, final
   DataOutputStream out) {
          FOR  attr : eclasz.getEAttributes
           ENDFOR
        }

      }

    '''
```

```
}


def buildWriter(List<EClass> classes) {
  '''
    public class TelemetryWriter {

      private final DataOutputStream out;

      public TelemetryWriter(final DataOutputStream out) {
        this.out = out;
      }

       FOR  eclasz : classes
         writeFunc  (eclasz)
        ENDFOR
    }
  '''
}

def buildReader(List<EClass> classes) {
  '''
    import  es .udima.cesarlaso.tfm.communicationstatus.
 CommunicationstatusPackage.package.name .*;

    public class TelemetryReader {

      private final DataInputStream in;

      public TelemetryReader(final DataInputStream in) {
        this.in = in;
      }

      Service readFromStream() throws IOException {
        final int serviceType = in.readInt();
        switch (serviceType) {
           FOR  eclasz : classes
            case  eclasz .classifierID :
                readFuncName  (eclasz)  ();
              break;
            ENDFOR
        }
      }

       FOR  eclasz : classes
         readFunc  (eclasz)
        ENDFOR
    }
  '''
}
```

```
private def writeFuncName(EClass eclasz) {
  '''write eclasz.name '''
}

def writeFunc(EClass eclasz) {
  '''
     IF !eclasz.isAbstract
      public
      ENDIF
     IF eclasz.isAbstract
      private
      ENDIF

    void writeFuncName (eclasz) (final eclasz .name value)
 throws IOException {
       IF !eclasz.isAbstract
        out.writeInt( eclasz .classifierID );
         FOR superType : eclasz.getESuperTypes
            writeFuncName (superType)
          ENDFOR

        ENDIF
       FOR attr : eclasz.getEAttributes
          generateWriterType (attr)
        ENDFOR
    }
  '''
}

def generateWriterType(EAttribute attribute) {
  if (attribute.upperBound != 1) {
    //es un array
    '''
      out.writeInt( attribute .name .length());
      for ( attribute .getEType.name element : attribute .
 name ) {
          writeFuncName (attribute.getEType.eClass) (element);
      }
    '''
  } else {
    '''out. simpleTypeWriter (attribute.getEType) ( attribute .
 name );'''
  }
}

def simpleTypeWriter(EClassifier etype) {
  switch (etype.name) {
    String case "EBoolean": "writeBoolean"
    String case "EInt": "writeInt"
    String case "ELong": "writeLong"
    String case "EString": "writeUTF"
```

```
    default: throw new Exception("Invalid type")
  }
}


private def readFuncName(EClass eclasz) {
  '''read eclasz.name '''
}

def readFunc(EClass eclasz) {
  '''
     eclasz .name  readFuncName (eclasz) () throws IOException
  {
       FOR attr : eclasz.getEAttributes
          generateReaderType (attr)
        ENDFOR

       eclasz .name obj =
        CommunicationstatusPackage.eINSTANCE.
 communicationstatusFactory.create eclasz.name ();

      //CommunicationstatusFactoryImpl.eINSTANCE;
       FOR attr : eclasz.getEAttributes
      obj.set attr ( attr .name );
        ENDFOR
      /*
      return new eclasz .name (
       FOR attr : eclasz.getEAttributes SEPARATOR ", "
         attr .name
        ENDFOR
      );
      */
    }
  '''
}

def generateReaderType(EAttribute attribute) {
  if (attribute.upperBound != 1) {
    //es un array
    '''
      final int attribute .name _size = in.readInt();
      final List< attribute .getEType.name > attribute .name
 = new ArrayList<>( attribute .name _size);
      for (int i=0; i< attribute .name _size; i++) {
          readFuncName (attribute.getEType.eClass) ();
      }
    '''
  } else {
    '''
      final attribute .getEType.name attribute .name = in.
    simpleTypeReader (attribute.getEType) ();
```

```
      '''
    }
  }

  def simpleTypeReader(EClassifier etype) {
    switch (etype.name) {
      String case "EBoolean": "readBoolean"
      String case "EInt": "readInt"
      String case "ELong": "readLong"
      String case "EString": "readUTF"
      default: throw new Exception("Invalid type")
    }
  }

}
```

ejemplos/telemetry_generator.xtend

Apéndice C

Casos de uso - Ejemplos de programas escritos en el lenguaje creado

C.1. Programa adaptación maquinas con monedero electrónico a su versión conectada a internet

Un simple dispositivo para tener control de las monedas que entran en un monedero de pulsos (cada moneda envía una anchura de pulso según su valor). Con este programa conseguimos poder tener una contabilidad de cuanto esta generando cada máquina de forma remota, almacenando en un servidor la fecha y hora de entrada de moneda en cada dispositivo. Permite de forma remota simular la entrada de moneda, permitiendo poder pagar en el dispositivo desde cualquier plataforma externa (por ejemplo móvil)

```
/*
 * Ejemplo programa monedero
 */
IotProyect monedero_remoto
{
  product_code 1
  version 1
  devices {
    simulator,
    esp8266 {
      ssid "monederos" with_password "monederos"
    }
```

```
}

servers {
  java_server {
    network_interface_address"localhost"
    network_interface_port 8080
  }
}

program {
  gpio_aliases {
    gpio_output_digital "status_led" connected_to_pin 1,

    gpio_input_digital "monedero_entrada" connected_to_pin 2,
    gpio_output_digital "monedero_salida" connected_to_pin 3
  }

  remote_identifiers {
    remote_event_identifier "dar_moneda",
    remote_actuator_identifier "moneda_entrante"
  }

  start_state "main"

  states {
    state "main" {
      now {
        change monedero_salida low
      },
      network_connected {
        blink status_led interval 500 Millisecond
      },
      network_disconnected {
        change status_led low
      },
      server_connected {
        change status_led high
      },
      server_disconnected {
        blink status_led interval 250 Millisecond
      },

      remote_event dar_moneda {
        blink_timer status_led {
          during 1 Second
          interval 50 Millisecond
        },
        pulse monedero_salida {
          times 1
          width 50 ms
          space_width 50 ms
```

```
          }
        },

        pulse_input monedero_entrada {
          mode pull_down
          count 1
          width 50
          space_width 50
          {
            remote_actuator moneda_entrante
          }
        }
      }
    }
  }
}
```

ejemplos/ejemplo_dsl_1.iotproyect

C.2. Programa modernización futbolín

Este ejemplo esta basado en un proyecto real, en conjunto con un cliente fabricante de futbolines

El proyecto consistía en un dispositivo IOT. Su primera versión distribuida en Raspberry PI, posteriormente se reescribió a una versión más económica para el módulo ESP8266. En este caso, el uso de un DSL hubiera permitido mantener los modelos en ámbas versiones. El dispositivo conectaba un futbolín con un sistema de gestión remota vía web de administración con las funcionalidades de alarmas, contabilidad, incidencias y junto a una aplicación para realizar torneos online entre jugadores para la plataforma Android.

Este ejemplo intenta emular un esqueleto de lo que podría ser la parte del proyecto IOT y su comunicación con un servidor.

```
/*
 * Ejemplo programa futbolin
 */
IotProyect futbolin
{
  product_code 123
  version 1
  devices {
    simulator,
    raspberry {
```

```
    ssid "cloudbolin" with_password "cloudbolin"
  }
}
servers {
  java_server {
    network_interface_address "localhost"
    network_interface_port 8080
  }
}

program {
  gpio_aliases {
    //Leds de estado
    gpio_output_digital "status_led" connected_to_pin 1,
    gpio_output_digital "error_led" connected_to_pin 2,

    //entradas gol
    gpio_input_digital "goals_left" connected_to_pin 3,
    gpio_input_digital "goals_right" connected_to_pin 4,

    //entradas monederos
    gpio_input_digital "coin_1_euro" connected_to_pin 5,
    gpio_input_digital "coin_50_cents" connected_to_pin 6,

    gpio_output_digital "coin_1_euro_salida" connected_to_pin 1,
    gpio_output_digital "coin_50_cents_salida" connected_to_pin
 1,


    //entradas alarmas
    gpio_input_digital "alarm_door" connected_to_pin 7,
    gpio_input_digital "alarm_coins" connected_to_pin 8,

    gpio_input_analog "humidity_relative" connected_to_pin 9,
    gpio_input_analog "temperature" connected_to_pin 10
  }

  remote_identifiers {
    remote_actuator_identifier "goal_left",
    remote_actuator_identifier "goal_right",
    remote_actuator_identifier "coin_1",
    remote_actuator_identifier "coin_50",

    remote_actuator_identifier "alarm1",
    remote_actuator_identifier "alarm2",

    remote_actuator_identifier "mojado",

    remote_event_identifier "send_money"
  }
```

```
  start_state "test"

  states {
    state "test" {
      now {
        //Arrancando programa....
        blink status_led interval 1000 Millisecond

      },
      repeat_every 10 Minute {
        blink status_led interval 10 Millisecond

      },
      clock 1h 10m 10s {
        blink status_led interval 10 Millisecond
      },

      network_connected {
        blink status_led interval 10 Millisecond
      },
      network_disconnected {
        change status_led low
      },
      server_connected {
        change status_led high
      },
      server_disconnected {
        blink status_led interval 10 Millisecond
      },


      //Dar partida, de forma remota se podra simular una entrada
 de
      //moneda en el sistema, permitiendo asi que los jugadores
      //desde un dispositvo con internet puedan pagar en la
maquina
      remote_event send_money {

        //ejemplo de actuador secuencial
        sequential {
          pulse coin_1_euro_salida {
            times 1
            width 50 ms
            space_width 50 ms
          },
          blink_timer status_led {
            during 1 Second
            interval 10 Millisecond
          }
        }
      },
```

```
      //goles
      interrupted goals_left falling {
        sequential {
          blink status_led interval 50 Millisecond,
          remote_actuator goal_left
        }
      },
      interrupted goals_right falling {
        blink status_led interval 50 Millisecond,
        remote_actuator goal_right
      },

      pulse_input coin_1_euro {
        mode pull_down
        count 1
        width 50
        space_width 50
        {
          blink status_led interval 50 Millisecond
        }
      },

      pulse_input coin_50_cents {
        mode pull_down
        count 1
        width 50
        space_width 50
        {
          blink status_led interval 50 Millisecond
        }
      },

      interrupted alarm_door falling {
        blink status_led interval 50 Millisecond
      },
      interrupted alarm_coins falling {
        blink status_led interval 50 Millisecond
      },

      analog_read_on_high humidity_relative
      every 1 loops
      value_higher_than 10
      {
        blink status_led interval 50 Millisecond,
        remote_actuator mojado
      }

    }
  }
```

```
  }
}
```

ejemplos/ejemplo_dsl_2.iotproyect

C.3. Programa modernización ascensor de hotel sin pantalla gráfica

Ejemplo de IOT conectado a un ascensor de un hotel. Utilizado para modernizar un ascensor antiguo sin modificar nada del sistema del ascensor original, simplemente leyendo las salidas generadas por el sistema.

Este ejemplo está basado en un estudio de proyecto que realizamos para la adaptación de ascensores sin pantalla gráfica a una adaptación a pantalla con visualización, desarrollando un prototipo en Raspberry PI.

El sistema muestra las señales recibidas por el dispositivo electrónico del panel del ascensor en un pantalla gráfica. La implementación corre mediante comandos que llaman al interfaz gráfico X-Window, mostrando imágenes y vídeos según la planta en la que se encuentre o eventos de tiempo determinados.

```
/*
 * Ejemplo programa ascensor
 */
IotProyect ascensor_hotel
{
  product_code 2
  version 1
  devices {
    simulator,
    //en el laboratorio probamos tambien con raspberry
    raspberry {
      ssid "prueba" with_password "prueba"
    },
    //en produccion, usaremos la beaglebone ira via ethernet sin
   wifi
    beaglebone
  }

  servers {
    java_server {
      network_interface_address "localhost"
      network_interface_port 8080
    }
```

```
}

program {
  gpio_aliases {
    gpio_input_digital "piso1" connected_to_pin 1,
    gpio_input_digital "piso2" connected_to_pin 2,
    gpio_input_digital "piso3" connected_to_pin 3,
    gpio_input_digital "piso4" connected_to_pin 4,
    gpio_input_digital "piso5" connected_to_pin 5,

    gpio_input_digital "subiendo" connected_to_pin 6,
    gpio_input_digital "bajando" connected_to_pin 7,
    gpio_input_digital "parado" connected_to_pin 8,

    gpio_input_digital "reparando" connected_to_pin 9
  }

  remote_identifiers {
    remote_event_identifier "mensaje_nuevo"
  }

  files {
    file_path "fondo_normal" with_path "/imagenes/fondo.png",
    file_path "fondo_reparando" with_path "/imagenes/
fondo_reparando.png",

    file_path "imagen_flecha_arriba" with_path "/imagenes/
flecha_arriba.png",
    file_path "imagen_flecha_abajo" with_path "/imagenes/
flecha_abajo.png",
    file_path "imagen_flecha_parado" with_path "/imagenes/
flecha_parado.png",

    file_path "imagen_piso1" with_path "/imagenes/piso1.png",
    file_path "imagen_piso2" with_path "/imagenes/piso2.png",
    file_path "imagen_piso3" with_path "/imagenes/piso3.png",
    file_path "imagen_piso4" with_path "/imagenes/piso4.png",
    file_path "imagen_piso5" with_path "/imagenes/piso5.png",

    file_path "video_diario" with_path "/imagenes/video.mp4",
    file_path "video_fin_semana" with_path "/imagenes/
video_fin_semana.mp4"
  }

  start_state "inicio"

  states {
    state "inicio" {

      //Cada 10 minutos se comprueban actualizaciones de los
archivos multimedia
```

```
      //El sistema es capaz de hacer un last-modified y/o etag
      //y no volver a descargar si no es necesario
      repeat_every 10 Minute {
        download "https://SERVER_HOST/images/fondo_normal.png" to
 fondo_normal,
        download "https://SERVER_HOST/images/fondo_reparando.png"
 to fondo_reparando,

        download "https://SERVER_HOST/images/flecha_arriba.png"
to imagen_flecha_arriba,
        download "https://SERVER_HOST/images/flecha_abajo.png" to
 imagen_flecha_abajo,
        download "https://SERVER_HOST/images/flecha_parado.png"
to imagen_flecha_parado,
        download "https://SERVER_HOST/images/piso1.png" to
imagen_piso1,
        download "https://SERVER_HOST/images/piso2.png" to
imagen_piso2,
        download "https://SERVER_HOST/images/piso3.png" to
imagen_piso3,
        download "https://SERVER_HOST/images/piso4.png" to
imagen_piso4,
        download "https://SERVER_HOST/images/piso5.png" to
imagen_piso5,

        download "https://SERVER_HOST/videos/diario.png" to
video_diario,
        download "https://SERVER_HOST/videos/fin_semana.png" to
video_fin_semana
      },

      now {
        background_image fondo_normal none,
        play_video video_diario {
          x 0
          y 0
          z 1
          color 0
          resize auto
        }
      },

      //fin semana a partir de las 8 del sabado
      cron "* 8 * * 6,0" {
        play_video video_fin_semana {
          x 0
          y 0
          z 1
          color 0
          resize auto
        }
```

```
      },

      //todos los dias reproducir el video
      cron "* 8 * * 1,2,3,4,5" {
        play_video video_diario {
          x 0
          y 0
          z 1
          color 0
          resize auto
        }
      },

      //si hubiera cambiado la imagen de fondo, se actualiza,
      //el resto como el ascensor se mueve no es necesario
repintar
      repeat_every 1 Minute {
        background_image fondo_normal none
      },

      interrupted piso1 falling {
        draw_image imagen_piso1 {
          x 100
          y 100
          z 0
          color 0
          resize auto
        }
      }
      ,
      interrupted subiendo falling {
        draw_image imagen_flecha_arriba {
          x 10
          y 10
          z 1
          color 0
          resize none
        }
      },

      interrupted bajando falling {
        draw_image imagen_flecha_abajo {
          x 10
          y 10
          z 1
          color 0
          resize none
        }
      },

      interrupted parado falling {
```

```
    draw_image imagen_flecha_parado {
      x 10
      y 10
      z 1
      color 0
      resize none
    }
  },

  interrupted piso1 falling {
    draw_image imagen_piso1 {
      x 10
      y 10
      z 2
      color 0
      resize none
    }
  },
  interrupted piso2 falling {
    draw_image imagen_piso2 {
      x 10
      y 10
      z 2
      color 0
      resize none
    }
  },
  interrupted piso3 falling {
    draw_image imagen_piso3 {
      x 10
      y 10
      z 2
      color 0
      resize none
    }
  },
  interrupted piso4 falling {
    draw_image imagen_piso4 {
      x 10
      y 10
      z 2
      color 0
      resize none
    }
  },
  interrupted piso5 falling {
    draw_image imagen_piso5 {
      x 10
      y 10
      z 2
      color 0
```

```
            resize none
          }
        },

        interrupted reparando falling {
          change_state estado_reparando
        }
      },

      state "estado_reparando" {
        now {
          background_image fondo_reparando none
        },

        interrupted reparando rising {
          change_state inicio
        }
      }
    }

  }
}
```

ejemplos/ejemplo_dsl_3.iotproyect

Lista de abreviaturas

BPMN Business Process Model and Notation. 21

CIM Computation-Independent Model. 21

CSS Cascading Style Sheets. 81, 84

CWM Common Warehouse Metamodel. 21

DSL Domain specific language. 11–17, 22, 24, 26, 28–33, 41, 47, 66, 70, 71, 74, 77, 81–85, 114

ECore ECore Eclipse Modeling framework. 4, 7–12, 17, 22–24, 29, 30, 34, 37, 45, 49, 51–57, 59–62, 65–67, 69–72, 77–80, 83

EDOC Enterprise Distributed Object Computing. 21

EMF Eclipse Model Framework. 11, 12, 22, 45, 49, 67, 70, 82, 83

ESA European Space Agency. 25

GMF Graphical modeling framework. 70

GMP Graphical modeling proyect. 69

GPIO General purpose input output. 17, 29–34, 47, 67, 74

HTML HyperText Markup Language. 81, 84

I2C Inter Integrated Circuit. 33

IDE Integrated Development Environment. 22, 25, 51, 57, 79, 81, 83

IO Input / Output. 83

IOT Internet of things. 11–13, 15–17, 28–32, 34, 37, 40, 41, 47, 76, 84, 114, 118

JSON JavaScript Object Notation. 81

JVM Java Virtual Machine. 25, 28, 77, 85

LED Light Emitting Diode. 31, 47, 67

MDA Model Driven Architecture. 20–22

MDE Model Driven Enginering. 11–17, 19, 80

MDSD Model Driven Software Development. 11, 12, 14, 15, 19

MOF Meta-Object Facility. 20

OCL Constraint Language Specification. 56

OMG Object Management Group. 20

PIM Platform-Independent Model. 16, 17, 21

PSM Platform-Specific Model. 16, 17, 21, 22

QVT Query/View/Transformation. 17, 21

RPC Remote Procedure Call. 17

RS232 Recommended Standard 232. 33

Sirius Eclipse Sirius. 30

SPEM Software Process Engineering Metamodel. 21

SQL Structured Query Language. 16, 81, 84

TCP Transmission Control Protocol. 37, 38

UML Unified Modeling Language. 20–22, 31, 47

XMI XML Metadata Interchange. 21

XML Extensible Markup Language. 7, 22, 24, 81, 84

Xtext Xtext framework for modeling languages. 24, 30, 47, 71, 74, 77, 80, 83

Glosario

bytecode Código intermedio más abstracto que el código máquina. Habitualmente es tratado como un archivo binario que contiene un programa ejecutable similar a un módulo objeto, que es un archivo binario producido por el compilador cuyo contenido es el código objeto o código máquina . 77

framework Conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar. 20, 22, 56, 82

housekeeping Comprobación de constantes de salud del sistema. 17, 40

keepalive Paquete de datos utilizado para mantener una conexión con estado activa. 38

Main loop Hilo principal de ejecución de eventos. Utilizado normalmente para eventos controlados por el usuario, tales como entradas/salidas, o interfaz gráfico. 33

metadatos Son datos que describen otros datos. En general, un grupo de metadatos se refiere a un grupo de datos que describen el contenido informativo de un objeto al que se denomina recurso. 51, 70

metamodelo Modelo que define el lenguaje para expresar un modelo. 30, 47, 60, 66, 70

payload Contenido de un paquete de datos. 37

plugin Es una aplicación que se relaciona con otra para agregarle una función nueva y generalmente muy específica. Esta aplicación adicional es ejecutada por la aplicación principal e interactúan por medio de un API.. 56

reflection Propiedad por la cual el propio lenguaje puede inspeccionar y manipular clases e interfaces (así como sus métodos y campos) en tiempo de ejecución, sin conocer a priori (en tiempo de compilación) los tipos y/o nombres de las clases específicas con las que está trabajando. 51

runtime Código ejecutado en tiempo de ejecución. 49, 51, 67, 77, 80, 81

script Un programa usualmente simple, que por lo regular se almacena en un archivo de texto plano. Los scripts son por lo general siempre interpretados, pero no todo programa interpretado es considerado un script.. 20

timestamp Marca de tiempo, por normal general segundos desde época Unix. 37

wizard Asistente paso a paso utilizado en la creación de elementos. 57

zip Formato de compresión de archivos sin perdida.. 30