UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Máster en Ingeniería Informática

Trabajo Fin de Máster

Aproximación MDD al desarrollo de Sistemas Embebidos para *Web Of Things*

Jorge Oíz Acosta

Febrero, 2015

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Máster en Ingeniería Informática

Trabajo Fin de Máster

Aproximación MDD al desarrollo de Sistemas Embebidos para *Web Of Things*

Autor: Jorge Oíz Acosta
Fdo:

Directores: Marino Linaje Trigueros
Cristina Vicente Chicote
Fdo:

<u>Tribunal Calificador</u>

Presidente:
Fdo:

Secretario:
Fdo:

Vocal:
Fdo:

CALIFICACIÓN:
FECHA:

A mis padres, por dedicar toda su vida a que conseguiría cualquier objetivo que me he propuesto.

A esa persona que aparece sin avisar en tu vida y te hace que todo merezca la pena.

A mis amigos, por pasar mis mejores momentos junto a ellos.

A mis directores del proyecto, Marino y Cristina, por motivarme y ayudarme a conseguir realizar este proyecto.

Tabla de contenidos

Listado de figuras 3
Summary 5
Capítulo 1. Introduction 6
1.1. Tools used in the project 9
1.2. Project Objectives 10
1.3. Project development phases 10
Capítulo 2. Estado de la técnica 12
2.1. Prototipado de aplicaciones WoT basadas en OpenPicus/Flyport 12
2.2. Desarrollo de Software Dirigido por Modelos 13
2.2.1. Eclipse Modeling Framework (EMF) 15
2.2.2. Graphical Modeling Framework (GMF) 15
2.2.3. El lenguaje XPAND 15
2.3. Trabajos relacionados 16
Capítulo 3. El entorno EMDSD 19
3.1. Retos del desarrollo 20
3.2. El lenguaje de modelado 22
3.2.1. Modelado de la estructura del sistema (SystemStructure) 23
3.2.2. Modelado de la lógica del sistema (SystemBehaviour) 26
3.2.3. Restricciones OCL 29
3.3. Herramientas de modelado 30
3.3.1. Modelado y uso del repositorio 30
3.3.2. Modelado de la estructura y el comportamiento del sistema 31
3.4. Motor de generación de código 33
3.5. Descarga y ubicación de las librerías del dispositivo 34
Capítulo 4. Manual de Instalación y Uso 39
4.1. Consideraciones Iniciales 39
4.2. Instalación del entorno EMDSD 41
4.3. Uso del entorno EMDSD 44
4.3.1. Diseño y validación de los modelos gráficos 44

4.3.2. Motor de generación de código ... 54
4.4. Ejemplo completo para OpenPicus ... 55
Capítulo 5. Conclusions and future work ... 60
5.1. Conclusions ... 60
5.2. Future work lines ... 62
Bibliography ... 63
Annex I. Implementation of EMDSD enviroment ... 67
I.1. Abstract syntax and EMF models editor ... 67
I.2. Graphic syntax and GMF model editor ... 72
I.3. Implementation of the code generation engine ... 80
Annex II. Restrictions OCL ... 84
Annex III. Graphic Editor Extensions ... 92
Annex IV. M2T Transformation GroveNest ... 108

Listado de figuras

Figure 1: Conceptual diagram of the development process with EMDSD. 8
Figura 2: Flyport WiFi Dispositivo OpenPicus. 12
Figura 3: Algunos de los conceptos básicos empleados en el DSDM. 14
Figura 4: Proceso de desarrollo propuesto basado en el uso del entorno EMDSD. 22
Figura 5: Parte del meta-modelo que describe la estructura del sistema (SystemStructure). 24
Figura 6: Parte del meta-modelo que describe el comportamiento del sistema (SystemBehaviour). 28
Figura 7: Modelo de repositorio desarrollado como parte del entorno EMDSD. 31
Figura 8: Aspecto del editor gráfico que permite modelar la estructura de los sistemas embebidos. 32
Figura 9: Wiki oficial del dispositivo 35
Figura 10: Algunos de los sensores/actuadores compatibles con OpenPicus. 35
Figura 11: Sensor temperatura dispositivo 36
Figura 12: Pantalla principal descarga automatizada de Librerías del dispositivo 37
Figura 13: Opción descarga automatizada de Librerías del dispositivo 37
Figura 14: Opción 2 descarga automatizada de Librerías del dispositivo 38
Figura 15: Configuración de memoria del Eclipse donde se ejecutará el editor gráfico. 40
Figura 16: Ubicación del modelo serializado dentro del proyecto XPAND. 40
Figura 17: Configuración del motor de generación de código en el workspace. 41
Figura 18: Ficheros incluidos en el CD que se adjunta a la memoria del TFM. 41
Figura 19: Selección del espacio de trabajo (workspace) para el entorno EMDSD. 42
Figura 20: Proceso de Import de los proyectos en el Workspace del entorno EMDSD 42
Figura 21: Pantalla inicial del proceso por lotes que automatiza la descarga de librerías OpenPicus. 44
Figura 22: Creación de un proyecto Java en Eclipse. 45
Figura 23: Creación de un modelo gráfico. 45
Figura 24: Selección repositorio Entorno EMDSD 46
Figura 25: Aspecto gráfico componentes mínimos programación dispositivo en Entorno EMDSD 47
Figura 26: Propiedades de los *Device*. 47
Figura 27: Aspecto gráfico de un dispositivo de tipo *GROVENEST*. 48
Figura 28: Propiedades de los *SystemElement*. 48
Figura 29: Aspecto gráfico de un sensor de tipo *GroveTemperature*. 48
Figura 30: Aspecto del *SystemStructure* tras unir el dispositivo con el sensor. 49
Figura 31: Propiedades variable ElementDeclaration Entorno gráfico EMDSD 50
Figura 32: Propiedades variable PrimitiveDeclaration Entorno gráfico EMDSD 50
Figura 33: Elementos de programación en Entorno gráfico EMDSD 51
Figura 34: Elementos de programación en Entorno gráfico EMDSD 53

Figura 35: Validación desde el elemento root en Entorno gráfico EMDSD 53
Figura 36: Estructura del proyecto XPAND. Carpeta donde se almacenan los modelos XMI de partida.54
Figura 37: Ejecución del motor de generación de código. 54
Figura 38: Estructura del proyecto XPAND. Carpeta donde se almacenan los ficheros de salida. 55
Figura 39: Entorno gráfico EMDSD – Diseño del SystemBehaviour – Declaración variables. 56
Figura 40. Entorno gráfico EMDSD – Diseño del SystemBehaviour –Workflow de tareas. 57
Figura 41: Entorno gráfico EMDSD – Modelo Serializable .xmi generado. 57
Figura 42: Creación de un proyecto OpenPicus GroveNest Web Server en IDE OpenPicus Flyport ...58
Figura 43: Estructura de un proyecto OpenPicus GroveNest en IDE OpenPicus Flyport. 59
Figura 44: Directorio “Web pages” en un proyecto OpenPicus GroveNest Web Server. 59
Figure 45: Creating an EMF Generator Model from meta-model. 68
Figure 46: Selecting the type of meta- model to create the EMF Generator Model. 69
Figure 47: Configuration parameters EMF Generator Model 69
Figure 48: EMF generation of Java code associated with the model editor in tree. 70
Figure 49: Repository model creation for EMDSD environment. 71
Figure 50: Definition Repository of EMDSD enviroment 71
Figure 51: Detailed modeling of the two devices included in the previous repository. 72
Figure 52: Creation *.gmfgraph* model associated with the EMDSD graphic editor 73
Figure 53: File *.gmfgraph* associated with the EMDSD enviroment graphic models editor. 74
Figure 54: Created *.gmftool* models attached in EMDSD enviroment graphic editor. 74
Figure 55: File *.gmftool* attached in EMDSD enviroment graphic editor. 75
Figure 56: Created *.gmfmap* model linked to the EMDSD enviroment graphic editor. 75
Figure 57: File *.gmfmap* associated with the EMDSD environment graphic editor 76
Figure 58: Configuration file *.gmfgen* extension in our EMDSD environment 76
Figure 59: File *.gmfgen* associated in EMDSD enviroment graphic models editor. 77
Figure 60: Configuration element Metamodel *.diagram* in file *.gmfgen*. 78
Figure 61: SystemEditPart element configuration file *.gmfgen* the section Diagram. 78
Figure 62: SystemEditPart element configuration file *.gmfgen* the section Providers. 78
Figure 63: GMF generation of Java code associated with the graphic model editor. 79
Figure 64: Eclipse memory configuration where the graphic editor is executed. 80
Figure 65: Directory hierarchy generation code of project *EMDSD.project* 81

Summary

Web of Things (WoT) describes an environment in which the objects are part of an interconnected network and are accessible only through the Internet using Web standards. To that physical objects can expose your identity and digital capabilities to other objects and people via the Internet is necessary for these objects incorporate some form of embedded with Internet connectivity system.

Despite the homogeneity of the communications protocols on which they are implemented (eg, HTTP , FTP, ...), the development of embedded systems of this kind is not easy, as there are different hardware platforms and programming languages that can be often require low-level coding (hardware).

The MDD techniques allow theoretically correct or reduce this problem, but there are two problems for use today: (1) lack of graphical modeling languages in order to design the structure and logic of embedded systems easily and intuitively; and (2) lack of code generation engine capable of obtaining valid implementations for different platforms, as they use a variety of programming languages (C, Java, Javascript, Wiring, Processing, etc.)

The solution adopted in this paper to address these challenges is based on a process Model Driven Software Development (DSDM) [1], supported by a set of tools, thanks to which the design and implementation of embedded systems is simplified to integrate objects in the WoT through the systematic use of models and model transformations .

Capítulo 1. Introduction

The term *Internet of Things (IoT)* represents the set of defined physical objects with a unique identifier that can be accessed through different protocols on the network layer / communication provided by the Internet. Existing communication protocols in the area of IoT are heterogeneous and involve a variety of protocols (TCP, IP, UDP, CoAP, etc.), often affecting various layers of the OSI communication architecture. Such solutions define a wide domain, which makes use of advantages as engines automatic code generation generic enough (able to generate implementations for several existing solutions) to be useful.

In this sense, the WoT be seen as a subdomain within IoT providing greater homogeneity for the integration of physical and virtual / digital objects. Thus, more specifically for the purpose of this project mode, the WoT concept represents a subdomain of IoT whose description and development could be capable of being systematized.

However, development of applications *Web of Things* (WoT) has a number of barriers to entry related mainly to the heterogeneity of the programming languages used and the need to acquire a number of technical specialized. The hardware [2] modular systems facilitate hardware development interconnections between sensors / actuators and microprocessor. However, it is not overly software facilitates the implementation of such systems, which hampers their development knowledge especially among people with at least basic programming skills.

The objective of this Master's Thesis (TFM) is trying to lower the entry barriers existing in the development of applications WoT by systematizing and simplifying the designs through the use of models and code generation engine capable of generating the software corresponding to the designs modeled for different platforms

To do this, he will use an approach *Model Driven Software Development* (DSMD) [1] based on: (1) the specification of a modeling language allowing designing the structure and logic of embedded systems that we use to develop WoT applications. The abstract syntax of this language is defined by a meta-model and concrete syntax is graphical type (as opposed to traditional textual syntax generally more complex and less intuitive); and (2) the implementation of a model to text conversion (*Model to Text, M2T*) which will produce, from graphical models designed with the previous language, the code to be run on the embedded system chosen as target platform.

Currently, DSDM is one of the paradigms of popular software development in the field of Software Engineering. The technologies developed around this new paradigm offers a promising solution to overcome the complexity of programming languages third generation, enabling describe designers more easily and intuitively increasingly complex systems through the use of concepts their application domains [3]. DSDM aims to raise the level of abstraction at which systems are specified, giving developers more high level primitives that allow you to work independently of the platform and programming languages of low level used in the final implementation of its systems.

DSDM has been applied successfully in some domains related to the project, and embedded systems design [4], [5], [6], or sensor networks [7]. Focusing on the domain of embedded systems, you may find multiple references that point to a growing interest of the community in this new paradigm of software development [8 [8], [9]. According to [10], the idea is to get all the objects of our daily lives are connected through Web standards at all times and anywhere. In this line, this project contributes to the modeling and implementation of embedded applications for WoT field.

The EMDSD environment (*Embedded Model-Driven Software Development*) developed as part of this TFM, it offers an approximation to software development for embedded systems using a model-driven approach. This environment provides a set of tools to: (1) graphically model the structure and behavior of embedded systems for connecting a number of sensors and actuators to the Internet; and (2) a set of motors to generate, from the previous graphical models, implementations for various platforms. Although the current version of the environment only supports code generation for OpenPicus [11], platform, it would be easy to incorporate new generation engines for other platforms. Figure 1 shows, conceptually, the development process of this TFM with EMDSD.

OpenPicus [11], resultaría sencillo incorporar nuevos motores de generación para otras plataformas. La Figure 1 muestra, de manera conceptual, el proceso de desarrollo con EMDSD.

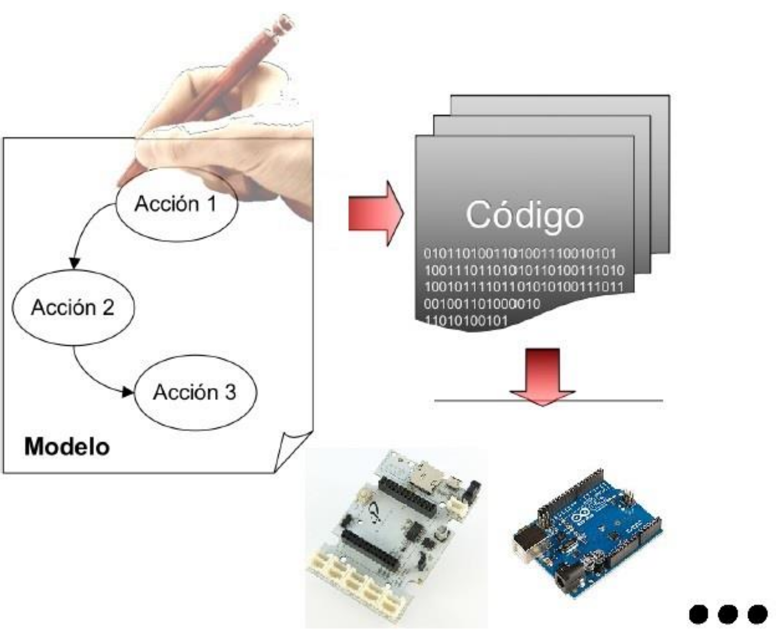


Figure 1: Conceptual diagram of the development process with EMDSD.

Throughout this report we have classified a number of sections covering many areas of programming for this project. In section 2 "State of Technique" tour the prototyping of these applications, the development environment EMDSD chosen for the model and text transformation carried out within the graphical environment is made. Finally the work related to the development of our project is told.

In section 3 "The EMDSD environment" is a tour for the challenges that have led to the development of this project, the language used for the meta-model and meta-model end obtained where is the concept of repository and tools that have helped us to supplement this meta- model through a series of restrictions and code generation with downloading libraries implemented through a process that provides very useful in this project.

Section 4 "Installation and Use" tells the initial considerations to use the project developed, with the two essential manuals: installation manual and use EMDSD environment. Finally, we develop a complete example as a demonstration of the resulting environment programmed device OpenPicus.

The last section 5 "Conclusions and future work" tells the conclusions we have reached upon completion of the development of this project along the lines that we should follow in a future version of the project for extending the base that we have built.

1.1. Tools used in the project

To carry out the implementation of the EMDSD environment has opted for the support offered by EMP (*Eclipse Modeling Project*) [12]. EMP provides a complete set of tools related to DSDM. EMP offers, among other tools as EMF (*Eclipse Modeling Framework*) [13], for the definition of models and meta- models based on a subset of the standard MOF (*Meta Object Facility*) [14], defined by the OMG (*Object Management Group*); OCLInEcore, an implementation of the OMG standard also known as OCL (*Object Constraint Language*) [15], which allows the definition of restrictions on EMF based models; GMF (***Graphical Modeling Framework***) [16], which provides a set of facilities for building graphical editors based models EMF meta- models; or more languages EMF transformation models or in other models (*Model- To- Model, M2M*) or text / code (*Model- To- Text , M2T*) .

The fact of using embedded systems as execution platform software generated from EMDSD environment due to the following reasons:

1) Systems are small and inexpensive.

2) They are extensible platforms and have their own programming tools.

3) Allow a large number of possibilities for the design of both simple and complex systems.

4) Its use is widespread in universities, both teaching and research level. There are also numerous initiatives such as, for example, [17] or [18], facilitating its use for people with little technical knowledge, having associated with many code examples available online.

5) They are reference platforms in the Web of Things field, having been used in several previous research work as described in [19].

1.2. Project Objectives

The objectives of this TFM are:

- Study of the tools included in EMP and serve as support for the development of EMDSD environment.

- Acquisition of technical knowledge necessary for understanding the languages used to program OpenPicus and Arduino [20] embedded systems using modular TinkerKit [21] or Grove [22] type systems.

- Enhancing the configuration of embedded system modeling chosen based on the paradigm in the DSDM environment scheduled EMDSD support mechanisms to enable more complex control flow.

- Developing necessary to support the processes of graphical modeling of complex applications for embedded systems and automatic code generation tools by M2T transformations.

- Obtaining results and draw conclusions on the developed tool, analysis of their impact in the field of WoT and approach possible improvements and future research.

1.3. Project development phases

The development of this TFM has been carried out at the stages that are summarized below:

1. ***Description of the problem.*** Understanding and description of the objectives of TFM on the basis of the constraints identified in the current processes for embedded software development in the field of WoT applications.
2. ***Review of the fundamental concepts and tools related to the DSDM.*** Reminder the concepts, processes and support tools available on the Eclipse platform from the knowledge acquired in the course on "Model Driven Software Development" (DSDM) taught at the Polytechnic School of Cáceres (EPCC) in University Extremadura (UEX) on September 2012.
3. ***Analysis of a complete example OpenPicus developed.*** Example developed on the platform OpenPicus, made as part of the project for the course Information Systems of Master in Computer Engineering, taught at the EPCC of UEx.

4. ***Development of a modeling language for specifying the structure and behavior of embedded systems independently of the final implementation platform.*** As noted above, to define the abstract syntax of this language meta-model based on EMF and a set of syntactic constraints described with OCLInEcore be used.

5. ***Development of a model editor, based on the above language, which will equip you with a concrete syntax graph type.*** This editor will enable designers to model and validate the syntactic correctness of their systems easily and intuitively.

6. ***Implementation of a code generation engine*** that it obtains, from the models developed with the previous graphic editor, the code for the selected target platform.

7. ***Obtaining results and analysis of conclusions.***

Capítulo 2. Estado de la técnica

Este capítulo introduce los conceptos básicos relacionados con el prototipado de sistemas WoT basados en la plataforma OpenPicus/Flyport, utilizada en el presente Proyecto, así como los relacionados con el DSDM. También se describen algunos de los trabajos previos relacionados con el Proyecto.

2.1. Prototipado de aplicaciones WoT basadas en OpenPicus/Flyport

La plataforma utilizada en este TFM para implementar aplicaciones embebidas WoT ha sido OpenPicus/Flyport.

Openpicus es una empresa italiana que creó un dispositivo inteligente basado en la plataforma Arduino, probablemente la más conocida a nivel de mercado. Este dispositivo se puede programar para que interactúe de forma "inteligente" con cualquier objeto de nuestra vida cotidiana como, por ejemplo, con un sistema de encendido/apagado de luces, un controlador que regule la presión del agua o un sensor que mida la temperatura ambiente.

Cabe destacar dentro de este dispositivo el elemento *Flyport (*véase Figura 2*)*. Es un módulo programable con conectividad WiFi 802.11g integrada y nos proporciona la manera más fácil de conectar sensores y dispositivos a Internet. Este elemento contiene un servidor web potente y personalizable a bordo que se puede utilizar como interfaz de usuario para sus sistemas recogiendo datos de los sensores analógicos y digitales, enviar dichos datos a Internet o de forma remota controlar otros dispositivos. El *Flyport* puede actuar como punto de acceso (limitado a 1 sólo cliente) o como infraestructura proporcionando conexión inalámbrica.

Figura 2: Flyport WiFi Dispositivo OpenPicus.

Flyport WiFi también cuenta con un Web Server incrustado que puede albergar archivos HTML. Esto significa que se puede acceder fácilmente a la información como las lecturas de los sensores utilizando una página web interna. El Web Server soporta Javascript y Ajax, por lo que las páginas web se pueden mostrar contenido dinámico para los usuarios.

En conclusión, este módulo proporciona datos en tiempo real que se pueden visualizar y/o actualizar desde un navegador web estándar, incluso en Smartphones o Tablets, porque Flyport soporta páginas web dinámicas a través de Internet, en un tamaño reducido (35*48*15mm), a baja potencia y de bajo coste.

Las principales razones que nos llevaron a elegir OpenPicus como plataforma para desarrollar nuestros prototipos fueron las siguientes:

- ✓ Cuenta con un servidor web integrado (*Flyport*) que permite conectar y controlar el dispositivo a través de la Web mediante el estándar TCP/IP.
- ✓ Cuenta con un entorno integrado de desarrollo (*Integrated Developpment Environment*, IDE) de libre distribución.
- ✓ El lenguaje de programación con el que se implementan las aplicaciones basadas en esta plataforma es C: un lenguaje sencillo y ampliamente difundido.
- ✓ Cuenta con una amplia gama de librerías de programación que se pueden descargar fácilmente a través de la wiki oficial de OpenPicus [23]. Estas librerías suelen incluir un ejemplo sencillo de la programación de cada sensor.
- ✓ Permite integrar sensores y actuadores modulares usando el sistema Grove creado por Seeduino y extendido a otras plataformas de prototipado. La gran variedad de sensores y actuadores disponibles permiten implementar prototipos similares con distintas implementaciones. Además, son compatibles con Flyport, lo que permite acceder/modificar su estado a través de una conexión Web.

2.2. Desarrollo de Software Dirigido por Modelos

Desde el lanzamiento de la iniciativa MDA (*Model-Driven Architecture*) por parte del OMG en el año 2000, el interés por el DSDM es cada vez mayor. El uso sistemático de modelos en las diferentes etapas que conforman el ciclo de vida del software se ha convertido en la base de la denominada Ingeniería del Software Dirigida por Modelos (MDE, de las siglas en inglés *Model-Driven Engineering*).

El uso de modelos permite elevar el nivel de abstracción de las especificaciones y aumentar el grado de automatización de los procesos de desarrollo de software, mejorando diferentes aspectos relacionados con su calidad y mantenimiento. Si el

DSDM alcanza el éxito esperado, los procesos de desarrollo basados en modelos se convertirán en algo tan habitual en la industria del software como lo son en la actualidad los procesos basados en objetos.

En la Figura 3 se ilustran algunos de los conceptos fundamentales entorno a los que gira el DSDM. Como muestra la figura, estos conceptos se organizan en forma de pirámide de cuatro niveles. En el nivel M0 se encuentra el código final de las aplicaciones que, mediante el uso de una o más transformaciones, conseguiremos a partir de los modelos definidos en el nivel M1. A partir de un mismo modelo es posible definir varias transformaciones que generen otros modelos (representaciones de la misma información utilizando otra representación de mayor, igual o menor nivel de abstracción) o varias implementaciones en distintos lenguajes de programación.

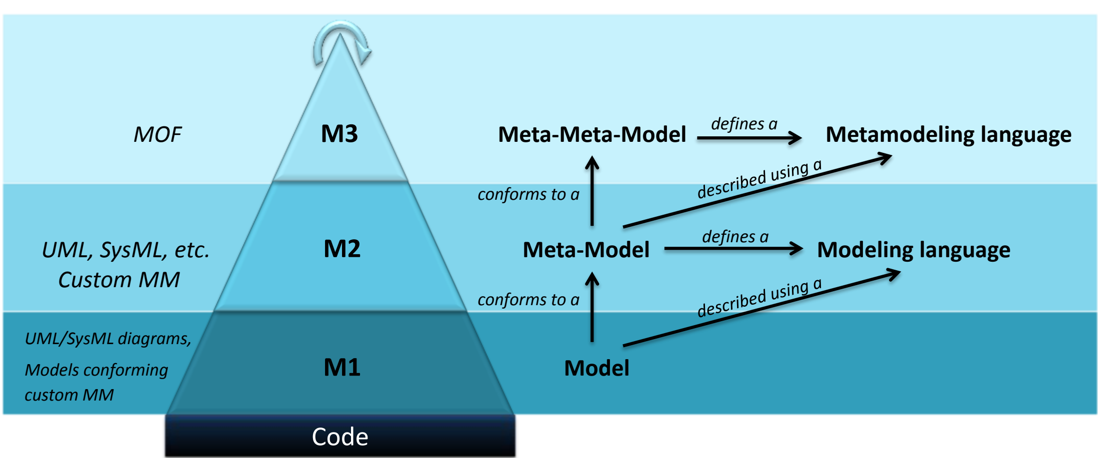


Figura 3: Algunos de los conceptos básicos empleados en el DSDM.

Cada modelo en el nivel M1 será conforme a uno de los meta-modelos ubicados en el nivel M2. Cada meta-modelo define la sintaxis abstracta de un lenguaje de modelado. Así, por ejemplo, el meta-modelo de UML (*Unified Modeling Language*) define la sintaxis abstracta de este lenguaje de modelado estandarizado por el OMG y gracias al cual es posible definir modelos de clases, de actividad, de estados, etc. Cada meta-modelo contiene los conceptos de modelado del lenguaje ("palabras") y las relaciones sintácticas existentes entre ellos (reglas que indican cómo combinar las "palabras" del lenguaje para formar "sentencias" sintácticamente correctas). Si a nivel M2 no encontramos un meta-modelo adecuado para especificar un determinado tipo de sistemas, siempre podemos definir nuestro propio meta-modelo incluyendo en él los conceptos de modelado necesarios.

Los meta-modelos ubicados en el nivel M2 pueden verse a su vez como modelos definidos conforme a un meta-meta-modelo definido a nivel M3. El meta-meta-modelo estándar y en base al cual están definidos la mayoría de los meta-modelos es MOF (*Meta Object Facility*), definido por el OMG. MOF es un lenguaje de meta-

modelado que define los conceptos básicos necesarios para definir meta-modelos y las reglas sintácticas que deben guiar su construcción. MOF es un lenguaje reflexivo, definido conforme a él mismo, lo que permite no considerar niveles adicionales en la pirámide mostrada en la Figura 3.

A continuación se describen brevemente algunas de las herramientas incluidas en EMP para dar soporte a los conceptos y procesos básicos del DSDM.

2.2.1. Eclipse Modeling Framework (EMF)

EMF [24] es un plug-in de Eclipse incluido en EMP. EMF proporciona un marco de modelado y de generación de código para la definición de modelos y meta-modelos basados en EMOF (*Essential MOF*). EMOF implementa un subconjunto del estándar de meta-modelado MOF, definido por el OMG.

Los modelos y meta-modelos definidos con EMF se almacenan en el formato estandarizado por la OMG conocido como XMI (*XML Metadata Interchange*). A partir de cada meta-modelo definido con EMF, es posible generar una implementación Java de un editor con el que es posible crear, editar y validar la corrección sintáctica de los modelos definidos conforme a dicho meta-modelo.

EMF puede considerarse el núcleo de EMP, ya que la mayoría del resto de herramientas incluidas en este proyecto de Eclipse se han definido entorno a él. Así, EMF proporciona la base para la interoperabilidad entre estas herramientas, entre las que encontramos: utilidades para la creación de editores gráficos y textuales de modelos, lenguajes para implementar transformaciones M2M y M2T, etc.

2.2.2. Graphical Modeling Framework (GMF)

GMF [16] es otro de los plug-ins incluidos en EMP. GMF proporciona un entorno de modelado y una infraestructura de ejecución para crear y ejecutar editores gráficos de modelos basados EMF. GMF permite modelar la sintaxis gráfica concreta asociada a los conceptos incluidos en cada meta-modelo EMF. Para ello, GMF hace uso también del plug-in llamado GEF (*Graphical Edition Framework*) [25], en el que se incluyen una serie de primitivas básicas de dibujo para crear las formas asociadas a cada elemento del lenguaje (cajas, círculos, flechas, etc.).

2.2.3. El lenguaje XPAND

El lenguaje XPAND [26] fue concebido como un lenguaje con un vocabulario reducido (aunque suficiente) para poder generar código a partir de una estructura basada en plantillas.

En este sentido, XPAND se implementó como parte de un framework potente como openArchitectureWare (oAW) [27] donde podía reutilizar la expresividad de otros lenguajes específicos de dominio como Xtend [28], que le permiten navegar por los meta-modelos.

Este lenguaje destaca sobre el resto de lenguajes de transformaciones modelo a texto (M2T) existentes por proporcionar una solución profesional para proyectos de una cierta envergadura. La mayor desventaja de XPAND es su curva de aprendizaje, mayor que la de otros lenguajes de transformación M2T.

Las plantillas de XPAND están compuestas por uno o más archivos textuales con la extensión .xpt. El generador enlazado al workflow referencia una de estas plantillas y actúa como punto de entrada para el inicio de la transformación. Dichas plantillas a su vez pueden organizarse en paquetes o directorios y subdirectorios del mismo modo que objetos Java.

Las características principales del lenguaje XPAND son:

- Es un lenguaje tipado estáticamente, lo que permite que se puedan validar sus plantillas sintácticamente desde el propio editor antes de su ejecución.
- Permite la integración de fragmentos de código protegidos dentro del código generado.
- Permite llamadas a funciones reutilizables ya sean implementadas en Xtend o en código Java.
- Permite invocación polimórfica de plantillas, basándose en la jerarquía de herencia definida en el meta-modelo.
- Soporta programación orientada a aspectos, permitiendo la introducción de puntos de extensión.
- Soporta la validación del meta-modelo mediante el lenguaje Check, en el que podremos introducir restricciones OCL.
- El editor incorpora características importantes como el coloreado sintáctico, la indicación de errores y la función de autocompletado.

2.3. Trabajos relacionados

Tras haber estudiado varios artículos, TFM y Tesis Doctorales, en esta sección se resumen las dos aportaciones consideradas más relevantes y directamente relacionadas con este Proyecto.

Engineering the development of systems for multisensory monitoring and activity interpretation [29]

En este artículo se identifican varios problemas a la hora de detectar situaciones y de tomar decisiones basadas en el seguimiento y la interpretación de información multisensorial. Entre otros, se identifica como uno de los problemas clave la falta de procesos formales para abordar el diseño de estos sistemas, cada vez más complejos y dinámicos.

Con el fin de superar estos problemas, el trabajo propone un proceso llamado INT3-SDP que ofrece una guía a la hora de desarrollar sistemas de seguimiento multisensorial y de interpretación de comportamientos y situaciones que les dote de un cierto nivel de inteligencia. Además, proporciona a los analistas las directrices y los modelos necesarios para que el entorno pueda ser monitorizado, indicando cómo deben instalarse los sensores y cómo deben implementarse los componentes software necesarios para realizar el seguimiento.

Para facilitar el proceso de toma de decisiones, en este trabajo se toma la información desde varios puntos de vista:

- El uso frecuente de la información capturada desde múltiples cámaras de vigilancia que detectan el tráfico.
- En la rama de la salud se utilizan sensores para detectar posibles enfermedades y evitar casos de emergencias.
- Algunos sensores para la visión o el tacto sirven para medir el grado de tensión de un usuario frente a una tarea de trabajo en su propio ordenador.

El trabajo presentado en este artículo dista mucho de las ideas principales de nuestro proyecto. Sin embargo, lo hemos querido mencionar para mostrar el alcance real que tienen en la actualidad las aplicaciones basadas en sensores, cómo tratan la información obtenida de ellos y las tareas que son capaces de desempeñar en base a dicha información.

HuRoME+: Entorno de Modelado de Coreografías Complejas para un Robot Humanoide [31]

En este TFM se aborda el problema del diseño de secuencias de movimientos predefinidas (coreografías) para robots humanoides.

Este proyecto ilustra los beneficios de aplicar el DSDM al ámbito de la robótica. Para ello, se define el entorno denominado HuRoME (*Humanoid Robot Modeling Environment*) en el que se integran un conjunto de herramientas de DSDM que facilitan: (1) el modelado gráfico de coreografías para el robot (secuencias de movimientos); (2) la generación del código correspondiente a dichas coreografías para el robot humanoide Robonova [32]; y (3) la modernización del software ya

existente para este robot, permitiendo obtener un modelo gráfico a partir de las coreografías previamente programadas de forma manual, con el fin de facilitar su actualización (resulta más sencillo e intuitivo modificar el modelo gráfico que el código correspondiente). Así pues, HuRoME permite a los numerosos usuarios de Robonova, incluso a aquellos que adolecen de formación técnica específica sobre control o programación de robots:

- Modelar gráficamente y validar formalmente las secuencias de movimientos del robot (coreografías).
- Generar automáticamente la implementación asociada a cada coreografía en el lenguaje específico del robot a través de un motor de generación de código.
- Modernizar y reutilizar el software ya existente, permitiendo la obtención de los modelos equivalentes a cualquier programa existente.

Aunque este TFM aplica los principios del DSDM en un dominio distinto al que nos ocupa en este trabajo, nos ha resultado de gran utilidad como guía en las distintas fases del Proyecto. En ambos TFM se ha utilizado prácticamente la misma tecnología (basada en la plataforma Eclipse) para implementar las distintas herramientas que integran los entornos de modelado y generación de código desarrollados. Además, existe un cierto paralelismo entre el lenguaje de modelado de coreografías, incluido en HuRoME, y el lenguaje de especificación de la lógica de los sistemas embebidos, desarrollado como parte de este trabajo: ambos permiten modelar flujos o secuencias, en el primer caso de movimientos del robot y, en el segundo, de actividades que se deben ejecutar en el sistema embebido.

En cuanto a las diferencias entre ambos trabajos, cabe destacar el hecho de que el editor gráfico de modelos implementado en este TFM permite modelar tanto la estructura como el comportamiento del sistema, mientras que el editor implementado en este trabajo citado sólo sirve para especificar la lógica del sistema. En contraposición, en este TFM no se ha automatizado la integración de las herramientas desarrolladas (debe hacerse de forma manual) y no se soportan determinados procesos, como el de modernización de código.

Capítulo 3. El entorno EMDSD

En este capítulo se describe el nuevo entorno de modelado de sistemas embebidos implementado como parte de este TFM y denominado EMDSD.

En la primera sección se describirán los retos que nos han llevado al desarrollo del entorno EMDSD. En la segunda, se detallarán los conceptos básicos del lenguaje modelado diseñado para especificar tanto la estructura como la lógica de los sistemas embebidos. Dentro de esta sección también se comentarán brevemente algunas de las restricciones que ha sido necesario añadir al lenguaje para completar su sintaxis abstracta[1].

En la tercera sección se mostrará el aspecto del editor gráfico de modelos, implementado a partir del lenguaje de anterior, y cuyo objetivo es facilitar a los diseñadores un entorno sencillo e intuitivo con el modelar sistemas embebidos sin necesidad de tener ningún conocimiento previo sobre programación. Dentro de esta sección se mostrará también el aspecto del repositorio utilizado para almacenar las definiciones de los dispositivos y elementos (sensores y actuadores) que tendrán disponibles los diseñadores para modelar sus sistemas. También se comentarán brevemente las extensiones[2] que hemos tenido que programar manualmente en el editor gráfico para poder extraer el máximo rendimiento posible (en cuanto a reutilización) al uso del repositorio.

En la cuarta sección se describirá la estructura de la transformación modelo a texto (*Model-to-Text*, M2T), implementada utilizando el lenguaje XPAND[3], gracias a la cual, a partir de los modelos gráficos especificados con el editor anterior, podremos obtener el código final de la aplicación para la plataforma seleccionada como

---

[1] El listado completo de restricciones OCL incorporadas al lenguaje de modelado se ha incluido en el Anexo II, al final de la memoria.

[2] El código completo asociado a las extensiones del editor gráfico se ha incluido en el Anexo III, al final de la memoria.

[3] La implementación completa del motor de generación de código implementado en XPAND se ha incluido en el Anexo IV, al final de la memoria.

destino. En la quinta sección se indicará cómo utilizar las facilidades implementadas para automatizar la descarga de las librerías asociadas a cada tipo dispositivo a través de las URLs correspondientes. Por último, en la sexta sección, se presentará un ejemplo completo implementado sobre una plataforma OpenPicus.

3.1. Retos del desarrollo

En este apartado se describen los retos u objetivos que nos han llevado a desarrollar el entorno EMDSD como parte de este TFM, así como las herramientas utilizadas para desarrollar cada uno de los elementos que integran dicho entorno.

Objetivo 1 – Definición de un lenguaje de modelado para especificar la estructura de los sistemas embebidos e implementación de un editor gráfico de modelos que facilite su uso.

El entorno EMDSD deberá proporcionar un lenguaje de modelado gráfico para especificar la estructura de los sistemas embebidos seleccionados como plataforma de destino. Este lenguaje permitirá especificar las características de todos los elementos involucrados en el diseño, esto es, del dispositivo y de los sensores y actuadores conectados a él. Para ello, como punto de partida, se utilizará un repositorio en el que, previamente, se habrán definido las características de varios de estos elementos de modo que, cuando deseemos añadir alguno de ellos al modelo gráfico, sólo tendremos que cargar su definición del repositorio. De este modo, las definiciones incluidas en el repositorio son completamente reutilizables y pueden instanciarse tantas veces y en tantos modelos gráficos como sea necesario. El aspecto del modelo gráfico deberá modificarse dinámicamente si se cambia el tipo de alguno de los elementos incluidos en él. Así, por ejemplo, si se cambia el tipo de dispositivo utilizado en el diseño, en el modelo gráfico deberá actualizarse su representación de modo que éste se muestre con sus pines correspondientes y su icono personalizado.

Objetivo 2 - Definición de un lenguaje de modelado para especificar la lógica de ejecución de los sistemas embebidos e implementación de un editor gráfico de modelos que facilite su uso.

El entorno EMDSD deberá proporcionar también un lenguaje de modelado gráfico para especificar las secuencias lógicas (programas) que se ejecutarán en el sistema embebido seleccionado como plataforma de destino. Estas secuencias podrán ser simples o compuestas (gracias al uso de mecanismos de repetición de tipo bucle) y, en todas ellas, deberá marcarse explícitamente su inicio y final. También se proporcionarán operaciones condicionales, de retardo y para poder declarar constantes y variables de distintos tipos, así como para leer o modificar éstas últimas.

Objetivo 3 – Implementación de un motor de generación de código capaz de obtener la implementación asociada a los modelos gráficos creados con los lenguajes anteriores al menos para una de las plataformas existentes.

Este motor de generación de código implementará una transformación M2T para cada plataforma de destino considerada.

Objetivo 4 - Integración de las herramientas desarrolladas.

Este último punto expresa la necesidad de conseguir un entorno compacto, en el que los editores y las transformaciones desarrolladas aparezcan integrados. La correcta integración de las herramientas desarrolladas redunda en una mayor usabilidad y facilidad de aprendizaje de las mismas por parte de los usuarios. Considerando que cada una de las herramientas implementadas utiliza una tecnología diferente (aunque relacionada), su integración no resulta trivial. Sería necesario implementar manualmente en Eclipse las extensiones necesarias para poder lanzar cada herramienta desde un menú contextual ubicado en alguna de las otras. Así, por ejemplo, desde el editor gráfico que permite modelar la lógica de los sistemas embebidos se debería poder cargar el modelo creado previamente con el editor gráfico de la vista estructural del sistema. Del mismo modo, una vez completado el modelo lógico, desde el propio editor, se debería poder lanzar el motor de generación para obtener el código asociado a la aplicación.

Las herramientas y procesos necesarios para abordar los objetivos anteriores aparecen ilustrados, de manera esquemática, en la Figura 4.

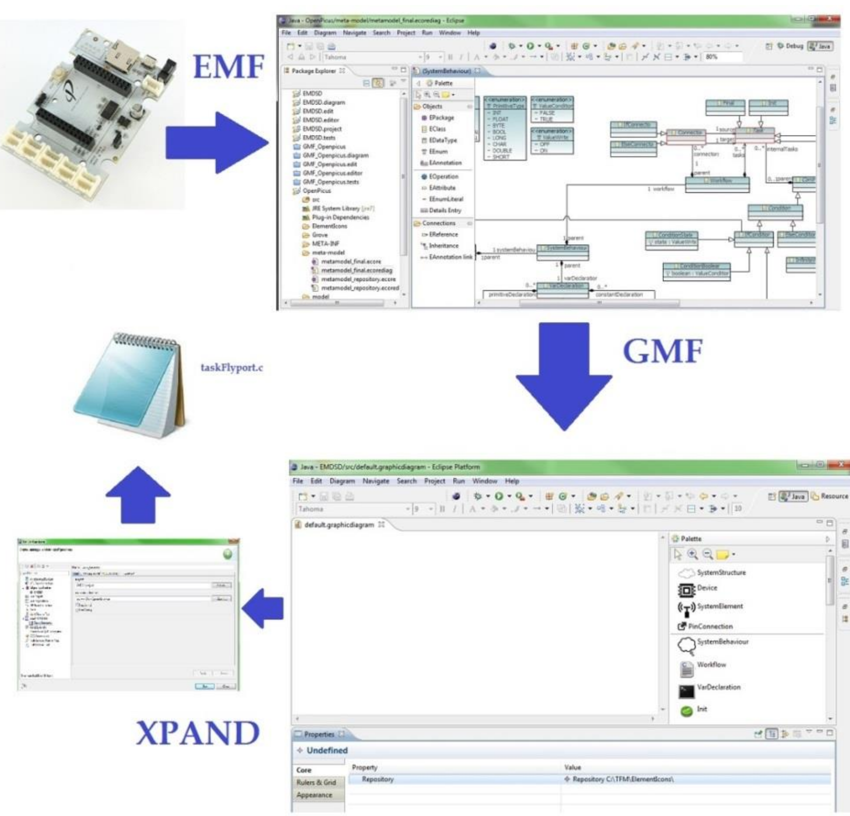


Figura 4: Proceso de desarrollo propuesto basado en el uso del entorno EMDSD.

A partir del estudio de lo que tienen en común distintos sistemas embebidos, se ha implementado un lenguaje de modelado para especificar su estructura y comportamiento. La sintaxis abstracta de este lenguaje se ha especificado utilizando un meta-modelo EMF. A partir de esta especificación se ha implementado un editor gráfico de modelos, que permite modelar tanto la vista estructural como la de comportamiento, utilizando GMF. Finalmente, los modelos desarrollados con este editor los utiliza como entrada el motor de generación de código, implementado en XPAND, para obtener la implementación final del sistema. Conviene señalar que el objetivo 4, relativo a la integración de las herramientas anteriores, no ha sido completado como parte de este trabajo y queda pendiente como una futura extensión, tal y como se comentará en el Capítulo 1.

3.2. El lenguaje de modelado

El meta-modelo desarrollado como parte de este trabajo define los conceptos (y las relaciones existentes entre ellos) necesarios para conformar un lenguaje de modelado que permite especificar tanto la estructura como el comportamiento de un sistema embebido. Aunque el meta-modelo se ha definido en un único fichero (con extensión .ecore), éste cuenta con dos partes claramente diferenciadas y débilmente acopladas: *SystemStructure* (ver Figura 5) y *SystemBehaviour* (ver Figura 6), que se detallarán a continuación en las secciones 3.2.1 y 3.2.2, respectivamente.

Como se puede observar, el diseño de un meta-modelo basado en EMF, como el desarrollado en este trabajo, se asemeja bastante a un diagrama de clases UML, en el que: (1) los conceptos del dominio (meta-clases) son representados usando cajas (*EClass* en EMF); (2) las flechas de punta triangular definen jerarquías de herencia o especialización de conceptos (EMF soporta herencia múltiple); y (3) las flechas simples representan relaciones entre conceptos (*EReference* en EMF) con o sin contención (composición).

El uso de meta-modelos basados en EMF para especificar la sintaxis abstracta de un lenguaje de modelado tiene sus limitaciones. Así, suele ser necesario completar esta sintaxis mediante la definición de un conjunto de reglas sintácticas adicionales. Para ello haremos uso del lenguaje de definición de restricciones OCL, tal y como se detalla más adelante en la Sección 3.2.3.

3.2.1. Modelado de la estructura del sistema (SystemStructure)

En esta parte del meta-modelo (ver Figura 5) se incluyen los conceptos relacionados con el modelado de la estructura del sistema, esto es, el tipo de *dispositivo* utilizado y los *pines* a través de los cuales éste se conecta con distintos elementos, por ejemplo, *sensores* y *actuadores*.

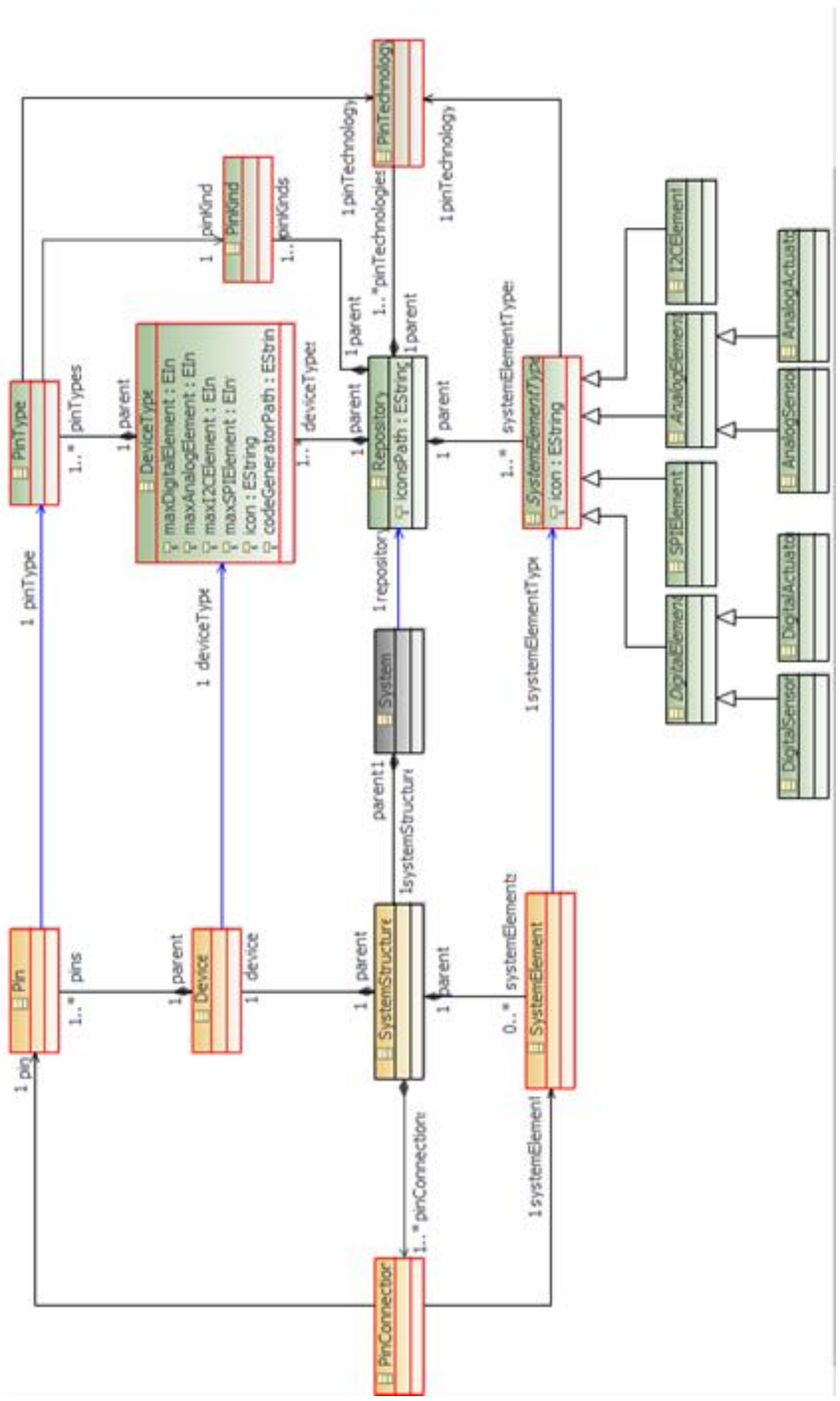


Figura 5: Parte del meta-modelo que describe la estructura del sistema (SystemStructure).

Dado un sistema embebido (System), su estructura (SystemStructure) se define en base a un dispositivo (Device) y a cero o más elementos (SystemElement) que se conectan a él a través de alguno de sus pines (Pin) por medio de conectores (PinConnection).

El dispositivo (Device) tiene asociado un tipo (DeviceType), que determina el número máximo de elementos de cada clase que se le pueden conectar (propiedades maxDigitalElement, maxAnalogElement, etc.). En el lenguaje se han considerado seis clases de elementos (ElementType): DigitalSensor y DigitalActuator (agrupados bajo la meta-clase abstracta *DigitalElement*), AnalogSensor y AnalogActuator (agrupados bajo la meta-clase abstracta *AnalogElement*), SPIElement e I2CElement.

Tanto DeviceType como ElementType tienen asociado un icono (propiedad icon) gracias al que es posible representar gráficamente cada uno de ellos de forma personalizada y bien diferenciada. Además, cada DeviceType tiene asociado un generador de código específico, cuya ruta se debe especificar en el atributo codeGeneratorPath.

La clase que modela el repositorio (Repository) juega un papel central en el modelado de nuevos sistemas embebidos ya que almacena las definiciones (tipos) de los distintos dispositivos, elementos y pines creados por uno o varios diseñadores. Estas definiciones son completamente reutilizables de forma que, para modelar un nuevo sistema, cada diseñador sólo necesitará cargar del repositorio el dispositivo y los elementos que le quieran conectar, pudiendo éstos haber sido diseñados por él mismo o por otros diseñadores.

El hecho de haber separado en el meta-modelo las definiciones de los dispositivos, los elementos y los pines (DeviceType, SystemElementType y PinKind) de las instancias concretas que se crean al modelar cada nuevo sistema embebido (Device, SystemElement y Pin) favorece no sólo la reutilización de los modelos sino también la separación de roles, permitiendo que el modelado de las definiciones (*modeling for reuse*, modelado para reutilización) y de las instancias (*modeling by reuse*, modelado mediante reutilización) puedan llevarla a cabo distintos actores.

Como se detallará más adelante en la memoria, esto ha tenido ciertas implicaciones a la hora de implementar el editor gráfico de modelos ya que la mayoría de las propiedades asociadas a (las instancias concretas de) cada dispositivo y elemento del sistema no las modela el diseñador sino que se cargan automáticamente una vez que éste indica su tipo. Así, por ejemplo, al crear un dispositivo (Device), el diseñador no tendrá que especificar cuántos pines tiene o de qué tipo son; bastará con que

indique el `DeviceType` asociado al dispositivo y los pines se crearán automáticamente conforme a lo indicado en la definición.

3.2.2. Modelado de la lógica del sistema (SystemBehaviour)

En esta parte del meta-modelo (ver Figura 6) se han incluido los conceptos necesarios para modelar la lógica del sistema embebido, esto es, el tipo de *variables* posibles utilizadas y las *tareas* que definirán cada paso en su comportamiento.

Dado un sistema embebido (`System`), su comportamiento (`SystemBehaviour`) se define en base a un flujo de trabajo (`Workflow`) y a una definición de variables (`VarDeclaration`).

Dicha definición de variables (`VarDeclaration`) nos permite, dentro de una declaración (`Declaration`), la distinción entre las variables primitivas, constantes y elementales. Las variables primitivas (`primitiveDeclaration`) definen el tipo (`typePrimitiveDecl`) que puede ser: Int, Boolean, Char, etc.. entre otros. Las variables constantes (`ConstantDeclaration`) definen el mismo tipo (`typePrimitiveDecl`) que las variables primitivas citadas anteriormente pero además ofrecen una funcionalidad especial, asignan un valor constante a lo largo del ciclo de vida de dicha variable. Por último, también podemos declarar variables elementales (`elementDeclaration`) asociadas a los distintos elementos que se pueden conectar con el sistema embebido y que realizarán operaciones de lectura o escritura.

El flujo de trabajo se define en base a una serie de tareas (`Task`) asociadas a un nombre (`NamedElement`) y conectadas entre sí, formando una secuencia, mediante conectores (`Connector`). Estos conectores son arcos dirigidos, esto es, tienen un origen (`source`) y un destino (`target`). El flujo de trabajo comienza siempre en una tarea inicial (`Init`) y termina en una tarea final (`Final`). Las tareas intermedias (`Task`) están asociadas a acciones (`Action`) que pueden ser: de lectura (`Read`), de escritura (`Write`) sobre variables elementales (`varElementDecl`) o de retardo (`DelayTime`). Éstas últimas establecen un tiempo de espera o inactividad en milisegundos (`milliseconds`).

También podemos encontrar tareas asociadas a estructuras de control (`ControlStructure`) de tipo condicional (`Condition`) o bucle (`Repeat`). Las tareas condicionales (`IfCondition`) determinan: (1) bifurcaciones basadas en condiciones booleanas (`ConditionBoolean`) cuyo valor (`ValueCondition`) sea verdadero o falso; (2) bifurcaciones basadas en el estado activo o inactivo de un dispositivo (`ValueWrite`), por ejemplo, en función de un sensor de luz está encendido (mode ON) o apagado (mode OFF); y (3) bifurcaciones basadas en el

valor de una variable (`ConditionValue`), pudiendo comparar si ésta es igual, distinta, mayor, menor, mayor o igual o menor o igual que una determinada constante. Si la condición de un IfCondition no se cumple, se ejecutarán las tareas asociadas a la rama “si no” (`ElseCondition`) del condicional.

Contamos con dos tipos de bucles: `Repeat` y `For`. Los primeros repiten una serie de tareas mientras: (1) se cumpla una determinada condición (`While`); (2) de manera indefinida (`WhileInfinite`); o (3) o un de veces (`WhileFinite`), conocido a priori (`numIterations`), y mientras se cumpla una determinada condición (`sign`). Por otro lado, los bucles de tipo `For` repiten una serie de tareas un número fijo de veces. El número de iteraciones se basa en un contador que comienza en `initIteration`, aumenta o disminuye en cada iteración en una cantidad fija (`valueIncrementOrDecrement`), y termina al alcanzar el valor `maxIteration`.

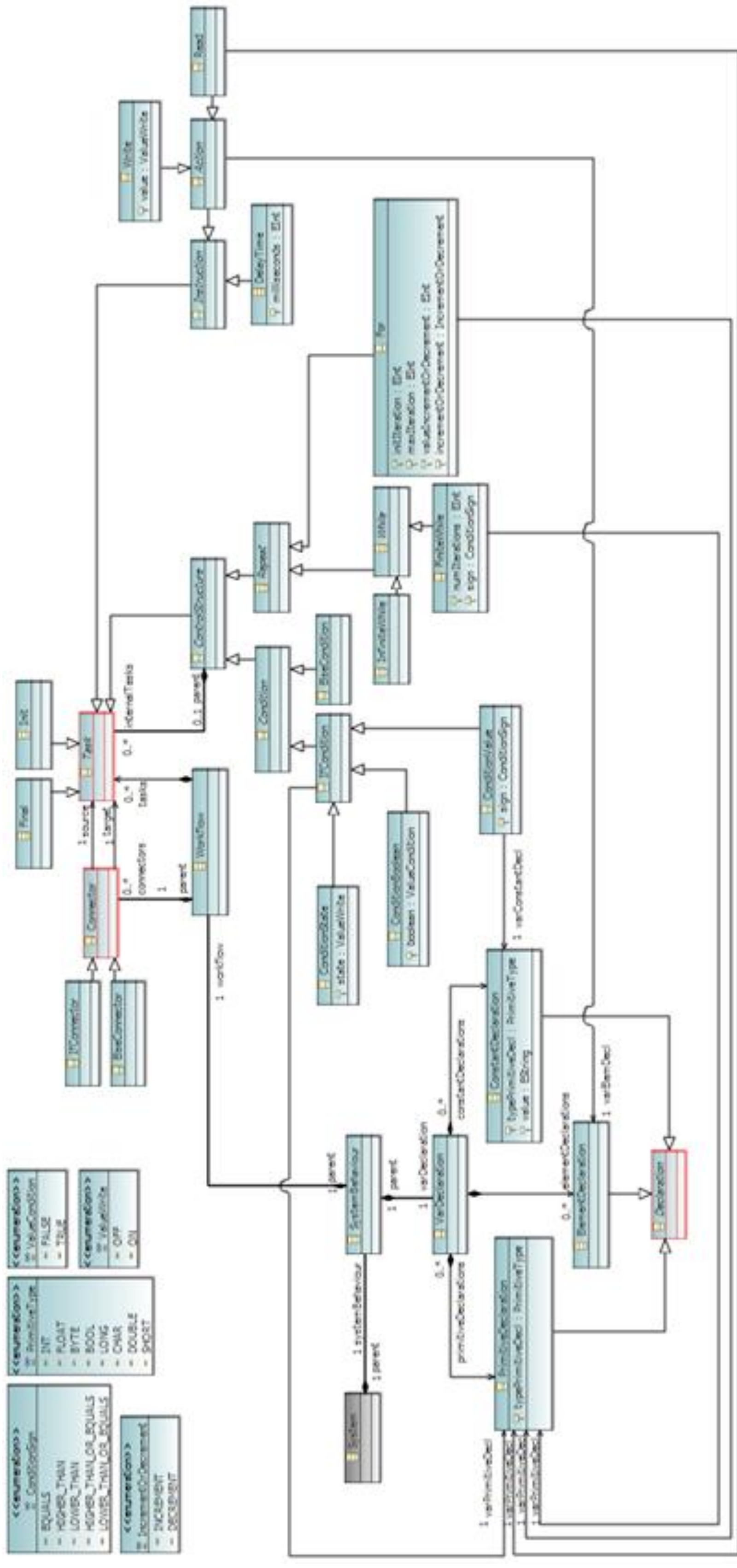


Figura 6: Parte del meta-modelo que describe el comportamiento del sistema (SystemBehaviour).

3.2.3. Restricciones OCL

Como ya se ha comentado anteriormente, las limitaciones de EMF como lenguaje para especificar la sintaxis abstracta de nuevos lenguajes de modelado mediante meta-modelos hace necesario incorporar restricciones sintácticas adicionales utilizando OCL. Para ello se ha utilizado la herramienta OCLInEcore [34] que permite abrir los meta-modelos EMF en un formato textual y añadir restricciones OCL (en forma de invariantes) a los elementos del meta-modelo. Los invariantes se definen del siguiente modo:

```
Class nameEClass
{
    invariant nameInvariant: restriction; // regla
    property nameProperty : typeProperty; // atributo
}
```

Las restricciones OCL aplicadas a nuestro meta-modelo han sido las siguientes:

- `Device`: El número de sensores/actuadores digitales, analógicos, I2C y SPI conectados a un dispositivo no puede superar el máximo estipulado en su tipo (atributos `maxAnalogElements`, `maxDigitalElements`, `maxI2CElements`, `maxSPIElements` del `DeviceType` asociado al dispositivo).
- Sensores/Actuadores:
  - ✓ Deben tener un nombre único que los identifique unívocamente.
- Variables Elementales:
  - ✓ Deben tener un nombre único que los identifique unívocamente.
  - ✓ Definición al menos una variable por sensor/actuador declarado.
  - ✓ Asignación de al menos una variable por sensor/actuador declarado.
- Variables Primitivas:
  - ✓ Deben tener un nombre único que los identifique unívocamente.
  - ✓ Definición de tantas variables primitivas como variables elementales haya declaradas, debido a que sólo podemos realizar operaciones de lectura y escritura a través de variables elementales a variables primitivas.
- Tareas:
  - ✓ Deben tener un nombre único que los identifique unívocamente.
  - ✓ Sólo puede haber una tarea Inicial.
  - ✓ Sólo puede haber una tarea Final.
  - ✓ La tarea Final sólo debe destino de un y sólo un conector.
  - ✓ La tarea Inicial solo debe ser origen de un y sólo un conector.
  - ✓ Los Bucles deben tener al menos una tarea.
  - ✓ Las tareas ElseCondition sólo tienen sentido si previamente se ha definido la correspondiente tarea IfCondition.

  - ✓ Los Condicionales Booleanos deben de tener variables primitivas booleanas.
- ➢ Conectores a/desde los Pines:
  - ✓ Los pines deben estar conectados con puertos de pines de su mismo tipo. Por ejemplo, los pines analógicos deben estar conectados a puertos analógicos del sistema embebido.
  - ✓ No puede haber dos pines conectados a un mismo puerto o viceversa.
  - ✓ Los pines deben conectar elementos de su misma tecnología.
  - ✓ Deben tener un nombre único que los identifique unívocamente.

En el Anexo II se detalla la especificación OCL de todas estas restricciones añadidas al meta-modelo. Para definir estas restricciones se ha utilizado la información incluida en [35], además de las referencias sobre OCL previamente incluidas en el texto.

3.3. Herramientas de modelado

En esta sección se describen brevemente las herramientas de modelado implementadas como parte del entorno EMDSD, así como el papel que juegan los modelos que se construyen con cada una de ellas dentro del proceso de DSDM propuesto.

3.3.1. Modelado y uso del repositorio

El repositorio es uno de los elementos centrales de este trabajo ya que incorpora la definición de los sistemas embebidos disponibles, así como de los sensores y actuadores que pueden conectarse a ellos a través de distintos tipos de pines. Por ello, su definición debe ser previa a la del resto de los modelos empleados en el proceso de diseño basado en el entorno EMDSD.

Cada elemento definido en el repositorio estará a disposición de los diseñadores, que podrá incluirlo tantas veces como quiera en cuantos modelos lo necesite. De este modo, las definiciones incluidas en el repositorio pueden considerarse artefactos completamente reutilizables.

Es posible crear un repositorio general con todos los elementos disponibles para todos los dispositivos considerados, o bien un repositorio asociado a los elementos de cada dispositivo. Ambas aproximaciones tienen sus ventajas e inconvenientes. Como en nuestro caso sólo hemos considerado dos posibles dispositivos (OpenPicus y TinkerKit) y un conjunto relativamente reducido de sensores y actuadores compatibles con ellos, hemos optado por la primera opción. La Figura 7 muestra el aspecto del modelo de repositorio desarrollado como parte del entorno EMDSD.

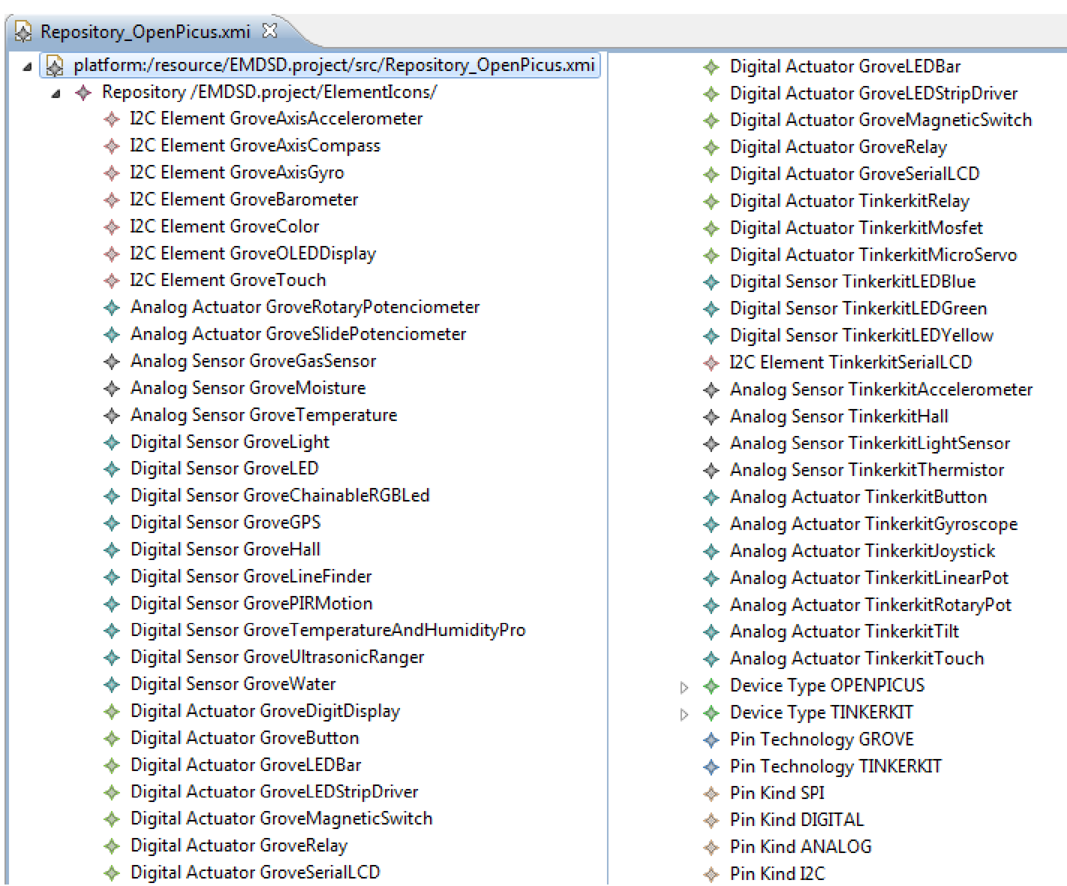


Figura 7: Modelo de repositorio desarrollado como parte del entorno EMDSD.

Una vez creado el repositorio, éste deberá cargarse en memoria cada vez que se inicie el diseño de un nuevo sistema embebido, esto es, cada vez que se cree un nuevo modelo con el editor gráfico utilizado para especificar su estructura, cuyo aspecto y funcionalidad se detallan en la siguiente sección.

3.3.2. Modelado de la estructura y el comportamiento del sistema

Para especificar la estructura y el comportamiento de los sistemas embebidos se ha implementado un editor gráfico de modelos haciendo uso de GMF. Los detalles del proceso seguido para implementar este editor pueden encontrarse en el apartado 2 del Anexo I. El aspecto de este editor es el que se muestra en la Figura 8.

Como se puede observar, el editor se ejecuta dentro de Eclipse, por lo que cuenta con algunas de las vistas propias de este entorno (p.ej., a la izquierda, el explorador del proyecto o, en la zona inferior de la ventana, la vista de propiedades). Propios del editor son la zona central (mostrada en blanco en la figura), en la que el diseñador podrá dibujar sus modelos y, a la derecha de la ventana, la paleta de herramientas que le permitirá seleccionar el elemento del lenguaje que desea añadir a su modelo en cada momento.

Los elementos incluidos en la paleta de herramientas están organizados en dos grupos: el primero contiene los elementos relacionados con la especificación de la estructura del sistema (`SystemStructure`, `Device`, `SystemElement` y `PinConnection`), mientras que el segundo contiene los relacionados con la especificación de su comportamiento (elementos descritos en la sección 3.2.2).

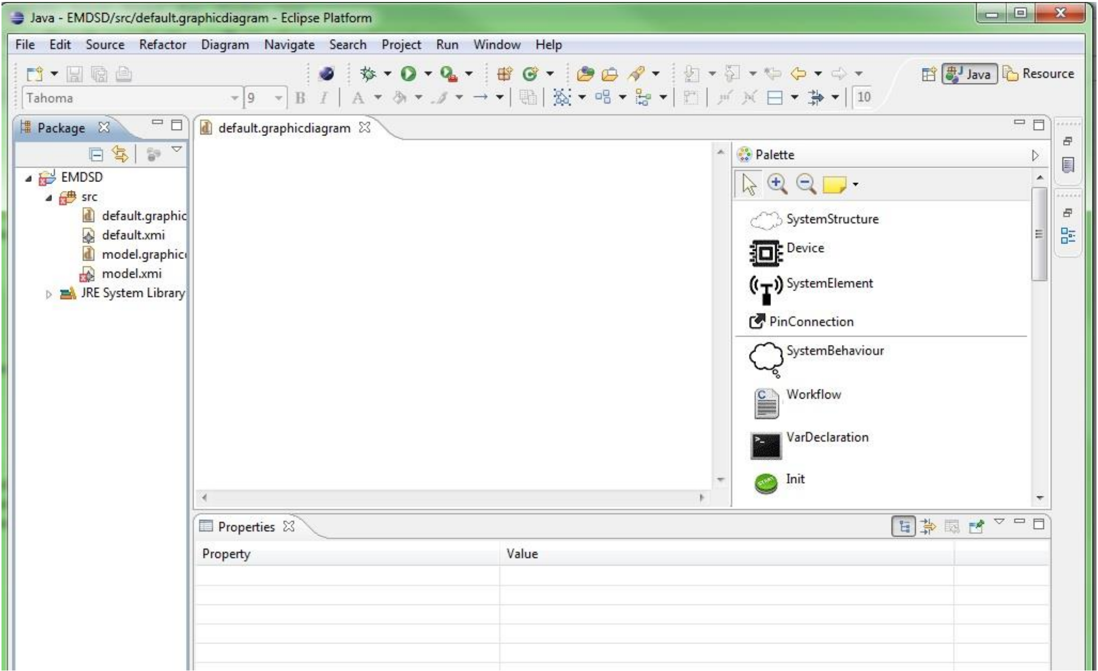


Figura 8: Aspecto del editor gráfico que permite modelar la estructura de los sistemas embebidos.

La implementación Java del editor gráfico generado por GMF ha sido extendida manualmente para añadir las siguientes funcionalidades, la mayoría de ellas relacionadas con el modelado de la estructura del sistema:

1) Conservación del nombre del paquete asociado al modelo serializado.

2) Selección y carga del repositorio. Al arrancar el editor gráfico y antes de comenzar cada nuevo diseño, es necesario cargar el modelo del repositorio en el que están contenidas las definiciones de todos los elementos (dispositivos, sensores, actuadores, pines, etc.) que el diseñador puede incorporar a sus modelos. Para ello, se mostrará una ventana en la que podremos navegar por la estructura de carpetas del ordenador hasta localizar y seleccionar el fichero en el que está almacenado el repositorio. Una vez seleccionado, éste se cargará en memoria de forma que todos sus elementos pasarán a estar automáticamente disponibles en el editor gráfico.

3) Carga dinámica de los iconos asociados al dispositivo y a los distintos sensores/actuadores incluidos en el modelo en función de su tipo. En tiempo de modelado, cuando se asocia un tipo a alguno de los elementos del sistema

(dispositivo, sensor o actuador), se carga y muestra en pantalla el icono asociado a los elementos de dicho tipo. Si se cambia el tipo con posterioridad, el icono se actualizará convenientemente.

4) Creación dinámica de los pines asociados a un dispositivo en función de su tipo. En tiempo de modelado, cuando se asocia un tipo a un dispositivo, se crean y muestran en pantalla los pines (número y tipo) asociados a esa clase de dispositivo. De hecho, el diseñador no puede modificar la configuración de pines que se carga automáticamente, ya que no puede eliminar ni añadir nuevos pines al dispositivo (la paleta de herramientas no incluye el elemento Pin). Si con posterioridad se cambia el tipo del dispositivo, los pines se reajustan convenientemente.

5) Decoración de los pines según su tipo (Analógico, Digital, I2C, SPI). Esta extensión va ligada a la anterior. De hecho, a medida que se van creando y añadiendo los pines al dispositivo al seleccionar su tipo, se les asocia el color que indica su tipo: Digitales: azul; Analógicos: rojo; SPI: amarillo; I2C: verde; Resto de pines: gris).

El código completo asociado a estas extensiones aparece recogido en el Anexo III, al final de la memoria.

3.4. Motor de generación de código

Para implementar el motor de generación de código del entorno EMDSD hemos optado por utilizar el lenguaje de transformación M2T denominado XPAND, cuyas características principales aparecen recogidas en la sección 2.2.3.

XPAND nos ha permitido implementar una transformación que, a partir de un modelo serializable *.xmi creado con el editor gráfico descrito en la sección previa, genera el código equivalente para el sistema embebido seleccionado como plataforma destino.

Como parte de este TFM se ha definido una única transformación M2T para generar código para la plataforma OpenPicus. La incorporación de otras transformaciones al motor de generación de código para dar soporte a otras posibles plataformas queda abierta para posibles ampliaciones futuras de este trabajo.

Los proyectos de programación que se desarrollan en OpenPicus utilizan varios ficheros. De todos ellos, el principal, es el fichero denominado ***“TaskFlyport.c”***, cuya estructura se muestra a continuación:

```
// Declaración de librerías
// Declaración de variables e inicialización
```

```
// Método que ayuda a la programación

void FlyportTask()
{
	// Declaración dispositivo GROVE
	// Declaración de sensores/actuadores
	// Conectar dispositivos a señales
	// Inicialización de sensores/actuadores

	while (1)
	{
		// Programación Dispositivo
	}
}
```

Cada uno de los apartados de este fichero, marcado con los caracteres ***"//"***, nos ha servido para identificar el código que se debe programar en la transformación M2T, así como los elementos del meta-modelo que intervienen en esa parte del código.

Una vez identificados los elementos necesarios para generar cada porción del código, deberemos acceder a ellos con XPAND recorriendo el modelo de partida (creado con el editor gráfico descrito en la sección anterior) desde su nodo raíz. Una vez localizados los elementos en cuestión, utilizaremos la información almacenada en ellos dentro del modelo para generar el texto correspondiente en el fichero de código.

La posibilidad de definir ***"templates polimórficos"*** en XPAND ha sido de gran utilidad, ya que hemos podido definir elementos reutilizables que pueden ser invocados utilizando diferentes parámetros de entrada, tal y como se ilustra en el siguiente ejemplo:

```
«DEFINE template FOR Elemento1»
	Programación template con elemento1
«ENDDEFINE»

«DEFINE template FOR Elemento2»
	Programación template con elemento2
«ENDDEFINE»
```

El código completo asociado al motor de generación de código implementado en XPAND aparece recogido en el Anexo IV, al final de la memoria.

3.5. Descarga y ubicación de las librerías del dispositivo

En la actualidad, descargar las librerías asociadas a los distintos dispositivos es un proceso manual y bastante tedioso. En primer lugar, hay que acceder a la página oficial del sistema embebido elegido coma plataforma de desarrollo y buscar el sensor/actuador del que queramos descargar la librería. El fichero que obtengamos,

habrá que descomprimirlo y copiarlo en la ubicación adecuada dentro de nuestro proyecto de desarrollo de software.

Por ejemplo, para OpenPicus, elegido como plataforma de desarrollo en este TFM, es necesario acceder a la wiki oficial: http://wiki.openpicus.com/index.php/Main_Page (ver Figura 9) y seleccionar el apartado *Grove Kits & Libraries*, en el que aparece una lista de las librerías asociadas a los distintos sensores y actuadores compatibles con esta plataforma (ver Figura 10).

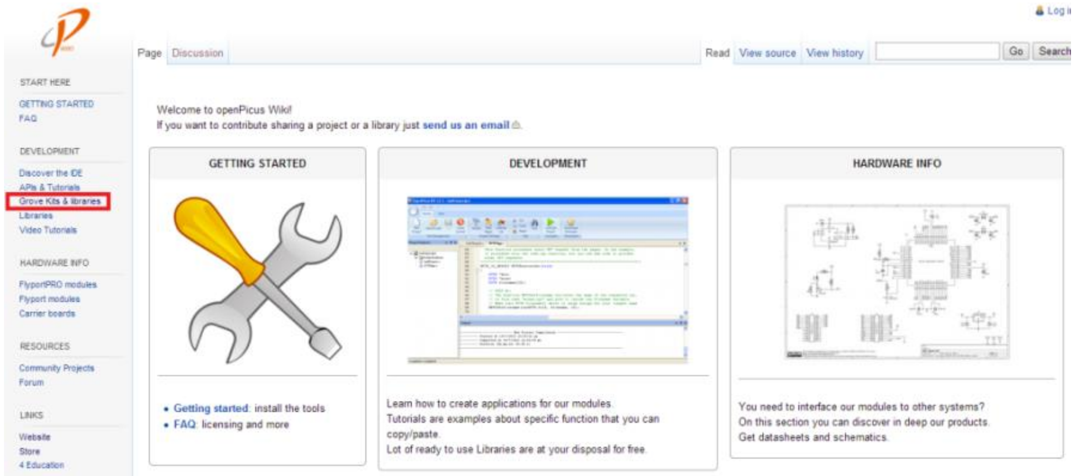


Figura 9: Wiki oficial del dispositivo

Figura 10: Algunos de los sensores/actuadores compatibles con OpenPicus.

Si elegimos, por ejemplo, el sensor de temperatura marcado con un cuadro rojo en la figura anterior, aparecerá la siguiente página en la que podremos descargar la librería asociada a este elemento (ver Figura 11).

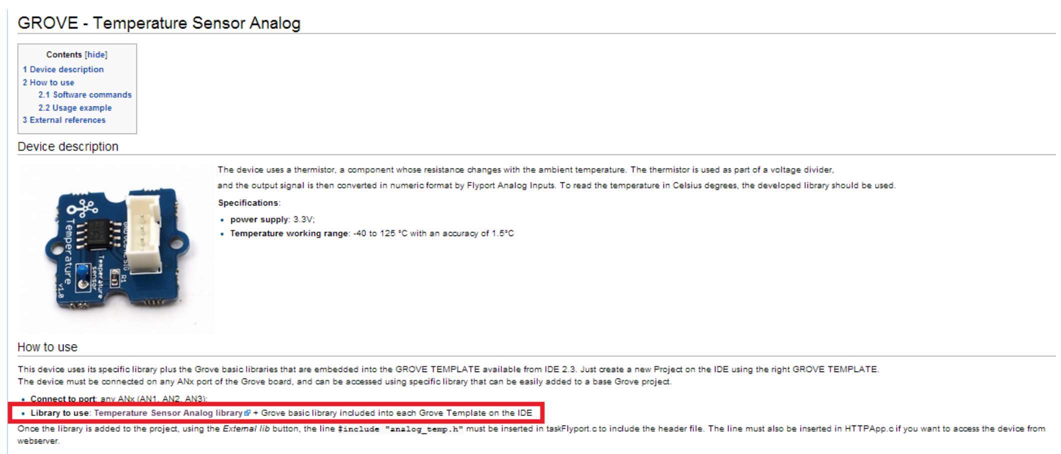


Figura 11: Sensor temperatura dispositivo

Los dos ficheros incluidos en cada librería descargada (normalmente comprimidos en formato.*zip*), deberían ubicarse dentro de las carpetas correspondientes de nuestro proyecto de desarrollo para la plataforma OpenPicus (*/proyecto*). Así el fichero con extensión *.c* debería incluirse dentro de la carpeta: */proyecto/Libs/ExternalLibs/* y el fichero con extensión *.h*, dentro de la carpeta: */proyecto/Libs/ExternalLibs/Include/*.

Como parte de este TFM se ha automatizado el proceso de descarga y ubicación de estas librerías mediante un archivo de procesamiento por lotes (archivo .bat). Cada una de las pantallas programadas en este proceso por lotes tiene una estructura como la siguiente (pantalla inicial):

```
:principal
cls
@echo off
color 30
...
echo ==================================================
echo. Descarga y movimiento de librerías externas.
echo. Acciones disponibles:
echo.     1.- Descarga de librerías externas.
echo.     2.- Movimiento de librerías externas.
echo.     3.- Salir
...
set /p var=Accion escogida:
if %var% ==1 goto descarga
if %var% ==2 goto movimiento
if %var% ==3 goto salir
```

Como podemos observar, cada pantalla comienza con una etiqueta que facilita su identificación de forma unívoca (en el ejemplo ***:principal***). Dentro de cada etiqueta, las líneas precedidas de la palabra "***echo***" se utilizan para mostrar mensajes por

pantalla (en el ejemplo, las distintas opciones de ejecución). La opción elegida por el usuario es recogida en la variable ***var*** y utilizada para pasar a la pantalla correspondiente mediante el comando ***"goto* et"**, que salta a la pantalla identificada con la etiqueta **et**.

Al ejecutar el proceso por lotes, el aspecto de esta primera pantalla es el que se muestra a continuación en la Figura 12.

Figura 12: Pantalla principal descarga automatizada de Librerías del dispositivo

Si en este menú principal seleccionamos la opción 1, aparecerá una pantalla como la mostrada en la Figura 13. En esta pantalla podremos seleccionar el tipo de elemento del que queremos descargar las librerías.

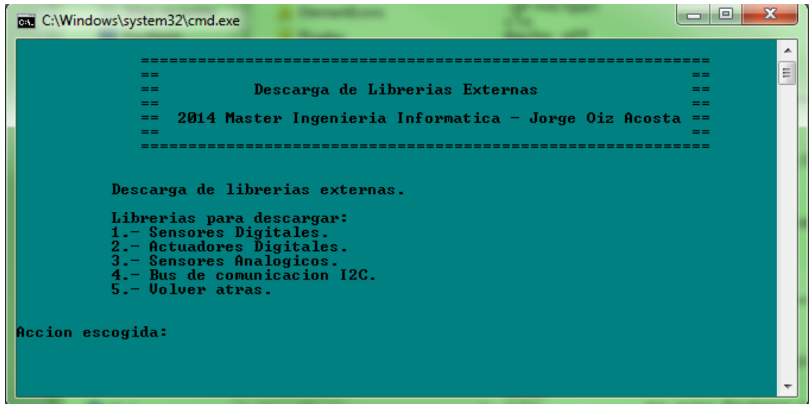


Figura 13: Opción descarga automatizada de Librerías del dispositivo

Del mismo modo, si en el menú principal seleccionamos la opción 2, aparecerá una pantalla como la mostrada en la Figura 14, en la que nos solicitará que introduzcamos la ruta en la que queremos guardar las librerías descargadas.

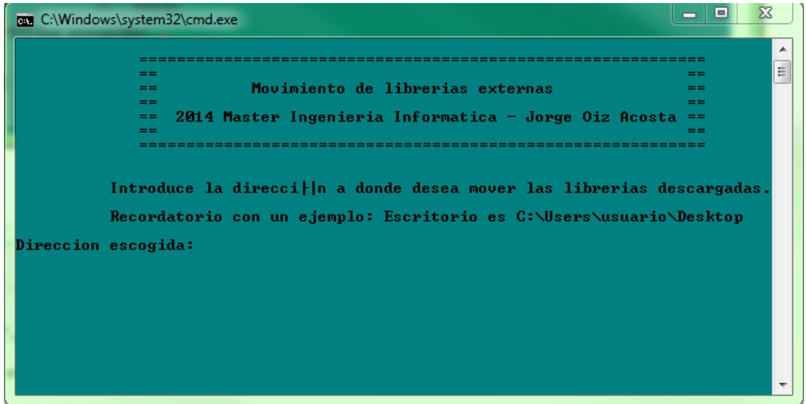


Figura 14: Opción 2 descarga automatizada de Librerías del dispositivo

Como se detalla en el siguiente capítulo, dentro de la sección 4.2, para poder ejecutar este proceso por lotes, es necesario instalar previamente un par de aplicaciones.

Capítulo 4. Manual de Instalación y Uso

En esta sección se describe el proceso que se debe seguir para instalar y utilizar el entorno EMDSD, desarrollado como parte de este TFM.

4.1. Consideraciones Iniciales

Antes de instalar y utilizar el entorno EMDSD deberemos prestar atención a las siguientes consideraciones:

- La versión de Eclipse [36] empleada en el desarrollo del proyecto ha sido la *Indigo (v 3.7)*, configurado con el bundle *Eclipse Modelling Tools v.1.4.2.*, el plug-in *Graphical Modelling Tools v 2.4.0*, el plug-in *Xpand SDK v.1.1.1* [26] *y el plug-in OCL Example and Editors v 3.1.2* [15].

- Para ejecutar los plug-ins asociados al editor gráfico del entorno EMDSD deberemos importar y mantener abiertos en Eclipse todos los proyectos desarrollados como parte del TFM. El usuario deberá acceder al menú *Run → Run Configurations*, seleccionar *Eclipse Application → EMDSD* y pulsar el botón *Run*. De esta forma se lanzará un nuevo Eclipse en el que estarán instalados y listos para usarse los plug-ins asociados al editor.

  Nota: Es recomendable ampliar la memoria asignada al Eclipse que se lanza desde el entorno de desarrollo. Para ello, en el paso anterior, antes de pulsar el botón *Run*, se debe acceder a la pestaña *Arguments* y añadir la siguiente línea en el campo *Program arguments* (ver Figura 15):

  -Dosgi.requiredJavaVersion=1.5
  -Xms512m
  -Xmx1024m
  -XX:PermSize=512M

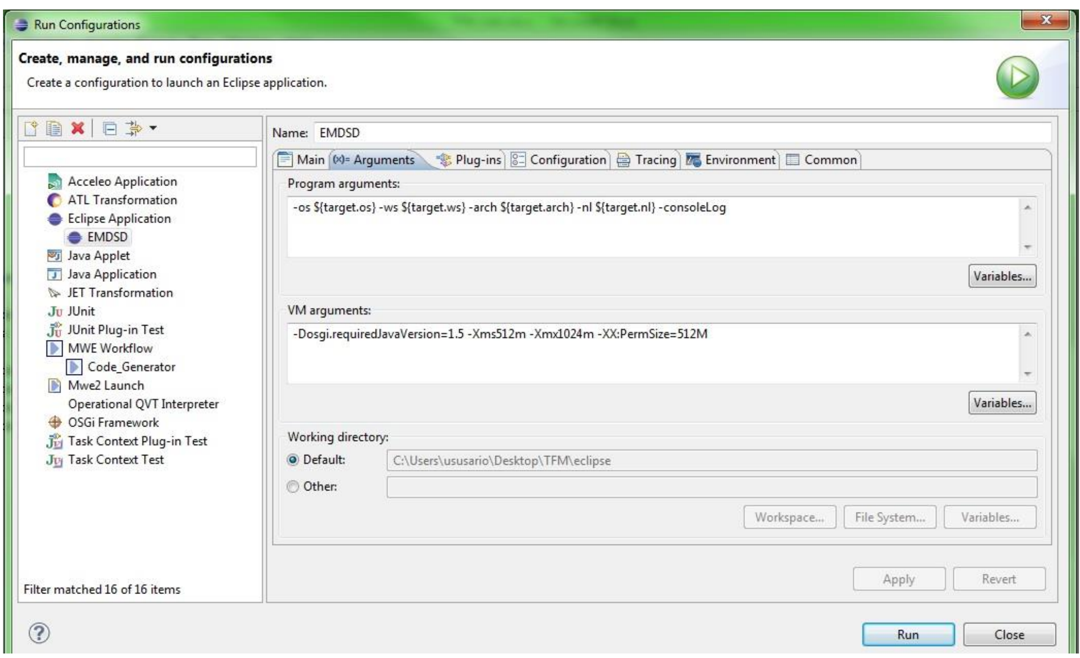


Figura 15: Configuración de memoria del Eclipse donde se ejecutará el editor gráfico.

- Para ejecutar el plug-in que implementa el motor de generación de código también deberemos mantener abiertos en Eclipse todos los proyectos desarrollados como parte del TFM. En este caso, el usuario deberá acceder al menú *Run* → *Run Configurations*, seleccionar la opción *MWE Workflow* → *Code Generator* y pulsar el botón *Run*. De esta forma se ejecutará (en el propio Eclipse de desarrollo, sin necesidad de lanzar un segundo Eclipse) el motor de generación de código asociado al modelo serializado que se haya obtenido previamente con el editor gráfico de modelos. Este modelo deberá haber sido copiado en la carpeta */src* del proyecto XPAND, denominado *EMDSD.project* (ver Figura 16).

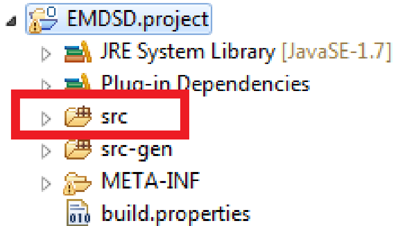


Figura 16: Ubicación del modelo serializado dentro del proyecto XPAND.

Nota: Antes de ejecutar el motor de generación de código pulsando el botón *Run*, como se indica en el párrafo anterior, deberán configurarse los siguientes parámetros en la ventana *Code Generator* (ver Figura 17).

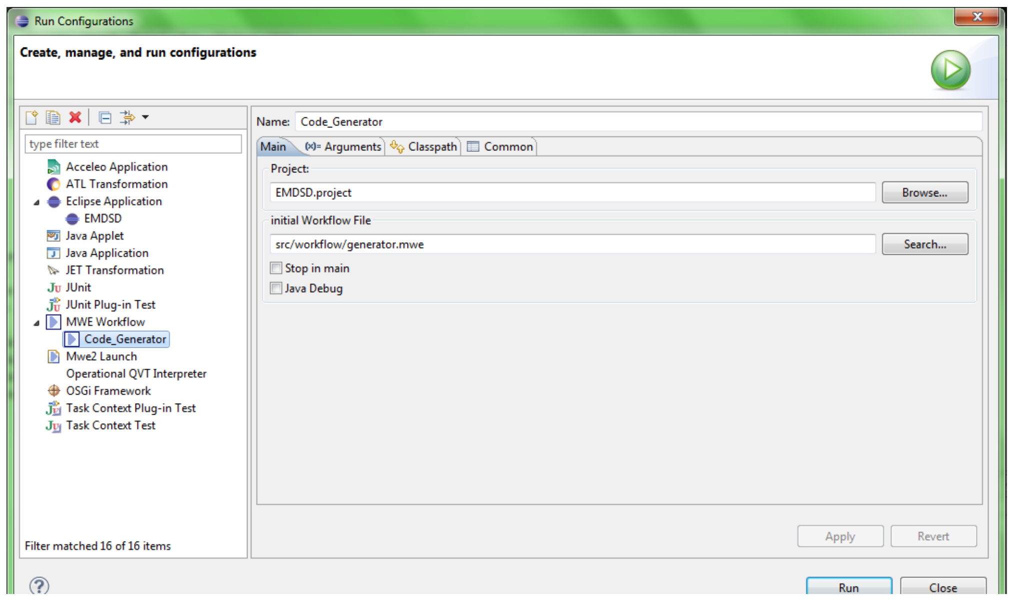


Figura 17: Configuración del motor de generación de código en el workspace.

- Por último, para ejecutar el proceso por lotes (fichero *.bat*) que permite la descarga y copia (en la carpeta correspondiente del proyecto) de las librerías asociadas a los dispositivos empleados en los modelos gráficos, debemos tener instalado: (1) el programa *Wget.exe* [37], que utilizará el proceso por lotes para descargar las librerías; y (2) el programa *7-Zip* [38], que utilizará el proceso por lotes para extraer, en la carpeta adecuada del proyecto, el contenido del archivo *.zip* descargado.

4.2. Instalación del entorno EMDSD

Para instalar el entorno EMDSD deberemos, en primer lugar, descomprimir los dos siguientes ficheros, incluidos en el CD que se adjunta con la memoria del TFM: *Workspace_EMDSD* y *Entorno_EMDSD* (Ver Figura 18). Ambos ficheros deben descomprimirse en el mismo directorio. Puede elegirse cualquier ubicación para este directorio aunque, dada la longitud de los nombres asociados a algunas de las carpetas y ficheros incluidos en ambos ficheros, se recomienda crearlo en una ruta de nombre corto, p.e., *C:\Eclipse* o directamente en *C:*.

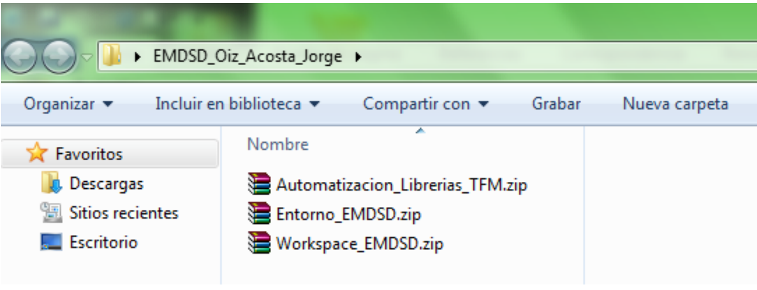


Figura 18: Ficheros incluidos en el CD que se adjunta a la memoria del TFM.

A continuación, ejecutaremos Eclipse y seleccionaremos, como ruta del espacio de trabajo (*workspace*), el directorio en el que hayamos descomprimido el fichero *Workspace_EMDSD.zip* (ver Figura 19).

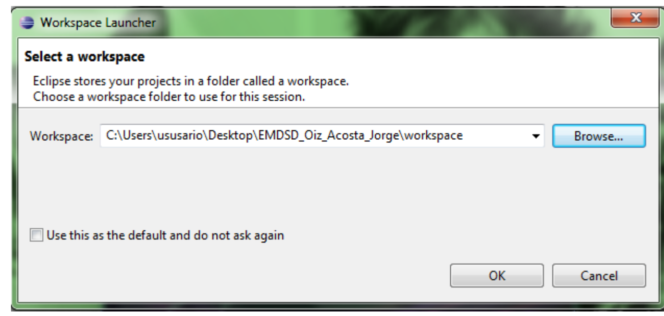


Figura 19: Selección del espacio de trabajo (workspace) para el entorno EMDSD.

Una vez arrancado Eclipse, deberemos importar los proyectos asociados al TFM utilizando la opción *File → Import → Existing Projects into Workspace*. Estos proyectos los hemos obtenido, en el primer paso, al descomprimir el fichero *Workspace_EMDSD.zip*. En la ventana de importación, deberemos seleccionar todos los proyectos (ver Figura 20) y pulsar el botón *Finish*.

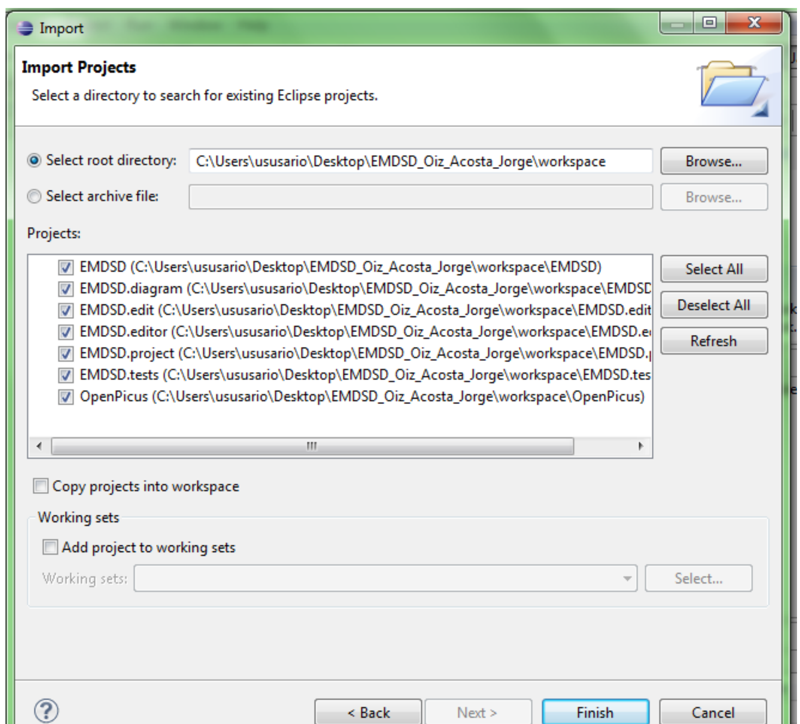


Figura 20: Proceso de Import de los proyectos en el Workspace del entorno EMDSD

Los proyectos que se importan son los siguientes:

- ***EMDSD***: proyecto que contiene: (1) una carpeta *metamodel*, en la que aparecen tanto el meta-modelo (en su versión XMI –fichero *.ecore*– y en su

versión gráfica –fichero *.ecorediag*–), como el generador EMF asociado a éste (fichero *.genmodel*); (2) una carpeta *GMFApp* en la que aparecen los modelos GMF asociados el editor gráfico (ficheros *.gmfgraph*, *.gmftools*, *.gmfmap* y *.gmfgen*); y (3) el código Java generado a partir del meta-modelo por EMF (carpeta */src*).

- ***EMDSD.edit***, ***EMDSD.editor*** y ***EMDSD.tests:*** código JAVA del editor de modelos en forma de árbol y de las pruebas unitarias obtenidas a partir del generador EMF (fichero *.genmodel*), incluido en la carpeta *metamodel* del proyecto EMDSD.
- ***EMDSD.diagram***: código JAVA del editor gráfico de modelos obtenido a partir del generador GMF (fichero *.gmfgen*), incluido en la carpeta *GMFApp* del proyecto EMDSD.
- **OpenPicus**: proyecto piloto del entorno EMDSD (primera versión). Contiene el meta-modelo base que sustenta todo el desarrollo del entorno EMDSD (en su versión XMI –fichero *.ecore*– y en su versión gráfica –fichero *.ecorediag*–), dónde se describe la sintaxis abstracta del lenguaje de modelado con el que especificaremos la estructura y el comportamiento de los sistemas embebidos.

En este punto, sólo resta configurar los lanzadores del editor gráfico de modelos y del motor de generación (ver consideraciones iniciales, incluidas en el apartado anterior) para tener listo y completamente operativo el entorno EMDSD.

Antes de concluir esta sección detallaremos el proceso que se debe seguir para instalar el programa que automatiza la descarga de las librerías asociadas a los sensores y actuadores que utilizaremos en los modelos gráficos. En primer lugar, deberemos crear una carpeta *TFM* en el directorio raíz del ordenador C:\ y descomprimir en ella el fichero *Automatizacion_Librerias_TFM.zip*, incluido en el CD que se adjunta a esta memoria (Ver Figura 18). A continuación, deberemos instalar los programas de libre distribución *7-zip* y *Wget* (incluidos en el fichero *Automatizacion_Librerias_TFM.zip*), tal y como se indica en las consideraciones iniciales, incluidas en el apartado anterior. Una vez instalados ambos programas ya podremos ejecutar el proceso por lotes, implementado en el fichero *grovenest.bat*, para descargar las librerías necesarias (ver Figura 21).

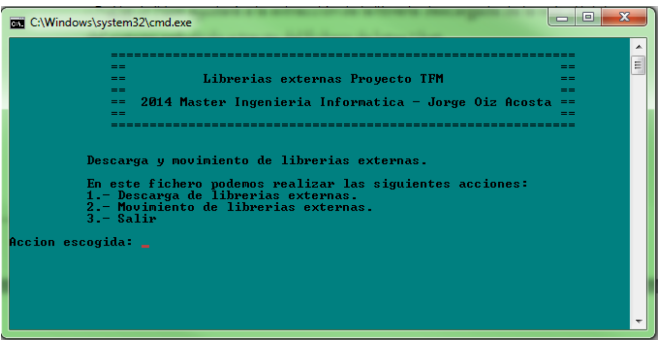


Figura 21: Pantalla inicial del proceso por lotes que automatiza la descarga de librerías OpenPicus.

Al seleccionar la opción 1 del menú, el proceso por lotes descarga las librerías, por defecto, en la carpeta *C:\ ExternLibrary_GroveNest*. Para moverlas a la carpeta adecuada dentro del proyecto Eclipse, deberemos ejecutar la opción 2 del menú.

4.3. Uso del entorno EMDSD

El entorno EMDSD permite: (1) la creación y la validación de modelos gráficos en los que se especifique la estructura y el comportamiento de los sistemas embebidos; y (2) la generación del código asociado a dichos modelos para una determinada plataforma. En esta sección se describe cómo utilizar las dos herramientas implementadas para dar soporte a estos dos procesos.

4.3.1. Diseño y validación de los modelos gráficos

Para diseñar cada nueva aplicación, el primer paso será utilizar el editor gráfico de modelos incluido dentro del entorno EMDSD. Para ello, deberemos ejecutar Eclipse, seleccionando el *workspace* en el que hayamos instalado el entorno siguiendo los pasos indicados en la sección anterior. A continuación, arrancaremos el editor gráfico de modelos, lanzando un nuevo Eclipse, tal y como se indica en la sección 4.1.

En el nuevo Eclipse, deberemos crear un proyecto Java seleccionando en el menú la opción: *File → New → Java Project*. Debemos dar un nombre al proyecto y seleccionar la versión adecuada de Java, en nuestro caso, Java 1.7 (ver Figura 22).

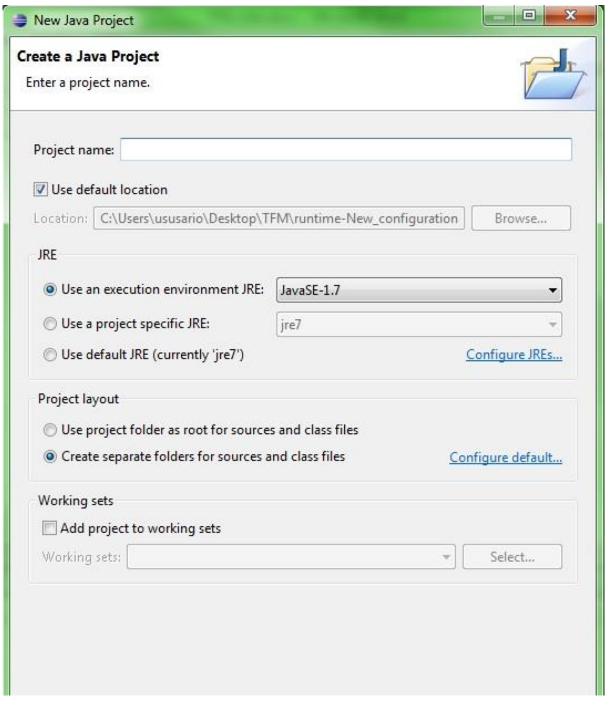


Figura 22: Creación de un proyecto Java en Eclipse.

A continuación, para crear un nuevo modelo utilizando el editor gráfico deberemos seleccionar la opción del menú: *File → New → Other→ Examples → EMDSD Diagram* (ver Figura 23).

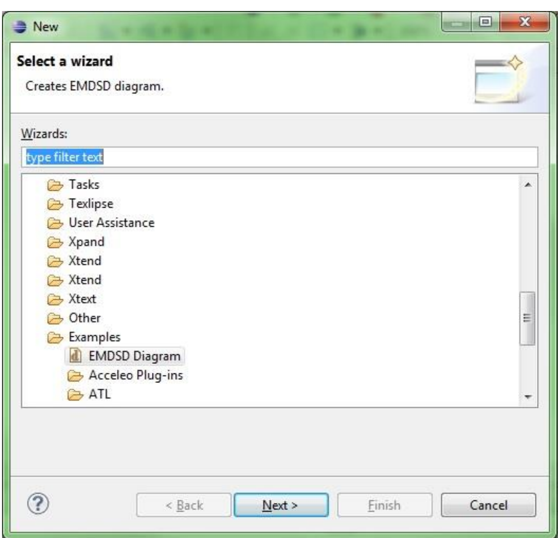


Figura 23: Creación de un modelo gráfico.

Antes de poder comenzar el diseño, aparecerá una ventana emergente en la que deberemos seleccionar el fichero que contiene el repositorio del entorno EMDSD (ver Figura 24). Recordemos que el repositorio contiene las definiciones de los

distintos elementos (dispositivos, sensores, actuadores, etc.) que tendremos disponibles para incluir en nuestros modelos gráficos.

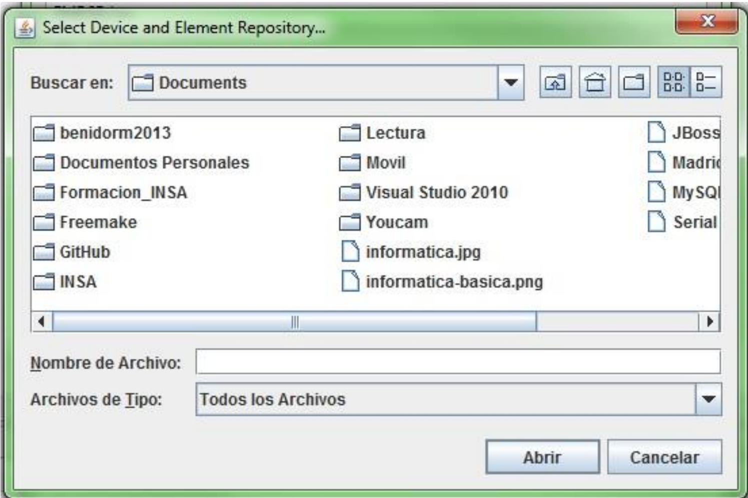
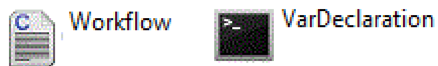


Figura 24: Selección repositorio Entorno EMDSD

Una vez cargado el repositorio, aparecerá el editor gráfico, con una paleta de herramientas lateral, en la que estarán todos los elementos que podemos incorporar a nuestro diseño. En primer lugar, deberemos añadir al modelo los dos elementos principales que componen el diseño, esto es: *SystemStructure* y *SystemBehaviour*. A continuación se muestran los iconos asociados a estos dos elementos:

Al seleccionar cada uno de ellos, deberemos pinchar sobre la zona central de modelado (inicialmente en blanco) para agregarlos al modelo.

Dentro del *SystemBehaviour* deberemos añadir un elemento *VarDeclaration* (declaración de variables) y otro *Workflow* (flujo de tareas), cuyos iconos asociados son los que se muestran a continuación:



A su vez, dentro del *Workflow*, deberemos añadir un elemento *Init* (para indicar dónde comienza el flujo de tareas) y otro *Final* (para indicar dónde termina), cuyos iconos asociados son los que se muestran a continuación:

En este momento, el aspecto de nuestro modelo gráfico debería ser como el que se muestra a continuación en la Figura 25.

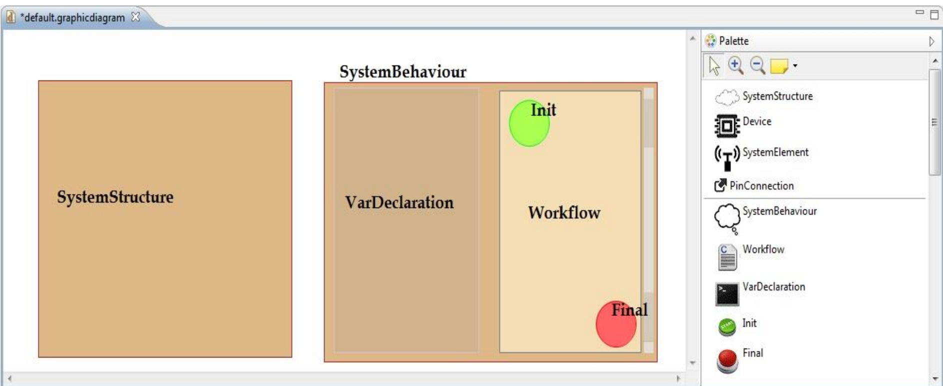


Figura 25: Aspecto gráfico componentes mínimos programación dispositivo en Entorno EMDSD

En el siguiente paso, definiremos la estructura del sistema, añadiendo dentro del componente *SystemStructure* un elemento de tipo *Device*. El icono por defecto asociado a este concepto del lenguaje, es el que se muestra a continuación:

Si seleccionamos el elemento *Device* que acabamos de añadir al modelo y mostramos sus propiedades (*Window* → *Show View* → *Properties*), veremos que éste tiene un *name* y un *DeviceType* (ver Figura 26). En la primera propiedad, deberemos indicar el nombre del dispositivo y, en la segunda, seleccionar del desplegable uno de los tipos disponibles (previamente cargados al importar el repositorio).

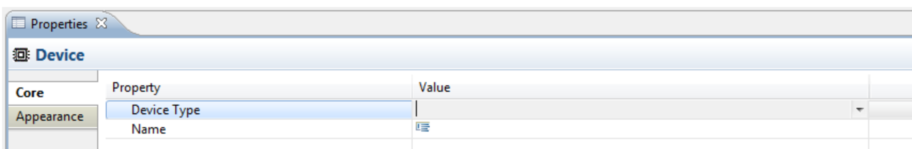


Figura 26: Propiedades de los *Device*.

Al rellenar la segunda propiedad, cambiará el aspecto del dispositivo en el modelo, mostrándose el icono y los pines correspondientes al tipo de dispositivo seleccionado. Por ejemplo, si rellenamos las propiedad del dispositivo indicando *name* = *OPENPICUS* y *DeviceType* = *GROVENEST*, el aspecto del dispositivo será el mostrado en la Figura 27.

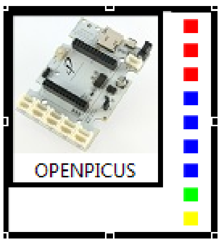
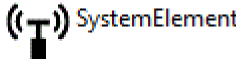


Figura 27: Aspecto gráfico de un dispositivo de tipo *GROVENEST*.

Como se puede observar en la figura anterior, los pines analógicos, digitales, I2C y SPI de este dispositivo se muestran en distinto color (rojo, azul, verde y amarillo, respectivamente), para facilitar su identificación.

Una vez añadido el dispositivo, a continuación, deberemos añadir al modelo los elementos (sensores o actuadores) que queramos conectarle. Para ello, dentro del *SystemStructure*, añadiremos tantos *SystemElement* como necesitemos. El icono por defecto asociado a este concepto del lenguaje, es el que se muestra a continuación:



Si seleccionamos cualquiera de los *SystemElement* que acabamos de añadir al modelo y mostramos sus propiedades, veremos que éstos tienen un *name* y un *SystemElementType* (ver Figura 28). Como hicimos al rellenar las propiedades del *Device,* deberemos indicar el nombre y el tipo de cada *SystemElement* del modelo. Como en el caso anterior, al cambiar el tipo asociado a cada elemento también cambiará su representación gráfica. Por ejemplo, al seleccionar *GroveTemperature* como tipo de un elemento, su representación será la que se muestra en la Figura 29. Además, el nombre del elemento (mostrado debajo de su icono) aparecerá en rojo, azul, verde o amarillo, en función de si éste es analógico, digital, I2C o SPI, respectivamente.

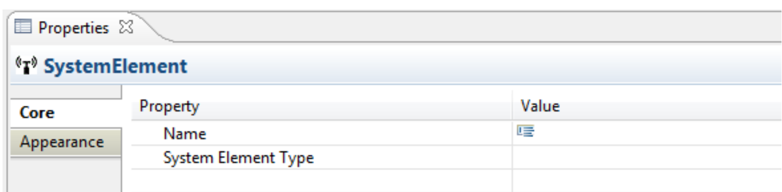


Figura 28: Propiedades de los *SystemElement.*

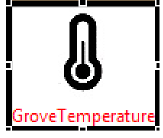


Figura 29: Aspecto gráfico de un sensor de tipo *GroveTemperature*.

Para completar el diseño de la estructura del sistema, sólo resta unir los distintos *SystemElement* con los pines correspondientes del *Device*. Para ello, seleccionaremos en la paleta de herramientas el componente *PinConnection* (cuyo icono se muestra a continuación) y, pinchando sobre uno de los *SystemElement*, arrastraremos el ratón hasta el pin correspondiente del *Device* al que queramos conectarlo.

PinConnection

El aspecto del elemento *SystemStructure*, al unir el elemento *GroveTemperature* con el primer pin analógico del dispositivo *OPENPICUS*, será el que se muestra a continuación en la Figura 30.

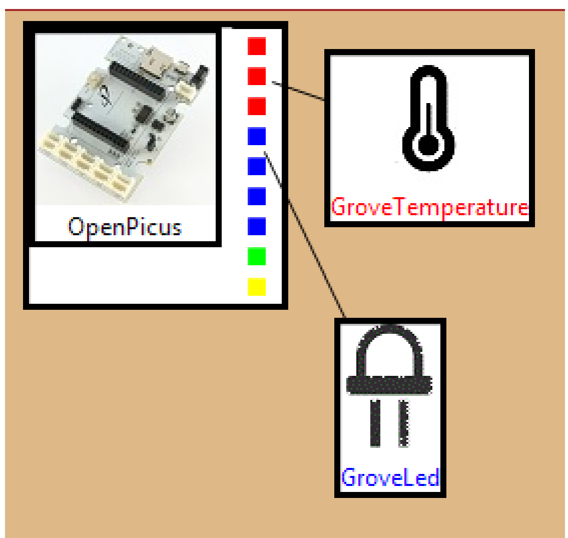


Figura 30: Aspecto del *SystemStructure* tras unir el dispositivo con el sensor.

Completado el diseño de la estructura del sistema (*SystemStructure*), ahora procederemos a diseñar su lógica (*SystemBehaviour*). En primer lugar, para cada *SystemElement* definido en la estructura, deberemos declarar dos variables: una asociada al elemento en sí (*ElementDeclaration*) y otra para almacenar los datos que éste lee o escribe, en función de si se trata de un sensor o de un actuador (*PrimitiveDeclaration*). Así, para cada SystemElement, añadiremos estos dos elementos dentro del *VarDeclaration* del *SystemBehaviour*, previamente creados en el modelo.

Tal y como se muestra en la Figura 31, para cada *ElementDeclaration* incluido en el modelo deberemos especificar sus propiedades *name* y *type*.

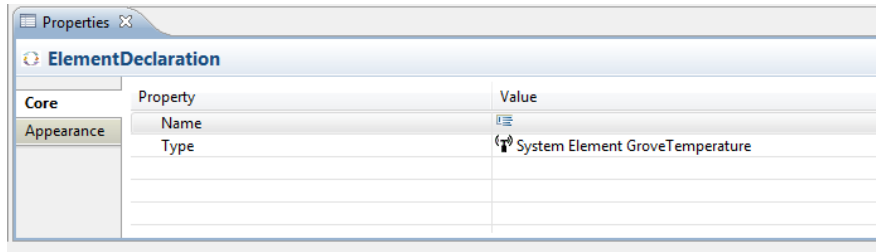


Figura 31: Propiedades variable ElementDeclaration Entorno gráfico EMDSD

De forma similar, como se muestra en la Figura 32, para cada *PrimitiveDeclaration* incluido en el modelo, deberemos especificar sus propiedades *name* y *TypePrimitiveDeclaration* (tipo primitivo de datos, que deberemos seleccionar de entre los disponibles en el lenguaje de modelado –ver los tipos recogidos en el tipo enumerado *PrimitiveType,* incluido en la Figura 6).

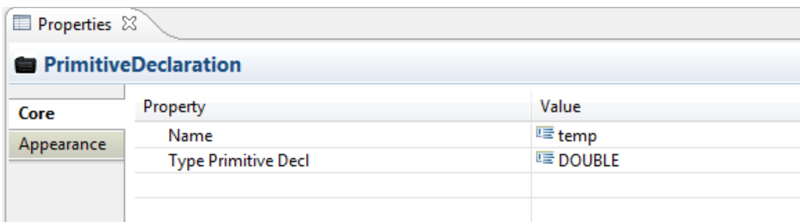


Figura 32: Propiedades variable PrimitiveDeclaration Entorno gráfico EMDSD

Una vez añadidas estas variables ya estamos en disposición de poder modelar el flujo de tareas que deberá realizar el dispositivo. La paleta de herramientas del editor ofrece una amplia gama de elementos para ello, algunos de los cuales aparecen recogidos a continuación en la Figura 33.

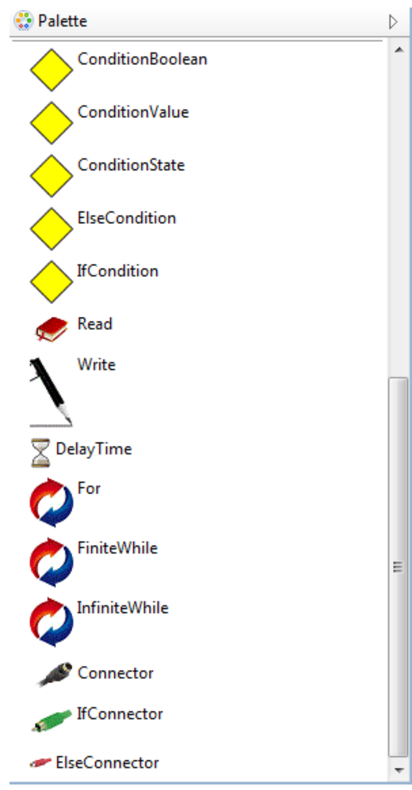


Figura 33: Elementos de programación en Entorno gráfico EMDSD

En el diseño podremos incluir bucles condicionales, bucles finitos (que terminan al alcanzar una determinada condición) y bucles infinitos. También podremos utilizar operaciones de lectura y escritura sobre los sensores y actuadores incluidos en el diseño. El flujo de ejecución de la aplicación (secuencia de tareas), vendrá determinado por cómo se enlazan mediante conectores (*Connector*, *IfConnector* o *ElseConnector*) los distintos elementos del *Workflow*.

Por ejemplo, si deseamos conseguir la siguiente programación que vamos a detallar a continuación deberemos definir lo siguiente:

```
While(1){
        If(varLuz == true)
                Set(sensorLuz, ON);
        Else
                Set(sensorLuz, OFF);
        varTemp = (double) get(sensorTemp);
}
```

Dos variables ElementDeclaration y PrimitiveDeclaration.

ElementDeclaration

- ➢ sensorLuz asociado al sensor de Luz
- ➢ sensorTemp asociado al sensor de Temperatura.

PrimitiveDeclaration

- ➢ Bool varLuz asociado al sensor de Luz.
- ➢ Double varTemp asociado al sensor de temperatura.

Y gráficamente el resultado gráfica sería parecido a lo siguiente:

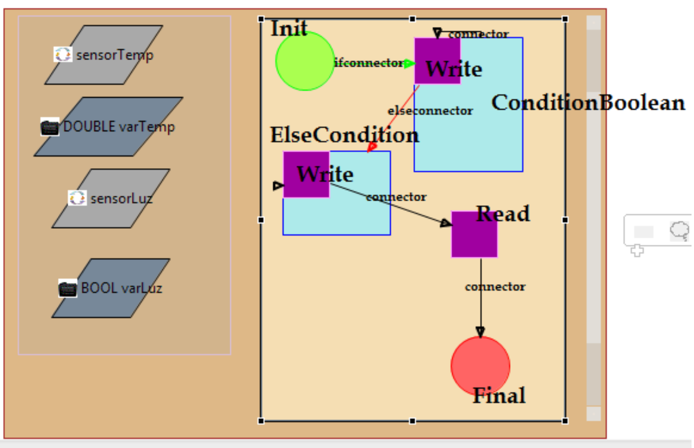


Ilustración 1: Elementos de programación en Entorno gráfico EMDSD

Para cumplir con las restricciones sintácticas impuestas por el lenguaje, deberemos tener en cuenta las siguientes consideraciones a la hora de diseñar el *Workflow*:

- A/de cada elemento del flujo de tareas debe llegar/salir un y sólo un conector, para que el flujo de ejecución sea determinista.
- Cada bucle debe contener al menos una tarea.
- Todos los elementos incluidos en el *SystemBehaviour*, tanto variables como tareas, deben tener nombres únicos.
- `IfConnector` se programa cuando el destino es una tarea `Condition` (excepto tarea `ElseCondition`) que sirven de bifurcación para la implementación del comportamiento del sistema embebido. `ElseConnector` se programa cuando el destino es una tarea `ElseCondition`, el cambio contrario a lo que marque la bifurcación justamente anterior. `Connector` se programa para el resto de conexiones de tareas.

Una vez completada la especificación del sistema con el editor gráfico y antes de utilizar el modelo para generar código a partir de él, deberemos validar su corrección sintáctica, esto es, comprobar si es conforme a la sintaxis definida tanto en el meta-modelo como en las restricciones OCL añadidas en él. Para ello tenemos dos posibles opciones:

1) ***Desde el editor gráfico***, podemos validar el modelo en cualquier momento con solo seleccionar la opción del menú: ***Edit →Validate*** (ver Figura 34).

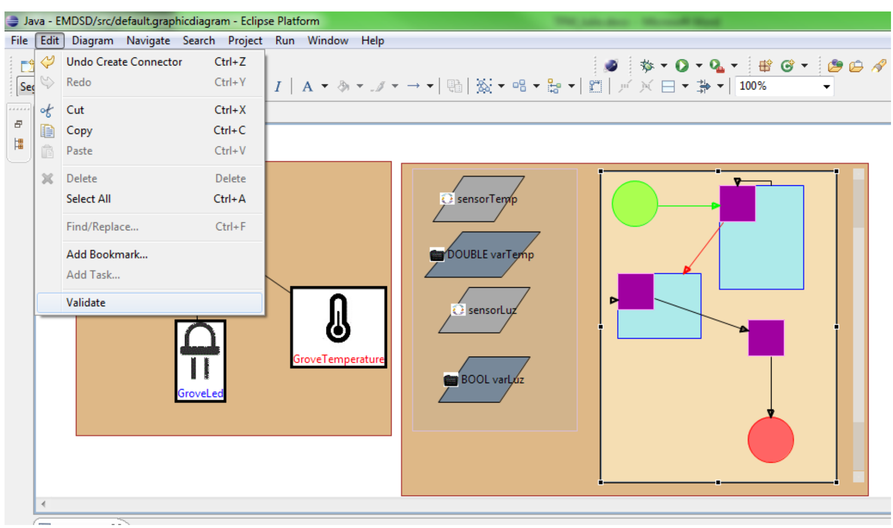


Figura 34: Elementos de programación en Entorno gráfico EMDSD

2) ***Desde el editor de modelos en forma de árbol***, podemos abrir el modelo serializado (fichero *.xmi*) y pinchando con el botón derecho del ratón sobre el elemento raíz del árbol con el que se representa el modelo, seleccionar la opción *Validate*.

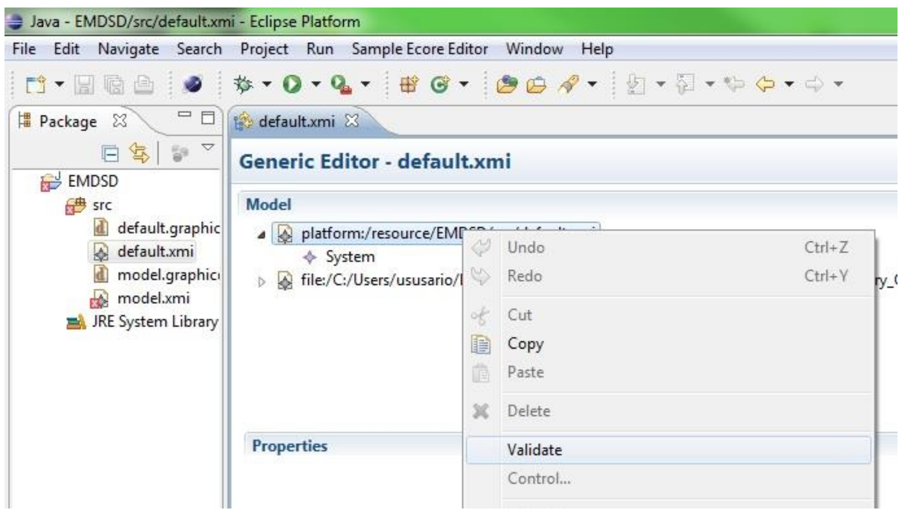


Figura 35: Validación desde el elemento root en Entorno gráfico EMDSD

4.3.2. Motor de generación de código

Una vez diseñado y validado el modelo del sistema, deberemos copiar su versión serializada (fichero *.xmi*) en el proyecto XPAND que contiene el motor de generación de código. Para ello, deberemos hacer lo siguiente: (1) localizar el fichero XMI en el proyecto Java creado en el Eclipse donde se ha ejecutado el editor gráfico; y (2) moverlo a la carpeta */src* del proyecto XPAND llamado EMDSD.project (ver Figura 36), disponible en el Eclipse donde inicialmente se importaron los proyectos incluidos en el TFM.

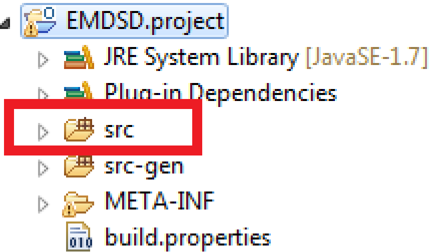


Figura 36: Estructura del proyecto XPAND. Carpeta donde se almacenan los modelos XMI de partida.

El motor de generación de código ya debería estar correctamente configurado (ver sección 4.1) por lo que, para ejecutarlo, sólo debemos seleccionar la opción del menú: *Run → Run Configurations → MWEWorkflow → Code_Generator → Run.*

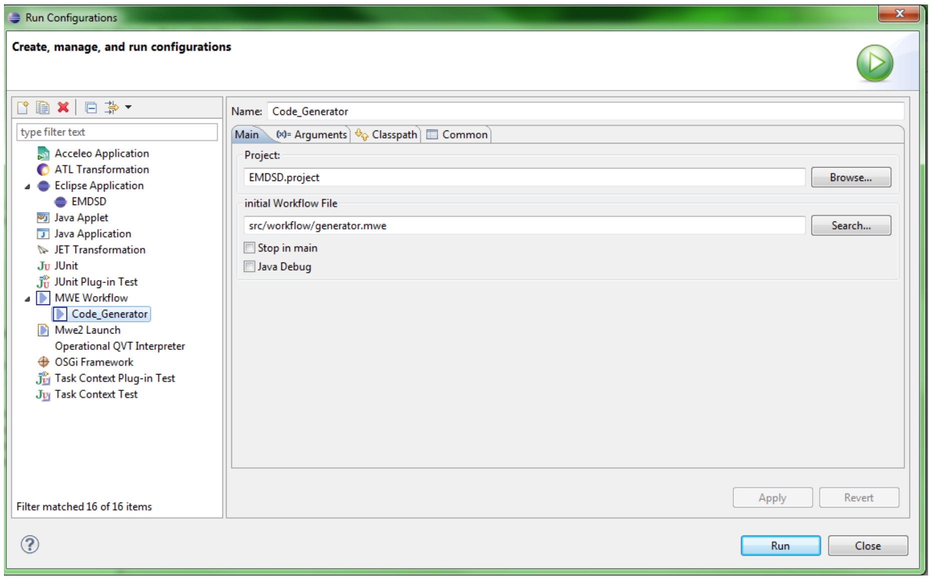


Figura 37: Ejecución del motor de generación de código.

Una vez terminado el proceso de generación de código, los ficheros resultantes aparecerán en la carpeta */src-gen* del proyecto XPAND (ver Figura 38).

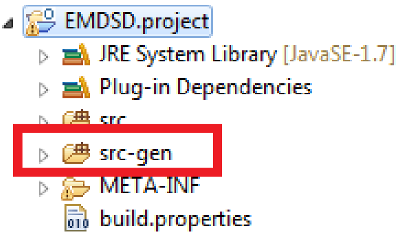


Figura 38: Estructura del proyecto XPAND. Carpeta donde se almacenan los ficheros de salida.

En la siguiente sección se detalla un ejemplo completo desarrollado con el entorno EMDSD. En él, además de comentarse los modelos desarrollados y cómo éstos se utilizan para generar el código correspondiente, se explica cómo ejecutar la aplicación resultante en la plataforma de destino seleccionada.

4.4. Ejemplo completo para OpenPicus

El ejemplo que se documenta en este apartado y a partir del que surgió la idea de desarrollar este Proyecto fue inicialmente planteado como un trabajo de la asignatura *Sistemas de Información* del Máster de Ingeniería en Informática, impartido en la EPCC de la UEx. El ejemplo consistía en programar un sensor digital Led y un sensor analógico Temperatura conectados a un dispositivo OpenPicus GroveNest. Para ello se utilizó el entorno *OpenPicus Flyport IDE* v 2.3 [39].

A continuación se explica cómo desarrollar este mismo ejemplo utilizando el entorno EMDSD, desarrollado como parte de este Proyecto.

En primer lugar tendremos que abrir el editor gráfico para crear un modelo de la estructura del sistema (SystemStructure). En este modelo tendremos que añadir los siguientes elementos:

- *Device* – Grovenest (name: OpenPicus)
- System Element – *Digital Sensor* – Sensor Led (name: GroveLed)
- System Element – *Analog Sensor* – Sensor Temperature (name: GroveTemperature).

Además, deberemos conectar cada sensor con su pin correspondiente:

- Sensor Led – *Pin Digital* – DIG1
- Sensor Temperature – *Pin Analog* – AN1

También deberemos conectar cada sensor con el pin correspondiente del dispositivo mediante sendos PinConnections:

- *Pin Connection 1* – Pin DIG1 – Digital Sensor
- *Pin Connection 2* – Pin AN1 – Analog Sensor

El resultado de esta serie de declaraciones y conexiones al dispositivo GroveNest en el entorno EMDSD podemos observarlo en la Figura 30.

En segundo lugar tendremos que modelar la lógica del sistema (SystemBehaviour), añadiendo los siguientes elementos:

- *Declaración de variables primitivas* (permiten leer/escribir datos de cada sensor/actuador)
  - Bool varLuz (permite saber si el Led está encendido o apagado).
  - Double varTemp (permite obtener la temperatura del sensor).

- *Declaración de variables elementales* (asignadas a cada sensor)
  - sensorLuz (asignado al sensor digital Led).
  - sensorTemperature (asignado al sensor analógico Temperatura).

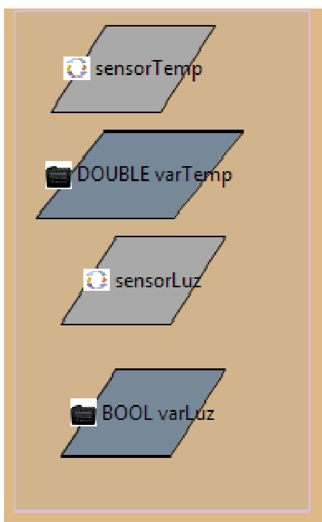


Figura 39: Entorno gráfico EMDSD – Diseño del SystemBehaviour – Declaración variables.

- *Diseño de la lógica del sistema.* La secuencia de tareas en pseudocódigo sería la siguiente:

```
// declaración e inicialización de variables primitivas

Bool varLuz = false;
Double varTemp;

// creación de dispositivo y los sensores creados
// asignar los sensores al dispositivo

Inicio
While (1) → bucle se programa solo
      If(varLuz == TRUE) → Condition Boolean
            Escritura sensorLuz encender
      Else → Else Condition
            Escritura sensorLuz apagar
```

```
        Fin If
        Lectura sensorTemperatura
Fin While
Fin
```

El modelo gráfico correspondiente a este pseudocódigo sería el mostrado en la Figura 40:

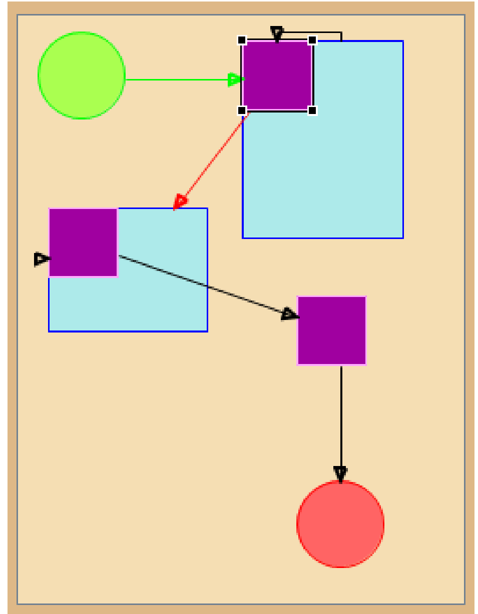


Figura 40. Entorno gráfico EMDSD – Diseño del SystemBehaviour –Workflow de tareas.

El modelo serializable que se obtiene tras completar la descripción de la estructura y la lógica del sistema es el siguiente:

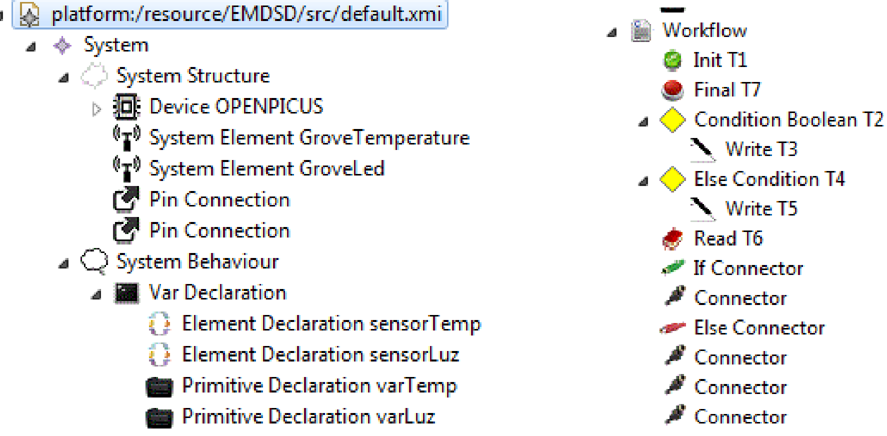


Figura 41: Entorno gráfico EMDSD – Modelo Serializable .xmi generado.

A continuación, deberemos generar la implementación asociada al modelo anterior utilizando el motor de generación de código implementado. Para ello debemos moverlo a la siguiente ruta dentro del proyecto Eclipse:

- Ruta: EMDSD.project/src/

Seguidamente, debemos arrancar el plugin de generación de código que tenemos disponible a través de la opción correspondiente de la barra de herramientas:

- Opción: Windows – Run Configurations – Code Generator.

Por último debemos recoger los siguientes ficheros generados en la siguiente ruta

- Ruta: EMDSD.project/src-gen/
- Ficheros: **taskFlyport.c**, **status.xml** y **HttpApp.c**

Una vez generados estos ficheros podremos utilizarlos en el entorno *OpenPicus Flyport IDE* v 2.3 (utilizado para desarrollar el ejemplo inicial) del siguiente modo. Una vez abierto el IDE, crearemos un proyecto OpenPicus GroveNest con Web Server (ver Figura 42), ya que queremos poder interactuar con la Web. La elección de esta opción hará que se genere un fichero status.xml (ver Figura 44).

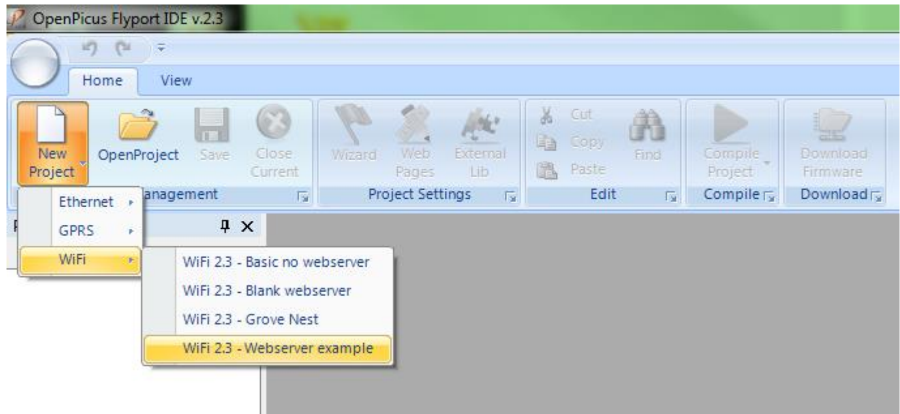


Figura 42: Creación de un proyecto OpenPicus GroveNest Web Server en IDE OpenPicus Flyport

Una vez creado el proyecto, iremos al directorio donde lo hayamos guardado y tendremos la siguiente estructura de directorio (ver Figura 43):

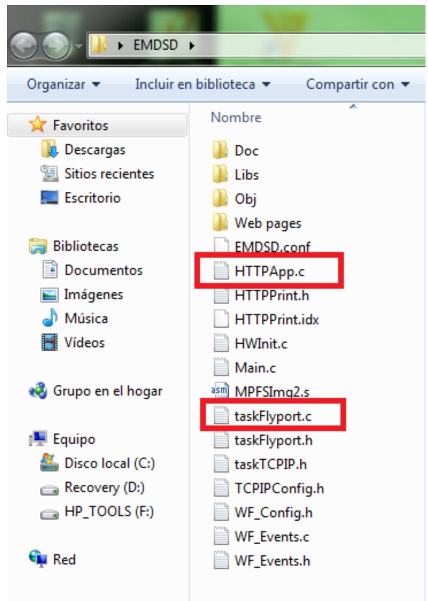


Figura 43: Estructura de un proyecto OpenPicus GroveNest en IDE OpenPicus Flyport.

En este directorio deberemos sustituir los ficheros **taskFlyport.c** y **HttpApp.c** (resaltados en rojo en la Figura 43), previamente generados en el entorno EMDSD. Del mismo modo, en la carpeta "Web pages" del proyecto (ver Figura 44) sustituiremos el fichero **status.xml** por el obtenido previamente por el motor de generación de EMDSD.

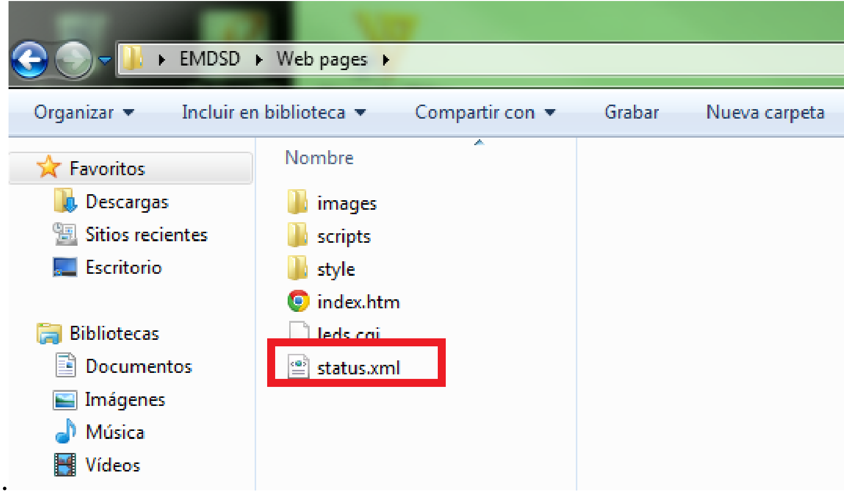


Figura 44: Directorio "Web pages" en un proyecto OpenPicus GroveNest Web Server.

Por último, deberemos mover las librerías asociadas a los sensores al directorio raíz del proyecto. Para ello, utilizaremos el fichero de procesamiento por lotes descrito en la Sección 3.5, seleccionando las siguientes opciones:

- Opción 1: Buscar los sensores de los que debemos descargar las librerías.
- Opción 2: Moverlos a la ruta raíz del proyecto OpenPicus desarrollado.

Capítulo 5. Conclusions and future work

In this chapter the project conclusions and some possible future work lines that could be used in the future, as a continuation of this work are narrated.

5.1. Conclusions

Currently, the increasing presence of embedded systems products and services offers a great opportunity for the development of all kinds of applications based on them. It has been estimated that 99 % of world production of microprocessors currently used to implement embedded systems, these being virtually invisible to consumers. According to the analyzing firm IDC, by 2015 the intelligent embedded systems will reach one billion units [40]. Also in numerous articles, we can find information about evolution [41], trends [42] and impact [43] embedded systems planned in the near future.

The addition of an embedded system in a particular product provides a significant added value, that allows distinguish it from competing products. This is possible because the solutions based on embedded systems are generally cheap, easily configurable, accurate and fast.

At present, the main barriers to more widespread use of embedded systems, both in universities and in business or technology, are derived from:

- The lack of interaction between the different existing platforms, hindering the development of cooperative solutions to achieving further progress in this field.
- The poor relationship between the various sectors that use these platforms, which would create a common bond to achieve common goals by enhancing interdisciplinary work groups.
- The limited transfer of knowledge about the advances being made in this field for industry, businesses, universities and research centers.

- The tendency to use embedded systems to develop ad- hoc applications, little or no reusable.
- Specifically, in Spain, university education is not based on development embedded systems specialized.

The development of this TFM has helped to achieve several goals, some of them related to overcoming some of these barriers. Among them are the following:

- We have developed a useful modeling language for specifying the structure and logic of embedded systems. Based on this language, it has developed a graphical editor that makes it easy for designers to create and model validation of their systems. We have implemented a code generation engine that allows obtain, from the previous models, the corresponding implementation for embedded systems based on OpenPicus platform. The set of all these tools is called EMDSD.

- The EMDSD environment, developed as part of this TFM, demonstrates the power and usefulness of a DSDM based on the domain of embedded system approach. Among other advantages , besides allowing to shorten development times and generally improve the quality of the developed software, the use of models increases the level of abstraction with which the systems are designed, making it possible to obtain different implementations (for example, for different embedded systems) from the same models. It also enables developers, not necessarily experts in low-level programming, make their, closer to his way of thinking that want to deploy applications using primitive designs (graphic) high-level languages (textual), typically they used to program these systems.

- The ease of learning and use of the tools included as part of the environment makes EMDSD be considered as a tool to support teaching in courses on design and programming of embedded systems.

Among the main difficulties we have encountered throughout the development of this TFM included the lack of documentation and examples related to the use of Eclipse plugins for DSDM. It should be noted, by way of exception, the plugin XPAND, which is not only extremely easy to use but also is quite well documented.

5.2. Future work lines

There are some extensions of EMDSD environment in this chapter that they would be interesting to use in the future.

The current version of EMDSD environment, developed as part of this TFM, can only generate code for OpenPicus platform. To improve usability would be interesting to extend: (1) the definition repository to include new devices, sensors and actuators; (2) Modeling Language to include new modeling primitives, in particular, to specify the logic of the embedded system; and (3) the code generation engine to allow the implementation of applications for other platforms.

In addition, the integration of the tools included in the EMDSD environment is currently manually. The designer must manually start both the graphical editor such as code generation engine and must move multiple files from one directory to another so that the tools work properly. Therefore, to improve the usability of the environment, you should automate the integration of all tools, offering the designer context menus or specific settings within Eclipse menus that allow you to perform, more simple and intuitive way, the different tasks process.

Another aspect that could be improved, especially in graphics editor, has to do with the inclusion of new extensions in the code generated by GMF (in addition to those already included detailed in Annex III). In this sense, you could add new extensions for each new graphics model that was created, is automatically generated (and the deletion was avoided) of all mandatory elements (similar to how and device pins are added from the definition of this type).

For example, to create a new model could automatically add the *SystemStructure and SystemBehaviour* elements. Within the first one you could automatically add a device, and in the second one *VarDeclaration and Workflow*. Within the latter , could be added marks the beginning and end of the workflow, for example the elements: *Init and Final* . Similarly, for each new *SystemElement* (sensor or actuator) that is added to SystemStructure, should automatically *PrimitiveDeclaration and ElementDeclaration* add the corresponding within *VarDeclaration* of *SystemBehaviour*. All these elements should disappear from the tool palette of graphic editor, because the designer does not need to manually add them to the model.

Bibliography

[1] T. Stahl, M. Voelter and K. Czarnecki, Model-Driven Software Development: Technology, Wiley, 2006.

[2] S. V. N. S. J. &. S. A. Hodges, A new era for ubicomp development., Pervasive Computing, IEEE, 11(1), 5-9, 2012.

[3] D. C. Schmidt, Model-Driven Engineering, vol. 39 (2), IEEE Computer, 2006, pp. 25-31.

[4] T. Strasser, M. Rooker, I. Hegny, M. Wenger, A. Zoitl, L. Ferrarini, D. A. and M. Colla, A Research Roadmap for Model-Driven Design of Embedded Systems for Automation Components, Proceedings of the 7th IEEE International Conference on Industrial Informatics, Cardiff, Wales, United Kingdom, 2009.

[5] Medeia, "Model-driven Embedded System Design Environment for the Industrial Automation sector," [Online]. Available: http://www.medeia.eu/.

[6] Quasimodo, "Quantitative system properties in model-driven design of embedded systems," [Online]. Available: http://www.quasimodo.aau.dk/.

[7] C. Vicente-Chicote, F. Losilla, B. Álvarez, A. Iborra and P. Sánchez, Applying MDE to the Development of Flexible and Reusable Wireless Sensor Networks, vol. 16 (3/4), International Journal of Cooperative Information Systems, Special Issue: Software Architecture — Towards the Software Engineering Core, 2007, pp. 393 - 412.

[8] OpenPicus, "Guide Programmer in OpenPicus," [Online]. Available: http://www.iiitd.edu.in/~amarjeet/EmSys2013/FLYPORT%20Programmer's%20Guide%202.2%20release%201.0.pdf.

[9] J. García Molina, F. O. García Rubio, V. Pelechano, A. Vallecillo, J. M. Vara and C.

Vicente-Chicote, Development of Model-Driven Software Concepts , Methods and Tools, RA-MA, 2013.

[10] L. Vanguardia, "The home of the future , smart homes controlled through smartphone," 2014. [Online]. Available: http://www.lavanguardia.com/tecnologia/mobile-world-congress/20140227/54402591229/hogar-futuro-casas-inteligentes-controladas-por-smartphone.html.

[11] OpenPicus, "Official Web OpenPicus," [Online]. Available: www.openpicus.com.

[12] Eclipse Foundation, "Eclipse Modeling Project," [Online]. Available: http://www.eclipse.org/modeling.

[13] Eclipse, "Eclipse Modeling Framework," [Online]. Available: http://eclipse.org/modeling/emf/.

[14] OMG, "Meta Object Facility," [Online]. Available: http://www.omg.org/mof/.

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

[16] Eclipse Foundation, "Eclipse Graphical Modeling Framework," [Online]. Available: http://eclipse.org/modeling/gmp/.

[17] SmartCitizen, "Smart Citizen Project," [Online]. Available: http://www.smartcitizen.me/.

[18] Telefónica, "X10RP Project," [Online]. Available: http://catedratelefonica.unex.es/?p=4775.

[19] IEEE, "IEEE Xplore Digital Library," [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6726199&tag=1.

[20] Arduino, "Official Web Arduino," [Online]. Available: http://www.arduino.cc/es/.

[21] Tinkerkit, "Official Web Tinkerkit," [Online]. Available: http://www.tinkerkit.com/.

[22] Seeduino, "Official Web Seeduino Grove," [Online]. Available: http://www.seeedstudio.com/wiki/GROVE_System.

[23] OpenPicus, "Official Wiki OpenPicus," [Online]. Available: http://wiki.openpicus.com.

[24] Eclipse Foundation, "Eclipse-Modeling-Framework," [Online]. Available: http://eclipse.org/modeling/emf/.

[25] Eclipse, "Graphical Edition Framework," [Online]. Available: https://eclipse.org/gef/.

[26] Eclipse, "Open Source Modeling – Generating Code with XPAND," [Online]. Available: http://www.eclipsecon.org/2008/sub/attachments/Code_Generation_with_M2Ts_Xpand.pdf.

[27] Eclipse, "Open Architecture Ware," [Online]. Available: http://www.voelter.de/data/articles/oAWArticleEM.pdf.

[28] Eclipse, "Xtend," [Online]. Available: http://eclipse.org/xtend/.

[29] Internacional Journal of Systems Science, "Engineering the development of systems for multisensory monitoring and activity interpretation," 3 Abril 2013. [Online]. Available: https://investigacion.uclm.es/documentos/fi_1394035084-00207721.2013.779048.pdf.

[30] WebRatio, "Official Web WebRatio," [Online]. Available: http://www.webratio.com/.

[31] University of Extremadura, *TFG - Modeling HuRoME + Environment,* Cáceres, Cáceres, 2011.

[32] Robonova, "Robot, Supernova," [Online]. Available: http://www.superrobotica.com/robonova.htm.

[33] University of Extremadura, *TFG - Real -A: A RIA approach to Augmented Reality on mobile using MDD,* Cáceres, Cáceres, 2011.

[34] Eclipse, "OCLInEcore," [Online]. Available: https://wiki.eclipse.org/OCL/OCLinEcore.

[35] Fixed exercises OCL Restrictions, "Software Engineering II - Exercises OCL," [Online].

[36] Eclipse, "Official Web Eclipse," [Online]. Available: https://eclipse.org/.

[37] Wget, "Wget, Command Line," [Online]. Available: http://www.gnu.org/software/wget/.

[38] 7-zip, "Utility tool file archiver," [Online]. Available: http://www.7-zip.org/.

[39] OpenPicus, "Good Practice Guide," [Online]. Available: http://wiki.openpicus.com/index.php/Getting_Started.

[40] I. Systems, "The Next Opportunity," 2011.

[41] Electrónicos.Online.Com, "Electronic Online," [Online]. Available: http://www.electronicosonline.com/2012/11/20/la-evolucion-de-los-sistemas-embebidos-inteligentes/?pagina=1.

[42] T. a. T. Ministry of Industry, "Trends and applications of Embedded Systems in Spain," [Online]. Available: http://www.opti.org/publicaciones/pdf/texto131.pdf.

[43] Mi+d, "MadriMasd," [Online]. Available: http://www.madrimasd.org/informacionIdi/analisis/opinion/opinion.asp?id=42611.

Annex I. Implementation of EMDSD enviroment

This chapter presents all the steps we have taken to develop various tools related to this project.First described the steps taken to design and configuration meta-model: the central element of this project.

Then we will focus on the process used to design and configure the graphic editor models built from previous meta-model.

The purpose of this chapter is not so much writing a "How-To" of the tools used within the Eclipse Modeling Project (EMP) for this project, such as providing an overview on which to assess the advantages and limitations of an approach DSDM using EMP.

I.1. Abstract syntax and EMF models editor

As already discussed in Section 3.2, the abstract syntax of the modeling language defined as part of EMDSD environment described using a EMF meta-model with two distinct parts: one to specify the structure of the embedded system (see Figura 5) and other to specify the logic of the embedded system (see Figura 6). In between the elements of that EMF it provides for defining meta-models, we have used the following:

- *EClass*: Each EClass is a meta- class or term of language.

  EClass

- *EAttribute*: Properties associated in each meta-classes of language.

  EAttribute

- *Ereference*: Associations between meta-class.

  EReference

  Each association has a Boolean attribute *IsContainment* (available in the properties tab), which indicates whether the elements of the meta-class marked how target association are contained or not the elements of the meta- class origin indicated as source the same.

  Is Containment

- *Inheritance*: Inheritance between meta- classes.

Inheritance

- *EEnum*: Definition of enum types.

EEnum

As indicated in section 3.2, the abstract syntax described by EMF meta- model needs to be supplemented by the inclusion of several OCL restrictions. For this we have used the OCLInEcore tool that it allow to edit the EMF meta- models in a textual format to add invariants associated meta- class or some of its attributes. The restrictions added to our meta-model, as discussed in Section 3.2.3, below are listed in Annex II of this memory.

Once complete the definition of the abstract syntax of the language, EMF will generate a simple model editor in tree (*Tree Editor*) from it. To do this, we create a model EMF code generation (*.genmodel* file) from the meta- model (*.ecore* file), selecting in the wizard that appears to create a new file (Menu: *File* → *New*), option **EMF Generator Model** (see Figure 45). We indicate that the starting meta- model is Ecore format (see Figure 46) and save the resulting file in the same folder as the meta- model.

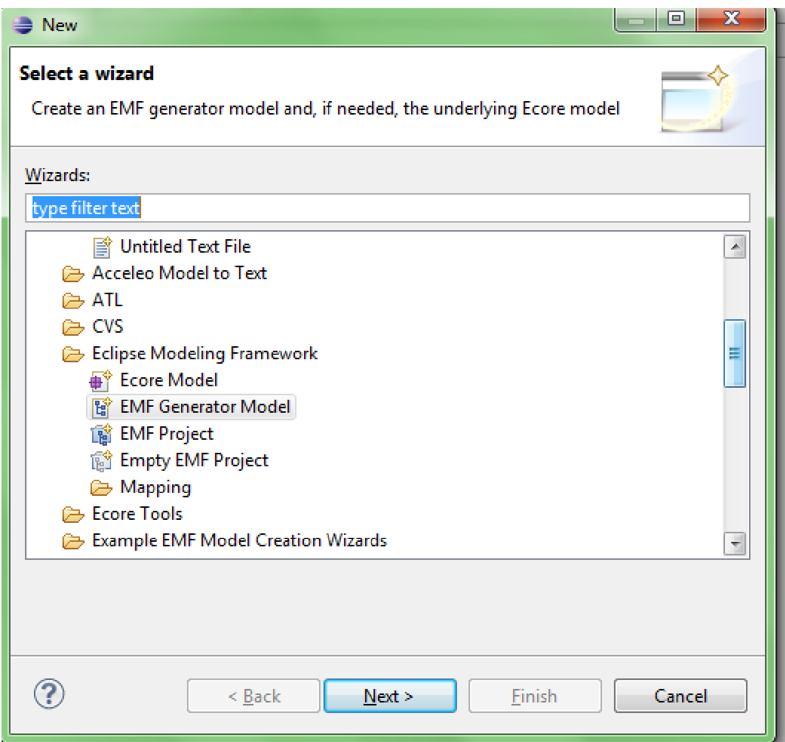


Figure 45: Creating an EMF Generator Model from meta-model.

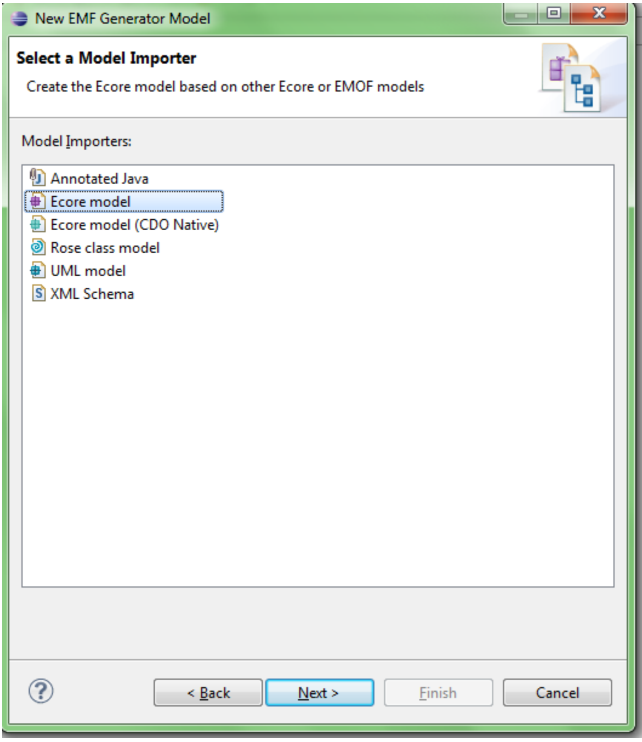


Figure 46: Selecting the type of meta- model to create the EMF Generator Model.

Once created the file, we open it to change some of the parameters defined in it:

- ***Compliance Level***: 6.0 to validate the constraints defined in the OCL meta-model.
- ***Copyright Fields***: true to have our code to get licensed.
- ***Copyright Text***: The text that gives you license our code.

To do this, we use the Eclipse Properties view (see Figure 47).

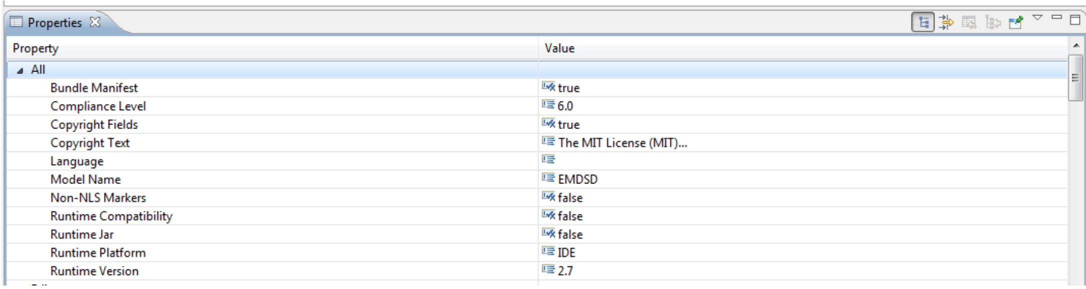


Figure 47: Configuration parameters EMF Generator Model

Once finalized the configuration of file *.genmodel* to generate the Java code editor tree models from it, we will select the "root" model, by clicking on it with the right mouse button and selecting item "Generate All" (see Figure 48).

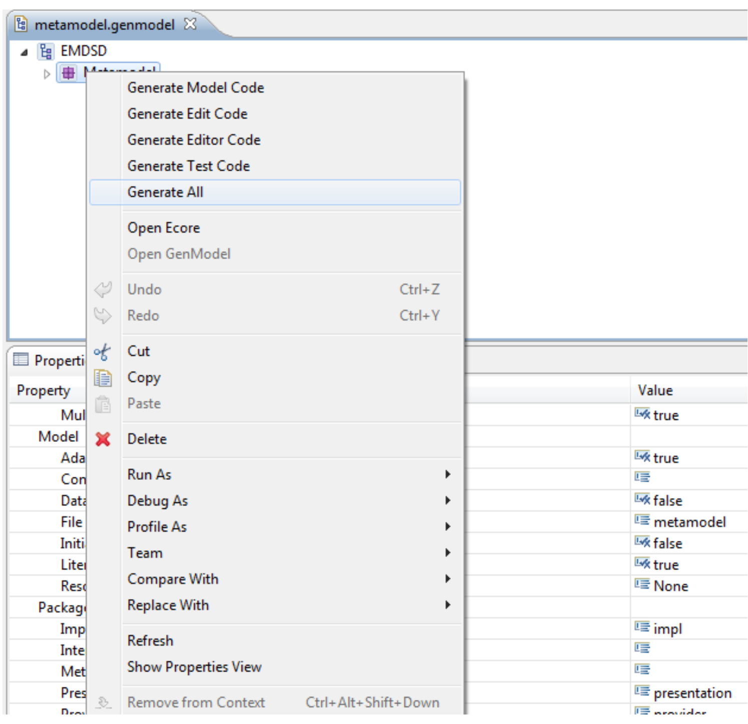


Figure 48: EMF generation of Java code associated with the model editor in tree.

Once completed the process of code generation we can see that it has created, within the main project, the *src* folder (with associated elements of Java code meta- model) and two new folders: *.edit* and *.editor* (with the Java code associated with the model editor in tree form). To run this editor we launch a new Eclipse using the *Run* → *Run Configurations* → *Eclipse Application* menu. In the new Eclipse we must create a new Java project (Menu: *File* → *New* → *Java project*). Within this project we can create models in a tree, meet our meta-model, using the menu: *File* → *New* → *Other* → *Example EMF Model Creation Wizards*. At design time we can check the syntactic validity of our model by selecting the menu: *Edit* → *Validate*.

To create models in a tree from the meta-model there is an alternative procedure that does not require previous EMF editor. This procedure consists of: open the meta-model (*.ecore* file); select the item you want to be the root (root) of our model; click with the right mouse button on it and select the "*Create dynamic instance*" option (see Figure 49). This will create in the local project folder that you indicate, a model format *.xmi* similar characteristics to those that can be created with the *tree-editor* generated by EMF.

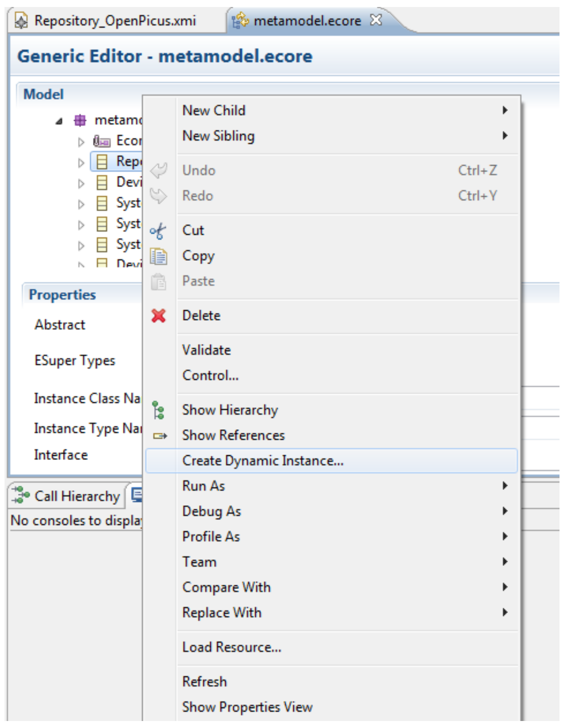


Figure 49: Repository model creation for EMDSD environment.

The last is the procedure we followed to create the model repository in our EMDSD environment (see Figure 50).

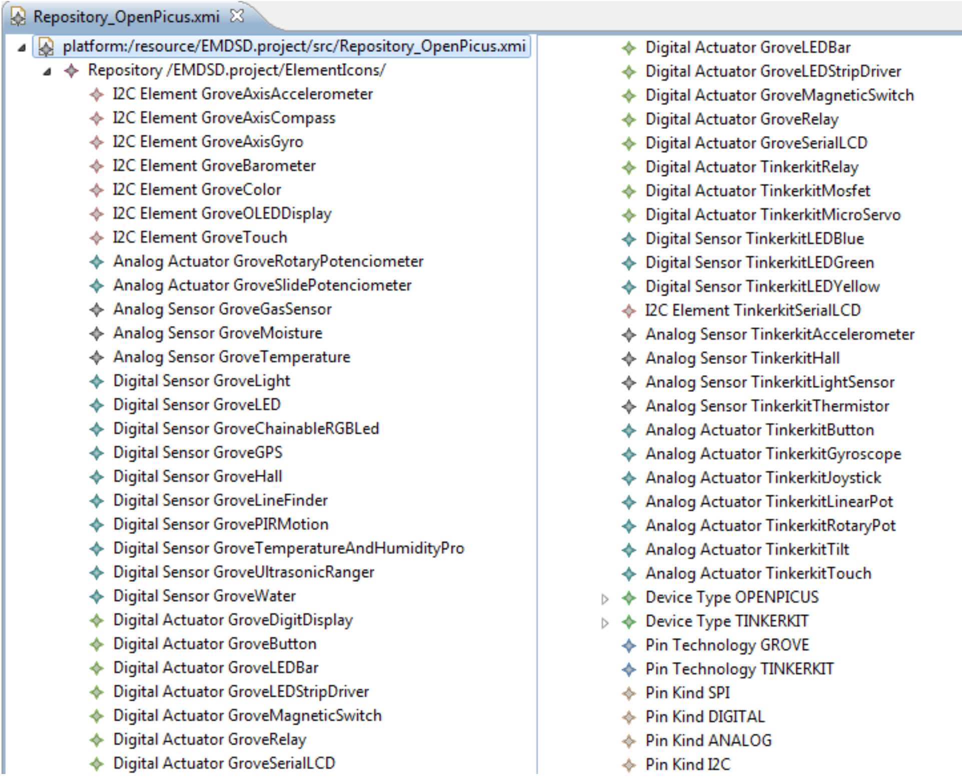


Figure 50: Definition Repository of EMDSD enviroment

- Device Type OPENPICUS
  - Pin Type AN1
  - Pin Type AN2
  - Pin Type AN3
  - Pin Type DIG1
  - Pin Type DIG2
  - Pin Type DIG3
  - Pin Type DIG4
  - Pin Type I2C
  - Pin Type SPI
- Device Type TINKERKIT
  - Pin Type ANALOG
  - Pin Type DIGITAL
  - Pin Type TWI
  - Pin Type SERIAL

Figure 51: Detailed modeling of the two devices included in the previous repository.

As can be seen in the two previous figures, the model repository contains the definition of all sensors and actuators, as well as of available devices, with a corresponding pin configuration (name and technology). The *path* of the root element of this model (*Repository*) contains attributes the directory path in which the icons associated with each of these items are stored. This path is relative to the project, this is the address: /EMDSD.project/

I.2. Graphic syntax and GMF model editor

As was said earlier, GMF is one of the plug-ins included in EMP and offers a number of facilities to implement graphics editors models from EMF meta-models (*.ecore* files). To do this, the designer must specify: (1) a model that defines the visual appearance (concrete graphical syntax) associated with each modeling concept included in the meta-model (*.gmfgraph* file); (2) a model that defines the elements that appear in the tool palette of graphic editor (*.gmftool* file); and (3) a model to define, univocally, the relationship between each element in the meta-model with its corresponding graphical representation and tool palette (*.gmfmap* file). It should be noted that the implementation of graphic editors with GMF requires prior generation associated with EMF editors code.

Next the steps to follow to define each of the three models needed to create the graphical editor with GMF plugin.

<u>Creation .gmfgraph file:</u>

We select option of the menu: *File* → *New* → *Other* → Graphical *Modeling Framework* → *Simple Graphical Definition Model* (see Figure 52).

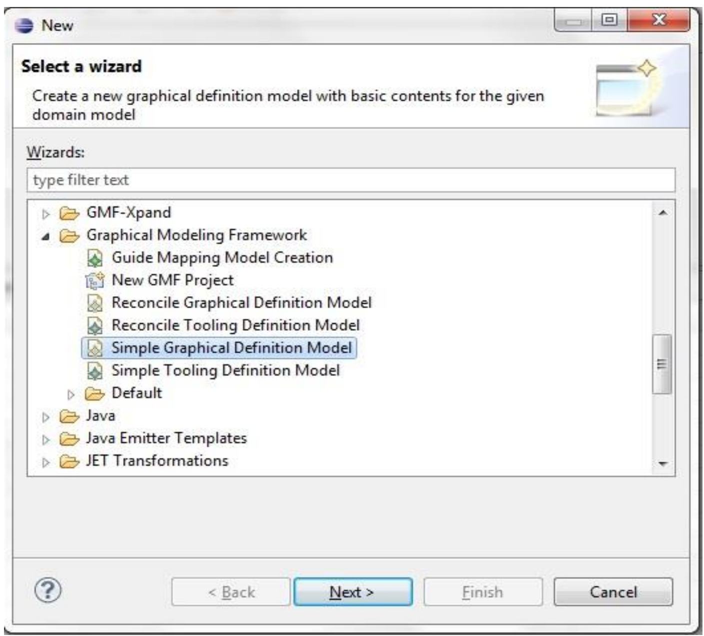


Figure 52: Creation *.gmfgraph* model associated with the EMDSD graphic editor .

GMF will create a default model in which partner: a node (*Node*) each EClass rectangular shape; a label (*Diagram Label*) to each attribute of each EClass; and a connection (*Connection*), type polyline, each EAssociation the meta-model. Graphic designer editor may change the appearance of these elements at will to suit the needs of end users modeling editor. In addition, you must add compartments (*Compartment*) to those elements that are to contain graphically others. It should be noted that the graph contention between elements is determined by the partnerships with existing contention between corresponding EClass defined in the meta-model.

After completing this process, the appearance of the graphical editor model *.gmfgraph* our EMDSD environment is shown in the Figure 53.

<u>Creation .gmftool file:</u>

We select option of the menu: *File* → *New* → *Other* → Graphical *Modeling Framework* → *Simple Tooling Definition Model* (see Figure 54).

Again, GMF will create a default model in which will include an entry in the tool palette for each *EClass* and *EAssociation* the meta-model. In most cases it is not necessary to change the model generated by GMF except perhaps to reorder or group items in the palette to suit the designer. The *.gmftool* model generated for the graphic editor EMDSD our environment is shown in the Figure 55.

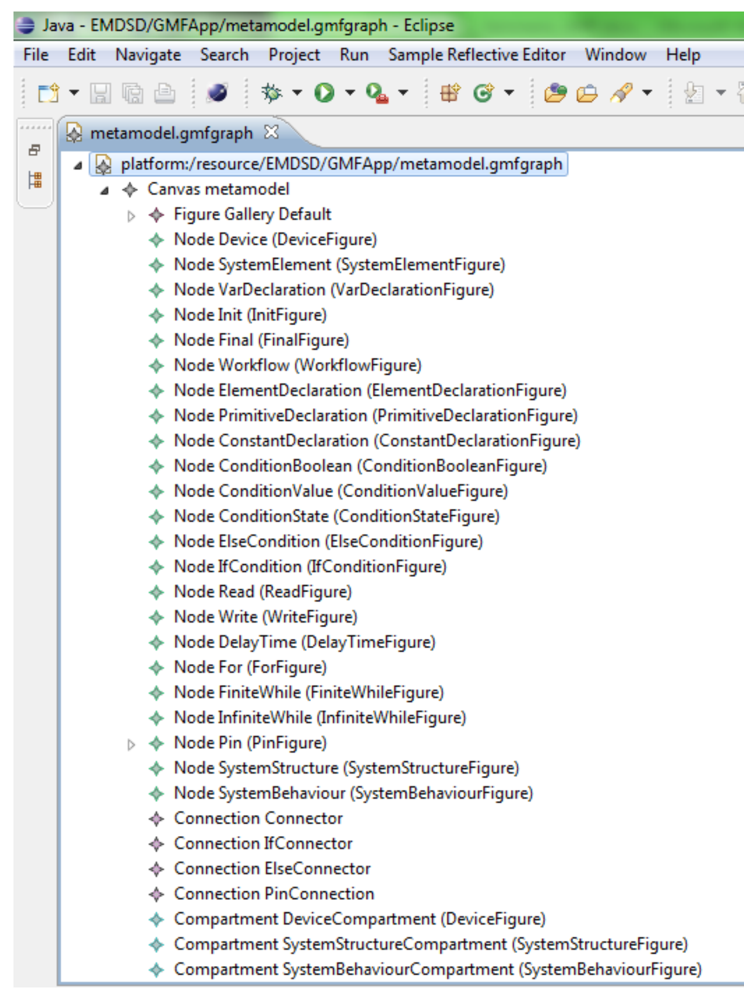


Figure 53: File *.gmfgraph* associated with the EMDSD enviroment graphic models editor.

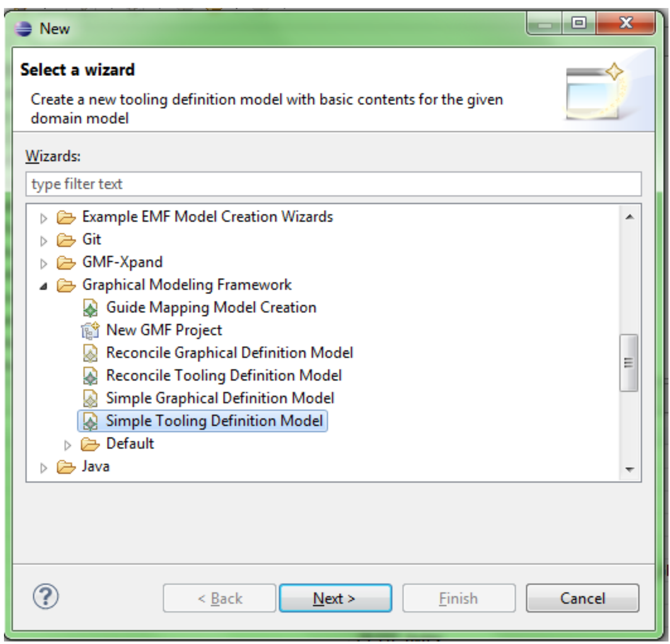


Figure 54: Created *.gmftool* models attached in EMDSD enviroment graphic editor.

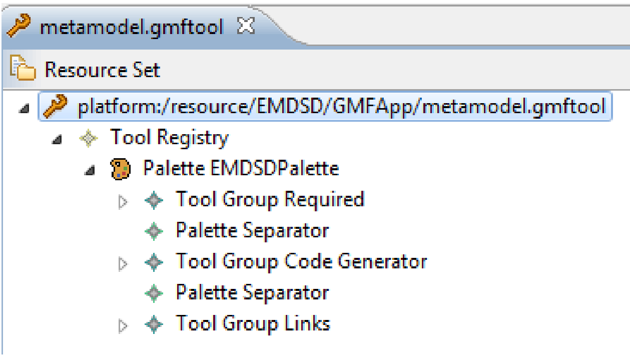


Figure 55: File .*gmftool* attached in EMDSD enviroment graphic editor.

Creation .gmfmap file:

We select option of the menu: *File* → *New* → *Other* → Graphical *Modeling Framework* → *Guide Mapping Model Creation* (see Figure 56).

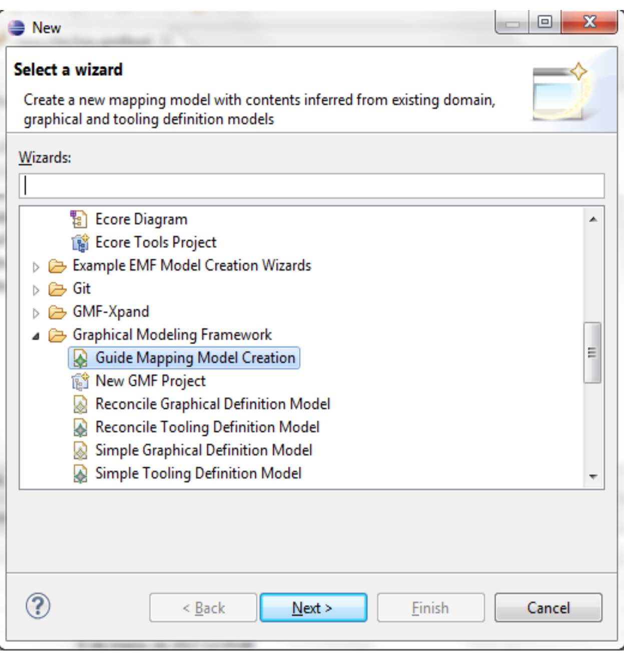


Figure 56: Created .*gmfmap* model linked to the EMDSD enviroment graphic editor.

Since this model establishes a relationship between each meta-model concept (.*ecore* file) with its associated graphical syntax (.*gmfgraph*) and the element of the tool palette necessary for its creation (.*gmftool*), GMF will ask us to we indicate the location of these three files to create the file .*gmfmap*. Once this is done, GMF will suggest a possible model configuration (nodes and connections) that we will review carefully and very likely change. The model will generate GMF .*gmfmap* default so we carefully review and complete, ensuring that all associations are defined between elements of the other three models and that it is correct. The appearance of the file .*gmfmap* end, obtained after all checks and appropriate modifications, is shown in Figure 57.

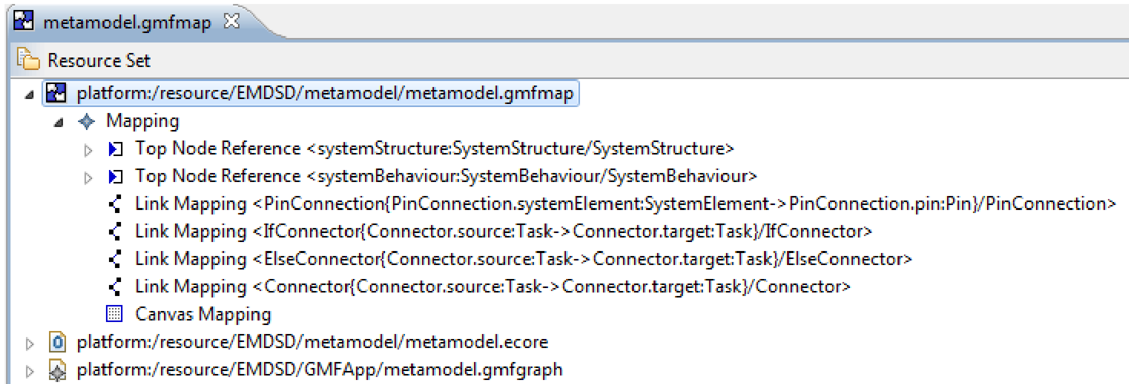


Figure 57: File *.gmfmap* associated with the EMDSD environment graphic editor

Once the definition of the three previous models is complete, we will create the GMF generator Editor (*.gmfgen* file) editor similar to EMF code (*.genmodel* file).

Creation .gmfgen file:

The .gmfgen file is generated from the corresponding *.gmfmap* obtained in the previous step. Clicking the right mouse button on the root element of the latter, we must select the option "*Create generator model*" and then the option *"Use IMapMode"* (see Figure 58).

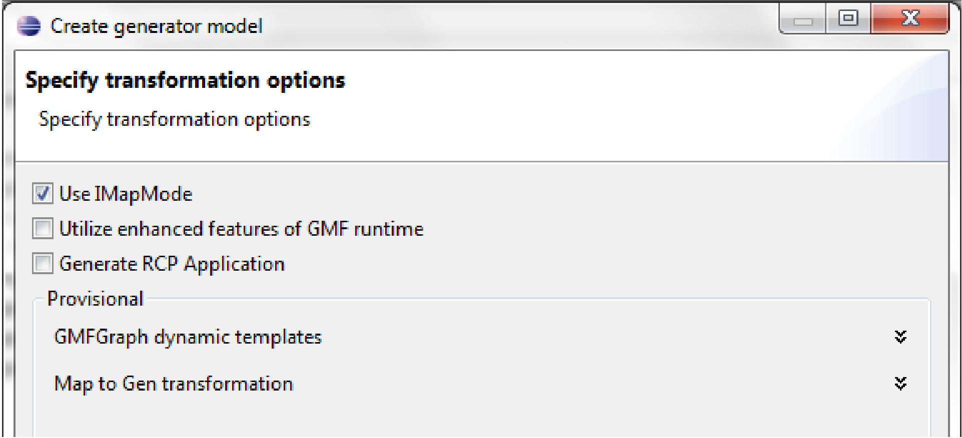


Figure 58: Configuration file *.gmfgen* extension in our EMDSD environment

In the file *.gmfgen* generated (see Figure 59), as we did in the file *.genmodel* associated EMF editor, we configure certain properties.

At the root element model (Gen Editor Generator metamodel.diagram) must set the following properties (see Figure 60):

- ***Copyright Text***: Text license each generate Java class in the later step.
- ***Diagram File Extension***: Extension of files containing the models developed with the graphical editor.
- ***Domain File Extension***: Extension of the files that contain the EMF models associated with each model created with the graphical editor.

At this point, it is important to note that the models designed with graphic editor only contain information on the visual aspect of the modeled elements (position, size, color, etc.). All the truly relevant information model and the elements included in it (containment hierarchy, values of its attributes, etc.) is stored in an EMF model underlying (*XMI* file format). The GMF model contains a reference to the EMF model, although the latter is completely independent of the first. This makes it possible to generate graphical models from the corresponding EMF model. Yes, the elements will be arranged arbitrarily and most likely will need retouching graphic pattern generated so that it is legible.

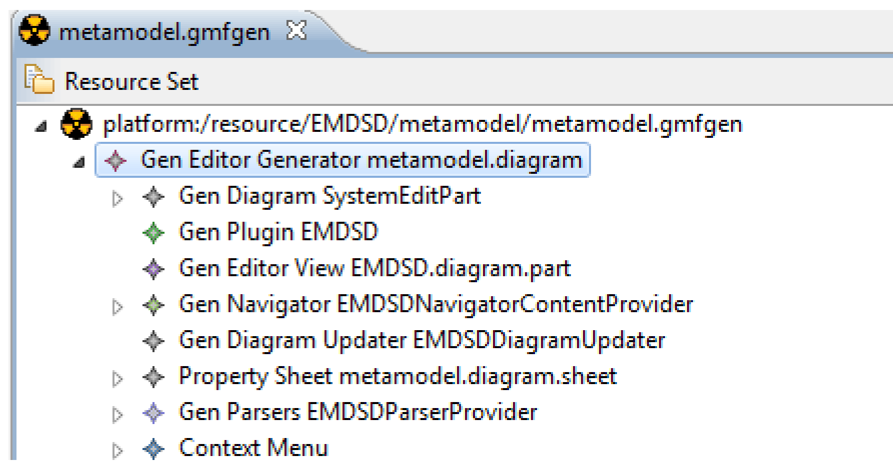


Figure 59: File *.gmfgen* associated in EMDSD enviroment graphic models editor.

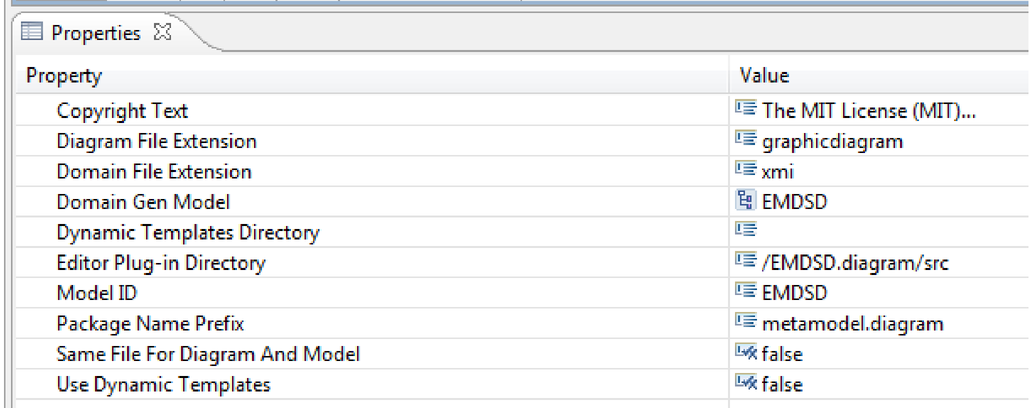
Furthermore, within the element *Gen Diagram SystemEditPart*, we must also set the following properties:

In paragraph *Diagram* (ver Figure 61):

- ***Validation Decorators***: decorators to display the associated validation errors.
- ***Validation Enabled***: validating enabled graphical models.

And in the paragraph *Providers* (Figure 62):

- ***Validation Decorator Provider Priority***: To make the validation of graphical models work properly we put on this property a *Medium* or *High* value.



Properties

Property	Value
Copyright Text	The MIT License (MIT)...
Diagram File Extension	graphicdiagram
Domain File Extension	xmi
Domain Gen Model	EMDSD
Dynamic Templates Directory	
Editor Plug-in Directory	/EMDSD.diagram/src
Model ID	EMDSD
Package Name Prefix	metamodel.diagram
Same File For Diagram And Model	false
Use Dynamic Templates	false

Figure 60: Configuration element Metamodel .*diagram* in file .*gmfgen*.

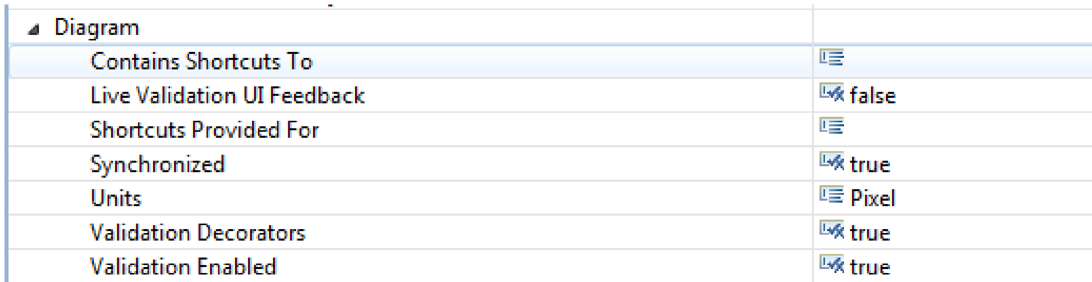


Diagram	
Contains Shortcuts To	
Live Validation UI Feedback	false
Shortcuts Provided For	
Synchronized	true
Units	Pixel
Validation Decorators	true
Validation Enabled	true

Figure 61: SystemEditPart element configuration file .*gmfgen* the section Diagram.

Providers	
Validation Decorator Provider Class Name	EMDSDValidationDecoratorProvider
Validation Decorator Provider Priority	Medium
Validation Provider Class Name	EMDSDValidationProvider

Figure 62: SystemEditPart element configuration file .*gmfgen* the section Providers.

After setting up our *.gmfgen* file, you can proceed to generate the Java implementation of graphic editor GMF. To do this, click on the right mouse button on this file, select the option: *Generate Code Diagram* (see Figure 63).

Once completed the process of code generation we can see that it has created a new folder containing the *.diagram* associated with our graphic model editor Java code. To run this editor we launch a new Eclipse using the *Run → Run Configurations → Eclipse Application* menu. In this case, we set some parameters of the new Eclipse to avoid memory problems (see Figure 64). At the both, in the "*Arguments*" section we include the following line:

-Dosgi.requiredJavaVersion=1.5 -Xms512m -Xmx1024m -XX:PermSize=512M

Once started the new Eclipse with the above configuration, we create a new Java project (Menu: *File → New → Java project*) to, within it, to create our graphics models (Menu: *File → New → Other → Examples*). We see that, to create each new graphic model, will also create an EMF model underlying, as we discussed above.

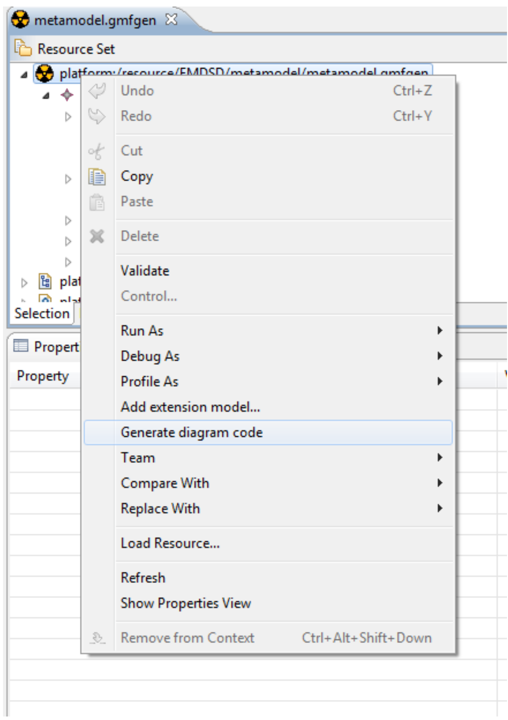


Figure 63: GMF generation of Java code associated with the graphic model editor.

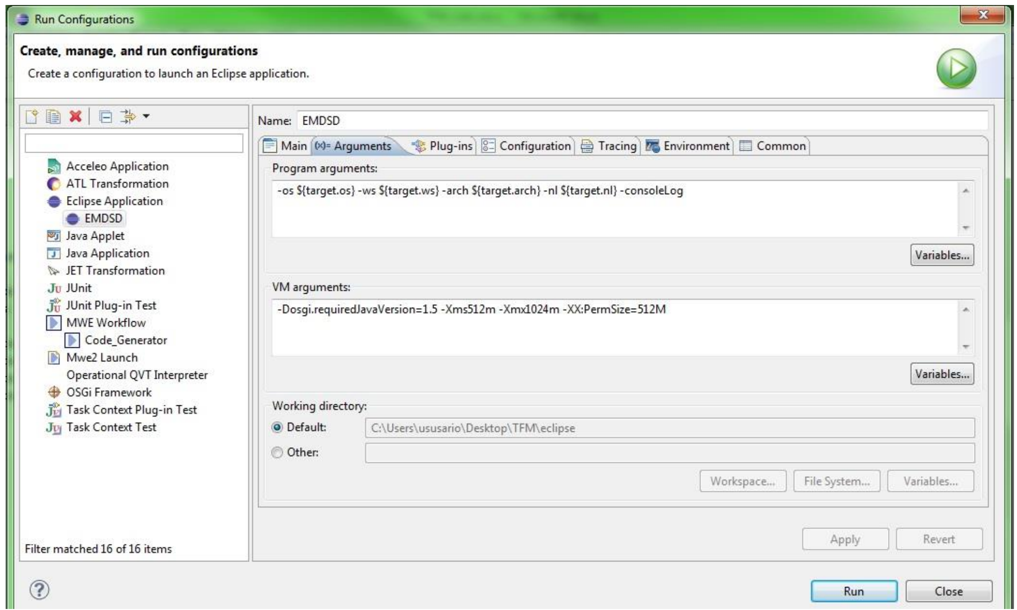


Figure 64: Eclipse memory configuration where the graphic editor is executed.

In Annex III extensions implemented on the graphic editor are detailed models implemented following the process described here.

I.3. Implementation of the code generation engine

In this section the structure of XPAND project that implements the engine generating the code EMDSD environment (see Figure 65) is described. The details of the M2T transformation, implemented as part of this project is in Annex IV of the report.

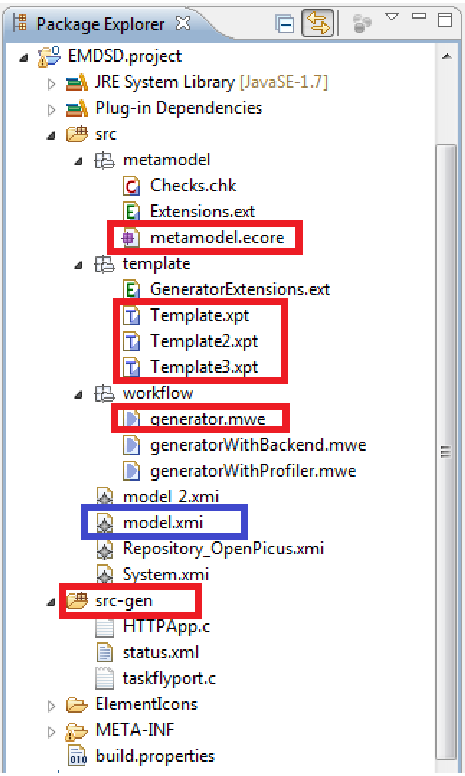


Figure 65: Directory hierarchy generation code of project *EMDSD.project*

The ***EMDSD.project / src*** folder will contain all the packages for code generation from models *.xmi* serializable format.

- ***Package metamodel.*** *This package contains the following files:*
  - **Checks.chk:** checks defined by the programmer.
  - **Extensions.ext:** extensions defined by the programmer.
  - **Metamodel.ecore:** meta-model developed in the EMF project.

  We have only had to use the last file because we have not needed to perform checks or extensions within the programmed code for this project.

- ***Package template***:
  - **GeneratorExtensions.ext:** Extensions for the programmed code.
  - **Template.xpt:** Generates the file *taskflyport.c*, main file of the program applications for this device.
  - **Template2.xpt:** Generates the file *HTTPApp.c,* associated sensors/actuators that they connect to OpenPicus device.
  - **Template3.xpt** Generates the file *status.xml,* associated sensors/actuadors that they connect to OpenPicus device.

  The first file has not been used since we have not needed to create extensions for our code.

- ***Package template***: Necessary configuration files to generate code for the device OpenPicus.
    - **Generator.mwe**
    - **GeneratorWithBackend.mwe**
    - **GeneratorWithProfiler.mwe**
    -

  In all these files we must pay special attention to the following property:

```
<property name="System" value="EMDSD.project/src/model.xmi" />
```

  System indicates the name of the EClass used as root (root) of the meta-model transformation. *Value* refers to the route that is the serialized version of the model from which we will generate the code. If the serializable model generated in the EMDSD graphical environment change of name, we should change that setting in all three configuration files for code generation.

In turn, the folder ***EMDSD.project / src -gen*** contain all the files resulting from the process of code generation from selected *.xmi* model as a starting point.

XPAND to implement the M2T transformation in which the engine code generation Project is based, we have defined three *templates* (files with extension *.tpx* included in the ***EMDSD.project / src / template*** package). The basic statements that XPAND offers to define these templates are as follows:

```
«IMPORT namePackageOfMetamodel»
«DEFINE main FOR rootOfMetamodel»
«FILE nameFile»
// code lines
«ENDFILE»
«ENDDEFINE»
```

Outside the space lines of code , we can use the following statements:

```
«IMPORT namePackageOfMetamodel»
```

It serves to define the package of meta-model.

```
«DEFINE main FOR rootOfMetamodel»
«ENDDEFINE»
```

It serves to define the root of meta-model.

```
«FILE nameFile»
«ENDFILE»
```

It serves to define the target file with the generation of the code programmed into the template file.

Within the space of code lines, we can use the following statements:

```
«DEFINE nameOfVariable FOR root.typeOfMetamodel»
«ENDDEFINE»
```

Assigning variables used to define one or Etype EClass meta-model. We can have multiple variables with the same name but assigning different types (concept of polymorphism).

```
«FOREACH root.typeOfMetamodel AS nameOfVariable»
«ENDFOREACH»
```

Serves to explore elements of the same type, it is equivalent to a loop in C language.

```
«IF condition» «ELSEIF condition»«ENDIF»
```

It serves to define the performance of a sentence if the right conditions are checked.

```
«ERROR "message"»
```

It serves to define an error message if an error occurs in this snippet.

```
«REM» commented code lines
«ENDREM»
```

It serves to include comments within your code.

You can consult a complete guide XPAND in the reference [26].

Annex II. Restrictions OCL

As discussed in previous chapters in this document , the meta- model requires the specification of OCL rules that complement it to their limitations. The rules that must follow our meta- model are:

Element:	Eclass Device
Restriction OCL:	Only_Maximum_Three_Analog_Elements
Description:	For the ' OpenPicus ' device must not define more than 3 sensors / actuators analog, otherwise, never more than the maximum of this elements defined in the class itself.

```
invariant Only_Maximum_Three_Analog_Elements:
not Device.allInstances()->exists(('Openpicus' = self.name and self.deviceType.maxAnalogElement > 3 or 'Openpicus'
<> self.name and self.deviceType.maxAnalogElement < AnalogElement.allInstances()->size()));
```

Element:	Eclass Device
Restriction OCL:	Only_Maximum_Four_Digital_Elements
Description:	For the ' OpenPicus ' device must not define more than 4 sensors / digital actuators, otherwise, never more than the maximum defined of this elements in the class itself.

```
invariant Only_Maximum_Four_Digital_Elements:
not Device.allInstances()->exists(('Openpicus' = self.name and self.deviceType.maxDigitalElement > 4 or 'Openpicus'
<> self.name and self.deviceType.maxDigitalElement < DigitalElement.allInstances()->size()));
```

Element:	Eclass Device
Restriction OCL:	Only_Maximum_One_SPI_Elements
Description:	There can have more than one SPI element for OpenPicus case, otherwise, never more than the maximum of this element in the repository (EClass DeviceType).

```
invariant Only_Maximum_One_SPI_Elements:
not Device.allInstances()->exists(('Openpicus' = self.name and self.deviceType.maxSPIElement > 4 or 'Openpicus' <>
self.name and self.deviceType.maxSPIElement < SPIElement.allInstances()->size()));
```

Element:	Eclass Device
Restriction OCL:	Only_Maximum_One_I2C_Elements
Description:	Same as last restriction but with the I2C restriction elements.

```
invariant Only_Maximum_One_I2C_Elements:
not Device.allInstances()->exists(('Openpicus' = self.name and self.deviceType.maxI2CElement > 1 or 'Openpicus' <>
self.name and self.deviceType.maxI2CElement < I2CElement.allInstances()->size()));
```

Element:	Eclass Device
Restriction OCL:	More_Than_Number_Maximum_Of_System_Elements_For_This_Device
Description:	Do not allow the definition of more sensors / actuators than the maximum defined in device of Repository.

```
invariant More_Than_Number_Maximum_Of_System_Elements_For_This_Device:
if self.deviceType.name = 'Openpicus' then not Device.allInstances() >exists((SystemElement.allInstances()->size()
>8))else not Device.allInstances()->exists((SystemElement.allInstances()->size() > self.deviceType.maxAnalogElement
+ self.deviceType.maxI2CElement + self.deviceType.maxDigitalElement + self.deviceType.maxSPIElement)) endif;
```

Element:	Eclass SystemElement
Restriction OCL:	Unique_Name_System_Elements
Description:	Sensors/Actuators with unique names.

```
invariant Unique_Name_System_Elements:
SystemElement.allInstances()->forAll(s1 : SystemElement, s2 : SystemElement | s1 <> s2 implies s1.name <> s2.name);
```

Element:	Eclass Task
Restriction OCL:	Only_One_Task_Init
Description:	It can only be a init task for programming the embedded system.

```
invariant Only_One_Task_Init: Init.allInstances()->size() < 2;
```

Element:	Eclass Task
Restriction OCL:	Only_One_Task_Final
Description:	It can only be a final task for programming the embedded system.

```
invariant Only_One_Task_Final: Final.allInstances()->size() < 2;
```

Element:	Eclass VarDeclaration
Restriction OCL:	Unique_Name_Var_Declaration
Description:	All variables must have unique names.

```
invariant Unique_Name_Var_Declaration:
not Declaration.allInstances()->select(decl : Declaration | (decl <> self))->collect(decl : Declaration |
decl.name)->includes(self.name);
```

Element:	Eclass ElementDeclaration
Restriction OCL:	Unique_Type_For_Each_Element_Declaration
Description:	Only can have a variable of type element assigned to a sensor / actuator.

```
invariant Unique_Type_For_Each_Element_Declaration:
not ElementDeclaration.allInstances()->select(ElementDecl : ElementDeclaration | (ElementDecl <> self))->collect(ElementDcl : ElementDeclaration | ElementDcl.type)->includes(self.type);
```

Element:	Eclass ElementDeclaration
Restriction OCL:	Should_Have_ElementDeclaration_ForEach_Sensor
Description:	Must have at least one variable defined for each sensor to read or write throughout the life cycle of the embedded coding system.

```
invariant Should_Have_ElementDeclaration_ForEach_Sensor:
ElementDeclaration.allInstances()->collect(type)->exists(sensor : SystemElement | sensor.name);
```

Element:	Eclass PrimitiveDeclaration
Restriction OCL:	Should_Have_PrimitiveDeclaration_As_ElementDeclaration
Description:	Must have so many primitive declaration declared as elements declaration.

```
invariant Should_Have_PrimitiveDeclaration_As_ElementDeclaration:
PrimitiveDeclaration.allInstances()->size() >= ElementDeclaration.allInstances()->size();
```

Element:	Eclass Task
Restriction OCL:	Unique_Name_Task
Description:	All tasks with unique names.

```
invariant Unique_Name_Task:
not Task.allInstances()->select(task : Task | (task <> self))->collect(task : Task | task.name.toLowerCase())->includes(self.name);
```

Element:	Eclass Connector
Restriction OCL:	Task_Final_Should_Have_Unique_Target_Connector
Description:	The final task should only have a target connector.

```
invariant Task_Final_Should_Have_Unique_Target_Connector:
Connector.allInstances()->select(connect : Connector | connect.target.oclIsTypeOf(Final))->size() < 2;
```

Element:	Eclass Connector
Restriction OCL:	Task_Final_Should_Not_Have_Source_Connect
Description:	The final task should only have any source connector.

```
invariant Task_Final_Should_Not_Have_Source_Connect:
not Connector.allInstances()->exists(self.source.oclIsTypeOf(Final));
```

Element:	Eclass Connector
Restriction OCL:	Task_Final_Should_Not_Have_Target_Connect
Description:	The init task should only have any target connector.

```
invariant Task_Init_Should_Not_Have_Target_Connect:
not Connector.allInstances()->exists(self.target.oclIsTypeOf(Init));
```

Element:	Eclass Connector
Restriction OCL:	Unique_Name_Connect
Description:	All connectors with unique names.

```
invariant Unique_Name_Connect:
not Connector.allInstances()->select(connect : Connector | (connect <> self))->collect(connect : Connector |
connect.name.toLowerCase())->includes(self.name);
```

Element:	Eclass Connector
Restriction OCL:	Task_Init_Should_Have_With_Unique_Source_Connector
Description:	The init task should only have a source connector.

```
invariant Task_Init_Should_Have_With_Unique_Source_Connector:
Connector.allInstances()->select(connect : Connector | connect.source.oclIsTypeOf(Init))->size() < 2;
```

Element:	Eclass Connector
Restriction OCL:	Connects_Must_Link_Different_Componentes
Description:	Source and target of connector must be different.

```
invariant Connects_Must_Link_Different_Componentes: source <> target;
```

Element:	Eclass ConditionBoolean
Restriction OCL:	Var_Primitive_Decl_Must_Be_Boolean
Description:	Primitive boolean variables in conditionals must be of boolean type.

```
invariant Var_Primitive_Decl_Must_Be_Boolean:
not ConditionBoolean.allInstances()->exists((self.varPrimitiveDecl.typePrimitiveDecl <> PrimitiveType::BOOL));
```

Element:	Eclass ControlStructure
Restriction OCL:	Loops_Should_Have_Instructions_Or_Loops_Internally
Description:	The loops must have at least one statement inside them (based programming).

```
invariant Loops_Should_Have_Instructions_Or_Loops_Internally:
not ControlStructure.allInstances()->exists((self.internalTasks->size() < 1));
```

Element:	Eclass Connector
Restriction OCL:	Should_Have_Before_Else_A_Condition
Description:	An else condition must have previous task called if condition (based programming).

```
invariant Should_Have_Before_Else_A_Condition:
Connector.allInstances()->exists(self.target.oclIsTypeOf(ElseCondition)and
self.source.parent.oclIsKindOf(Condition));
```

Element:	Eclass PinConnection
Restriction OCL:	Pins_Must_Link_Unique_Analog_Elements
Description:	The pins that connect sensors / actuators analog must be analog type.

```
invariant Pins_Must_Link_Unique_Analog_Elements:
not PinConnection.allInstances()->exists((self.systemElement.systemElementType.oclIsKindOf(AnalogElement) and
self.pin.pinType.pinKind.name <> 'ANALOG'));
```

Element:	Eclass PinConnection
Restriction OCL:	Pins_Must_Link_Unique_SPI_Elements
Description:	The pins that connect sensors / actuators SPI must be SPI type.

```
invariant Pins_Must_Link_Unique_SPI_Elements:
not PinConnection.allInstances()->exists((self.systemElement.systemElementType.oclIsKindOf(SPIElement) and
self.pin.pinType.pinKind.name <> 'SPI'));
```

Element:	Eclass PinConnection
Restriction OCL:	Pins_Must_Link_Unique_Digital_Elements
Description:	The pins that connect sensors / actuators digital must be digital type.

```
invariant Pins_Must_Link_Unique_Digital_Elements:
not PinConnection.allInstances()->exists((self.systemElement.systemElementType.oclIsKindOf(DigitalElement) and
self.pin.pinType.pinKind.name <> 'DIGITAL'));
```

Element:	Eclass PinConnection
Restriction OCL:	Element_Only_Can_Have_Pins_Same_Technology
Description:	The pins that connect sensors / actuators, they must be the same technology.

```
invariant Element_Only_Can_Have_Pins_Same_Technology:
not PinConnection.allInstances()->exists((self.systemElement.systemElementType.pinTechnology <> self.pin.pinType.pinTechnology));
```

Element:	Eclass PinConnection
Restriction OCL:	*Pins_Must_Link_Unique_I2C_Elements*
Description:	The pins that connect sensors / actuators I2C must be I2C type.

```
invariant Pins_Must_Link_Unique_I2C_Elements:
not PinConnection.allInstances()->exists((self.systemElement.systemElementType.oclIsKindOf(I2CElement) and self.pin.pinType.pinKind.name <> 'I2C'));
```

Element:	Eclass PinConnection
Restriction OCL:	*Pins_Must_Be_Diferent_For_Each_Element*
Description:	For connections of elements and pins, the pins can not connect two pins identical names.

```
invariant Pins_Must_Be_Diferent_For_Each_Element:
PinConnection.allInstances()->forAll(e1 : PinConnection, e2 : PinConnection | e1 <> e2 implies e1.pin.pinType.name <> e2.pin.pinType.name);
```

Element:	Eclass PinConnection
Restriction OCL:	Only_Can_Have_System_Element_Unique
Description:	For connections elements with pins, the elements can’t have equals names.

```
invariant Only_Can_Have_System_Element_Unique:
PinConnection.allInstances()->forAll(e1 : PinConnection, e2 : PinConnection | e1 <> e2 implies e1.name <> e2.name);
```

Annex III. Graphic Editor Extensions

This section describes the extensions implemented on the graph editor described models generated using GMF to support the following functionalities:

1) Selecting the name of the package for the serialized version of the meta-model used in the graphical environment EMDSD.
2) Load model repository located on the route indicated by the user at the time of creating a new model of embedded system.
3) Dynamic loading of icons associated with the embedded and sensor / actuator based repository of its kind system.
4) Automatic inclusion of the pins of device depending on its type.
5) Decoration of the pins by type (analog, digital , I2C , SPI) .

Extension number: Description:	1 Selecting the name of the package for the serialized version of the meta- model used in the graphical environment EMDSD
Modified package: Modified class: Modified method:	metamodel.impl MetamodelPackageImpl.java createResource

```
protected Resource createResource(String uri) {
     return super.createResource("metamodel/metamodel.ecore");
}
```

Modified package: Modified class: Modified method:	EMDSD.diagram.part EMDSDDiagramEditorUtil.java getSaveOptions

```
public static Map<?, ?> getSaveOptions() {
     HashMap<String, Object> saveOptions = new HashMap<String, Object>();
     saveOptions.put(XMLResource.OPTION_ENCODING, "UTF-8");    //$NON-NLS-1$
     saveOptions.put(XMLResource.OPTION_SCHEMA_LOCATION,Boolean.TRUE);
     saveOptions.put(Resource.OPTION_SAVE_ONLY_IF_CHANGED,Resource.OPTION_SAVE_ONLY_IF_CHANGED_MEMORY_BUFFER);
     return saveOptions;
}
```

Extension number:	2
Description:	Load model repository located on the route indicated by the user at the time of creating a new model of embedded system
Modified package:	metamodel.diagram.part
Modified class:	MetamodelDiagramEditorUtil.java
Modified method:	createDiagram
Comments:	It's necessary import of packages: `javax.swing.JFrame`, `javax.swing.JFileChooser` and `java.IO.File`

```
public static Resource createDiagram ( URI diagramURI, URI modelURI, IProgressMonitor progressMonitor) {

     TransactionalEditingDomain editingDomain = GMFEditingDomainFactory.INSTANCE .createEditingDomain();
     progressMonitor.beginTask(Messages.MetamodelDiagramEditorUtil_CreateDiagramProgressTask, 3);
     final Resource diagramResource = editingDomain.getResourceSet().createResource(diagramURI);
     final Resource modelResource = editingDomain.getResourceSet().createResource(modelURI);
     final String diagramName = diagramURI.lastSegment();

     JFrame frame = new JFrame("");
     frame.setVisible(false);
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     JFileChooser fileChooser = new JFileChooser(new File(""));
     fileChooser.setDialogTitle("Select Device and Element Repository...");
     fileChooser.showOpenDialog(frame);

     final Resource loadedRepository =
          editingDomain.loadResource(fileChooser.getSelectedFile().toURI().toString());

     AbstractTransactionalCommand command = new AbstractTransactionalCommand(
          editingDomain,
```

```
            Messages.MetamodelDiagramEditorUtil_CreateDiagramCommandLabel,
            Collections.EMPTY_LIST) {
            protected CommandResult doExecuteWithResult(IProgressMonitor monitor, IAdaptable info)
                  throws ExecutionException {
                        System model = createInitialModel();
                        attachModelToResource(model, modelResource);
                        Diagram diagram = ViewService.createDiagram(model, SystemEditPart.MODEL_ID,
                              MetamodelDiagramEditorPlugin.DIAGRAM_PREFERENCES_HINT);
                        if (diagram != null) {
                              diagramResource.getContents().add(diagram);
                              diagram.setName(diagramName);
                              diagram.setElement(model);

                              if (modelResource.getContents().get(0) instanceof metamodel.impl.SystemImpl) {
                                    if (loadedRepository.getContents().get(0)instanceof metamodel.impl.RepositoryImpl) {
                                       metamodel.impl.SystemImpl si =
                                              (metamodel.impl.SystemImpl) modelResource.getContents().get(0);
                                       metamodel.impl.RepositoryImpl ri =
                                              (metamodel.impl.RepositoryImpl) loadedRepository.getContents().get(0);

                                       si.setInputRepository(ri);
                                       java.lang.System.out.println("Repository of system initialized how: "
                                                    + si.getInputRepository());
                                    }
                                    else    java.lang.System.out.println("This repository has a strange root");
                              }
                              else java.lang.System.out.println("Error of initialize repository of system");
                  }

   try {
                  modelResource.save(metamodel.diagram.part.MetamodelDiagramEditorUtil.getSaveOptions());
                  diagramResource.save(metamodel.diagram.part.MetamodelDiagramEditorUtil.getSaveOptions());
            } catch (IOException e) {
                  MetamodelDiagramEditorPlugin.getInstance().
```

```
                    logError("Unable to store model and diagram resources", e); //$NON-NLS-1$
            }
            return CommandResult.newOKCommandResult();
            }
        };
        try {
            OperationHistoryFactory.getOperationHistory().execute(command,
                new SubProgressMonitor(progressMonitor, 1), null);
        } catch (ExecutionException e) {
            MetamodelDiagramEditorPlugin.getInstance().
                logError("Unable to create model and diagram", e); //$NON-NLS-1$
        }
        setCharset(WorkspaceSynchronizer.getFile(modelResource));
        setCharset(WorkspaceSynchronizer.getFile(diagramResource));
        return diagramResource;
    }
```

Extension number:	3
Description:	Dynamic loading of icons associated with the embedded and sensor / actuator based repository of its kind system
Modified package:	metamodel.diagram.edit.parts
Modified class:	DeviceNameEditPart.java
Modified method:	**setLabel** (text under icon)

```
public void setLabel(Label figure) {
     unregisterVisuals();
     setFigure(figure);
     defaultText = getLabelTextHelper(figure);
     figure.setTextPlacement(org.eclipse.draw2d.PositionConstants.SOUTH);
     registerVisuals();
     refreshVisuals();
}
```

Modified package:	metamodel.diagram.edit.parts
Modified class:	DeviceNameEditPart.java
Modified method:	**setLabelIconHelper** (update icon dinamic)

```
protected void setLabelIconHelper(IFigure figure, Image icon) {
     icon = getNewIcon(figure, icon);
     if (figure instanceof WrappingLabel) {
          ((WrappingLabel) figure).setIcon(icon);
     } else {
          ((Label) figure).setIcon(icon);
     }
}
```

Modified package:	metamodel.diagram.edit.parts
Modified class:	DeviceNameEditPart.java
Modified method:	**handleNotificationEvent** (when attribute type change)

```
protected void handleNotificationEvent(Notification event) {
	Object feature = event.getFeature();
	if (NotationPackage.eINSTANCE.getFontStyle_FontColor().equals(feature)) {
		Integer c = (Integer) event.getNewValue();
		setFontColor(DiagramColorRegistry.getInstance().getColor(c));
	} else if (NotationPackage.eINSTANCE.getFontStyle_Underline().equals(
			feature)) {
		refreshUnderline();
	} else if (NotationPackage.eINSTANCE.getFontStyle_StrikeThrough()
			.equals(feature)) {
		refreshStrikeThrough();
	} else if (NotationPackage.eINSTANCE.getFontStyle_FontHeight().equals(feature)
				|| NotationPackage.eINSTANCE.getFontStyle_FontName().equals(feature)
				|| NotationPackage.eINSTANCE.getFontStyle_Bold().equals(feature)
				|| NotationPackage.eINSTANCE.getFontStyle_Italic().equals(feature)) {
		refreshFont();
	} else if (feature instanceof org.eclipse.emf.ecore.impl.EReferenceImpl
				&& ((org.eclipse.emf.ecore.impl.EReferenceImpl) feature).getName().equals("deviceType")) {
		// Se actualiza el icono asociado al DeviceType
		setLabelIconHelper(figure, this.getLabelIcon());
	} else {
		if (getParser() != null && getParser().isAffectingEvent(event,getParserOptions().intValue())) {
			refreshLabel();
		}
		if (getParser() instanceof ISemanticParser) {
			ISemanticParser modelParser = (ISemanticParser) getParser();
			if (modelParser.areSemanticElementsAffected(null, event)) {
```

```
                        removeSemanticListeners();
                        if (resolveSemanticElement() != null) {
                             addSemanticListeners();
                        }
                        refreshLabel();
                   }
              }
         }
         super.handleNotificationEvent(event);
    }
```

Modified package:	metamodel.diagram.edit.parts
Modified class:	DeviceNameEditPart.java
New method:	**getNewIcon** (obtain icon by device type)

```
    protected Image getNewIcon(IFigure figure, Image icon) {
         Image newIcon;
         try {
              org.eclipse.swt.graphics.Device visualizationDevice = icon.getDevice();
              org.eclipse.gmf.runtime.notation.Node graphicalDeviceNode =
                    (org.eclipse.gmf.runtime.notation.Node) this.getParent().getModel();
              metamodel.Device deviceModel = (metamodel.Device) graphicalDeviceNode.getElement();
              String deviceIconFile = deviceModel.getDeviceType().getIcon().toString();
              String iconsPath = deviceModel.getParent().getParent().getRepository().getIconsPath();
              String deviceIconFullPath = iconsPath + deviceIconFile;
              newIcon = new Image(visualizationDevice, deviceIconFullPath);
         } catch (Exception e) {
              newIcon = icon;
         }
         return newIcon;
    }
```

Modified package:	metamodel.diagram.edit.parts
Modified class:	SystemElementNameEditPart.java
Modified method:	**setLabel** (text under icon)

```
public void setLabel(Label figure) {
	unregisterVisuals();
	setFigure(figure);
	defaultText = getLabelTextHelper(figure);
	figure.setTextPlacement(org.eclipse.draw2d.PositionConstants.SOUTH);
	registerVisuals();
	refreshVisuals();
}
```

Modified package:	metamodel.diagram.edit.parts
Modified class:	SystemElementNameEditPart.java
Modified method:	**setLabelIconHelper** (update icon dynamic)

```
protected void setLabelIconHelper(IFigure figure, Image icon) {
	icon = getNewIcon(figure, icon);
	if (figure instanceof WrappingLabel) {
		((WrappingLabel) figure).setIcon(icon);
	} else {
		((Label) figure).setIcon(icon);
	}
}
```

Modified package:	metamodel.diagram.edit.parts
Modified class:	SystemElementNameEditPart.java
Modified method:	**handleNotificationEvent** (when attribute type change)

```
protected void handleNotificationEvent(Notification event) {
     Object feature = event.getFeature();
     if (NotationPackage.eINSTANCE.getFontStyle_FontColor().equals(feature)) {
          Integer c = (Integer) event.getNewValue();
          setFontColor(DiagramColorRegistry.getInstance().getColor(c));
     } else if (NotationPackage.eINSTANCE.getFontStyle_Underline().equals( feature)) {
          refreshUnderline();
     } else if (NotationPackage.eINSTANCE.getFontStyle_StrikeThrough().equals(feature)) {
          refreshStrikeThrough();
     } else if (NotationPackage.eINSTANCE.getFontStyle_FontHeight().equals(feature)
                    || NotationPackage.eINSTANCE.getFontStyle_FontName().equals(feature)
                    || NotationPackage.eINSTANCE.getFontStyle_Bold() .equals(feature)
                    || NotationPackage.eINSTANCE.getFontStyle_Italic().equals(feature)) {
          refreshFont();
     } else if (feature instanceof org.eclipse.emf.ecore.impl.EReferenceImpl
               && ((org.eclipse.emf.ecore.impl.EReferenceImpl)feature).getName().equals("systemElementType")) {
          setLabelIconHelper(figure, this.getLabelIcon());
          org.eclipse.gmf.runtime.notation.Node graphicalElementNode =
                 (org.eclipse.gmf.runtime.notation.Node) this.getParent().getModel();
          metamodel.SystemElement elementModel = (metamodel.SystemElement) graphicalElementNode.getElement();
          if (elementModel.getSystemElementType() instanceof metamodel.DigitalActuator
                    || elementModel.getSystemElementType() instanceof metamodel.DigitalSensor)
               this.setForegroundColor(org.eclipse.draw2d.ColorConstants.blue);
          else {
               if (elementModel.getSystemElementType() instanceof metamodel.AnalogActuator
                         || elementModel.getSystemElementType() instanceof metamodel.AnalogSensor)
                    this.setForegroundColor(org.eclipse.draw2d.ColorConstants.red);
```

```
				else {
					if (elementModel.getSystemElementType() instanceof metamodel.SPIElement)
						this.setForegroundColor(org.eclipse.draw2d.ColorConstants.yellow);
					else {
						if (elementModel.getSystemElementType() instanceof metamodel.I2CElement)
							this.setForegroundColor(org.eclipse.draw2d.ColorConstants.green);
					}
				}
			}
		} else {
			if (getParser() != null && getParser().isAffectingEvent(event, getParserOptions().intValue())) {
				refreshLabel();
			}
			if (getParser() instanceof ISemanticParser) {
				ISemanticParser modelParser = (ISemanticParser) getParser();
				if (modelParser.areSemanticElementsAffected(null, event)) {
					removeSemanticListeners();
					if (resolveSemanticElement() != null) {
						addSemanticListeners();
					}
					refreshLabel();
				}
			}
		}
		super.handleNotificationEvent(event);
	}
```

Modified package:	metamodel.diagram.edit.parts
Modified class:	SystemElementNameEditPart.java
New method:	**getNewIcon** (obtain icon by element type) => Not equal implement in type of device

```
protected Image getNewIcon(IFigure figure, Image icon) {
	Image newIcon;
	try {
		org.eclipse.swt.graphics.Device visualizationDevice = icon.getDevice();
		org.eclipse.gmf.runtime.notation.Node graphicalElementNode =
			(org.eclipse.gmf.runtime.notation.Node) this.getParent().getModel();
		metamodel.SystemElement elementModel = (metamodel.SystemElement) graphicalElementNode.getElement();
		String elementIconFile = elementModel.getSystemElementType().getIcon().toString();
		String iconsPath = elementModel.getParent().getParent().getRepository().getIconsPath();
		String elementIconFullPath = iconsPath + elementIconFile;
		newIcon = new Image(visualizationDevice, elementIconFullPath);

	} catch (Exception e) {
		newIcon = icon;
	}
	return newIcon;
}
```

Extension number:	4
Description:	Automatic inclusion of the pins of device depending on its type
Modified package:	metamodel.diagram.edit.parts
Modified class:	DeviceEditPart.java
New method:	**createPins** (created of pins how indicated in definition of Device enabled on Repository)
Comments:	It's necessary import of package `org.eclipse.emf.common.notify.Notification`

```
public void createPins() {
	try {
		org.eclipse.gmf.runtime.notation.Node graphicalDeviceNode =
			(org.eclipse.gmf.runtime.notation.Node) this.getModel();
		metamodel.Device deviceModel = (metamodel.Device) graphicalDeviceNode.getElement();
		deviceModel.getPins().clear();

		int numDeviceTypePins = deviceModel.getDeviceType().getPinTypes().size();
		metamodel.PinType pinType;
		for (int i = 0; i < numDeviceTypePins; i++) {
			pinType = (metamodel.PinType) deviceModel.getDeviceType().getPinTypes().get(i);
			metamodel.Pin newPin = metamodel.MetamodelFactory.eINSTANCE.createPin();
			newPin.setPinType(pinType);
			deviceModel.getPins().add(newPin);
		}
	} catch (Exception e) {
		System.out.println("Error in method createPins");
	}
}
```

Modified package:	metamodel.diagram.edit.parts
Modified class:	DeviceEditPart.java
New method:	**handleNotificationEvent** (when update type of device)
Comments:	It's necessary import of package `org.eclipse.emf.common.notify.Notification`

```
public void handleNotificationEvent(Notification event) {
     Object feature = event.getFeature();
     if (feature instanceof org.eclipse.emf.ecore.impl.EReferenceImpl
                    && ((org.eclipse.emf.ecore.impl.EReferenceImpl) feature).getName().equals("deviceType")) {
          org.eclipse.gmf.runtime.notation.Node graphicalDeviceNode =
                 (org.eclipse.gmf.runtime.notation.Node) this.getModel();
          metamodel.Device deviceModel = (metamodel.Device) graphicalDeviceNode.getElement();
          deviceModel.getParent().getPinConnections().clear();
          createPins();
     }
}
```

Extension number:	5
Description:	Decoration of the pins by type (analog, digital , I2C , SPI)
Modified package:	metamodel.diagram.edit.parts
Modified class:	PinEditPart.java
Modified method:	**PinFigure**

```
public PinFigure() {
	FlowLayout layoutThis = new FlowLayout();
	layoutThis.setStretchMinorAxis(false);
	layoutThis.setMinorAlignment(FlowLayout.ALIGN_LEFTTOP);
	layoutThis.setMajorAlignment(FlowLayout.ALIGN_LEFTTOP);
	layoutThis.setMajorSpacing(5);
	layoutThis.setMinorSpacing(5);
	layoutThis.setHorizontal(true);
	this.setLayoutManager(layoutThis);
	this.setLineWidth(3);
	this.setForegroundColor(ColorConstants.white);
	updateBackground();
}
```

Modified package:	metamodel.diagram.edit.parts
Modified class:	PinEditPart.java
New method:	**updateBackground**

```
public void updateBackground() {
     metamodel.Pin pin = (metamodel.Pin) ((org.eclipse.gmf.runtime.notation.Node)
                              PinEditPart.this.getModel()).getElement();
     if (pin.getPinType().getPinKind().getName().equals("DIGITAL"))
          this.setBackgroundColor(ColorConstants.blue);
     else {
          if (pin.getPinType().getPinKind().getName().equals("ANALOG"))
               this.setBackgroundColor(ColorConstants.red);
          else {
               if (pin.getPinType().getPinKind().getName().equals("SPI"))
                    this.setBackgroundColor(ColorConstants.yellow);
               else {
                    if (pin.getPinType().getPinKind().getName().equals("I2C"))
                         this.setBackgroundColor(ColorConstants.green);
                    else
                         this.setBackgroundColor(ColorConstants.gray);
                    }
               }
          }
     }
```

Annex IV. M2T Transformation GroveNest

This last annex describes sections with M2T transformation implemented within the template for the device GroveNest studied on the graphic editor of models generated using GMF:

0) General block GroveNest M2T transformation.
1) Declaration libraries needed for transformation.
2) Declaration variables (constants and locals).
3) Declaration device, sensors, actuators and sensor - device or device - actuator connections.
4) Programming the sensors and actuators programmable in device chosen.
5) Programming on behaviour of the sensors and actuators in device.
6) Polymorphism sensors and actuators in device.

Section number: Description:	0 General block GroveNest M2T transformation
Modified package: Modified template::	EMDSD.project/src/template template.xpt

```
// Declaration includes (Section number 1)
// Declaration constants and locals variables (Section number 2)
// Main Method FlyportTask
void FlyportTask()
{
      vTaskDelay(300);
      //WIFI Connection
      WFConnect(WF_DEFAULT);

      // Declaration device (Section number 3)
      // Declaration sensors and actuators (Section number 3)
      // Connect sensors and actuators in device (Section number 3)
      // Initialize sensors and actuators (Section number 4)
      // Behaviour sensors and actuators (Section number 5)
}

// Polymorphism of elements in device (Section number 6)
```

Section number: Description:	1 Declaration libraries needed for transformation.
Modified package: Modified template::	EMDSD.project/src/template template.xpt

```
// Declaration includes
#include "taskFlyport.h"
#include "grovelib.h"

«FOREACH this.systemBehaviour.varDeclaration.elementDeclarations AS varDecl»«IF
varDecl.type.systemElementType.name.toString().contains("GroveChainableRGBLed") ||
varDecl.type.systemElementType.name.toString().contains("GroveLEDStripDriver") »#include "rgb.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveGPS")»#include "gps.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveUltrasonicRanger")»#include "sonic_ranger.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveTemperatureAndHumidityPro")»#include "rht03.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveTemperature")»#include "analog_temp.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveSerialLCD")»#include "serial_lcd.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveLEDBar")»#include "led_bar.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveDigitDisplay")»#include "4digitdisplay.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveColor")»#include "color_sens.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveAxisAccelerometer")»#include "accel.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveAxisGyro")»#include "3axisgyro.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveAxisCompass")»#include "3axiscompass.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveTouch")»#include "touch.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveBarometer")»#include "barometer.h"
«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveOLEDDisplay")»#include "oled.h"
   #include "logo.h"«ENDIF»«ENDFOREACH»
```

Section number:	2
Description:	Declaration variables (constants and locals).
Modified package: Modified template::	EMDSD.project/src/template template.xpt

```
// Declaration local variables
«FOREACH this.systemBehaviour.varDeclaration.primitiveDeclarations AS varDecl»«varDecl.typePrimitiveDecl» «varDecl.name»«IF
varDecl.typePrimitiveDecl.toString().contains("BOOL")» = FALSE;
«ELSE»;
«ENDIF»«ENDFOREACH»

// Declaration constant variables
«FOREACH this.systemBehaviour.varDeclaration.constantDeclarations AS varDecl»«varDecl.typePrimitiveDecl» «varDecl.name» =
«varDecl.value»;«ENDFOREACH»
```

Section number:	3
Description:	Declaration device, sensors, actuators and sensor - device or device - actuator connections
Modified package: Modified template::	EMDSD.project/src/template template.xpt

```
// Declaration device and elements
      // GROVE board
      void *«this.systemStructure.device.name» = new(GroveNest);

    // Digital and Analog Elements
    «FOREACH this.systemBehaviour.varDeclaration.elementDeclarations AS varDecl»«IF
varDecl.type.systemElementType.name.toString().contains("GroveRelay")»void *«varDecl.name» = new(Dig_io, OUT);
```

```
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveDigitDisplay")»void *«varDecl.name» = 
new(_4Digit,6,POINT);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveLEDBar")»void *«varDecl.name» = new(LedBar,2);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveLEDStripDriver")»void *«varDecl.name» = new(Rgb, 
2);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveMagneticSwitch")»void *«varDecl.name» = new(Dig_io, 
IN);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveSerialLCD")»void *«varDecl.name» = 
new(SerialLcd,2);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveButton")»void *«varDecl.name» = new(Dig_io, IN);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveChainableRGBLed")»void *«varDecl.name» = new(Rgb, 
2);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveGPS")»void *«varDecl.name» = new(Gps,2);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveHall")»void *«varDecl.name» = new(Dig_io, IN);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveLED")»void *«varDecl.name» = new(Dig_io, OUT);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveLineFinder")»void *«varDecl.name» = new(Dig_io, 
IN);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GrovePIRMotion")»void *«varDecl.name» = new(Dig_io, IN);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveTemperatureAndHumidityPro")»void *«varDecl.name» = 
new(RHT03);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveUltrasonicRanger")»void *«varDecl.name» = 
new(Ranger);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveWater")»void *«varDecl.name» = new(Dig_io, IN);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveGasSensor")»void *«varDecl.name» = new(An_i);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveMoisture")»void *«varDecl.name» = new(An_i);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveTemperature")»void *«varDecl.name» = new(An_Temp);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveLight")»void *«varDecl.name» = new(An_i);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveRotaryPotenciometer")»void *«varDecl.name» = 
new(An_i);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveSlidePotenciometer")»void *«varDecl.name» = 
new(An_i);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveTouch")»void *«varDecl.name» = new(Touch, 
TOUCH_ADDR0, 8);
```

```
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveOLEDDisplay")»void *«varDecl.name» = new(Oled, 
OLED_ADDR0);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveColor")»void *«varDecl.name» = new(ColorSens, 
COLORSENS_ADDR, GAIN_16, PRESCALER_1);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveBarometer")»void *«varDecl.name» = 
new(Baro,BAROM_ADDR);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveAxisGyro")»void *«varDecl.name» = new(Gyro, 
GYRO_ADDR0);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveAxisCompass")»void *«varDecl.name» = new(Compass, 
HMC5883_ADDR,SCALE1,46330);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveAxisAccelerometer")»void *«varDecl.name» = 
new(Accel, ACC_ADDR0);
    «ENDIF»«ENDFOREACH»

    // Connect elements to device
    «FOREACH this.systemBehaviour.varDeclaration.elementDeclarations AS 
varDecl»attachToBoard(«this.systemStructure.device.name», «varDecl.name», «FOREACH this.systemStructure.pinConnections AS 
elemPinLink»«IF elemPinLink.systemElement.name == varDecl.type.name»«elemPinLink.pin.pinType.name»«ENDIF»«ENDFOREACH»);
    «ENDFOREACH» «FOREACH this.systemBehaviour.varDeclaration.elementDeclarations AS varDecl»«IF 
varDecl.type.systemElementType.name.toString().contains("GroveTouch") ||
varDecl.type.systemElementType.name.toString().contains("GroveAxisAccelerometer") ||
varDecl.type.systemElementType.name.toString().contains("GroveAxisCompass") ||
varDecl.type.systemElementType.name.toString().contains("GroveAxisGyro") ||
varDecl.type.systemElementType.name.toString().contains("GroveBarometer") ||
varDecl.type.systemElementType.name.toString().contains("GroveColor") ||
varDecl.type.systemElementType.name.toString().contains("GroveOLEDDisplay")»
    configure(«varDecl.name»);
    «ENDIF»«ENDFOREACH»
```

Section number: Description:	4 Programming the sensors and actuators programmable in device chosen
Modified package: Modified template::	EMDSD.project/src/template template.xpt

```
    // Initialize elements
    «FOREACH this.systemBehaviour.varDeclaration.elementDeclarations AS varDecl»«IF
varDecl.type.systemElementType.name.toString().contains("GroveLED") ||
varDecl.type.systemElementType.name.toString().contains("GroveChainableRGBLed")»set(«varDecl.name» ,off);
    «ELSEIF
varDecl.type.systemElementType.name.toString().contains("GroveTemperatureAndHumidityPro")»configure(«varDecl.name», 3);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveRelay")»configure(«varDecl.name», ON);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveUltrasonicRanger")»configure(«varDecl.name», 1);
    «ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveLEDBar")»
    /*Configure Ledbar chain one singular led to maximum level*/
      configure(«varDecl.name»,LED,1,2,0xFF);
      configure(«varDecl.name»,LED,1,30,0xFF);
      configure(«varDecl.name»,LED,2,2,0xFF);
      configure(«varDecl.name»,LED,2,30,0xFF);
      configure(«varDecl.name»,LED,1,4,0xFF);
      configure(«varDecl.name»,LED,1,7,0xFF);

      /*Perform the previously LedBar chain configuration*/
      set(«varDecl.name»,ON);
      vTaskDelay(500);

      /*Switch off the LedBar chain*/
      set(«varDecl.name»,OFF);

      if(configure(«varDecl.name»,BAR,1,0,REVERSE))
```

```
	{
		UARTWrite(1,"Error!\r\n");
		while(1);
	}
	«ELSEIF varDecl.type.systemElementType.name.toString().contains("GroveSerialLCD")»
	if(!configure(«varDecl.name»))
		UARTWrite(1,"Serial LCD initialized properly!\r\n");
	else
		UARTWrite(1,"Error! The Serial LCD didn't reply\r\n");

	/***** Turn on the backlight ****/
	set(«varDecl.name»,SLCD_CNTL,SLCD_BACKLIGHT_ON);

	/***Display One Character ***/
	set(«varDecl.name»,SLCD_PRNT,"H");

	/***** Turn on the cursor ****/
	set(«varDecl.name»,SLCD_CNTL,SLCD_CURSOR_ON);
	vTaskDelay(300);

	/***** Blinking the cursor ****/
	set(«varDecl.name»,SLCD_CNTL,SLCD_BLINK_ON);
	vTaskDelay(300);

	set(«varDecl.name»,SLCD_CNTL,SLCD_CURSOR_OFF);
	set(«varDecl.name»,SLCD_CNTL,SLCD_BLINK_OFF);

	/***Display "Hello GroveNest" ***/
	set(«varDecl.name»,SLCD_PRNT,"ello GroveNest");
	vTaskDelay(300);

	/***Scroll to left" ***/
	set(«varDecl.name»,SLCD_CNTL,SLCD_SCROLL_LEFT);
```

```
vTaskDelay(300);

/***Scroll to right" ***/
set(«varDecl.name»,SLCD_CNTL,SLCD_SCROLL_RIGHT);
vTaskDelay(300);

/***Move the cursor to the last column on the first row" ***/
set(«varDecl.name»,SLCD_POST,15,0);

/***Set the writing form RIGHT to LEFT" ***/
set(«varDecl.name»,SLCD_CNTL,SLCD_RIGHT_TO_LEFT);

/***Display "tseNevorG olleH" ***/
set(«varDecl.name»,SLCD_PRNT,"Hello GroveNest");
vTaskDelay(300);

/***Clear LCD Display****/
set(«varDecl.name»,SLCD_CNTL,SLCD_CLEAR_DISPLAY);

/***Set the writing from LEFT to RIGHT" ***/
set(«varDecl.name»,SLCD_CNTL,SLCD_LEFT_TO_RIGHT);

/***Back to home position (0,0)" ***/
set(«varDecl.name»,SLCD_CNTL,SLCD_RETURN_HOME);
set(«varDecl.name»,SLCD_PRNT,"Hello GroveNest");
vTaskDelay(300);

/***Turn off the display ***/
set(«varDecl.name»,SLCD_CNTL,SLCD_DISPLAY_OFF);
vTaskDelay(300);

/***Turn on the display ***/
set(«varDecl.name»,SLCD_CNTL,SLCD_DISPLAY_ON);
```

```
        vTaskDelay(300);

        set(«varDecl.name»,SLCD_POST,0,1);
        char *message = "Serial LCD Grove example";
        set(«varDecl.name»,SLCD_PRNT,"Serial LCD Grove example");
        vTaskDelay(300);«ENDIF»«ENDFOREACH»
```

Section number:	5
Description:	Programming on behaviour of the sensors and actuators in device
Modified package:	EMDSD.project/src/template
Modified template::	template.xpt

```
«FOREACH this.systemBehaviour.workflow.connectors AS c»«IF c.source.toString().contains("Init")»«EXPAND treatedTask FOR
c.source»«EXPAND program (c.target) FOR this»«ENDIF»«ENDFOREACH»
«FOREACH this.systemBehaviour.workflow.connectors AS c»«IF c.target.toString().contains("Final")»«EXPAND treatedTask FOR
c.target»«ENDIF»«ENDFOREACH»
«FOREACH this.systemBehaviour.varDeclaration.constantDeclarations AS varDecl»«varDecl.typePrimitiveDecl» «varDecl.name» =
«varDecl.value»;«ENDFOREACH»
```

Section number: Description:	6 Polymorphism sensors and actuators in device
Modified package: Modified template::	EMDSD.project/src/template template.xpt

```
«DEFINE tasks FOR Task»
«ERROR "should not happen"»
«ENDDEFINE»
«DEFINE program (Task t) FOR System»«EXPAND wayTreatedTask (t) FOR this»«ENDDEFINE»
«DEFINE wayTreatedTask (Task t) FOR System»«EXPAND wayTasks (t, t) FOR this.systemBehaviour.workflow»«ENDDEFINE»
«DEFINE wayTasks (Task taskTratada, Task taskInitial) FOR Workflow»«FOREACH this.connectors AS connect»«IF
connect.source.name == taskTratada.name»«EXPAND treatedTask FOR connect.source»«IF connect.source.parent == null &&
connect.source.parent.parent == null»«EXPAND closingKeys (connect.source, null , null, taskInitial) FOR this»«ELSEIF
connect.source.parent != null && connect.source.parent.parent == null»«IF connect.source.parent.name ==
taskInitial.name»«EXPAND closingKeys (connect.source, null, null, taskInitial) FOR this»«ELSE»«EXPAND closingKeys
(connect.source, connect.source.parent, null, taskInitial) FOR this»«ENDIF»«ELSEIF connect.source.parent == null &&
connect.source.parent.parent != null»«IF connect.source.parent.parent.name == taskInitial.name»«EXPAND closingKeys
(connect.source, null, null, taskInitial) FOR this»«ELSE»«EXPAND closingKeys (connect.source, null,
connect.source.parent.parent, taskInitial) FOR this»«ENDIF»«ELSEIF connect.source.parent != null &&
connect.source.parent.parent != null»«EXPAND closingKeys (connect.source, connect.source.parent,
connect.source.parent.parent, taskInitial) FOR this»«ENDIF»«EXPAND wayTasks (connect.target, taskInitial) FOR
this»«ENDIF»«ENDFOREACH»«ENDDEFINE»
«DEFINE treatedTask FOR Task»«IF this.toString().toString().subString(0,30).contains("metamodel::Init")»// Function
Main«ELSEIF this.toString().toString().subString(0,30).contains("metamodel::Final")»«ELSEIF
this.toString().toString().subString(0,30).contains("metamodel::Condition")» «EXPAND condition FOR this»«ELSEIF
this.toString().toString().subString(0,30).contains("metamodel::InfiniteWhile")»
«EXPAND whileInf FOR this»«ELSEIF this.toString().toString().subString(0,30).contains("metamodel::Write")»
«EXPAND opWrite FOR this»«ELSEIF this.toString().toString().subString(0,30).contains("metamodel::Read")»
«EXPAND opRead FOR this»«ELSEIF this.toString().toString().subString(0,30).contains("metamodel::DelayTime")» «EXPAND
opDelayTime FOR this»«ELSEIF this.toString().toString().subString(0,30).contains("metamodel::ElseCondition")» «EXPAND
```

```
elseCondition FOR this»«ELSEIF this.toString().toString().subString(0,30).contains("metamodel::FiniteWhile")» «EXPAND
whileFin FOR this»«ELSEIF this.toString().toString().subString(0,30).contains("metamodel::For")» «EXPAND for FOR
this»«ENDIF»«ENDDEFINE»

«DEFINE closingKeys (Task taskTratada, ControlStructure taskparent1, ControlStructure taskparent2, Task taskInitial) FOR
Workflow»«IF taskparent1 == null && taskparent2 == null»«IF taskTratada.parent.name == taskInitial.name»«IF
taskTratada.parent.internalTasks.last().name == taskTratada.name»
             }«ENDIF»«ENDIF»«ENDIF»«IF taskparent2 != null && taskTratada.parent.parent.name == taskparent2.name &&
taskTratada.parent.parent.internalTasks.size > 1 && taskTratada.parent.parent.internalTasks.last().name ==
taskTratada.parent.name && taskTratada.parent.internalTasks.last().name == taskTratada.name && taskparent2.name !=
taskInitial.name»«IF taskTratada.toString().toString().subString(0,30).contains("metamodel::Read") ||
taskTratada.toString().toString().subString(0,30).contains("metamodel::Write") ||
taskTratada.toString().toString().subString(0,30).contains("metamodel::DelayTime")»
             }«ENDIF»«ENDIF»«IF taskparent1 != null && taskTratada.parent.name == taskparent1.name &&
taskTratada.parent.internalTasks.size > 1 && taskTratada.parent.internalTasks.last().name == taskTratada.name &&
taskparent1.name != taskInitial.name»«IF taskTratada.toString().toString().subString(0,30).contains("metamodel::Read") ||
taskTratada.toString().toString().subString(0,30).contains("metamodel::Write") ||
taskTratada.toString().toString().subString(0,30).contains("metamodel::DelayTime")»
             }«ENDIF»«ENDIF»
«ENDDEFINE»
«DEFINE whileInf FOR Task»
      «ERROR "should not happen"»
«ENDDEFINE»
«DEFINE whileInf FOR InfiniteWhile»    while(1){«ENDDEFINE»
«DEFINE whileFin FOR Task»«ERROR "3should not happen"»«ENDDEFINE»
«DEFINE whileFin FOR FiniteWhile»
             «IF this.sign.toString().contains("HIGHER_THAN_OR_EQUALS")»
             while(«this.varPrimitiveDecl.name» >= «this.numIterations»){«ELSEIF
this.sign.toString().contains("LOWER_THAN_OR_EQUALS")»
             while(«this.varPrimitiveDecl.name» <= «this.numIterations»){«ELSEIF
this.sign.toString().contains("HIGHER_THAN")»
             while(«this.varPrimitiveDecl.name» > «this.numIterations»){«ELSEIF this.sign.toString().contains("LOWER_THAN")»
             while(«this.varPrimitiveDecl.name» < «this.numIterations»){«ELSEIF this.sign.toString().contains("EQUALS")»
```

```
            while(«this.varPrimitiveDecl.name» == «this.numIterations»){«ENDIF»«ENDDEFINE»
«DEFINE for FOR Task»
      «ERROR "should not happen"»
«ENDDEFINE»
«DEFINE for FOR For»
            for(int i=«this.initIteration» ; i<«this.maxIteration» ; i = i«IF
this.incrementOrDecrement.toString().contains("INCREMENT")»+«ELSE»-«ENDIF»«this.valueIncrementOrDecrement»)«IF
this.internalTasks.size == 1»«IF
!this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Write") ||
!this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Read")»{«ENDIF»«ELSEIF
this.internalTasks.size >= 1»{«ENDIF»«ENDDEFINE»
«DEFINE condition FOR Task»
      «ERROR "should not happen"»
«ENDDEFINE»
«DEFINE condition FOR ConditionBoolean»
            if(«this.varPrimitiveDecl.name» == «this.boolean»)«IF this.internalTasks.size == 1»«IF
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Write") ||
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Read")»«ELSE»{«ENDIF»«ELSEIF
this.internalTasks.size >= 1»{«ENDIF»«ENDDEFINE»
«DEFINE condition FOR ConditionValue»
      «IF this.sign.toString().contains("HIGHER_THAN_OR_EQUALS")»
            if(«this.varPrimitiveDecl.name» >= «this.varConstantDecl.value»)«ELSEIF
this.sign.toString().contains("LOWER_THAN_OR_EQUALS")»
            if(«this.varPrimitiveDecl.name» <= «this.varConstantDecl.value»)«ELSEIF
this.sign.toString().contains("HIGHER_THAN")»
            if(«this.varPrimitiveDecl.name» > «this.varConstantDecl.value»)«ELSEIF
this.sign.toString().contains("LOWER_THAN")»
            if(«this.varPrimitiveDecl.name» < «this.varConstantDecl.value»)«ELSEIF this.sign.toString().contains("EQUALS")»
            if(«this.varPrimitiveDecl.name» == «this.varConstantDecl.value»)«ENDIF»«IF this.internalTasks.size == 1»«IF
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Write") ||
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Read")»«ELSE»{«ENDIF»«ELSEIF
this.internalTasks.size >= 1»{«ENDIF»«ENDDEFINE»
«DEFINE condition FOR ConditionState»
```

```
	if(«this.varPrimitiveDecl.name» == «this.state.toString()»)«IF this.internalTasks.size == 1»«IF
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Write") ||
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Read")»«ELSE»{«ENDIF»«ELSEIF
this.internalTasks.size >= 1»{«ENDIF»«ENDDEFINE»
«DEFINE elseCondition FOR Task»
	«ERROR "should not happen"»
«ENDDEFINE»
«DEFINE elseCondition FOR ElseCondition»
		else«IF this.internalTasks.size == 1»«IF
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Write") ||
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Read")»«ELSE»{«ENDIF»«ELSEIF
this.internalTasks.size >= 1»{«ENDIF»«ENDDEFINE»
«DEFINE opWrite FOR Task»
	«ERROR "should not happen"»
«ENDDEFINE»
«DEFINE opWrite FOR Write» set(«this.varElemDecl.name», «this.value.toString()»);«ENDDEFINE»
«DEFINE opRead FOR Task»
	«ERROR "should not happen"»
«ENDDEFINE»
«DEFINE opRead FOR Read» «this.varPrimitiveDecl.name» = («this.varPrimitiveDecl.typePrimitiveDecl.toString()»)
get(«this.varElemDecl.name»);
«ENDDEFINE»
«DEFINE opWriteControl FOR Task»
	«ERROR "should not happen"»
«ENDDEFINE»
«DEFINE opWriteControl FOR Write»		set(«this.varElemDecl.name», «this.value.toString()»);«ENDDEFINE»
«DEFINE opReadControl FOR Task»
	«ERROR "should not happen"»
«ENDDEFINE»
«DEFINE opReadControl FOR Read»		«this.varPrimitiveDecl.name» =
(«this.varPrimitiveDecl.typePrimitiveDecl.toString()») get(«this.varElemDecl.name»);«ENDDEFINE»
«DEFINE opDelayTime FOR Task»
	«ERROR "should not happen"»
```

```
«ENDDEFINE»
«DEFINE opDelayTime FOR DelayTime»vTaskDelay(«this.milliseconds»);«ENDDEFINE»
«DEFINE opDelayTimeControl FOR Task»«ERROR "should not happen"»«ENDDEFINE»
«DEFINE opDelayTimeControl FOR DelayTime»                  vTaskDelay(«this.milliseconds»);«ENDDEFINE»
«DEFINE conditionControl FOR Task»«ERROR "should not happen"»«ENDDEFINE»
«DEFINE conditionControl FOR ConditionBoolean»                  if(«this.varPrimitiveDecl.name» == «this.boolean»)«IF
this.internalTasks.size == 1»«IF
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Write") ||
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Read")»«ELSE»{«ENDIF»«ELSEIF
this.internalTasks.size >= 1»{«ENDIF»«ENDDEFINE»
«DEFINE conditionControl FOR ConditionValue»«IF this.sign.toString().contains("HIGHER_THAN_OR_EQUALS")»
      if(«this.varPrimitiveDecl.name» >= «this.varConstantDecl.value»)«ELSEIF
this.sign.toString().contains("LOWER_THAN_OR_EQUALS")»          if(«this.varPrimitiveDecl.name» <=
«this.varConstantDecl.value»)«ELSEIF this.sign.toString().contains("HIGHER_THAN")»        if(«this.varPrimitiveDecl.name» >
«this.varConstantDecl.value»)«ELSEIF this.sign.toString().contains("LOWER_THAN")»         if(«this.varPrimitiveDecl.name» <
«this.varConstantDecl.value»)«ELSEIF this.sign.toString().contains("EQUALS")»      if(«this.varPrimitiveDecl.name» ==
«this.varConstantDecl.value»)«ENDIF»«IF this.internalTasks.size == 1»«IF
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Write") ||
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Read")»«ELSE»{«ENDIF»«ELSEIF
this.internalTasks.size >= 1»{«ENDIF»«ENDDEFINE»
«DEFINE conditionControl FOR ConditionState» if(«this.varPrimitiveDecl.name» == «this.state.toString()»)«IF
this.internalTasks.size == 1»«IF
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Write") ||
this.internalTasks.first().toString().toString().subString(0,30).contains("metamodel::Read")»«ELSE»{«ENDIF»«ELSEIF
this.internalTasks.size >= 1»{«ENDIF»«ENDDEFINE»
```