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

HuRoME: Entorno de Modelado para el Software de un Robot Humanoide

Juan Francisco Inglés Romero

Título del Proyecto

HuRoME: Entorno de Modelado para el Software de un Robot Humanoide

Autor

Juan Francisco Inglés Romero

Titulación

Master en Tecnologías de la Información y las Comunicaciones

Directores

Cristina Vicente Chicote
Diego Alonso Cáceres

Defensa

Cartagena, 27 de Septiembre de 2010

ÍNDICE

CAPÍTULO I

Introducción

Actualmente, el *Desarrollo de Software Dirigido por Modelos* (DSDM) [1] representa uno de los paradigmas de desarrollo software más en boga en el ámbito de la Ingeniería del Software. Las tecnologías en torno a este nuevo enfoque ofrecen una aproximación prometedora para superar las limitaciones expresivas de los lenguajes de programación de tercera generación, permitiendo a los diseñadores describir sistemas cada vez más complejos de manera más simple, gracias a la utilización de conceptos propios de sus dominios de aplicación [2]. El DSDM busca, por lo tanto, elevar el nivel de abstracción utilizado durante las distintas etapas del ciclo de vida del software.

El DSDM ha sido aplicado de forma exitosa en algunos dominios, como en el diseño de sistemas empotrados [3-5], o en redes de sensores [6]. Sin embargo, en el dominio de la robótica sólo es posible encontrar algunas referencias muy recientes, que apuntan hacia un creciente interés de la comunidad en este nuevo paradigma de desarrollo software [7-8]. Según [9] esto se debe a la "*falta crónica de normalización, interoperabilidad y reutilización del software*", especialmente en áreas como la robótica.

El presente proyecto trata de ilustrar los beneficios de aplicar el DSDM al ámbito de la robótica. Para ello, el entorno HuRoME (*Humanoid Robot Modeling Environment*), que se presenta en este proyecto, ofrece una aproximación al desarrollo de software para robótica utilizando un enfoque dirigido por modelos. Este entorno se plantea como un conjunto de herramientas diseñadas para facilitar el modelado de coreografías (secuencias de movimientos) y la modernización del software existente para el robot humanoide Robonova [10]. Así pues, HuRoME permitirá a los numerosos usuarios de Robonova, incluso a aquellos que adolecen de formación técnica específica sobre control o programación de robots, (1) modelar gráficamente y validar formalmente las secuencias de movimientos del robot (coreografías), (2) generar

automáticamente la implementación asociada a cada coreografía en el lenguaje específico, y (3) modernizar y reutilizar el software ya existente, permitiendo la obtención de los modelos equivalentes a cualquier programa existente. De esta forma, HuRoME no sólo facilita la generación automática de código a partir de modelos, sino también el proceso inverso, con la transformación de programas existentes (legacy code) en modelos, facilitando así su tratamiento (depuración, extensión, reutilización, análisis, simulación, etc.) con un nivel de abstracción mayor que el que proporciona el código fuente. En la Figura 1.1 podemos observar, de manera conceptual, el flujo de trabajo que presenta HuRoME.

Para llevar a cabo la implementación de HuRoME se ha optado por el soporte ofrecido por *Eclipse Modeling Project* [11] que provee un completo conjunto de herramientas relacionadas con el DSDM que permite, por ejemplo, transformaciones de modelos y generación de código, la definición de restricciones sobre los modelos utilizando el estándar OCL (*Object Constraint Language*) [12] o la creación de editores gráficos de modelos, mediante la herramienta GMF (*Graphical Modeling Framework*) [13]. Por último, la adopción del robot Robonova como plataforma robótica obedece a las siguientes razones: (1) Se trata de un robot comercial de bajo coste, pequeño y fácil de manipular. (2) Es una plataforma sencilla y extensible, con herramientas de simulación y programación. (3) El gran número de grados de libertad de sus articulaciones posibilita el diseño de movimientos y coreografías complejas. (4) Es un robot bastante extendido en el ámbito docente universitario e investigador (véase, por ejemplo, [14-15]) con multitud de código existente.

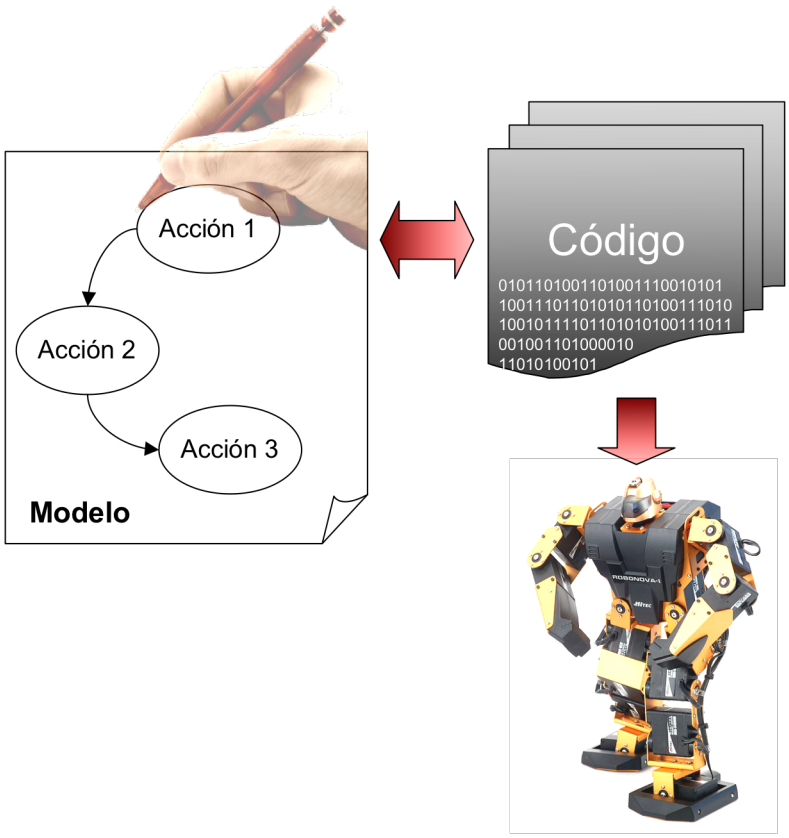


Figura 1.1.: Diagrama conceptual que refleja el proceso de desarrollo que propone el entorno HuRoME.

1.1. Objetivos

El primer objetivo que se planteó al inicio de este Proyecto fue el estudio del conocimiento técnico relacionado con la problemática introducida, en las siguientes líneas:

- Los fundamentos teóricos del Desarrollo de Software Dirigido por Modelos y las herramientas que dan soporte a este paradigma, en concreto, aquellas que componen el *Eclipse Modeling Project*.
- Tecnologías que conforman la plataforma robótica Robonova, como es el lenguaje de programación *RoboBASIC* [16], simuladores, etc.

En segundo lugar y como objetivo principal del Proyecto, proponemos el diseño e implementación de un entorno para el modelado de coreografías del robot Robonova basado en el paradigma DSDM.

Por último, la obtención de resultados y extracción de conclusiones sobre la herramienta desarrollada como iniciativa basada en DSDM y su impacto en el ámbito de la robótica. Planteamiento de mejoras y líneas futuras de trabajo.

1.2. Fases de desarrollo del Proyecto

El desarrollo de este Proyecto se ha llevado a cabo siguiendo las etapas que se resumen a continuación:

a) Definición y formalización del problema y su contexto. Concepción inicial de los objetivos del proyecto dirigida a perfilar la revisión bibliográfica posterior. Decisión del robot y la plataforma de desarrollo que permitirá abordar el proyecto en base a trabajos previos del grupo de investigación DSIE (División de Sistemas e Ingeniería Electrónica) y la propia experiencia de los directores del presente proyecto.

b) Estudio y análisis del robot humanoide Robonova. Estudio bibliográfico y experimentación práctica con el robot Robonova. Análisis del lenguaje de programación *RoboBASIC*.

c) Estudio y análisis de los fundamentos del DSDM y de las herramientas Eclipse asociadas.

d) Desarrollo del lenguaje de modelado para definir los movimientos del robot. Extracción de los conceptos y las relaciones entre éstos para conformar el lenguaje de modelado que se aplicará para describir los movimientos del robot.

e) Implementación de la herramienta. La implementación básica se lleva a cabo en tres etapas diferentes: (1) desarrollo del entorno gráfico de modelado de coreografías conforme al lenguaje especificado anteriormente, (2) generación automática de código *RoboBASIC* y *RoboScript*, y (3) modernización de código existente con la creación de un editor textual específico que posibilite la traducción de los programas a modelos.

f) Extracción de resultados y conclusiones, redacción de la memoria.

1.3. Organización de la memoria

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

- **Capítulo 2**: Breve revisión del robot humanoide Robonova.
- **Capítulo 3**: Revisión de los fundamentos del DSDM y la plataforma Eclipse Modeling Project.
- **Capítulo 4**: Desarrollo del entorno HuRoME.
- **Capítulo 5:** Guía de uso del entorno HuRoME.
- **Capítulo 6:** Exposición de resultados, conclusiones y trabajos futuros.
- Bibliografía

CAPÍTULO II

Breve revisión del robot humanoide Robonova

En este capítulo se introducen los conceptos básicos acerca del robot Robonova de la compañía Hitec Robotics. En la primera sección se describen algunos aspectos físicos relevantes del robot. En la sección segunda, se ofrece una visión general del lenguaje de programación roboBASIC empleado en el robot Robonova. Por último, en la sección tercera se analizan algunas herramientas que soportan el desarrollo software del robot.

2.1. Hardware del robot

El robot Robonova es un robot humanoide, completamente articulado, de algo más de 30cm de altura. La cinemática de este robot se compone de 5 articulaciones por cada pierna y 3 articulaciones por cada brazo, dando un total de 16 articulaciones. Así pues, el cuerpo del robot lo compone una estructura formada por 16 servomotores digitales, uno para cada articulación, unidos por pletinas de aluminio anodizado que le confieren gran rigidez. El resto del cuerpo lo forman el circuito de control, una batería recargable de 5 células NiMH y piezas de plástico rígido de protección, opcionalmente, el robot puede estar dotado con distintos tipos de sensores. A continuación repasaremos rápidamente algunos de los detalles más importantes de estos elementos.

2.1.1. Servomotor digital

Un servomotor, básicamente, es un motor donde la posición angular de su eje es controlada por un microcontrolador, éste eleva el nivel de abstracción de la interfaz de control con el objeto de realizar operaciones de mayor complejidad y precisión. Los servomotores empleados en el robot Robonova, son los HSR-8498 de Hitec basados en el microcontrolador de 8 bits ATmega8 de Atmel. Este servomotor destaca por sus características avanzadas como son la interfaz HMI (Hitec Multi-protocol Interface) que incluye funciones programables así como la posibilidad de

leer desde el controlador la posición, la tensión y el consumo actual de corriente, lo que permite crear sistemas articulados avanzados.

El término multi-protocolo en HMI indica la habilidad del servomotor de funcionar bajo tres modos diferentes:

- ***Standard Pulse Mode.*** Se trata de un modo de operación compatible con la mayoría de servomotores actuales. En este modo el servomotor responde a señales PWM (Pulse Width Modulation) en el rango de 550-2450μs de duración. La duración de un pulso simple codificará la posición a la que se desea mover el servomotor, siguiendo la siguiente relación matemática,

$$\alpha_{servo}(grados) = \frac{Pulso(\mu s) - 1500}{10}$$

  Por ejemplo, un pulso de 2100μs indica un movimiento hasta la posición de 60º respecto a un punto central predefinido.

- ***Extended Pulse Mode***. Además de la funcionalidad estándar expresada en el punto anterior, este modo permitiría leer la posición actual del servomotor, para ello, se genera un pulso de entrada de 50μs, el servomotor responderá con un pulso de duración proporcional a la posición actual según la fórmula añadida anteriormente. Este modo también facilitaría la configuración de parámetros que describen el movimiento del servomotor, en concreto, *D Gain* (indica cómo de suave el servomotor se detiene tras su accionamiento), *Dead Zone* (especifica el rango de precisión del punto de parada), *P Gain* (determina la velocidad y potencia del movimiento).

- ***Serial Mode.*** Se trata de un enlace bidireccional serie a 19200bps, a través de éste pueden ser ejecutados comandos complejos y recibir un feedback más rico, por ejemplo, lectura de voltajes y corrientes, configuración precisa de velocidad, etc.

Por último, comentaremos que es el ancho de los pulsos lo que determina el modo de operación del servomotor. Así, pulsos entre 550-2450μs corresponden al *Standard Pulse Mode*, pulsos entre 50-200μs al *Extended Pulse Mode* y pulsos de 416μs al *Serial Mode*.

2.1.2. Circuito de control

El elemento principal del circuito de control lo compone el microcontrolador Atmel ATmega128 [17]. Se trata de un microcontrolador de 8 bits de bajo consumo y altas prestaciones. Presenta una arquitectura RISC (Reduced Instruction Set Computer) con 133 instrucciones, la mayoría ejecutables en un solo ciclo de reloj. Además, incluye un banco de 32 registros de uso general más otros destinados al control de periféricos, e incorpora hardware específico para realizar multiplicaciones en tan sólo dos ciclos de reloj, todo ello, permite alcanzar una potencia de cálculo de 16MIPS (Millones de Instrucciones por Segundo) a 16MHz. La memoria de programa del microcontrolador lo compone una Flash de 128KB, esta memoria interna está ampliada con una memoria externa EEPROM de 64Kb lo que le permite ejecutar gran cantidad de pasos y movimientos de forma autónoma, además incorpora una memoria RAM de datos de 256 bytes. Cuenta con un total de 40 puertos de entrada y salida capaces de controlar 24 servos, también se incorporan 8 puertos conversores A/D, 3 salidas PWM y un puerto serie de alta velocidad. En la figura 2.1c podemos observar una imagen del circuito de control empleado en el Robonova.

2.1.3. Sensores

Aunque en su versión más básica el robot Robonova no incluye sensores, con el fin de desarrollar programas complejos y dado el elevado número de puertos de E/S que presenta la arquitectura de control, el usuario puede añadir de forma flexible algunos elementos adicionales de sensorización. A continuación, detallamos algunos de éstos.

- ***Sensor de proximidad***. Este sensor permite detectar objetos situados a una cierta distancia a través de la reflexión sobre el obstáculo de una onda emitida desde el robot. En función de la naturaleza de esta onda podemos encontrar sensores de proximidad basados en ultrasonidos, por ejemplo, el sensor Maxsonar EZ1 que detecta objetos en un rango entre 0 y 6.45 metros de distancia. Y sensores de proximidad basados en infrarrojos, como el Sharp GP2D12 que mide distancias en un rango de 10 a 80cm.
- ***Sensor de sonidos***. Éste produce una señal de salida en función del nivel sonoro. Lo que permite que el robot reaccione cuando se produce cualquier tipo de sonido, por ejemplo, es posible programar que el robot haga un movimiento cada vez que se da una palmada.
- ***Sensor de luz***. Este sensor reacciona ante el nivel de luz ambiental, de manera, que se facilita la discriminación de zonas con diferentes iluminaciones. Resulta útil para hacer seguimiento de haces luminosos o responder ante cambios repentinos de iluminación.
- ***Sensor de inclinación***. Este sensor se basa en un acelerómetro de 2 ejes. Por ejemplo, para el robot Robonova encontramos el circuito ADXL322 de Analog Devices donde las salidas suministran un nivel de tensión proporcional al ángulo en el plano X e Y del sensor. De esta forma, el sensor puede utilizarse para saber el grado de inclinación en el que se encuentra el circuito en dos planos.

(a) (b) (c)

Figura 2.1.: (a) Robot Robonova. (b) Servomotor HSR-8498 de Hitec. (c) Circuito de control Robonova.

2.2. El lenguaje de programación roboBASIC

RoboBASIC es un lenguaje de programación diseñado para el control de robots y originado a partir de BASIC, por lo tanto, se trata de un lenguaje no estructurado extendido con comandos de control específicos del robot. Seguidamente, se resumen los aspectos y capacidades principales del lenguaje basándonos en las especificaciones [16].

2.2.1. Declaración y definición de los datos

En cuanto a la declaración de tipos de datos, dado que roboBASIC está orientado al control del hardware de un robot, éste sólo soporta dos tipos: BYTE y INTEGER. Como su nombre indica, el tipo BYTE quedaría limitado a datos de un byte de tamaño, por lo que, se podrían representar enteros comprendidos entre 0 y 255. En cuanto a INTEGER, se emplean dos bytes para codificar el dato, lo que correspondería a un entero entre 0 y 65535. Además, roboBASIC no soporta la representación de valores con signo, ni representación decimal, por lo que todas las operaciones se basan en aritmética entera sin signo. Aunque en las especificaciones del lenguaje se recomienda, por el carácter cercano al hardware, seguir una notación hexadecimal para escribir el valor de las variables y constantes que definamos en un programa, es posible emplear también una notación octal, binaria o decimal, tan sólo bastaría anteponer al valor una cadena especial en función de la representación, por ejemplo, en &B111101, "&B" indica el uso de una notación binaria. Por último, comentar que el lenguaje no define funciones de conversión entre tipos de datos, tampoco adoptaría el concepto de array de datos.

2.2.2. Operadores

Al igual que en el lenguaje BASIC, en roboBASIC podemos encontrar operadores aritméticos, relacionales, lógicos y de control de bits de un valor.

- ***Operadores aritméticos.*** Como en la mayoría de lenguajes, se define la operación de suma (+), diferencia (-), multiplicación (*), división (/) y módulo (% o MOD). Sin embargo, existen diferencias importantes que deben tenerse en cuenta. En primer lugar, tal y como se introdujo anteriormente, roboBASIC sólo soporta aritmética entera. En segundo lugar, no se define un orden de precedencia de las operaciones ni se permite el uso de paréntesis para este mismo fin, así, por ejemplo, mientras en BASIC "1+2*3 = 1+6 = 7", en roboBASIC "1+2*3 = 3*3 = 9". Por último, secuencias complicada de operaciones pueden causar errores inesperados (por ejemplo, según la documentación, la operación "D = A*B+C" podría ejecutarse sin problemas, mientras que "F = A*B/C*D+E" acarrearía errores).
- ***Operadores relacionales.*** Son utilizados para comparar dos valores, su uso es idéntico al lenguaje BASIC.
- ***Operadores lógicos.*** Son utilizados para realizar combinaciones de operaciones relacionales para su uso en sentencias condicionales, de nuevo, no aporta ninguna diferencia sustancial respecto a BASIC.
- ***Operadores de bit.*** Son operadores que permiten realizar acciones a nivel de bit sobre las variables, éstos facilitan algunas acciones fundamentales del robot, como el control de los puestos de entrada y salida. Las acciones posibles son desplazamiento lógico a la derecha (>>) o a la izquierda (<<), acceso a un bit específico de la variable (por ejemplo, A.0 representa el bit de menor peso de una variable A), y operaciones lógicas (OR, XOR, AND).

2.2.3. Control de flujo

El lenguaje roboBASIC presenta los mismos comandos básicos que BASIC para manejar el flujo de un programa. Encontramos la sentencia condicional IF…THEN, IF anidada, etc. Sentencia iterativa FOR…NEXT. Sentencia GOTO para ir a puntos etiquetados del programa. Llamadas a

subrutinas y retorno GOSUB…RETURN (con un máximo de 5 llamadas anidadas). Por otro lado, el sistema empotrado del Robonova permite control de tiempo real y por lo tanto, da soporte a las sentencias WAIT y DELAY. En cuanto a WAIT, normalmente, cuando un movimiento es ejecutado, el programa continua su ejecución secuencial independientemente de si los servomotores han terminado de moverse, así, empleando el comando WAIT es posible controlar que la ejecución del programa se reanude sólo cuando se concluye la ejecución del comando anterior.

2.2.4. Control de los motores

El lenguaje roboBASIC incluye comandos que permiten controlar los servomotores del robot. El rango de un servomotor se extiende desde -90º a +90º, en el lenguaje, un valor angular en grados es expresado como un valor numérico entre 10 y 190, dado que éste no soporta valores negativos. En este apartado, algunos de los comandos más significativos son MOVE, SPEED y ACCEL. El primero permite operar sobre varios servomotores al mismo tiempo, la sintaxis de éste la conforma una cadena de valores numéricos separados por comas, donde el valor i-ésimo indica la posición angular a la que debe derivar el movimiento del i-ésimo servomotor. Un espacio en blanco en lugar de un valor indica que no se desea mover el correspondiente motor. El comando SPEED, como su nombre indica, permite configurar la velocidad a la que se ejecutará los posteriores movimientos en los servomotores, la velocidad es cuantificada como un valor entero entre 1 y 16, valor proporcional a la velocidad que se adoptará. Por último, el comando ACCEL configura el grado de aceleración de los consiguientes movimientos en los servomotores, este parámetros es ajustado como entero entre 0 y 15, siendo 0 la mínima aceleración.

2.2.5. Entrada y salida digital

En el apartado 2.1.2. comprobamos que el microcontrolador montado en el robot Robonova está dotado con 40 puertos de entrada y salida, dirigidas al control de estos puertos roboBASIC incluye algunas sentencias útiles. Por ejemplo, para leer el valor de los puertos puede ser empleado el comando IN o BYTEIN. Ejecutando IN seguido de un entero que identifique el puerto en cuestión, podremos almacenar la señal binaria (‘0’ o ‘1’) del puerto en una variable. Con BYTEIN podremos almacenar los valores de un grupo de 8 puertos consecutivos como un BYTE. Para escribir los puertos encontramos comandos OUT y BYTEOUT cuyo comportamiento es similar al comentado con la acción de lectura. También se puede recurrir a las funciones PULSE y TOGGLE para modificar el valor de los puertos de salida, la primera permite enviar un pulso de 5μs de duración por el puerto de salida seleccionado, la segunda conmuta el valor actual del puerto de salida. Por último, resaltar dos funciones incluidas en el lenguaje, especialmente diseñadas para obtener información de botones. Cuando un botón es pulsado una vez, se producen cientos o miles de fluctuaciones eléctricas transitorias, este fenómeno es llamado *Chattering* y puede producir errores de lectura, para evitar este fenómeno se ha incluido una protección a nivel de software que se materializa en los comandos INKEY y KEYIN.

2.2.6. Manejo de la memoria

En el lenguaje RoboBASIC aparecen comandos que facilitan el uso de las memorias disponibles en el circuito de control del robot, estas memorias son, la RAM interna del microcontrolador y una memoria EEPROM externa de 64Kb. El programador podrá emplear estos elementos para almacenar datos durante la ejecución del programa. Los comandos empleados para leer y

escribir una dirección de la memoria RAM son PEEK y POKE, en cambio para acceder a la memoria EEPROM se emplean los comandos ROMPEEK y ROMPOKE.

2.2.7. Comunicaciones externas

El robot Robonova implementa una interfaz compatible con RS232 para realizar comunicaciones en serie con un PC o con otro robot a una velocidad máxima de 115200bps. Adicionalmente, estas comunicaciones pueden ser sin cables en el caso de que se adopten los dispositivos de radiofrecuencia oportunos. Actualmente, RoboBASIC cuenta con comandos que permiten transmitir (comandos TX y ETX) y recibir información (comandos RX y ERX) de forma sencilla. Por último, comentar que versiones anteriores del hardware permitían realizar comunicaciones en bus a través de la interfaz llamada miniBUS, así pues, podemos encontrar en el lenguaje algunas funciones para el manejo de estas comunicaciones aunque éstas estén, a día de hoy, en desuso (comandos MINIIN y MINIOUT).

2.2.8. Procesamiento de señales analógicas

Tal y como se introdujo en el correspondiente apartado destinado al hardware, el microcontrolador Atmel ATmega128 dispone de 8 puertos de entrada analógicos que permiten realizar conversiones analógico - digitales, la sentencia en roboBASIC encargada del control de esta funcionalidad es AD, que devuelve el resultado de la conversión pudiendo ser éste asignado a una variable. Además, el puerto analógico 7 es utilizado para recibir la señal infrarroja del control remoto que incluye de serie el robot, normalmente, con este control el usuario puede seleccionar la ejecución de uno entre varios programas almacenados en la memoria EEPROM, sin embargo, el lenguaje permite al programador utilizar esta capacidad para realizar algoritmos que respondan a este control remoto. Para leer el botón pulsado por el usuario se utiliza el comando REMOCON. Por otro lado, dado que el robot soporta la extensión de su hardware con múltiples sensores, roboBASIC incluye funciones específicas para el control de las extensiones más habituales, así, los puertos digitales de entrada y salida 0 a 23 permiten ser empleados como puertos dedicados a sensores de proximidad de ultrasonidos, el comando que maneja esta funcionalidad es SONAR. De la misma forma, el lenguaje dispone de sentencias para control de un posible giroscopio (GYRODIR, GYROSET y GYROSENSE).

2.2.9. Control de una pantalla LCD

Hemos comentado anteriormente que el diseño del robot aboga por una extensión sencilla del hardware, con el fin de mejorar la interacción con el usuario y posibilitar el desarrollo de una interfaz robot-humano más sofisticada, una de estas extensiones es el módulo LCD. Se trata de una pantalla de dimensiones reducidas que permite visualizar información textual y numérica. Para el control de este elemento, se dedican comandos específicos en el lenguaje, por ejemplo, para inicializar el módulo (LCDINIT), para borrar la pantalla (CLS), para escribir información en la pantalla (PRINT) y otros relativos al formato de la información (FORMAT, DEC, BIN y HEX).

2.2.10. Otras funciones de interés

El lenguaje roboBASIC incluye un comando (RND) para generar números aleatorios enteros entre 0 y 255, éste puede resultar útil para desarrollar algoritmos de inteligencia artificial. También, el lenguaje nos permite diseñar sonidos y melodías que podrán ser reproducidos por el robot, así encontramos la sentencia BEEP que reproduce un pitido y otras sentencias más

complejas como PLAY, SOUND o MUSIC que permiten componer melodías y sonidos complejos.

2.3. Herramientas software de desarrollo

En esta sección se revisan las principales herramientas disponibles en la versión 2.72 del SDK (Software Development Kit) del robot Robonova que se distribuye de forma gratuita [18]. La empresa titular del copyright del software que se examina a continuación es Hitec RCD Korea, Inc.

2.3.1. Entorno de programación RoboBASIC

Como se puede contemplar en la figura 2.2, se trata de un entorno de programación al estilo tradicional, donde el elemento fundamental lo compone un editor que resalta y formatea el código.

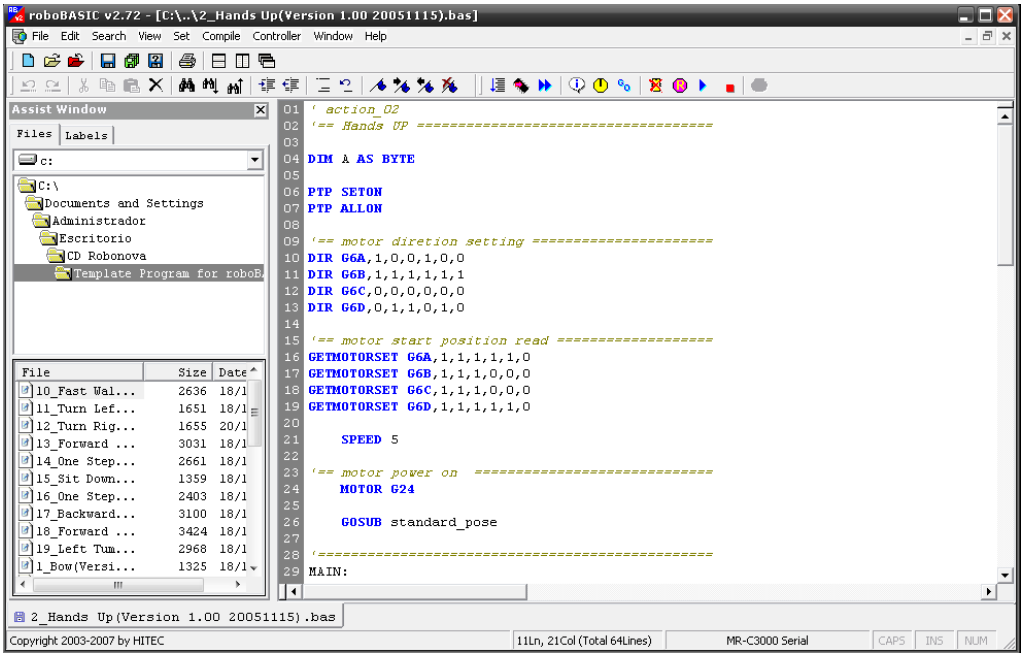
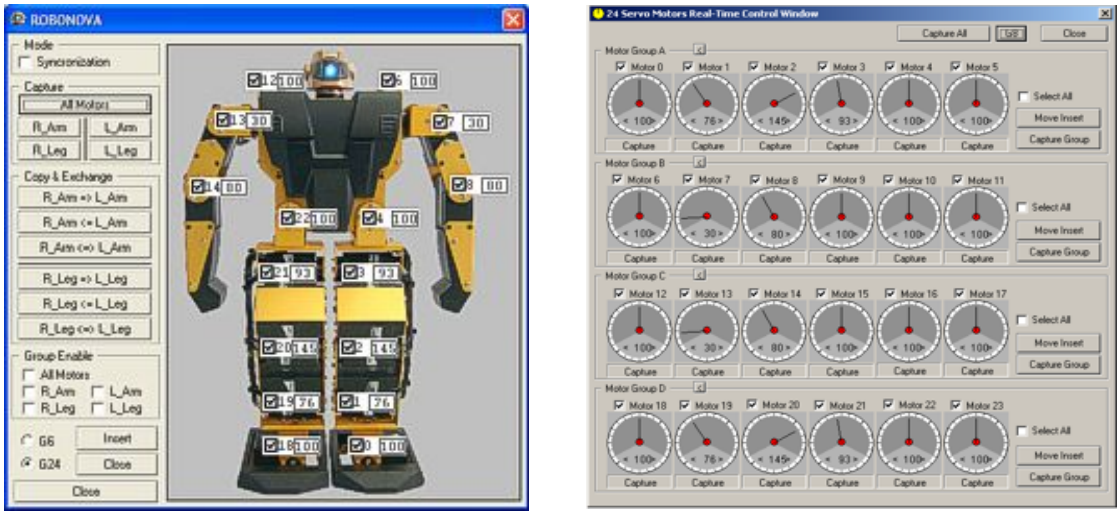


Figura 2.2.: Entorno de programación RoboBASIC v2.72

Algunas de las acciones más importantes a las que el usuario tiene acceso son:

- Compilación del código y descarga directa en el robot.
- *Direct Line Control*. Utilizado para ejecutar directamente sobre el robot una línea de código (válido solamente para sentencias que acarrean movimientos en el robot o accionamientos sobre puertos de salida: OUT, MOVE, SERVO, POS y MOVEPOS).
- *Robonova Motor Control*. Permite configurar un movimiento del robot de forma intuitiva y gráfica, seleccionando los servomotores y la posición deseada de éstos (figura 2.3a).
- Función *Catch&Play*. Permite monitorear la posición de los servomotores del robot, así, un usuario podrá manipular el robot para adoptar la postura deseada, tras ello, accionar el correspondiente botón para insertar en el programa el código de un nuevo movimiento que derive a dicha posición (figura 2.3b).



(a) (b)

Figura 2.3.:(a) Ventana correspondiente a la función Robonova Motor Control. (b) Ventana Catch&Play

2.3.2. Entorno de programación RoboScript

RoboScript es un intento de simplificar la programación del robot Robonova, la idea que gravita en RoboScript consiste en reducir al máximo el lenguaje roboBASIC para obtener un subconjunto esencial de instrucciones que permita describir las acciones más habituales en el robot, esto es, secuencias de movimientos. Los comandos que conforman roboScript son los siguientes: MOVE24, SPEED, DELAY, GOTO, PTP ALLON y PTP ALLOFF, con estos comandos es posible definir movimientos, configurar la velocidad, insertar retardos, ir a una determinada línea de código y activar o desactivar el control simultáneo de los servomotores (indica si, tras una sentencia MOVE24, todos los servomotores terminan a la vez o no). En la figura 2.4 puede observarse la interfaz del entorno de programación roboScript.

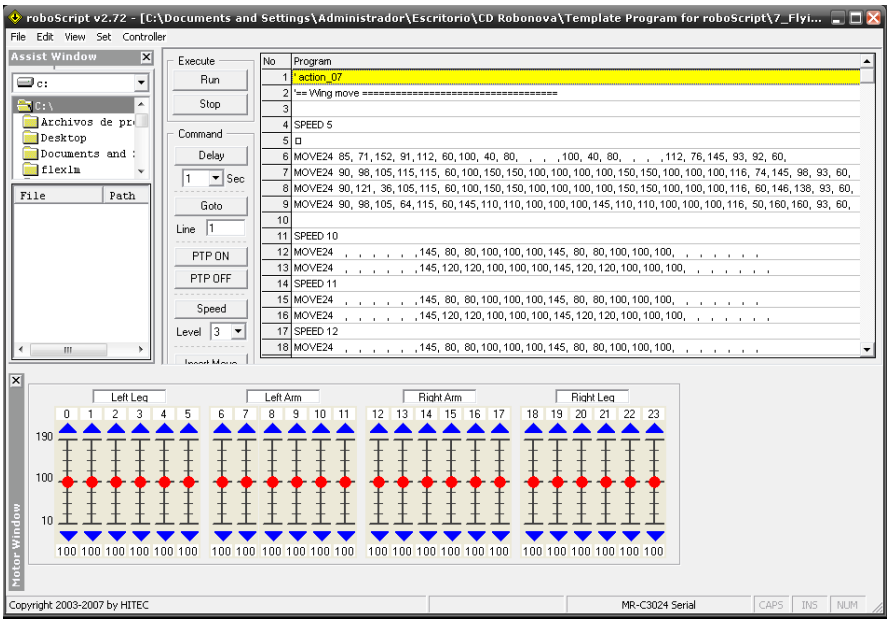


Figura 2.3.: Entorno de programación RoboScript v2.72

2.3.3. RoboRemocon

El propósito de esta herramienta no está dirigido a programar el robot, sino que proporciona un control remoto virtual para testear las acciones programadas cuando se pulsa un botón del mismo. De esta manera, se facilita la configuración y comprobación de las comunicaciones infrarrojas conservando la batería e integridad del dispositivo físico. En la figura 2.4 podemos apreciar la apariencia de esta herramienta.

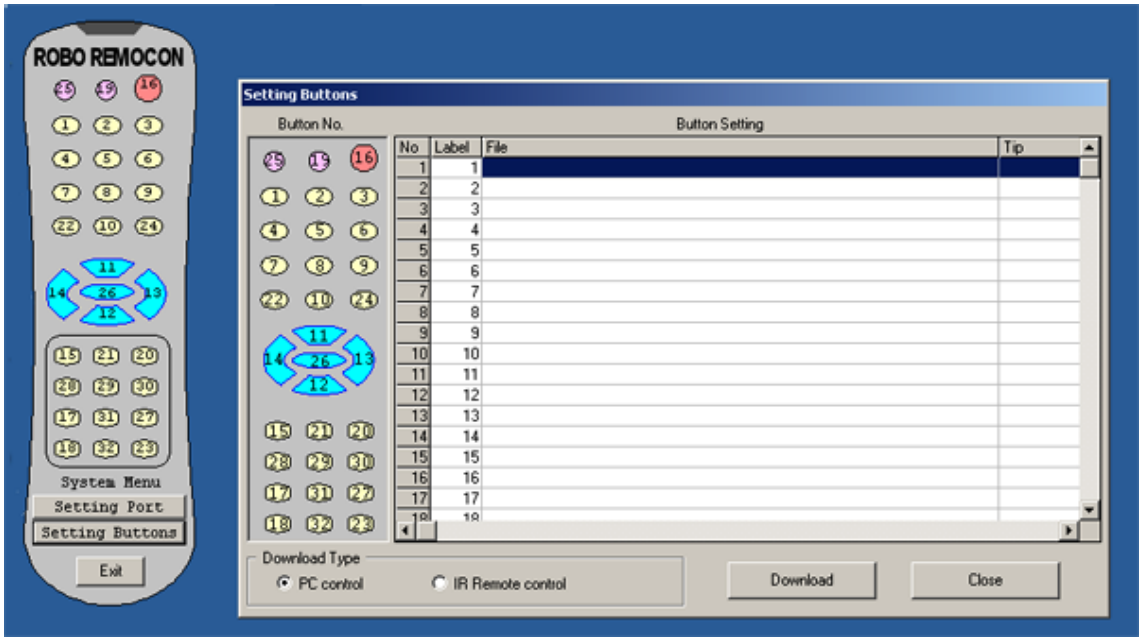


Figura 2.4.: Entorno RoboRemocon v2.72

2.3.4. Simulador SimROBOT0

Se trata de un entorno de simulación exclusivo del robot Robonova ofrecido por Hitec RCD Korea, Inc. Actualmente, el simulador se encuentra en una temprana versión 1, por lo que, aunque permite multitud de funcionalidades, es posible encontrar algún que otro *bug*. Las capacidades principales del simulador son las siguientes:

- El simulador se centra en la recreación de los movimientos del robot, así pues, ofrece una vista virtual del robot en 3D para que el usuario pueda observar en detalle la ejecución de la coreografía. Permite cambiar la perspectiva de la vista y añadir diferentes efectos visuales.
- Admite ficheros escritos en roboScript (extensión *.rsf) para ser simulados, sin embargo, no soporta programas en roboBASIC dado que el entorno no es capaz de simular todas las capacidades que ofrece este lenguaje.
- Permite la edición de secuencias de movimientos sobre la misma representación virtual del robot, así, el usuario puede mover las diferentes articulaciones del robot virtual empleando el ratón e insertar la posición configurada en el programa. También, puede definirse un nuevo movimiento añadiendo directamente los valores angulares de los distintos servomotores. Una vez editado el programa es posible descargarlo en el propio robot físico conectado al PC.

- Dispone de una librería con 43 movimientos que el usuario puede emplear para hacer composiciones más complejas. Por otro lado, el entorno facilita la sincronización de la secuencia de movimientos con un clip musical o vídeo.
- Emplea ficheros de formato propietario con extensión *.srm para almacenar el programa editado.

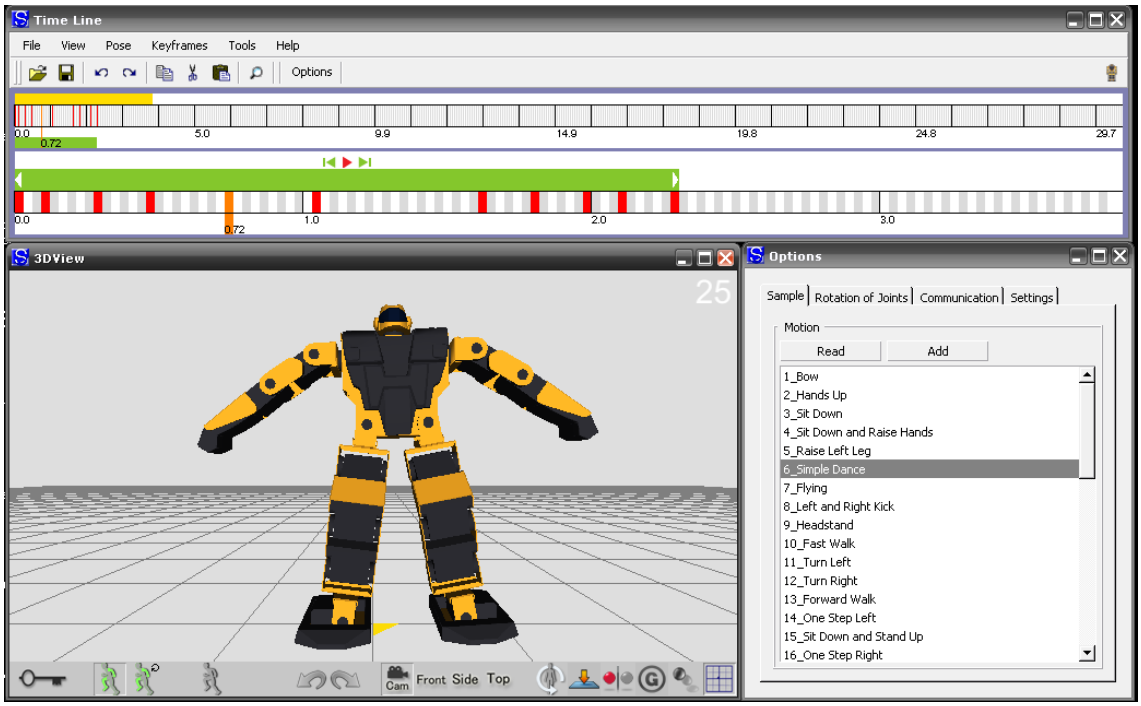


Figura 2.5.: Interfaz de usuario del entorno SimROBOT0

CAPÍTULO III

Desarrollo de Software Dirigido por Modelos

En este capítulo de la memoria se estudian las nociones fundamentales y las técnicas más relevantes del Desarrollo de Software Dirigido por Modelos. Así pues, el capítulo comienza con la revisión de algunos conceptos teóricos de este paradigma, tras ello, nos centramos en la propuesta de implementación MDA (Model Driven Architecture) de la OMG (Object Management Group) [26], y por último, abordamos herramientas concretas y funcionales enmarcadas en el Eclipse Modeling Project que permiten desarrollar los diferentes aspectos del paradigma.

3.1. Fundamentos básicos

El Desarrollo de Software Dirigido por Modelos (DSDM) comprende un conjunto de técnicas y herramientas que permiten modelar formalmente los sistemas que se quieren desarrollar para, posteriormente, aplicando una serie de transformaciones automáticas, obtener el código final de las aplicaciones. Así, el DSDM gira entorno a la definición y el uso sistemático de modelos y de transformaciones de modelos a lo largo de todo el ciclo de vida del desarrollo de software.

La noción de modelo es muy antigua y puede definirse como “una abstracción o simplificación de la realidad con un cierto propósito” [19]. Más concretamente, de acuerdo a la definición proporcionada por Bézivin y Gerbé en [20], “un modelo es una simplificación de un sistema construido con un objetivo definido. El modelo ha de ser capaz de responder a las preguntas que se le formulen, como si se tratara del sistema real. Las respuestas que proporciona el modelo deberán ser exactamente las mismas que si respondiera el sistema, con la condición de que las preguntas se formulen en los mismos términos en los que se definió el modelo”. Así, se puede decir que un modelo es una simplificación de un sistema que proporciona información sobre el mismo en el contexto de unos ciertos objetivos.

Un sistema puede estar representado por uno o más modelos, cada uno de ellos centrado en representar un determinado aspecto de interés con un cierto grado de detalle (nivel de abstracción). Así, es posible crear modelos de análisis, de diseño, e incluso modelos de implementación, muy próximos a la plataforma de desarrollo que se empleará para codificar el sistema final. Los modelos de más alto nivel pueden evolucionar a otros de más bajo nivel, mediante la aplicación de *transformaciones automáticas de modelos*, hasta obtenerse un modelo lo suficientemente detallado como para poder generar, a partir de él, el código final del sistema. Así, el modelo (o conjunto de modelos) que representa al sistema junto con las transformaciones *modelo-a-modelo* y *modelo-a-código*, permiten automatizar, en gran medida, el proceso de desarrollo de software.

Para definir cualquier modelo es necesario contar con un lenguaje de modelado. Los meta-modelos definen formalmente la sintaxis abstracta de los lenguajes de modelado [21], recogiendo sus conceptos (palabras del lenguaje), así como las reglas que indican cómo se pueden combinar dichos conceptos para formar modelos válidos. De este modo, como se indica en [20], "*un modelo solamente será válido si es conforme a su meta-modelo*". Por otra parte, la representación (ya sea gráfica o textual) asociada a cada uno de los elementos del meta-modelo (conceptos y relaciones) constituye la sintaxis concreta del lenguaje. Por último, la semántica asociada a los elementos de cada meta-modelo, esto es, su significado o interpretación, tiene impacto fundamentalmente en las transformaciones de modelos, en las que se define cómo transformar (interpretar) cada concepto del modelo de partida, en términos del otro lenguaje (ya sea de modelado o de programación).

La figura 3.1 sitúa los conceptos anteriormente abordados. El esquema 3.1a ayuda a situar la noción de meta-meta-modelo, que permite definir un lenguaje para especificar otros lenguajes, así, los meta-modelos originados serían conformes a su meta-meta-modelo. No obstante, dado que esta dinámica de ir acumulando términos "meta" podría resultar infinita, en el DSDM, los meta-meta-modelos se definen de forma reflexiva, es decir, se emplea el propio metalenguaje para definirse a sí mismo (ver pirámide en la figura 3.1b), a este nivel, podemos encontrar el estándar MOF (MetaObject Facility Specification) [27] de la OMG.

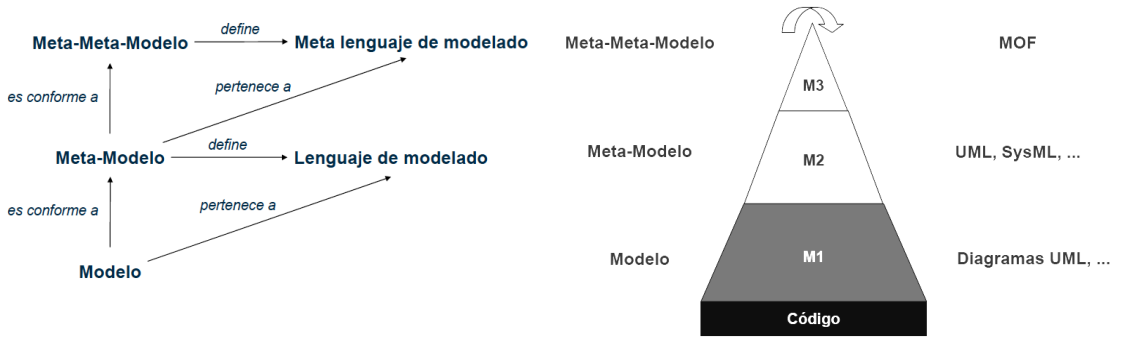


Figura 3.1.: Esquemas conceptuales que sitúan los conceptos del DSDM. (Fuente [33])

Actualmente, el DSDM es uno de los paradigmas de desarrollo de software más en boga en el ámbito de la Ingeniería del Software. Los fundamentos sobre los que asienta el DSDM fueron establecidos hace ya un par de décadas. Sin embargo, el auge de este enfoque sólo ha sido posible en los últimos años gracias, por una parte, a la encomiable labor de estandarización que, en el ámbito del DSDM, ha llevado a cabo la OMG (Object Management Group) con su iniciativa MDA (Model-Driven Architectura) [22], y por otra, a la aparición en el mercado de las primeras herramientas que dan soporte a este nuevo enfoque, permitiendo explotar todo su potencial.

Entre las herramientas que actualmente dan soporte al DSDM, cabe mencionar las siguientes: DSL Tools (de Microsoft) [23], Meta-Edit+ (de la empresa Meta-Case) [24] y Eclipse [25]. Esta última es una plataforma abierta y de libre distribución que ofrece, entre otras muchas funcionalidades relacionadas con el desarrollo de software, un nutrido grupo de plug-ins relacionados con el DSDM. En los últimos años, Eclipse se ha convertido en un estándar de facto para la comunidad de DSDM, ya que implementa las principales tecnologías estandarizadas por el OMG para dar soporte a este enfoque.

3.2. Arquitectura Dirigida por Modelos

La Arquitectura Dirigida por Modelos (MDA – Model Driven Architecture) es una iniciativa del OMG que propone utilizar modelos durante todas las etapas del proceso de desarrollo de software. En este sentido, los modelos se podrían considerar como un nuevo 'lenguaje de programación' (es decir, se alcanza un nuevo nivel de abstracción con respecto a lenguajes de alto nivel como Java o C++). De este modo, como indica [26], se pueden encontrar compiladores capaces de traducir modelos de datos y de aplicaciones definidos en MOF [27] y UML [28] a un lenguaje de alto nivel y a plataformas que implementan sistemas vigentes. En MDA, MOF es el meta-meta-modelo que permite definir lenguajes de meta-modelado, mientras que UML es un meta-modelo conforme a MOF que define un lenguaje de modelado.

Algunas de las ideas centrales de MDA son las siguientes:

- ***Especificación vs. Implementación.*** Tal y como se explica en [26], MDA es "una técnica que permite separar la especificación de funcionalidad de la especificación, de la implementación de esa funcionalidad en una plataforma tecnológica concreta". Claramente, esto supone un gran avance ya que se va a favorecer la reutilización de todos los modelos que sean independientes de una implementación concreta, porque aunque cambie la tecnología, estos modelos permanecerán inalterados. Sólo habrá que modificar aquellos modelos que representen una tecnología concreta para que se adapten a las nuevas propiedades.
- ***Integración***. Uno de los objetivos de MDA consiste en resolver los problemas de integración proporcionando unas especificaciones detalladas basadas en modelos estándar, que permiten obtener modelos de datos y componentes software interoperables, reutilizables, y portátiles mediante la separación arquitectónica de vistas. De este modo, MDA va a permitir especificar la estructura de cada componente, así como toda la semántica asociada relacionada con el comportamiento del componente a lo largo de su ciclo de vida (es decir, MDA va a permitir realizar una especificación funcional completa).

- ***Arquitectura software***. La arquitectura en MDA queda definida como una especificación de la estructura, pero no de un sistema, sino de un conjunto de tecnologías y estándares, las interrelaciones entre sus partes, y de cómo utilizar estas tecnologías y estándares en el desarrollo de un sistema. Efectivamente, tal y como afirma [29] modelar la arquitectura del software ayuda a los arquitectos a desarrollar sus soluciones de una manera flexible, con la posibilidad de reutilizar los esfuerzos existentes en el contexto de nuevos servicios que implementan la funcionalidad del negocio en un tiempo aceptable, aun cuando la infraestructura deseada siga evolucionando.

Así pues, MDA tiene que ver con todo aquello que sea *lenguajes de modelado*, *entornos de desarrollo*, herramientas que permitan especificar *lenguajes de modelado específicos del dominio*, tecnología para el intercambio de modelos a través de una cadena de herramientas, *transformaciones* que usan *reglas de negocio* para el desarrollo de modelos software, traducción de modelos a otros tipos de lenguajes, utilización de modelos en la ingeniería de sistemas, redirección de modelos y software de una plataforma a otra, y finalmente, por supuesto, generación de sistemas software a partir de modelos y especificaciones de las arquitecturas de las plataformas.

El hecho es que MDA debe su origen a la gran variedad de tecnologías, paradigmas, etc., que existen en la actualidad, y las que van apareciendo. Como ya se ha dicho, MDA parte de la idea de elevar el nivel de abstracción en el desarrollo de sistemas software a los modelos, y propone utilizar modelos para dirigir las distintas actividades que comprende todo desarrollo de un sistema software a través de transformaciones: entendimiento, diseño, construcción, despliegue, operación, mantenimiento y modificación. La *transformación de modelos* se puede definir como el proceso que convierte un modelo del sistema en otro modelo del mismo sistema. Así, a través de transformaciones de modelos se podrá convertir un modelo que especifica el sistema en otro modelo del mismo sistema, pero para una plataforma concreta.

MDA clasifica los modelos en tres categorías o vistas: a) CIM (*Computation Independent Model*): este modelo del sistema se correspondería con el modelo de dominio expresado en el lenguaje propio de los expertos del dominio, y que es independiente (es decir, no aporta ningún tipo de información) de la arquitectura e implementación del sistema; b) PIM (*Platform Independent Model*): este modelo se obtiene a partir del CIM y es independiente de la plataforma de desarrollo (es decir, tecnología, entorno de ejecución) que se utilice para desarrollar el sistema. El PIM constituye la arquitectura del sistema (es decir, describe la estructura del sistema que es independiente de la plataforma, y describe las interrelaciones entre las partes del sistema); y c) PSM (*Platform Specific Model*): este modelo se obtiene a partir del PIM y se correspondería con el modelo que tiene en cuenta los detalles de la plataforma que se va utilizar para desarrollar el sistema.

Con lo cual, en MDA se utilizan modelos, que pueden estar expresados en distintos lenguajes, para representar distintas vistas del sistema (CIM, PIM y PSM) bajo diferentes perspectivas que constituyen las dimensiones de modelado. Esta transformación permite pasar a través de correspondencias (*mappings*) entre los modelos, de niveles más altos de abstracción del sistema a niveles más concretos que permitirán obtener el sistema implementado (es decir, el código). Es decir, este tipo de transformación se podría ver como una *evolución* del modelo a través de las distintas vistas que representa. Con lo cual, si se tienen en cuenta todos los modelos

considerados en MDA se deduce que un proceso de desarrollo del sistema software (ciclo de vida) comenzaría con la creación de un CIM que se utiliza para separar posteriormente la especificación de la operación de un sistema a través del PIM de los detalles de implementación en una plataforma concreta que quedan definidos en el PSM.

3.3. Eclipse Modeling Project

Con el término Eclipse Modeling Project [11] se identifica a un conjunto de proyectos destinados a abordar el modelado y las tecnologías relativas al DSDM empleando la plataforma Eclipse. Concretamente, las capacidades que estos proyectos tratan de afrontar son las siguientes: (1) desarrollo de sintaxis abstractas, (2) desarrollo de sintaxis concretas, (3) transformaciones modelo-a-modelo y, (4) transformaciones modelo-a-texto.

En la figura 3.2 puede observarse la imagen originalmente propuesta como logotipo para el Eclipse Modeling Project. Esta imagen refleja claramente la estructura y las áreas funcionales de esta plataforma. Como se puede ver, EMF (Eclipse Modeling Framework) conforma el núcleo principal, éste provee soporte al desarrollo de sintaxis abstractas. La capa *EMF Query, Validation, and Transformation* es un complemento orientado a la gestión de las instancias de modelos. Entorno a éstos componentes para el manejo de la sintaxis abstractas encontramos las tecnologías de transformación de modelos, modelo-a-texto (Java Emitter Templates y Xpand) y modelo-a-modelo (QVT y ATL). La última capa representa las herramientas destinadas al desarrollo de sintaxis concretas, en sus dos versiones, por un lado sintaxis gráficas (GMF, Graphical Modeling Framework) y por otro, sintaxis textuales (TMF, Textual Modeling Framework). Finalmente, una serie de elementos aparecen orbitando el núcleo, estos componentes constituyen proyectos e iniciativas, algunas de ellas en fase de desarrollo e investigación, enfocadas a completar y mejorar las capacidades de la plataforma, así, por ejemplo, encontramos MOFScript para la definición de transformaciones modelo-a-texto.

A continuación, revisaremos las tecnologías principales del Eclipse Modeling Project que se han empleada en el desarrollo del presente proyecto.

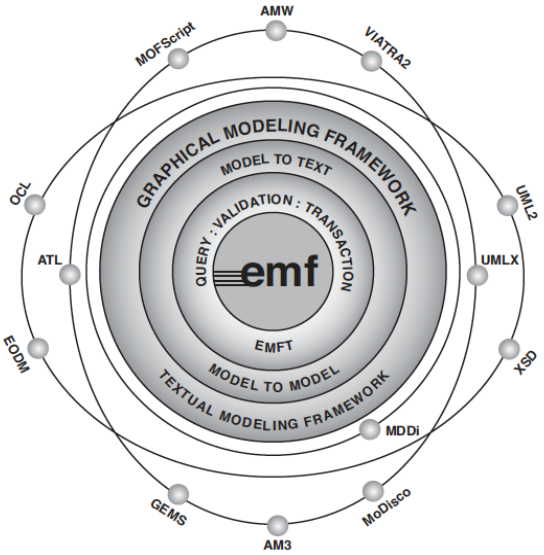


Figura 3.2.: Eclipse Modeling Project

3.3.1. Desarrollo de sintaxis abstractas usando EMF

El Eclipse Modeling Framework (EMF) [36] es una herramienta que proporciona una estructura de modelado y facilidades para la generación de código con el objeto de construir herramientas u otras aplicaciones basadas en un modelo de datos estructurado. A partir de una especificación XML de un modelo, EMF suministra herramientas y soporte de ejecución para producir un conjunto de clases Java en base a ese modelo, un conjunto de clases *Adapter*, que permiten su visualización y edición basándose en comandos del modelo, y un editor básico.

EMF permite importar modelos que han sido previamente especificados usando documentos *Ecore.* Una de las características más importantes de EMF es que suministra mecanismos de interoperabilidad con otras herramientas y aplicaciones basadas en EMF. A continuación, se abordan los aspectos más relevantes de EMF.

a) Meta-modelo Ecore

Ecore es el meta-modelo sobre el que se fundamenta EMF para dar soporte al modelado, realmente se trata de una implementación de EMOF. En la figura 3.1 podemos contemplar el meta-modelo Ecore, a partir de éste un usuario de EMF podría crear modelos para representar la sintaxis abstracta del sistema deseado. Así pues, podemos considerar que el meta-modelo expuesto en la figura 3.3 es, en verdad, el meta-meta-modelo sobre el que conforman todos los meta-modelos en Eclipse Modeling Project.

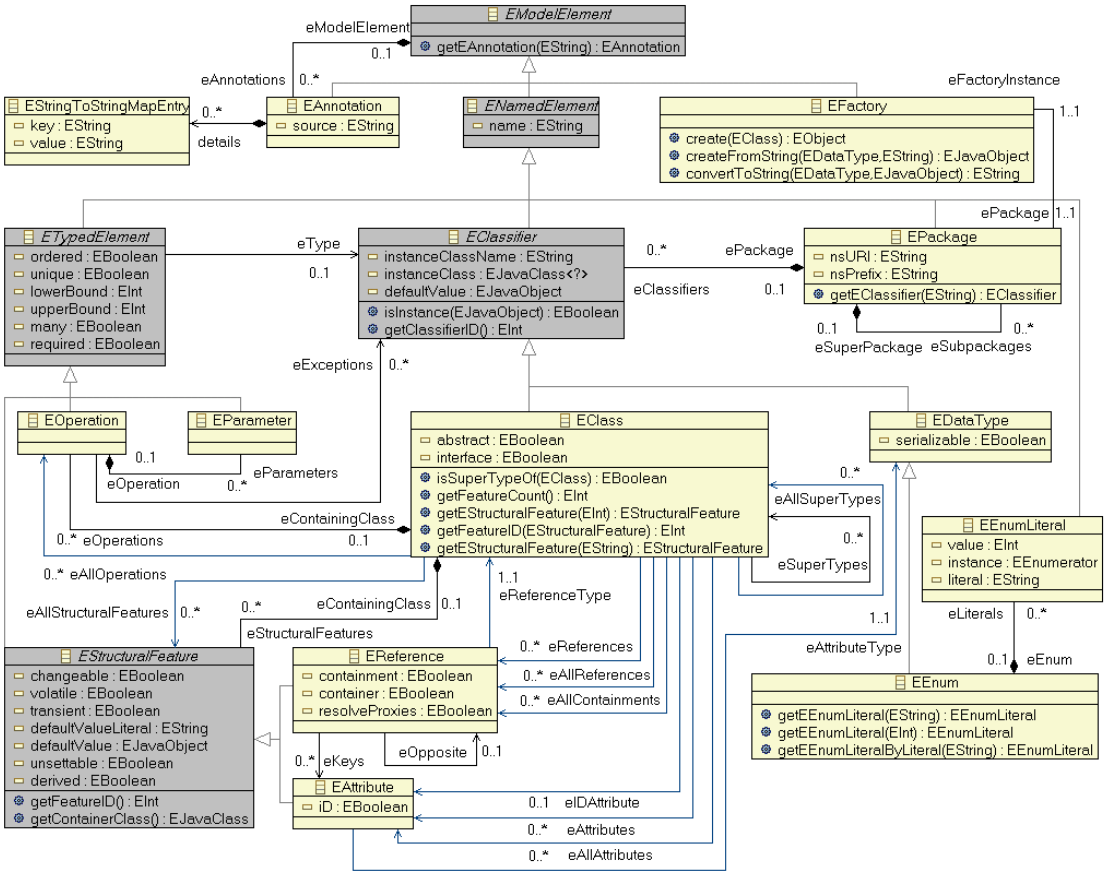


Figura 3.3.: Meta-modelo Ecore

Los elementos más importantes de Ecore son los siguientes:

- ***EClass***: Representa al elemento meta-clase con el que podemos definir los conceptos del meta-modelo desarrollado.
- ***EAttribute***: Define una propiedad de una EClass en forma de tipos primitivos (int, float, String, enumerados, etc).
- ***EReference***: Especifica relaciones entre conceptos (EClass), este elemento señala multiplicidad, rol, navegabilidad y contención de los conceptos relacionados. En cuanto a la contención, debemos señalar que EMF sólo distingue relaciones de composición y asociación, no siendo posible la definición de agregaciones.
- ***EEnum***: Define tipos de datos con valores enumerados.
- ***EPackage***: Paquetes para organizar los elementos anteriores. Son sólo elementos organizativos sin funcionalidad adicional.
- ***EOperación***: Define métodos en los conceptos (EClass).

b) Creación de un meta-modelo

Existen varias formas de crear un meta-modelo: (1) Utilizando el editor de EMF para tal propósito llamado tree-editor, dado que emplea una representación en árbol de los elementos del meta-modelo. (2) A través de un editor gráfico como el *Ecore diagram editor* que facilita GMF o TOPCASED [32]. (3) Otras herramientas ofrecen sus propios editores, por ejemplo, Emfatic [31] para trabajar con sintaxis textual. (4) También se puede utilizar un modelo UML de entornos como Rational, etc.

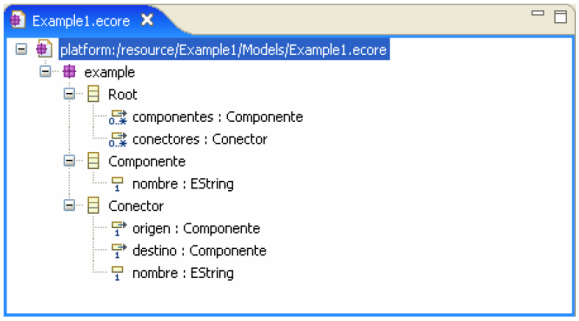
Los modetamodelos creados pueden encontrarse serializados en ficheros con extensión *.ecore* empleando el formato *XML Metadata Interchange* (XMI). Como su nombre indica, XMI es un formato que sigue una sintaxis XML, por lo que, su contenido es legible para una persona, lo que implica que puede editarse "a mano", aunque solo se recomienda hacerlo en casos excepcionales, siendo la mejor opción el uso de algún editor que ofrezca una representación más amigable para el usuario.

Para ilustrar la creación de un meta-modelo con EMF supongamos que deseamos desarrollar una herramienta que nos permita modelar sistemas de componentes sencillos descritos con "cajas y flechas" (hoy en día, esta sencilla terminología es la base de multitud de representaciones más complejas, p.e, los diagramas de estados). La sintaxis abstracta la forman los conceptos (EClass): *Componente* y *Conector*, estos dos elementos tienen un atributo (EAttribute) llamado "nombre" y dos asociaciones (EReference), dado que por definición un elemento Conector relaciona un Componente origen con otro destino. En la figura 3.4a observamos el aspecto de esta sintaxis abstracta tras ser creada empleando el tree-editor de EMF. La figura 3.4b se ha utilizado un editor gráfico, en concreto, el *Ecore diagram editor* que facilita GMF, en el que una sintaxis concreta gráfica se sobrepone en el meta-modelo Ecore, puede percibirse que el aspecto es muy similar a los diagramas de clases de UML. También, se muestra en la figura 3.4c el código XMI de esta sintaxis abstracta. En las figuras nos percatamos de la presencia de un elemento extraño, identificado como Root, este elemento no tendría sentido por si solo, es un mero artificio que resuelve una premisa fundamental en EMF, y es que, en EMF las relaciones de composición controlan la serialización de los modelos y la posición de

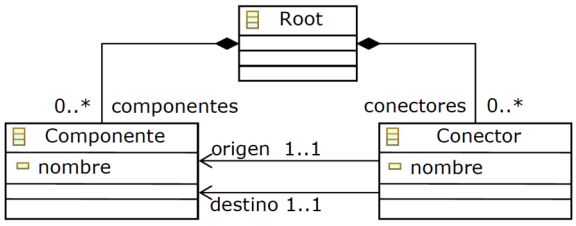
los elementos en el editor, por lo que siempre es necesaria la existencia de un elemento raíz que contenga a todos los demás, si este elemento no aparece de forma natural en la sintaxis deberá ser creado.

c) Generación automática de código

EMF es capaz de crear de forma automática un plug-in Eclipse, de modo que, integre la sintaxis abstracta del usuario en una implementación Java del tree-editor específica para nuestro lenguaje de modelado, así, una vez instalado este nuevo plug-in en cualquier entorno Eclipse, dispondríamos de facilidades para la creación de modelos conformes a la sintaxis abstracta definida. Más concretamente, en un paso previo a la generación de código, debe obtenerse un fichero .genmodel con la información relativa a las especificaciones del usuario, a partir de este fichero, se genera, por un lado, el código asociado a los elementos del meta-modelo, y por otro, el código del plug-in correspondiente al editor reflexivo de modelos. En la figura 3.5a podemos ver una muestra de modelo en el tree-editor generado para ejemplo de componentes anterior. La figura 3.5b ilustra la versión gráfica creada con GFM del mismo editor. Por último, en la figura 3.5c se muestra el código XMI del modelo creado.



(a)



(b)

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
  ...
 <eClassifiers xsi:type="ecore:EClass" name="Root">
     <eStructuralFeatures xsi:type="ecore:EReference"
       name="componentes" upperBound="-1"
       eType="#//Componente" containment="true"/>
     <eStructuralFeatures xsi:type="ecore:EReference"
       name="conectores" upperBound="-1"
       eType="#//Conector" containment="true"/>
 </eClassifiers>
 <eClassifiers xsi:type="ecore:EClass“ name="Componente">
     <eStructuralFeatures xsi:type="ecore:EAttribute"
       name="nombre" lowerBound="1"
       eType="ecore:EDataType
 </eClassifiers>
 <eClassifiers xsi:type="ecore:EClass" name="Conector">
     <eStructuralFeatures xsi:type="ecore:EReference"
       name="origen" lowerBound="1"
       eType="#//Componente"/>
     <eStructuralFeatures xsi:type="ecore:EReference"
       name="destino" lowerBound="1"
       eType="#//Componente"/>
     <eStructuralFeatures xsi:type="ecore:EAttribute"
       name="nombre" lowerBound="1"
       eType="ecore:EDataType
  </eClassifiers>
</ecore:EPackage>
```

(c)

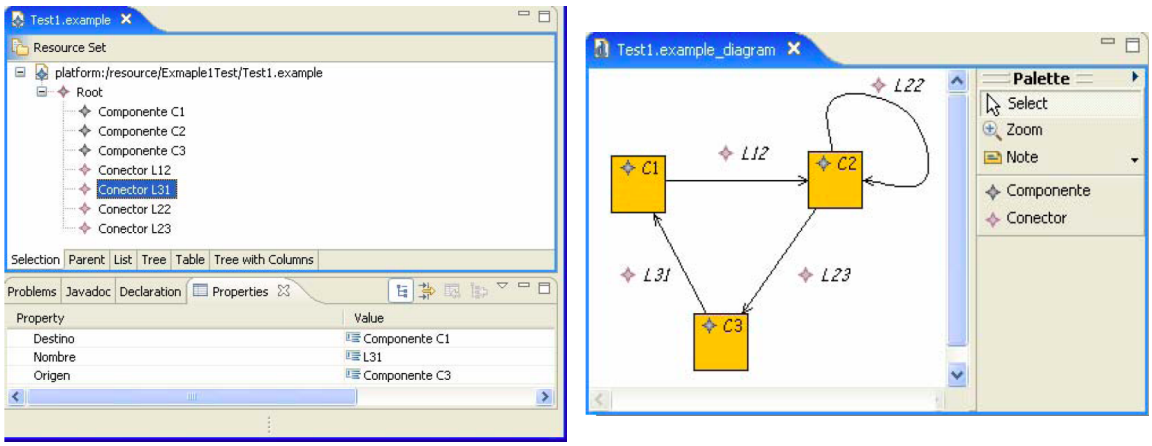
Figura 3.4.: Ejemplo de meta-modelo EMF. (a) Tree-editor, example.ecore. (b) Editor GMF, example.ecorediag. (c) Vista del código XMI contenido en example.ecore. (Fuente [33]).

3.3.2. Desarrollo de sintaxis concretas gráficas usando GMF

Graphical Modeling Framework (GMF) [13] proporciona una herramienta para la generación de editores gráficos a partir de meta-modelos EMF. Así, GMF depende de los siguientes plug-ins Eclipse: (1) EMF, para la definición de modelos y meta-modelos; (2) GEF (Graphical Editing Framework) para la definición de los elementos gráficos de la sintaxis concreta del lenguaje; y (3)

EMF OCL/Query/Validation/Transaction, para la definición de restricciones OCL sobre elementos pertenecientes al meta-modelo (sintaxis abstracta) y a los modelos gráficos (sintaxis concreta) y para el manejo, validación y control de modelos y meta-modelos. Una característica destacable en GMF es la reutilización de la definición gráfica para diferentes dominios y aplicaciones, esto es, se pueden reutilizar las metáforas gráficas ya definidas para conceptos de diferentes dominios y aplicaciones. Esta característica se consigue modelando por separado los componentes gráficas que se corresponden con cada uno de los elementos del dominio y la definición de la paleta de herramientas, la cual tendrá una herramienta por cada primitiva. Para completar el proceso de generación de un editor gráfico de dominio, GMF proporciona una definición de mapping o correspondencia mediante la que se asocia cada primitiva de modelado con su componente gráfica y con su herramienta dentro del editor que se está generando.

Como se puede observar en el diagrama de la figura 3.6, el centro del proyecto es el *Domain Model*, es decir, el modelo del dominio o meta-modelo que será el origen del proceso y del cuál se derivarán el resto de subprocesos. Como abordamos en el apartado anterior, el modelo del dominio se define utilizando EMF mediante un lenguaje de definición de modelos Ecore. Todos y cada uno de los subprocesos de GMF están relacionados con la sintaxis abstracta y como indica su nombre, Ecore, constituirá el centro del proyecto en todo momento. A continuación, se explican cada uno de los subprocesos que conforman el desarrollo del proyecto.



(a) (b)

```
<?xml version="1.0" encoding="UTF-8"?>
<example:Root xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:example="example">
      <componentes nombre="C1"/>
      <componentes nombre="C2"/>
      <componentes nombre="C3"/>
      <conectores origen="//@componentes.0" destino="//@componentes.1" nombre="L12"/>
      <conectores origen="//@componentes.2" destino="//@componentes.0" nombre="L31"/>
      <conectores origen="//@componentes.1" destino="//@componentes.1" nombre="L22"/>
      <conectores origen="//@componentes.1" destino="//@componentes.2" nombre="L23"/>
</example:Root>
```

(c)

Figura 3.5.: Ejemplo de modelo EMF. (a) Modelo en Tree-editor. (b) Modelo en editor gráfico GMF. (c) Vista del código XMI del modelo. (Fuente [33]).

a) Definición del meta-modelo

GMF precisa de un meta-modelo y para ello se sirve de EMF. Una vez especificada la sintaxis abstracta, se obtiene a partir del fichero *.ecore* el *.genmodel* que permite generar el código del plug-in correspondiente al editor reflexivo en árbol.

b) Definición de la sintaxis concreta gráfica

Una vez concretado la sintaxis abstracta, se puede empezar a diseñar la definición gráfica de las primitivas de modelado. La definición gráfica consiste en decidir qué primitivas de modelado harán la función de nodos (elementos de la herramienta de modelado que se desea construir), cuáles serán conectores (enlaces entre los elementos de la herramienta de modelado) y cuáles etiquetas (propiedades de los elementos y enlaces de la herramienta de modelado), así como el diseño del aspecto visual de los elementos como colores, formas, etc. La información de este proceso queda codificada en un fichero con extensión *.gmfgraph*.

c) Definición de la paleta de herramientas

La creación y especificación del panel de herramientas se ha de realizar después de la definición de la metáfora gráfica. Consideramos panel de herramientas como la paleta de herramientas de modelado a crear que proporcionará una funcionalidad "drag and drop", mostrándonos las primitivas de modelado en la forma que se han diseñado en la definición gráfica. En este paso también se definen los iconos de la herramienta de modelado, que constituyen la iconografía de la paleta. El fichero resultante de este proceso es identificado con la extensión *.gmftool*.

c) Correspondencia entre los elementos del modelos y gráficos

La correspondencia entre los elementos del modelo, la metáfora gráfica y herramientas se ha de realizar después de hacer la especificación de las herramientas. En este estadio todo lo creado anteriormente cobra un sentido, ya que, que se relacionan y aúnan todos y cada uno de los elementos creados con anterioridad. La información de este mapping se serializa en un fichero con extensión *.gmfmap*.

d) Generación de la herramienta gráfica

Previamente a la generación del editor gráfico, a partir del fichero .gmfmap se genera *.gmfgen*. Una vez obtenido el código es posible arrancar un nuevo Eclipse desde el que ejecutar el nuevo editor. En la figura 3.5b se muestra un ejemplo de editor final generado con GMF.

3.3.3. Desarrollo de sintaxis concretas textuales usando xText

xText es un framework, componente de TMF, que soporta el desarrollo de gramáticas de lenguajes textuales de aplicación especifica (DSL, Domain Specific Language). Permite generar automáticamente desde la definición de la gramática, el meta-modelo Ecore, un editor textual con formateo léxico y validación sintáctica del código, y el correspondiente convertidor texto-modelo basado en ANTLR [34]. Recientemente, en la versión 1.0, xText también ofrece la generación automática de una posible gramática del lenguaje partiendo de un meta-modelo existente. Además, el framework se integra con otras tecnologías de modelado bajo Eclipse, como EMF, GMF, etc. Todas estas capacidades hacen de xText una herramienta potente para realizar transformaciones texto-a-modelo.

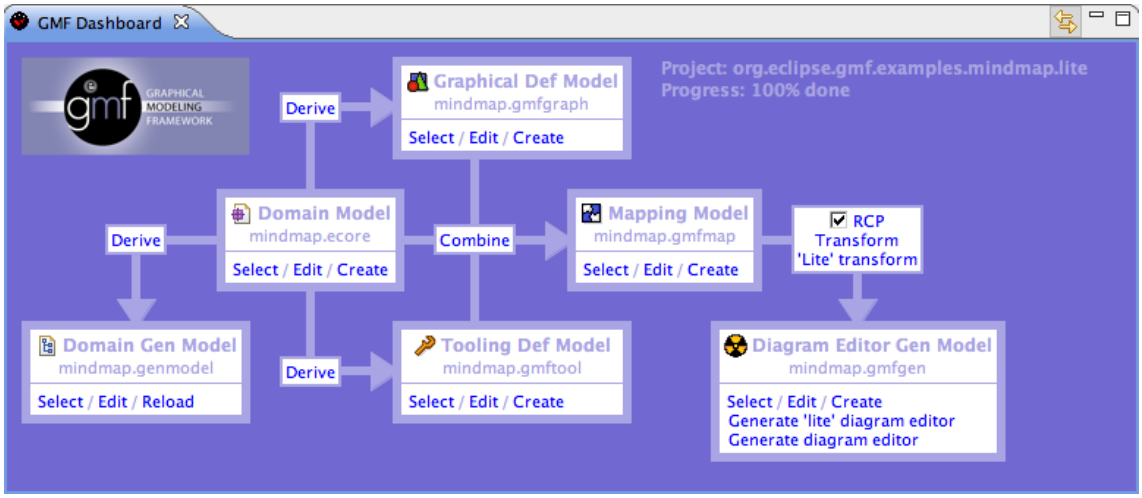


Figura 3.6.: Desarrollo de un proyecto GMF. (Fuente [13]).

El siguiente código es un ejemplo de definición de gramática de un lenguaje sencillo. Podemos ver en la figura 3.7 una muestra del uso de este lenguaje, notar que se emplea el editor textual generado por xText, el cuál permite reconocer palabras clave del lenguaje (obsérvese que '*type*', '*entity*', '*property*' y '*extends*' aparecen resaltadas), ofrece autocompletado de términos y una vista en árbol del modelo resultante.

En cuanto al código, las dos primeras líneas declaran la gramática del nuevo lenguaje con un identificador unívoco (xtext.example.Domainmodel) y el paquete Ecore donde se derivará el meta-modelo generado de la gramática. Los conceptos del lenguaje, que se traducirán posteriormente a elementos *EClass* del meta-modelo, son identificadores seguidos por ':', podemos observar que se definen los siguientes conceptos: *DomainModel*, *Type*, *SimpleType*, *Entity* y *Property*. El primero de ellos constituye el elemento raíz del lenguaje, recordad que EMF requiere que exista una relación de composición que albergue todos los elementos, así pues, la única función de este elemento sería el de actuar de root (para efectuar esta relación de composición aparece un atributo llamado *elements* que referencia a un conjunto de *Type*). El elemento *Type* se define como un elemento genérico para identificar a los conceptos *SimpleType* y *Entity*, ello será traducido en el meta-modelo como una relación de herencia. Por último, los conceptos *SimpleType*, *Entity* y *Property* se conforman como términos del lenguaje con una determinada sintaxis concreta textual (las comillas simples delimitan palabras), también encontramos relaciones entre términos empleando corchetes (observar en la figura que la propiedad *Attendees* de *Conference* es de tipo *Person*, entidad definida posteriormente).

```
grammar xtext.example.Domainmodel with org.eclipse.xtext.common.Terminals
generate domainmodel"http://www.eclipse.org/xtext/example/Domainmodel"
DomainModel:
    (elements+=Type)*;
Type:
    SimpleType | Entity;
SimpleType:
    'type' name=ID;
Entity:
    'entity' name=ID ('extends' superType=[Entity])? '{'
    (properties+=Property)*
    '}';
Property:
    'property' name=ID ':' type=TypeRef;
```

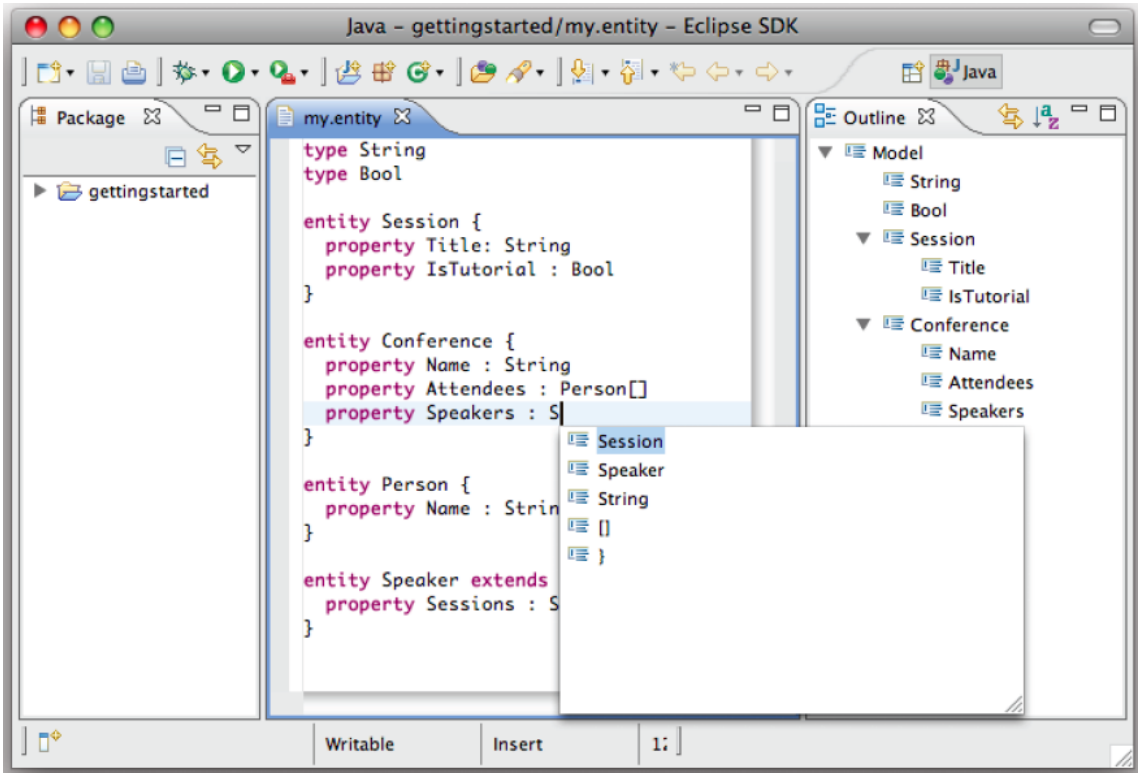


Figura 3.7.: Ejemplo de editor textual generado por xText GMF. (Fuente [35]).

3.3.4. Transformacion Modelo-a-Modelo con ATL

Las transformaciones de modelos permiten hacer evolucionar los modelos a lo largo del proceso de desarrollo. Actualmente, podemos encontrar numerosas herramientas para realizar transformaciones de modelo-a-modelo, entre ellas: ATL (ATLAS Transformation Language) [37], SmartQVT [38], Xpand [39], MediniAVT [40] y QVTOperational [41]. También podríamos considerar las herramientas generales de manipulación XML que, aunque fuera del paradigma de DSDM, trabajan con los modelos como mero código XML.

En cuanto a ATL, fue desarrollado por el grupo de investigación ATLAS como respuesta a las especificaciones RFP-M2M de la OMG. Se trata de un lenguaje híbrido, declarativo e imperativo basado en la sintaxis de OCL. Así, la parte declarativa del lenguaje, establece reglas de correspondencia entre los conceptos de los meta-modelos de origen con los de destino de la transformación. La parte imperativa ayuda a terminar de especificar las reglas declarativas con sentencias condicionales o de selección. Además, el lenguaje textual de ATL es conforme a un meta-modelo, por lo que podemos ver ATL realmente como un lenguaje de modelado dentro de la filosofía del DSDM. A continuación, se enumeran las características más importantes de ATL.

- ATL permite transformaciones MIMO (Multiple Input Multiple Output), lo que implica la posibilidad de poder definir cualquier tipo de transformación, esto es, uno o más modelos de entrada resultan uno o más modelos de salida.
- En ATL se pueden definir tres tipos de ficheros: *Module*, define la transformación principal; *Library*, alberga funciones auxiliares reutilizables; *Query*, devuelve los elementos del modelo que cumplen determinadas propiedades o restricciones.

- El compilador de ATL es capaz de detectar pocos errores en el código, la comprobación de tipos se realiza en tiempo de ejecución, lo cual lo hace difícil de depurar, a pesar de incorporar un depurador de transformaciones.
- Se definen tres tipos de reglas de transformación:
- *Matched rules*. Son reglas puramente declarativas. Se tratan de reglas principales que especifican cómo se transforma un elemento del meta-modelo de entrada en uno o varios elementos del meta-modelo de salida.
- *(Lazy) called rules*. Son reglas auxiliares imperativas. Son invocadas por el usuario para crear elementos en el meta-modelo de destino.
- *Helper rules*. Son reglas auxiliares invocadas por el usuario. No crean elementos en el meta-modelo de destino.
- Al tratarse de un lenguaje con sentencias declarativas, la ejecución del código ATL no es secuencial.

La figura 3.8 muestra un sencillo ejemplo de transformación entre, lo que se ha llamado, un modelo de componentes y un modelo de figuras. Así, la diferencia entre el meta-modelo de origen y destio radica en cómo se modelan los elementos de tipo *Círculo* y *Cuadrado*. A continuación, podemos ver el código que definiría la transformación (trazada por líneas discontinuas en la figura). Observamos que las reglas *conector* y *root* no son más que correspondencias directas entre origen y destino, por otro lado, las reglas *componente2cuadrado* y *componente2circulo* permiten adaptar los modelos a los nuevos aspectos, así, por ejemplo, la regla *componente2circulo* generaría un nuevo elemento *Circulo* si el atributo 'tipo' del componente explorado es igual a "Circulo", además, se añade nueva información del elemento (vea el atributo 'lado').

```
module comp2fig;

create outM : outMM from inM : inMM

rule conector {
       from   f: inMM!Conector
       to     t: outMM!Conector(nombre<-f.nombre,origen<-f.origen,destino<-
f.destino)
}

rule componente2circulo {
       from   f: inMM!Componente (f.tipo = #"Circulo")
       to     t: outMM!Circulo (nombre<-f.nombre, radio<-8)
}

rule componente2cuadrado {
       from   f: inMM!Componente (f.tipo = #"Cuadrado")
       to     t: outMM!Cuadrado (nombre<-f.nombre, lado<-3)
}

rule root {
       from   f:inMM!Root
       to     t: outMM!Root (componentes<-f.componentes, conectores<-
f.conectores)
}
```

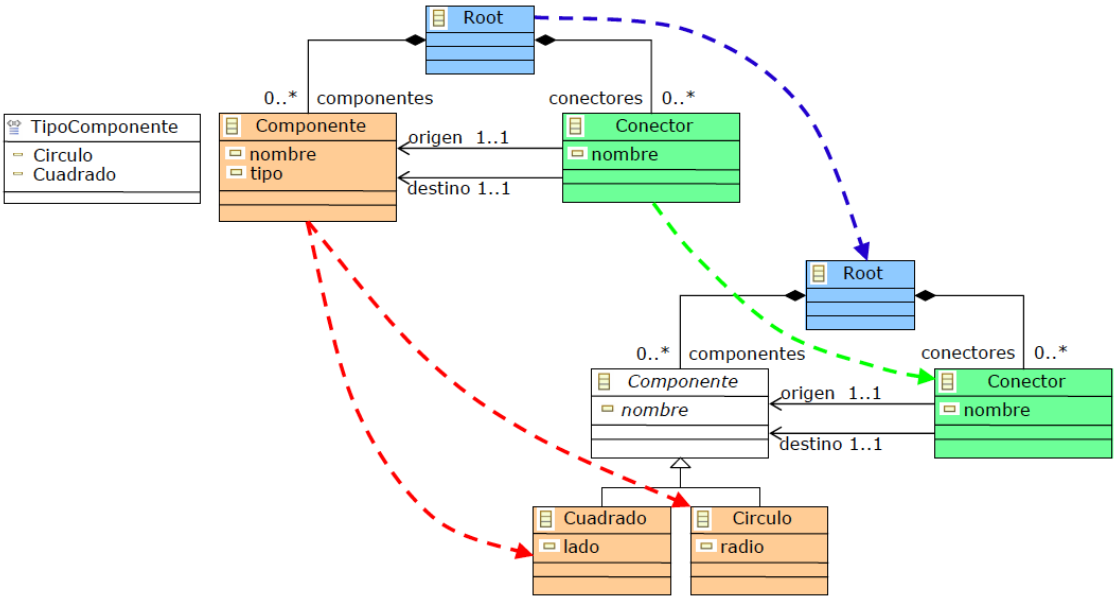


Figura 3.8.: Ejemplo transformación modelo-a-modelo. (Fuente [33]).

3.3.5. Transformacion Modelo-a-Texto con JET

Al igual que en las transformaciones modelo-a-modelo podemos encontrar múltiples tecnologías para realizar transformaciones modelo-a-texto, también llamadas transformaciones modelo-a-código. Entre ellas encontramos dos grandes grupos [42], por un lado, las soluciones basadas en plantillas: JET (Java Emmiter Tempates), VT (Velocity Templates) o Xpand, y las soluciones basadas en lenguajes imperativos, como MOFScript. Concretamente, JET se encuentra integrado en EMF y su lenguaje está inspirado en JSP (Java Server Pages). Seguidamente, se añaden las características más destacables de JET.

- JET genera automáticamente código Java para ejecutar la transformación.
- Integra JMerge, tecnología Java que permite diferenciar y mantener el código generado automáticamente del introducido “a mano” por el usuario, con el fin de respetar dicho código y no sobrescribirlo tras sucesivas generaciones de código.
- Se trata de una herramienta independiente, por lo que no requiere la utilización de tecnología de DSDM, así que, podría utilizarse con código XML.
- Permite insertar código Java en el código JET que define la transformación.
- Utiliza Xpath 1.0 [43] para recorrer el modelo.
- Sólo se admiten transformaciones SIMO (Single Input Multiple Output), lo que indica que sólo se puede emplear un único modelo de entrada para generar múltiples ficheros de código de salida.
- Define las interfaces para la extensión de la funcionalidad del plug-in.
- Distingue mayúsculas de minúsculas.
- Las extensión de los ficheros que especifican las transformaciones JET es *.jet*. El fichero *main.jet* es, por defecto, el punto de entrada de la transformación que se encarga de organizar la transformación.

Seguidamente podemos ver el aspecto que adopta el código JET. Al ejecutar este código de ejemplo se crea un fichero de texto que enumera todos los componentes directamente conectdos a otro en el modelo de entrada.

```
<c:iterate var="c" select="/Root/componentes">
    <c:get select="$c/@nombre"/> ::
    <c:iterate var="cn" select="/Root/conectores">
        <c:if test="$cn/origen/@nombre=¢c/@nombre">
        </c:if>
    </c:iterate>
</c:iterate>
<c:iterate var="cn" select="/Root/conectores[origen/@nombre=$c/@nombre]">
    <c:get select="$cn/destino/@nombre"/> ::
</c:iterate>
```

CAPÍTULO IV

Desarrollo del entorno HuRoME

En este capítulo pretendemos mostrar las bondades del DSDM en el ámbito de la robótica. En las siguientes secciones se describe el desarrollo de las herramientas que componen HuRoME enfocadas al modelado de secuencias de movimientos, generación automática de código y soporte a procesos de reingeniería en un robot humanoide Robonova. En primer lugar, presentamos los retos que se pretenden abordar en la realización del entorno HuRoME, tras ello, el meta-modelo diseñado y el desarrollo de los editores y transformaciones.

4.1. Retos del desarrollo

Actualmente, existen líneas de investigación que trabajan en mecanismos para elevar el nivel de abstracción del proceso de desarrollo del software para robótica. En este sentido, el objetivo principal del entorno HuRoME, presentado en este proyecto, consiste en acercar el concepto de modelo y meta-modelo al mundo de la robótica, realizando para ello, una aproximación al desarrollo de software utilizando un enfoque de DSDM y centrado en una plataforma sencilla, como es el caso del robot humanide Robonova. A continuación, puntualizamos las capacidades fundamentales que deberá adoptar el entorno HuRoME, también se expone cómo *Eclipse Modeling Project* puede hacer frente a los retos que suponen.

a) Lenguaje de coreografías para el robot Robonova

Uno de los objetivos es el de diseñar un lenguaje de modelado de coreografías para la plataforma específica seleccionada. Para ello, se requiere de:

- Un lenguaje constituido por los conceptos básicos necesarios para formar coreografías, como puede ser la noción de “movimiento” o “transición”. Las coreografías concretas que se diseñen deberán poder ser validadas conformes a este lenguaje.

- Reglas dirigidas a restringir las coreografías válidas para impedir, por ejemplo, la especificación de secuencias de movimientos que formen bucles infinitos. Es decir, no toda combinación de conceptos presentes en el lenguaje tiene porqué tener sentido.

Según se revisó en el capítulo 3, el framework EMF proporciona soporte a la definición e implementación de meta-modelos. Esto es, la sintaxis abstracta del lenguaje de modelado puede especificarse utilizando EMF, obteniendo de forma automática un validador que permite verificar la conformidad de los modelos creados. Dado que se emplean estándares reconocidos, como la serialización en XMI, y que EMF forma la base del resto de frameworks de Eclipse Modeling Project, queda asegurada la compatibilidad y aplicabilidad de nuestro lenguaje, por ejemplo, para desarrollar editores o transformaciones asociados a éste. Por otro lado, vimos que para crear un meta-modelo se puede emplear algunos recursos, como la generalización o la composición de conceptos (meta-clases), sin embargo, a veces estos mecanismos no son suficientes para expresar de forma precisa todos los aspectos del lenguaje, en este caso, se pueden emplear reglas expresadas según el estándar OCL que restringen y completan la sintaxis abstracta expuesta en el meta-modelo. EMF no proporciona un soporte fácil y natural a la definición de reglas OCL, sin embargo, el framework dedicado a la implementación de editores gráficos, GMF, sí proporciona este soporte.

b) Modelado gráfico de las coreografías

La aplicación de la sintaxis abstracta del lenguaje pasa por disponer de una herramienta que permita el modelado gráfico de coreografías donde el usuario pueda, de forma sencilla e intuitiva, trazar secuencias de movimientos y concretar sus características. Para ello, se requiere de:

- Una representación visual asociada a cada uno de los conceptos del lenguaje, de modo que sea fácilmente reconocible por el usuario.
- Un editor donde desplegar la notación gráfica del lenguaje y que permita, a partir de ésta, desarrollar diagramas de secuencias de movimientos. Dado que no toda la información necesaria para especificar una coreografía puede estar presente de forma visual en el diagrama, es necesario que el editor disponga de diferentes vistas que den acceso a esta información, generalmente, de naturaleza textual.
- El editor deberá reportar de forma gráfica o textual las incidencias y errores que se encuentren en el diagrama diseñado por el usuario.
- El editor deberá serializar en un fichero toda la información de una coreografía diseñada por el usuario.

Los puntos anteriores pueden ser abordados utilizando GMF. Pueden verse, en la correspondiente sección, los detalles del desarrollo.

c) Generación automática de código

En última instancia la finalidad que persigue el paradigma de DSDM es la generación de código. El entorno HuRoME deberá tener la capacidad de traducir las coreografía diseñadas a código, con el objeto de poder ejecutarlas en el robot. Para ello, proponemos:

- Una transformación a código RoboBASIC. El código que pueda emplearse para describir secuencias de movimientos emplea, fundamentalmente, sentencias asociadas a acciones de los motores, esto hace que sólo se utilice una pequeña parte de

RoboBASIC, ya que la mayoría de las funcionalidades que puede expresar este lenguaje no se modelan (por ejemplo, las comunicaciones o la sensorización).

- Una transformación a código RoboScript. La justificación de porqué proponer dos transformaciones a códigos equivalentes es, en primer lugar, una cuestión de forma, ya que, ante la situación expresada en el primer punto, roboScript fue diseñado para describir exclusivamente coreografías, lo cual encaja perfectamente con el modelado que se realiza en HuRoME. En segundo lugar, se trata de una cuestión práctica, pues el simulador del robot Robonova sólo permite la lectura de ficheros escritos en roboScript y no en roboBASIC. Por último, abogar por una transformación a RoboBASIC es lo mismo que abogar por la flexibilidad y el potencial que pudieran proporcionar futuras mejoras del entorno, con el fin de incorporar en el modelado aspectos más avanzados, como, por ejemplo, el control de flujo.

Las transformaciones modelo-a-código quedarán definieras y podrán llevarse a cabo, gracias a la tecnología JET.

d) Generación de modelos a partir de código

Una característica deseable en el entorno HuRoME es el soporte a la modernización y reutilización de código existente. De forma que no sólo se facilite la generación automática de código a partir de modelos, sino también el proceso inverso, permitiendo la transformación de programas ya existentes (legacy code) en modelos, posibilitado así su tratamiento (depuración, extensión, reutilización, análisis, simulación, etc) con un nivel de abstracción mayor que el que proporciona el código fuente. El desarrollo de esta funcionalidad conlleva:

- La definición de la gramática del lenguaje de entrada, en este caso, dadas las limitaciones del modelado consideraremos a RoboScript como el lenguaje principal de entrada.
- Considerando que el código no es más que un modelo con una sintaxis concreta textual conforme a un meta-modelo que ha sido inferido desde la gramática del lenguaje RoboScript. Podemos plantear la creación de un editor específico que permita representar adecuadamente estos modelos, y además, si el meta-modelo que describe el código difiere del meta-modelo de planteado para modelar las coreografías, se necesitará realizar una transformación modelo-a-modelo.

Para llevar a cabo estas acciones se utilizará el framework xText encargado de generar y manejar DSLs (Domain Specific Language) y ATL para las transformaciones modelo-a-modelo.

e) Integración de las herramientas

Este último punto expresa la necesidad de conseguir un entorno compacto, donde los editores y transformaciones desarrolladas se encuentren cohesionadas en HuRoME. Las bondades de una correcta integración puede reflejarse en una mayor usabilidad y un aprendizaje más rápido de las herramientas por parte del usuario. Considerando que cada faceta anteriormente comentada es implementada utilizando una tecnología diferente (aunque relacionada), la integración supone un esfuerzo extra que requiere codificación manual, es decir, los distintos frameworks no incluyen un mecanismo de integración natural para que, por ejemplo, las acciones destinadas a ejecutar la generación automática de código a partir de un modelo se reduzcan simplemente a hacer clic

sobre una opción del menú del propio editor gráfico de modelos, y no requiera estar moviendo ficheros de uno a otro proyecto Eclipse.

De forma esquemática en la figura 4.1 puede observarse el proceso completo que aborda HuRoME, en esta representación se aprecia cada una de las funcionalidades comentadas anteriormente. En las siguientes secciones del capítulo se detallará el desarrollo de cada una de las facetas envuelven HuRoME.

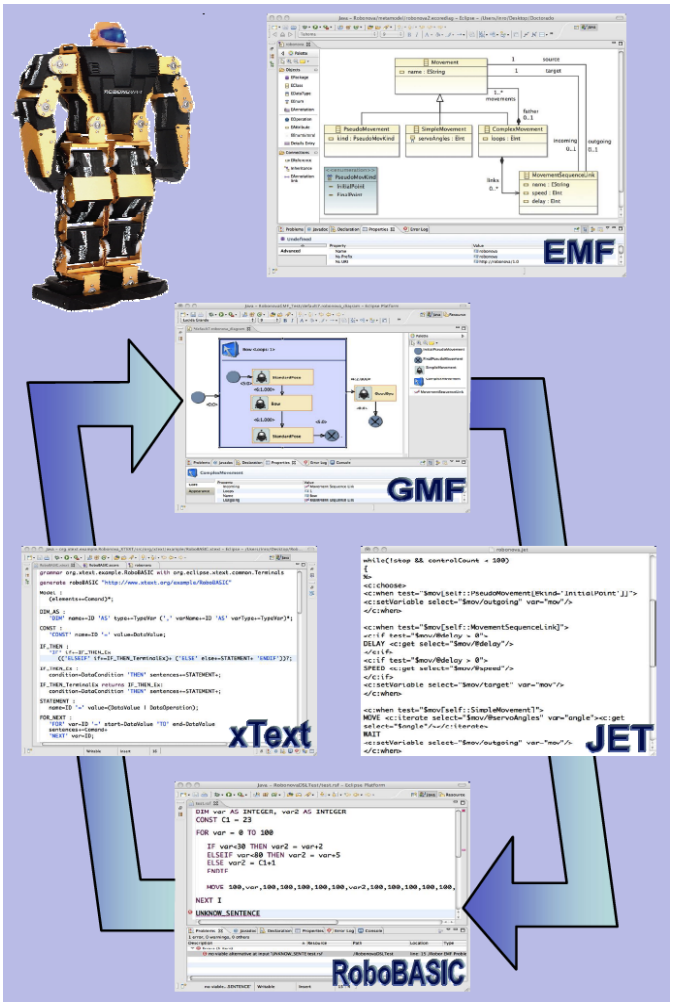


Figura 4.1.: Representación del proceso de desarrollo en HuRoME.

4.2. Lenguaje de modelado de coreografías

4.2.1. Descripción del meta-modelo diseñado

El meta-modelo desarrollado como parte de este trabajo, define los conceptos (y las relaciones existentes entre ellos) necesarios para conformar un lenguaje que permita modelar los movimientos de un robot humanoide. La figura 4.2 ilustra el meta-modelo propuesto. Podemos observar que el diseño del meta-modelo se asemeja bastante al de un diagrama de clases UML, en el que: (1) los conceptos del dominio (meta-clases) son representados usando cajas (*EClass* en EMOF), (2) las flechas de punta triangular definen jerarquías de herencia o especialización de conceptos (EMOF soporta herencia múltiple), y (3) flechas simples representan relaciones entre conceptos (*EReference* en EMOF) con o sin contención (composición).

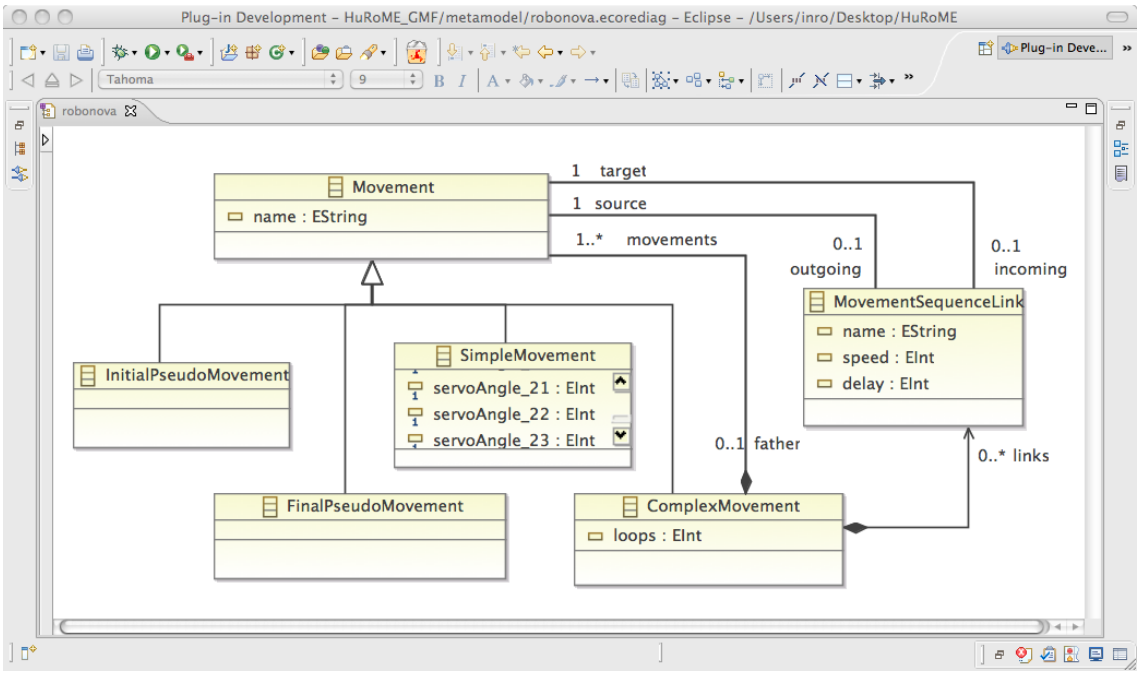


Figura 4.2.: Meta-modelo EMF propuesto para modelar los movimientos de un robot humanoide.

Los movimientos del robot se definen mediante el concepto *Movement* (meta-clase abstracta), que se especializa en tres subtipos de movimiento: (1) *SimpleMovement*, que modela un cambio de postura del robot mediante el accionamiento de uno o más actuadores mecánicos (servomotores), (2) *ComplexMovement*, que modela una secuencia de movimientos (simples o compuestos) y, (3) *InitialPseudoMovement y FinalPseudoMovement*, que modela, no tanto un movimiento, sino un punto de control que establece el inicio o el final de una secuencia. Las instancias de *PseudoMovement* permiten al diseñador determinar si el punto de control es de inicio o de final de secuencia, tal como se indica en el propio nombre.

Cada instancia de *SimpleMovement* tiene un conjunto de atributos (*servoAngles*), correspondientes a los valores angulares de cada uno de los servomotores, lo que permite controlar las distintas articulaciones del robot para que éste adopte una postura determinada. Otro atributo de *SimpleMovement* es el booleano *ptpall* que indica si todos los movimientos especificados en el elemento *SimpleMovement* deben finalizar a la vez (*ptpall* true) o no. Por otro lado, cada movimiento compuesto (*ComplexMovement*) contiene dos o más movimientos (*Movement*) enlazados secuencialmente mediante un *MovementSequenceLink*, de modo que la secuencia siempre se inicia con un *InitialPseudoMovement* y termina con un *FinalPseudoMovement*. En cada instancia de *ComplexMovement* se puede determinar el número de veces que debe repetirse la secuencia de movimientos modelada mediante el atributo *loops*. En cuanto al elemento de enlace *MovementSequenceLink,* es posible seleccionar la velocidad de ejecución del siguiente movimiento mediante el atributo *speed,* y el retardo de la transición entre dos movimientos mediante el atributo *delay*.

El meta-modelo diseñado recoge algunas de las reglas sintácticas que determinan cuándo un modelo es válido o no. Así, según indica el meta-modelo, cada *ComplexMovement* debe contener al menos una instancia de tipo *Movement*. Si embargo, EMF tiene ciertas limitaciones a la hora de permitir al diseñador describir ciertas restricciones sintácticas que necesariamente deben cumplir sus modelos. Así, por ejemplo, reglas como las que establecen que los

MovementSequenceLink nunca deberían poder conectar un movimiento consigo mismo, o que toda secuencia de movimientos debe comenzar con un *InitialPseudoMovement* y terminar con un *FinalPseudoMovement* no pueden definirse en EMF. En este sentido, necesitamos establecer ciertas reglas sintácticas que completen el lenguaje de modelado diseñado, haciendo uso de restricciones OCL. En la siguiente sección, en la que se presenta la herramienta gráfica de modelado implementada a partir del meta-modelo de la figura 4.2, se comenta cómo y dónde se han añadido estas restricciones.

4.2.2. Descripción del procedimiento

Este apartado se irá emplazando a lo largo del capítulo para las distintas facetas del desarrollo. En él se realiza una breve descripción del proceso de desarrollo que se siguió, con la finalidad, no de escribir un "How-to" de las herramientas del Eclipse Modeling Project, sino de ofrecer una perspectiva general que refleje tanto el grado de sencillez o dificultad del proceso como los conceptos que aborda. De este modo, al final del capítulo quedarán patentes las ventajas y desventajas del enfoque de DSDM utilizando Eclipse Modeling Project.

Como se comentó en el capítulo anterior, GMF depende de otros plug-in Eclipse, entre ellos de EMF, así pues, dado que el meta-modelo definido irá dirigido a la implementación de un editor gráfico de modelos, la definición de estos se realiza en el marco de un proyecto "GMF Project". A continuación, se describen los pasos realizados para especificación del meta-modelo.

1. ***Definición del meta-modelo:*** Se añade una nueva fuente al proyecto de tipo *Ecore Diagram*. Esta acción genera dos ficheros, uno **.ecorediag* y otro **.ecore*. El primero de ellos es una serialización XMI del meta-modelo Ecore. El segundo está dirigido a almacenar sólo información relativa a la representación gráfica del meta-modelo. El editor asociado a este último fichero, permite dibujar usando el ratón los meta-modelos, para ello, dispone de una paleta de herramientas desde donde seleccionar los nuevos elementos (*EClass, EPackage, EAttribute*, etc) que se desean trazar. En la figura 4.2 puede verse este editor con el meta-modelo de HuRoME. Ambos ficheros están sincronizados, de modo, que una modificación en uno de ellos produce la actualización automática del otro.
2. ***Validación del meta-modelo:*** Una vez finalizado el meta-modelo, éste ha de ser validado para verificar, por ejemplo, que no se han dejado relaciones al "aire" y que es conforme al meta-meta-modelo Ecore. Para ello, sólo hay que seleccionar la correspondiente opción del menú contextual.
3. ***Creación del EMF Generador Model:*** Añadimos al proyecto un nuevo fichero de tipo *EMF Generator Model*, durante el proceso de creación, sólo habrá que indicar el meta-modelo Ecore al que está asociado. Tras su creación, este fichero **.genmodel* centraliza toda la información del meta-modelo diseñado, además alberga un listado de propiedades cuya configuración afectará a aspectos avanzados de la generación automática de código del siguiente punto. En nuestro caso, estas propiedades se mantienen con su valor por defecto.
4. ***Generación de código Java:*** Por último, se genera el código Java asociado a los elementos del meta-modelo y el código del plug-in correspondiente al editor reflexivo de modelos (**.edit* y **.editor*). Para ello, se selecciona la opción "Generate All" en el menú contextual del nodo raíz del *genmodel*.

4.3. Herramienta gráfica de modelado

Como ya se ha comenado, se ha implementado una herramienta gráfica de modelado utilizando las facilidades proporcionadas por el plug-in de Eclipse conocido como GMF. Este editor gráfico de modelos se ha construido sobre el meta-modelo descrito anteriormente, con el fin de facilitar a los usuarios un entorno amigable e intuitivo con el que podrán realizar, de forma 'visual', programas para el robot, sin necesidad de tener ningún conocimiento previo sobre su lenguaje de programación.

En la figura 4.3 puede observarse el aspecto del editor gráfico desarrollado. Básicamente, éste se compone de: (1) una paleta de herramientas (panel derecho) desde la que el usuario puede seleccionar los conceptos que desea incorporar a sus modelos, (2) un área de trabajo (panel central) donde podrá modelar las secuencias de movimientos que desee para el robot, y (3) una vista de propiedades (panel inferior) para añadir y completar la información asociada a los distintos elementos del modelo (por ejemplo, el valor angular de los servos del robot). Esta herramienta proporciona una sintaxis concreta, en este caso gráfica, a los conceptos descritos en el meta-modelo (sintaxis abstracta). La tabla 4.1 recoge la relación entre los conceptos del meta-modelo y la sintaxis concreta que se les ha asignado.

Por último, debemos indicar que la herramienta gráfica de modelado que se ha creado incorpora facilidades de validación, que permiten comprobar si un modelo es o no correcto de acuerdo al meta-modelo y a las restricciones OCL adicionales añadidas en el editor. Cuando se detecta algún error, los elementos que incumplen alguna de las reglas son marcados gráficamente con un círculo rojo y una cruz. Una vez que el modelo es validado correctamente, el usuario ya puede proceder a generar, de forma totalmente automática, el código correspondiente a dicho modelo para el robot, aplicando la transformación que se describe en la siguiente sección.

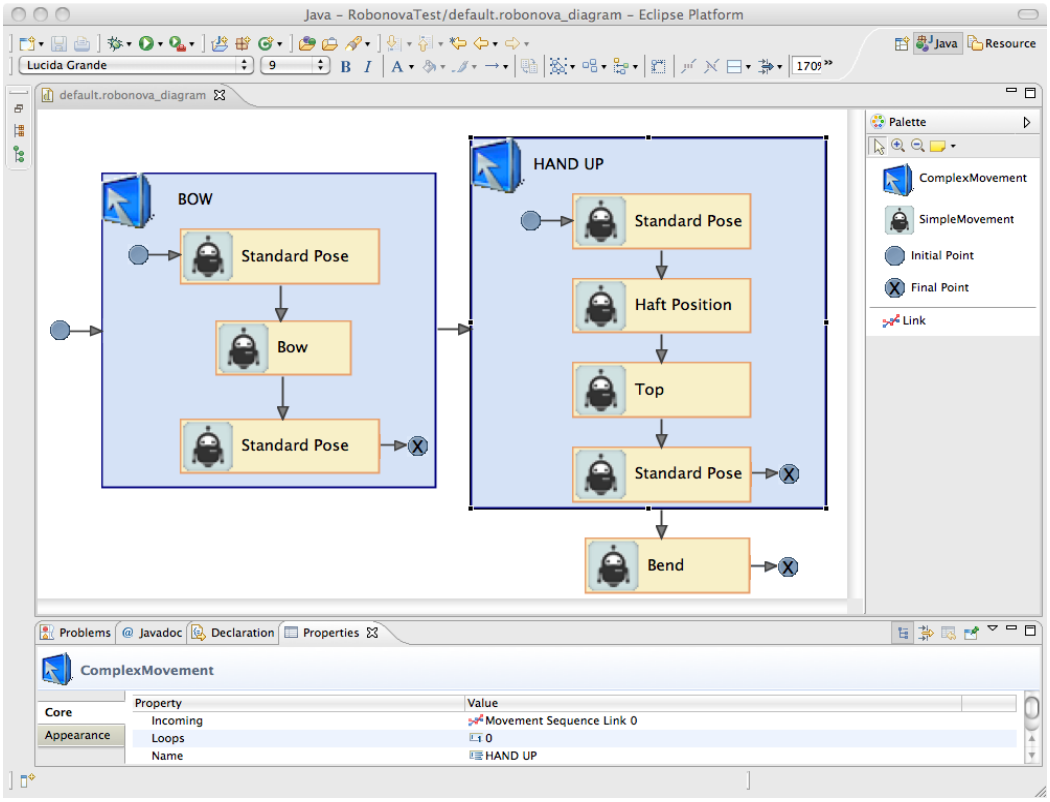


Figura 4.3.: Herramienta de modelado gráfico implementada.

4.3.1. Descripción del procedimiento

A continuación se comenta el proceso y los elementos que han sido necesarios para construir la herramienta gráfica con GMF. Para realizar este proceso es necesario contar con el meta-modelo EMF (fichero *.ecore*), previamente descrito en la sección anterior e ilustrado en la figura 4.2.

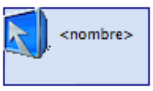
a) Definición de los elementos gráficos.

1. ***Creación fichero *.gmfgraph:*** Se añade al proyecto GMF una nueva fuente llamada *Simple Graphical Definition Model*. Durante el proceso de creación es necesario indicar, primero, el root del meta-modelo asociado, y tras ello, el tipo de representación de cada elemento del meta-modelo. En GMF se consideran, básicamente, dos tipos de representaciones gráficas para los elementos del meta-modelo, esto es, "forma" o "enlace". Un elemento con una sintaxis gráfica de tipo "forma" tomará una apariencia, por ejemplo, de caja con icono. Al contrario, un elemento de tipo "enlace" queda representado con una línea, generalmente, destinada a unir formas. En HuRoME, el root lo conforma la meta-clase *ComplexMovement* ya que contiene, con relaciones de composición, al resto de conceptos del meta-modelo. En cuanto a la definición gráfica, *ComplexMovement*, *SimplexMovement* y *PseudoMovement* son descritos como formas, mientras *MovementSequenceLink* es un enlace.

2. ***Configuración de los elementos gráficos:*** La figura 4.4 puede verse el contenido del fichero gmfgraph, este fichero agrupa todas las propiedades y definiciones de los gráficos asociados a cada concepto del meta-modelo. Básicamente podemos diferenciar tres tipos de nodos:

   - Los *Figure Descriptor* están destinados a crear una colección de objetos gráficos. Define el aspecto de cada objeto gráfico. En la figura 4.5 ilustra la especificación de algunos elementos.
   - Elementos *Node* (por ejemplo, *Node SimpleMovement*) y *Connection* (por ejemplo, *Connection MovementSequenceLink*). Mapean cada concepto del meta-modelo con un objeto gráfico *Figure Descriptor*.
   - *Diagram Label* permte establecer algunos aspectos visuales relativos a una etiqueta, como si la etiqueta tiene un icóno, por ello, este elemento va enlazado con a un *Figure Descriptor*.

TABLA 4.1
RELACIÓN ENTRE SINTAXIS ABSTRACTA Y CONCRETA

Conceptos del dominio	Sintaxis gráfica	Paleta de herramientas
ComplexMovement		
SimpleMovement		
PseudoMovement		
MovementSequenceLink		

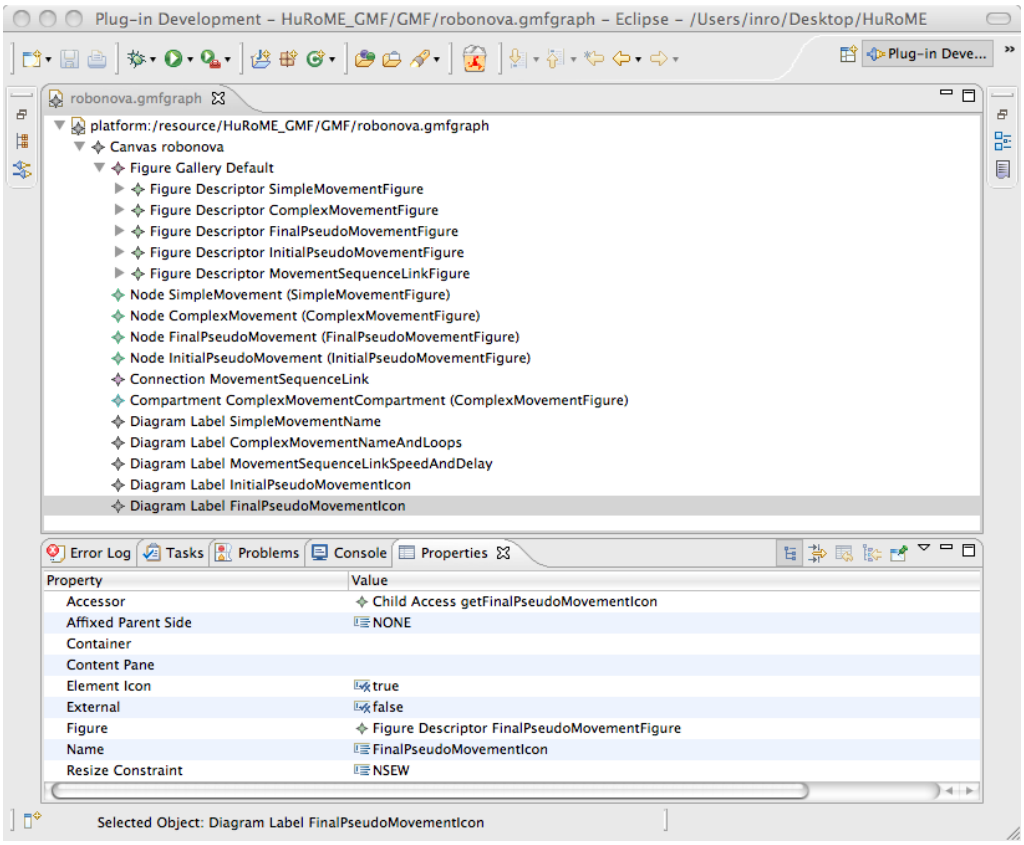


Figura 4.4.: Contenido fichero *.gmfgraph.

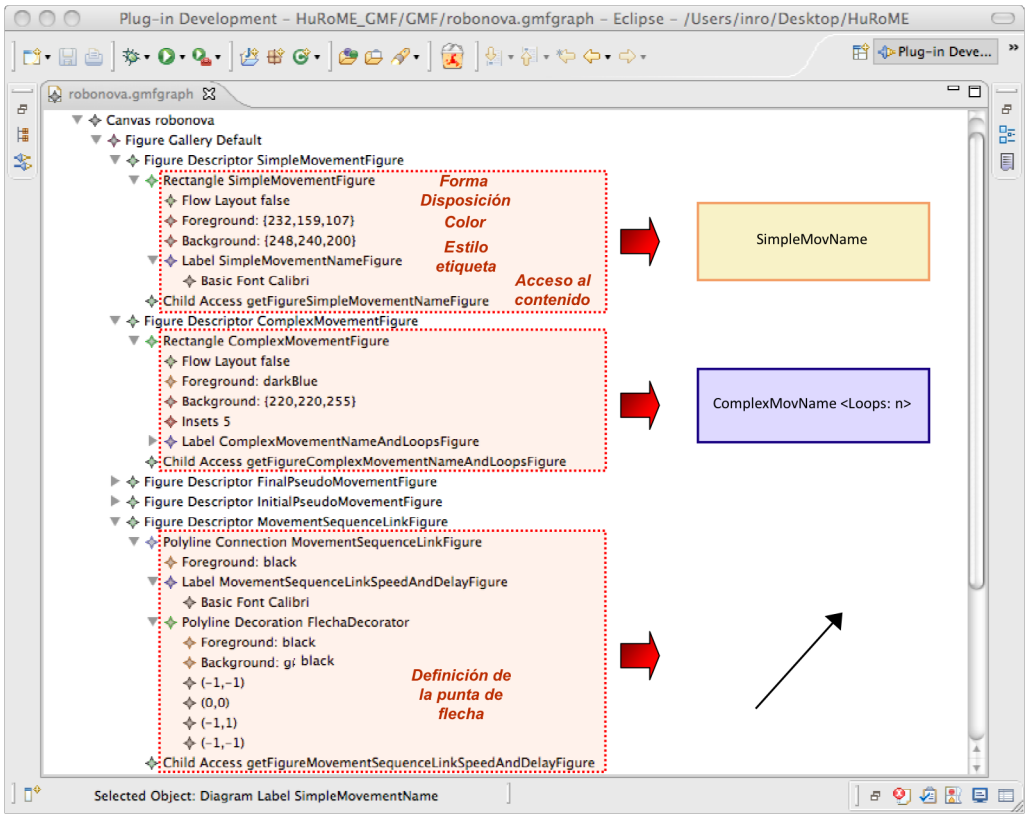


Figura 4.5.: Detalle definición de los objetos gráficos de HuRoME.

Un elemento opcional que aparece en la figura 4.4 es *Compartment* que define que un objeto gráfico concreto pueda contener a otros en su interior. En este caso establece que *ComplexMovement* podría albergar a todos los demás elementos.

b) Definición de la paleta de herramientas.

Añadimos al proyecto un *Simple Tooling Definition* creándose un nuevo fichero de extensión **.gmftool*. Durante el proceso de creación es necesario indicar, primero, el root del meta-modelo y tras ello, se selecciona el tipo de herramienta asociada a cada uno de los elementos que se visualizarán en la paleta. Por último, tras finalizar la creación, se genera una paleta por defecto que en la mayoría de casos suele ser válida y no requiere modificación. La figura 4.6 muestra la definición de la paleta de herramientas para el entorno HuRoME. Observamos que se definen un nodo para hacer un grupo de herramientas (*Tool Group*), también podemos ver la disposición de una línea separadora que permite organizar los ítems de la paleta (*Palette Separator*).

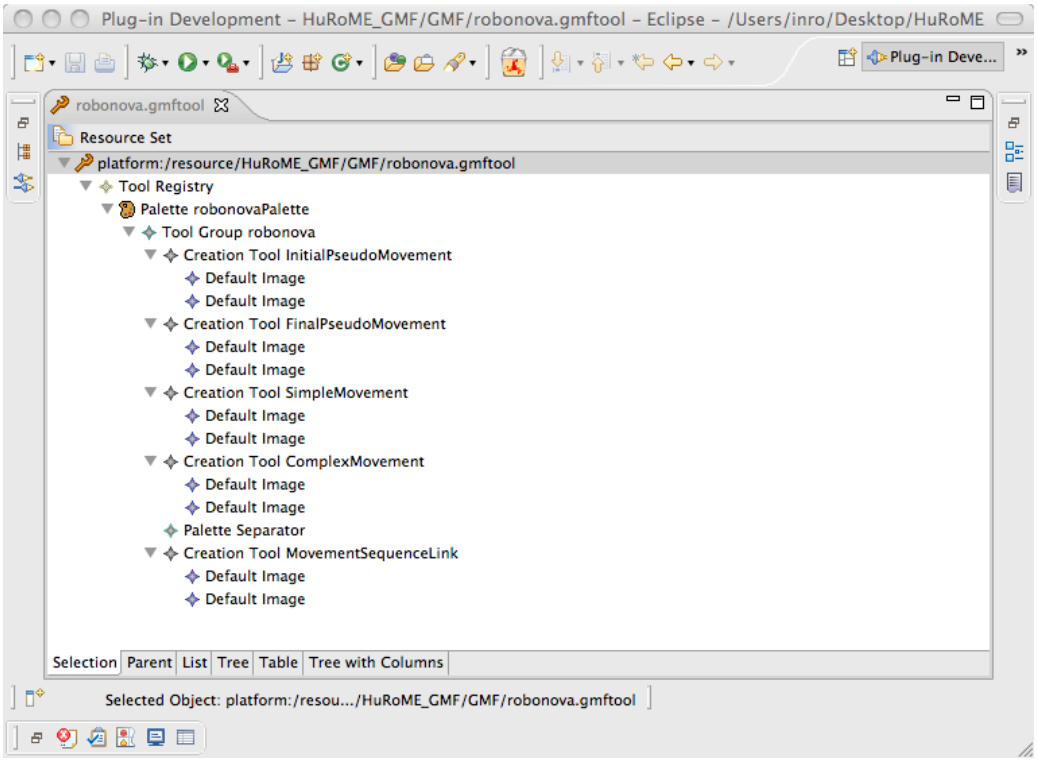


Figura 4.6.: Detalle definición de la paleta de herramientas de HuRoME.

c) Mapeado entre los elementos gráficos y la paleta de herramientas.

Añadimos al proyecto un *Guide Mapping Model Creation* creándose un fichero de extensión **.gmfmap*: Éste permite relacionar todos los elementos anteriores, es decir, los conceptos del meta-modelo Ecore, las figuras creadas en *gmfgraph* y las herramientas de la paleta. Al finalizar la creación se genera una propuesta de mapping que normalmente requiere ser revisada y a veces modificada, tras ello, debe verificarse el fichero *gmfmap* seleccionando la opción "*Validate*" del menú contextual. En figura 4.7 puede verse el contenido del fichero gmfmap de HuRoME. Observamos en la vista de propiedades que el clase "SimpleMovement" del meta-modelo (Campo *Element* en *Domain meta information*) queda asociada a la figura "SimpleMovementFigure" definida en *gmfgraph* (campo *Diagram Node en Visual representation*) y a la herramienta de la paleta "Creation Tool SimpleMovement" (campo *Tool* en *Visual representation*).

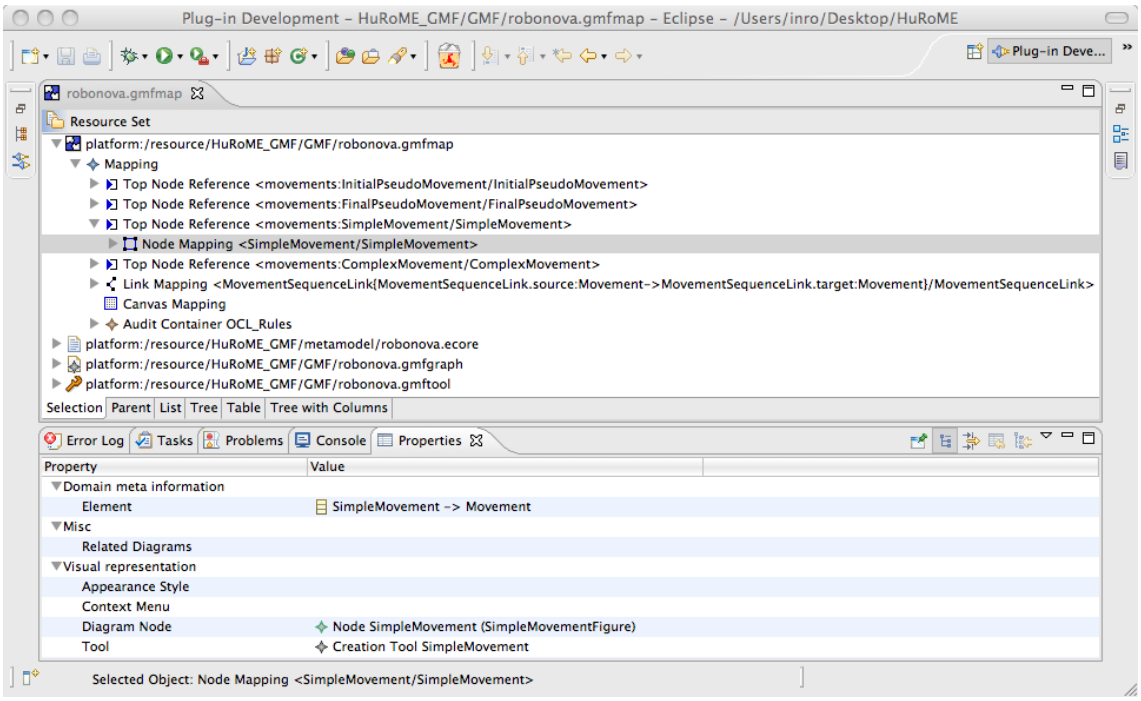


Figura 4.7.: Detalle definición del mapeado gmfmap de HuRoME.

d) Generación de la herramienta gráfica.

1. ***Generación fichero *.gmfgen:*** Tras validar el *gmfmap* y estando seleccionado este fichero, se hace clic sobre la opción "Create generador mode" del menú contextual. El fichero generado tendría una finalidad equivalente al *genmodel* de la sección anterior, *gmfgen* aglutina toda la información y opciones de control necesarias para realizar la generación de código del editor gráfico.
2. ***Generación de la herramienta gráfica:*** La generación automática de código Java comienza cuando seleccionado el genmodel se aplica la opción del menú "Generate diagram code". Esta acción genera el plug-in del editor de modelado gráfico identificado como **.diagram*.

4.3.2. Definición de las restricciones OCL

En el fichero *.gmfmap los diseñadores pueden añadir restricciones sintácticas adicionales a las ya incluidas en el meta-modelo, utilizando para ello el lenguaje OCL. Las restricciones que se han incluido en HuRoME son las siguientes:

(1) No se permiten conexiones reflexivas, esto es, desde un movimiento a él mismo (las referencias source y target de MovementSequenceLink han de ser distintas).

(2) Sólo se pueden enlazar elementos de movimiento del mismo nivel de composición (dos elementos enlazados deber tener el mismo valor de referencia parent).

(3) El elemento inicial de una secuencia de movimientos (PseudoMovement configurado como InitialPoint) debe tener un enlace de salida y ninguno de entrada (la referencia incoming debe ser nula).

(4) El elemento final de una secuencia de movimientos (PseudoMovement configurado como FinalPoint) debe tener un enlace de entrada y ninguno de salida (la referencia outgoing debe ser nula).

(5) Los elementos SimpleMovement y ComplexMovement han de presentar obligatoriamente enlaces de entrada y salida a otras instancias Movement.

La definición de estas reglas se incluye en la tabla 4.2.

TABLA 4.2
REGLAS OCL IMPLEMENTADAS

Regla	Elemento objetivo	Código OCL
(1)	MSL	self.source <> self.target
(2)	MSL	self.source.father = self.target.father
(3)	PM	self.kind=InitialPoint implies self.outgoing->size()=1 and self.incoming->isEmpty()
(4)	PM	self.kind=FinalPoint implies self.incoming->size()=1 and self.outgoing->isEmpty()
(5)	M	not(self.oclIsTypeOf(PseudoMovement)) implies self.autgoing->size()=1 and self.incoming->size()=1

Claves empleadas en la columna "Elemento objetivo": Movement (M), MovementSequenceLink (MSL), PseudoMovement (PM).

Para crear una regla OCL en el fichero gmfmap, en primer lugar, se añade un nuevo elemento al árbol de *gmfmap* de tipo *Audit Container*, sobre éste, insertamos un nodo hijo de tipo *Audit Rule*. Para indicar sobre que clase del meta-modelo tiene efecto la restricción, se añade a *Audit Rule* un *Domain Element Target* donde el objeto de la regla queda configurado en la vista de propiedades. Por último, se añade a *Audit Rule* un elemento *Constraint* y se introduce el código OCL de la regla en la propiedad *Body* de este elemento. Comentar que para que la restricción tenga efecto será necesario regenerar el fichero **.gmfgen*, activar las propiedades de validación en éste y regenerar el código **.diagram*.

4.4. Generación automática de código

Entre las herramientas existentes para realizar transformaciones M2T, hemos optado por el lenguaje basado en plantillas JET, uno de los muchos plug-ins relacionados con DSDM que ofrece Eclipse. Esta decisión se fundamenta en que es el lenguaje de transformación empleado en la propia implementación de EMF, es decir, la tarea de generación automática de código en este framework es realzada con JET, quedando, por lo tanto, sobradamente manifiesta su utilidad y potencia. Así pues, el objetivo de esta fase es convertir los modelos diseñados con la herramienta gráfica de modelado a código RoboBASIC y RoboScript, con el fin, de ejecutar la coreografía en el robot.

4.4.1. Generación de código RoboBASIC

Como ya se abordó en el capítulo 2, RoboBASIC es un lenguaje de programación sencillo basado originalmente en BASIC, por lo que se trata de un lenguaje no estructurado, sin embargo, incorpora algunas sentencias de control de flujo. Contiene instrucciones específicas del dominio de la robótica, dirigidas al movimiento de actuadores y a la lectura de sensores. La tabla 4.3 refleja la identificación de estructuras de código RoboBASIC con las que serán traducidas los conceptos definidos en el meta-modelo.

TABLA 4.3
TRANSFORMACIÓN MODELO A CÓDIGO ROBOBASIC

Conceptos del dominio	Código RoboBASIC
Código de inicio de programa	GETMOTORSET G24,1,1,1,1,1,0,1,1,1,0,0,0,1,1,1, 0,0,0,1,1,1,1,1,0 MOTOR G24
ComplexMovement	DIM var1 AS INTEGER FOR var1=0 TO *loops* *(Código elementos contenidos)* NEXT
SimpleMovement	PTP [ALLON | ALLOFF] MOVE24 *servoAngles*
MovementSequenceLink	SPEED *speed* DELAY *delay*

El procedimiento de transformación lo dirige el motor de ejecución de JET. Éste recorre el modelo especificando qué plantilla de código se va a invocar para cada tipo de elemento encontrado. Así pues, según la tabla 4.3, las instancias de clase *ComplexMovement* serían traducidas a estructuras iterativas de número de repeticiones establecido por el atributo *loops*, el código que encerrará estas estructuras vendrá como resultado de procesar los elementos contenidos en cada *ComplexMovement*. Por otro lado, los elementos *SimpleMovement* quedan convertidos a una sentencia de movimiento simultáneo de los servosmotores, con ello, el robot transiciona a una pose determinada por el valor de la propiedad *servoAngles*. Por último, las instancias de clase *MovementSequenceLink* se traducen a sentencias que permiten controlar la velocidad de ejecución de los movimientos y retardos (SPEED y DELAY).

4.4.2. Generación de código RoboScript

El lenguaje RoboScript lo forma un subconjunto de sentencias de RoboBASIC destinadas a especificar sólo secuencias de movimientos en el robot. En la tabla 4.4 se resume la equivalencia entre los conceptos del meta-modelo y el código RoboScript. Podemos apreciar que la implementación es muy similar a la expuesta en RoboBASIC, la mayor diferencia se encuentra en la implementación de los bucles, pues al no existir sentencias de control de flujo, es necesario repetir el código tantas veces como indique el atributo *loops*.

TABLA 4.4
TRANSFORMACIÓN MODELO A CÓDIGO ROBOSCRIPT

Conceptos del dominio	Código RoboScript
ComplexMovement	*Código elementos contenidos repetido loops veces*
SimpleMovement	PTP [ALLON | ALLOFF] MOVE24 *servoAngles*
MovementSequenceLink	SPEED *speed* DELAY *delay*

Dada la similitud entre la transformación a código RoboBASIC y a código RoboScript. Para ilustrar la programación en JET seguidamente se añade un fragmento correspondiente a la transformación a RoboBASIC.

```
1       <c:setVariable select="/ComplexMovement/InitialPseudoMovement" var="mov"/>
2       <% boolean stop = false;
3       while(!stop) { %>
4       <c:choose>
5       <c:when test="$mov[self::InitialPseudoMovement]">
6       <c:setVariable select="$mov/outgoing" var="mov"/></c:when>
7       <c:when test="$mov[self::MovementSequenceLink]">
8       <c:if test="$mov/@delay > 0">
9       DELAY <c:get select="$mov/@delay"/></c:if>
10      <c:if test="$mov/@delay > 0">
11      SPEED <c:get select="$mov/@speed"/>
12      </c:if>
13      <c:setVariable select="$mov/target" var="mov"/></c:when>
14      <c:when test="$mov[self::SimpleMovement]">
15      MOVE24
16      <c:if test="$mov/@servoAngle_00>0"><c:get select="$mov/@servoAngle_00"/></c:if>,
17      <c:if test="$mov/@servoAngle_01>0"><c:get select="$mov/@servoAngle_01"/></c:if>,
18      <c:if test="$mov/@servoAngle_02>0"><c:get select="$mov/@servoAngle_02"/></c:if>,
        (...)
19      <c:setVariable select="$mov/outgoing" var="mov"/>
20      </c:when>
21      <c:when test="$mov[self::ComplexMovement]">
22      <c:if test="$mov/@loops > 1">
23      FOR var<%=--varCount%>=0 TO <c:get select="$mov/@loops"/>
24      </c:if>
25      <c:setVariable select="$mov/movements[self::InitialPseudoMovement]" var="mov"/>
26      </c:when>
27      <c:when test="$mov[self::FinalPseudoMovement]">
28      <c:if test="$mov/father/@loops > 1">
29      NEXT
30      </c:if>
31      <c:setVariable select="$mov/father/outgoing/target" var="mov"/>
32      </c:when>
33      <c:otherwise>
34      <% stop = true; %>
35      </c:otherwise>
36      </c:choose>
```

Podemos observar que se trata de un lenguaje de etiquetas que también permite insertar código Java de propósito general. Así pues, el objetivo del código anterior sería el de recorrer los elementos del modelo e ir traduciéndolos en código RoboBASIC. Para ello, se emplea la sentencia iterativa *while* (línea 3) y una estructura selectora <*c:choose*><*c:when…*>, podemos ver resaltadas las diferentes opciones de esta estructura. En cada iteración, la variable *$mov* almacena una instancia del modelo, cada concepto del meta-modelo queda representado en sentencias *when* (líneas 5, 7, 14, 21 y 27), de modo, que se definen las acciones concretas para cada elemento determinado. Tras procesar un item, la variable *$mov* se actualizaría al siguiente elemento enlazado, por ejemplo, si se ha procesado un elemento de tipo *SimpleMovement*, *$mov* adoptará la referencia a *MovementSequenceLink* contenida en su atributo *outgoing* (línea 19). De esta forma, se recorrien todos los enlaces desde *InitialPseudoMovement* a *FinalPseudoMovement.* Cuando encontramos varios niveles jerárquicos por la presencia de subsecuencias de movimientos definidos en un *ComplexMovement,* descendemos en la jerarquía apuntando la variable *$mov* al *InitialPseudoMovement* de la subsecuencia y ascendemos al alcanzar el correspondiente *FinalPseudoMovement* configurando la variable *$mov* para que apunte al siguiente elemento enlazado con el padre (línea 31). Los caracteres que no aparecen contenidos dentro de una etiqueta son escritos de forma literal en el fichero de salida, esta técnica permite añadir en el propio código JET las sentencias RoboBASIC con las que se traducen cada una de las clases del meta-modelo.

4.4.3. Descripción del procedimiento

A continuación, son descritos los pasos que se han de seguir para definir una transformación M2T en el entorno Eclipse.

1. Creación de un proyecto JET. Dado que el motor de transformación de JET es una herramienta independiente, es decir, no requiere la utilización de tecnología enfocada al DSDM, es necesario configurar el campo *modelLoader* en *plugin.xml* con el valor "org.eclipse.jet.emf", de esta forma, se indica que la transformación tendrá como entrada modelos EMF.
2. Tras la creación del proyecto, automáticamente se genera el fichero *main.jet*. Se trata del punto de entrada de la transformación y el código que alberga permite inicializar y organizar la transformación. El código real de la transformación se distribuye en plantillas (ficheros **.jet*), donde cada plantilla podría generar un fichero de salida. Normalmente, las plantillas son invocadas desde el *main.jet*. Así pues, para definir la transformación M2T en HuRoME, primero se ha añadido al proyecto dos nuevos ficheros de texto nombrados con la extensión *.jet*, tras ello, se escribe el código de la transformación a RoboBASIC en uno de ellos y en el otro, el correspondiente a RoboScript. Por último, se modifica el contenido de *main.jet* para enlazar las plantillas y especificar la ruta de los ficheros de salida. En la figura 4.8 puede observarse el contenido de main.jet de HuRoME donde se han resaltado las líneas modificadas que corresponden a las invocaciones.

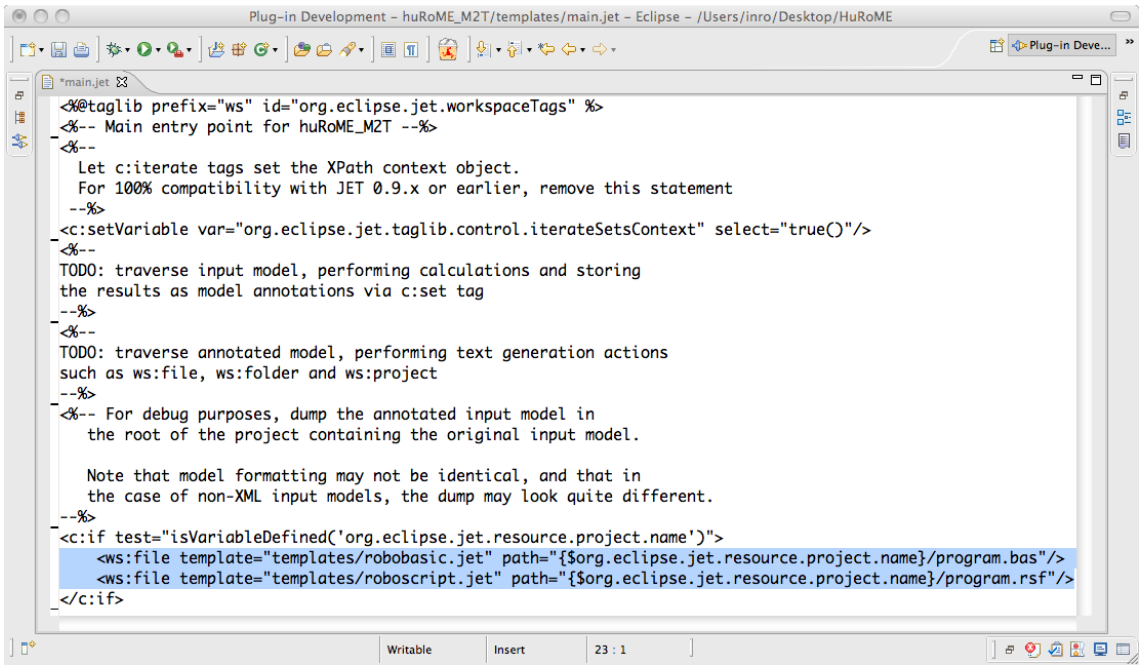


Figura 4.8.: Main.jet de HuRoME con las invocaciones a las plantillas resaltadas.

4.5. Generación de modelos a partir de código

Con el objetivo de soportar el proceso inverso al descrito en las secciones anteriores, proponemos incorporar a HuRoME una transformación texto-a-modelo (T2M) que permita obtener modelos de coreografías a partir de código RoboScript ya existente (bien generado automáticamente por la herramienta anterior o codificado manualmente por un desarrollador). El

framework fundamental empleado para llevar a cabo esta funcionalidad es xText. En términos generales, el procedimiento de transformación de código a modelos de HuRoME sigue dos etapas. En la primera, partiendo del código RoboScript, se realiza una transformación T2M para obtener un modelo conforme a un meta-modelo intermedio, cercano al código, generado de forma automática por xText. Además, xText generará un editor textual para el lenguaje RoboScript, con formateo léxico y validación sintáctica del código. En una segunda etapa, se implementa en ATL una transformación M2M con el fin de obtener modelos conformes al meta-modelo de HuRoME a partir de los modelos xText obtenidos en la etapa anterior.

4.5.1. Desarrollo del editor de código RoboScript

En este apartado expondremos los pasos que se siguen para conseguir un editor textual funcional para el lenguaje de dominio específico (DSL), en este caso RoboScript, utilizando el plug-in de Eclipse xText.

1. Tras crear un proyecto xText vacío, el primer paso consistiría en detallar la sintaxis del lenguaje RoboScript en un fichero de extensión **.xtext* (xText genera este fichero cuando se crea un nuevo proyecto). En la figura 4.9 puede observarse la gramática desarrollada.

2. A partir de la especificación de la gramática, se genera de forma automática la implementación Java del editor textual para código RoboScript. Para ello, hay que ejecutar el workflow (fichero con extensión *.mwe2) que se creó con el proyecto.

En cuanto a la definición de la gramática, debemos comentar que se ha considerado todas las sentencias incluidas en RoboScript excepto GOTO. Se trata de una medida para simplificar el proceso puesto que la sentencia GOTO incita a la aparición de saltos en el código y la formación de programas no estructurados. Ello eleva gravemente la complejidad de la transformación M2M comentada al inicio de la sección ya que se requeriría interpretar el flujo que sigue el código y transformarlo a una versión estructurada de éste. La repercusión negativa en la usabilidad de HuRoME al no considerar GOTO se prevé escasa. En la mayoría de programas que han sido revisados el uso de GOTO es pequeño. Así pues, será responsabilidad del programador que utilice HuRoME el realizar sus programas sin la sentencia GOTO o adaptar aquellos ya existentes para eliminar el uso de esta sentencia. El resto de la definición de la gramática (véase la figura 4.9), podemos contemplar cómo se ha introducido un elemento raíz (*Program)*, tal que todos los demás elementos queden contenidos. Observar que según el atributo *codelines*, éste puede tener cero o más elementos de tipo *Sentence*. El componente *Sentence* sería un elemento que permite generalizar para referirse a todas las posibles instrucciones consideradas, en este caso, apreciamos *Move24*, *Speed*, *Delay* y *Ptpall*, cada una de estas instrucciones son especificadas utilizando diferentes notaciones de escritura (todo mayúsculas, todo minúsculas o primera letra en mayúscula). Por último, se define un enumerador con los posibles valores de la sentencia *Ptpall* y el tipo de datos entero (*INT*).

4.5.2. Transformación a modelo xText

La finalidad principal que persigue el framework xText generando un editor textual con reconocimiento de sintaxis y otras facilidades, no es crear un simple editor específico, sino dar soporte al proceso de transformación T2M. Es decir, cuando un usuario escribe código en el editor textual implementado por xText, no sólo se está coloreando las palabras clave y chequeando la sintaxis del lenguaje, sino que, a la misma vez que se escribe, se almacena en memoria un modelo. Desde esta perspectiva, la gramática del DSL define una sintaxis concreta

textual asociada a un meta-modelo. Este meta-modelo se genera de forma automática, tras el paso 2 visto en el apartado anterior, a partir de los conceptos que aparecen en la misma gramática del lenguaje. En la figura 4.10 puede verse el meta-modelo creado a partir de la gramática de RoboScript. Podemos apreciar que este meta-modelo refleja las relaciones y elementos contenidos en la gramática, de modo que es mucho más cercano al lenguaje que el meta-modelo diseñado para el modelado de coreografías (representado en la figura 4.2).

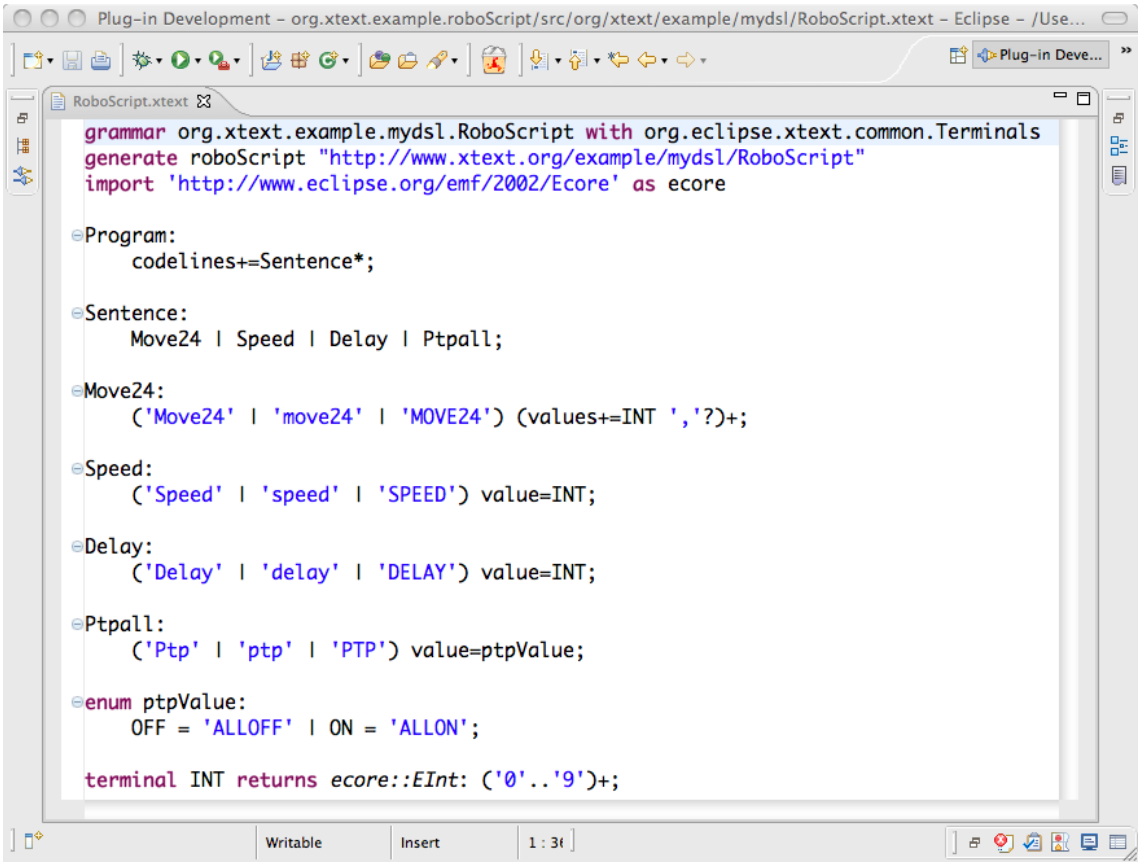


Figura 4.9.: Definición realizada para HuRoME de la gramática del lenguaje RoboScript.

Como hemos comentado anteriormente, el modelo es completado en memoria de forma transparente al usuario, así, será necesario definir un proceso de serialización dirigido a obtener el correspondiente fichero **.xmi*. El plug-in xText no ofrece esta serialización de forma directa desde la interfaz gráfica, será necesario codificar manualmente las llamadas a la API para realizar esta función. Existen pues, dos mecanismos para implementar la serialización de un modelo en memoria, uno tradicional, basado en la codificación manual en Java y otra basada en la tecnología de oAW, con el uso de workflow. La figura 4.11 muestran el método Java y el workflow implementados para el desarrollo del presente proyecto.

4.5.3. Transformación modelo xText a modelo HuRoME

El objetivo final de la transformación es obtener un modelo conforme el meta-modelo de HuRoME a partir de un modelo obtenido con xText con el fin de poder abrirlo usando el editor gráfico de modelos desarrollado con GMF. Para definir esta transformación M2M empleamos la herramienta ATL. A continuación, se dan algunos apuntes sobre el procedimiento que se ha seguido para definir la transformación ATL en HuRoME.

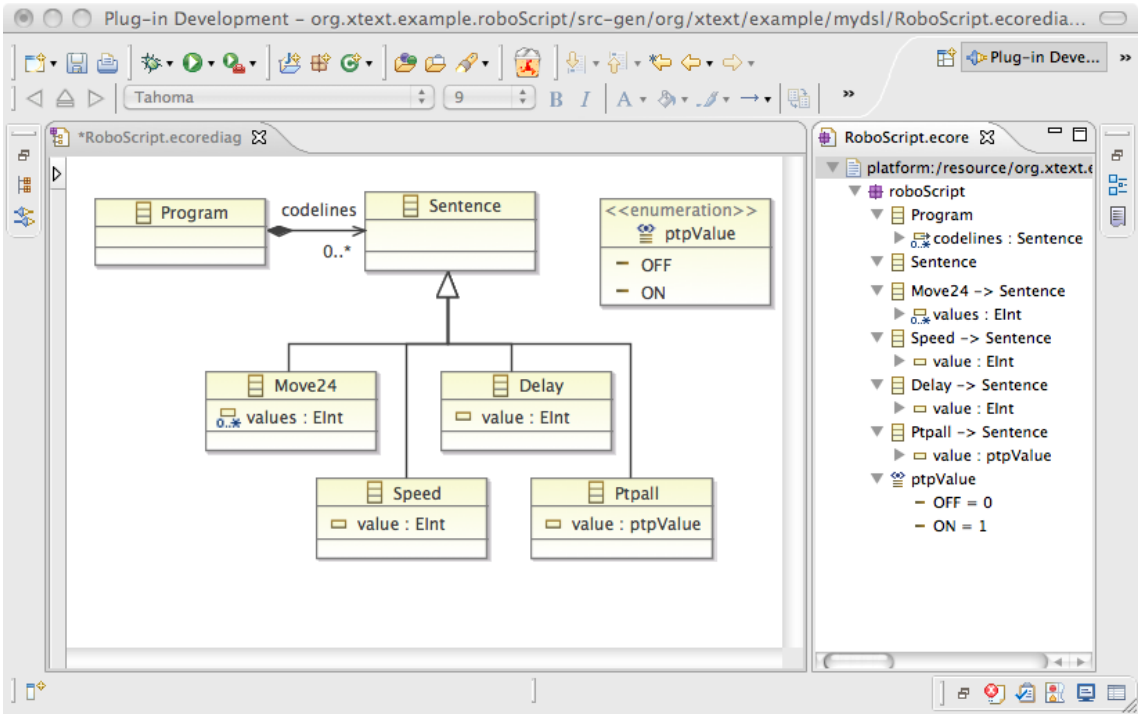


Figura 4.10.: Representación gráfica y en árbol del meta-modelo creado por xText desde la gramática de RoboScript.

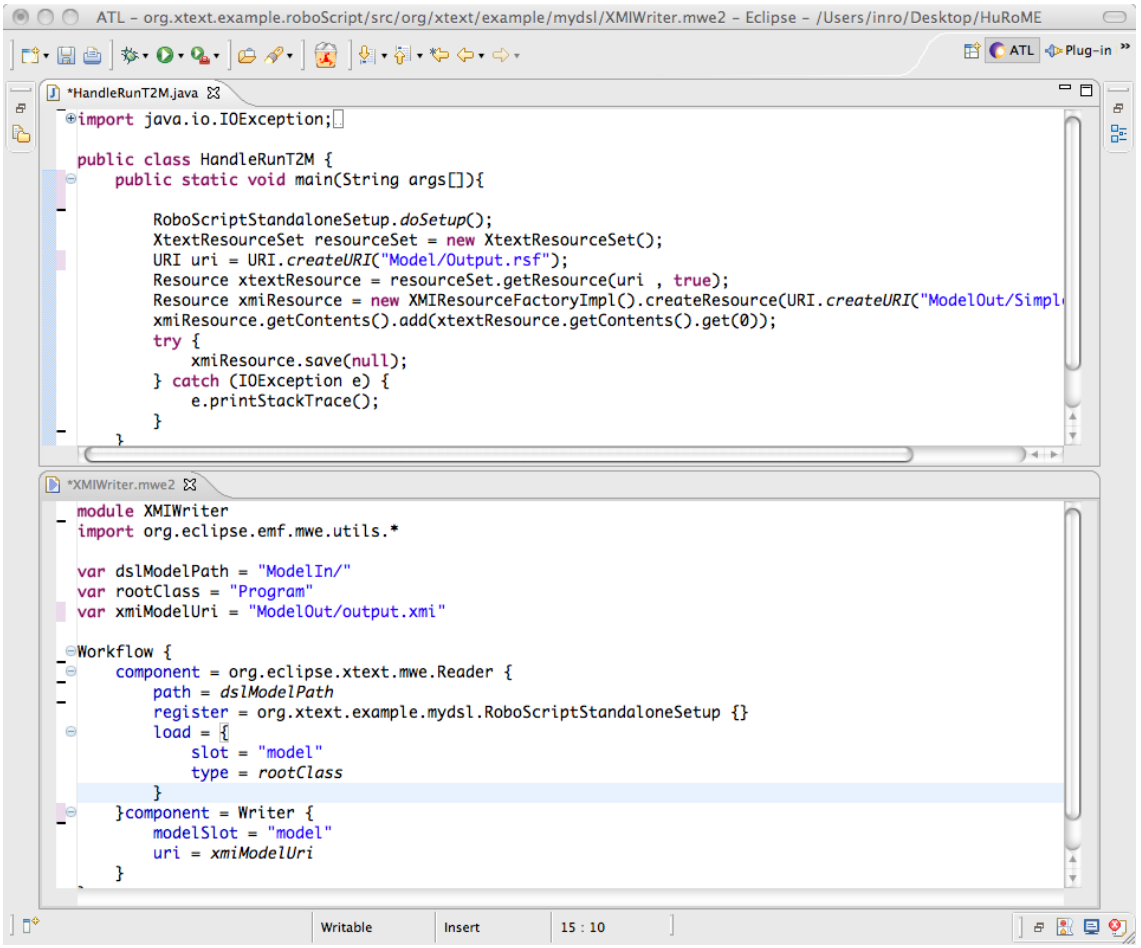


Figura 4.11.: Serialización a XMI de un modelo. Código Java en la parte superior, workflow.en la inferior.

1. Para trabajar con ATL es necesario crear un nuevo proyecto ATL dado que el motor ATL sólo ejecutará la transformación si se encuentra en este tipo de proyecto. Tras ello, se crea un fichero de transformación (fichero ATL con extensión *.atl) que contendrá el código que se defina para la transformación (véase figura 4.12).
2. Aunque no sea necesario para la ejecución de la transformación ATL, conviene registrar los meta-modelos (tanto el de HuRoME como el de xText), por ejemplo, esta acción será fundamental para ejecutar el código asociado a la serialización XMI de los modelos. Para registrar un meta-modelo, debemos tener abierta la perspectiva ATL (puede activarse desde el menú Window), se selecciona el fichero *.ecore y se hace clic sobre la opción *Register Metamodel*) de su menú contextual. En el siguiente capítulo se detallarán más los aspectos relativos al uso de las herramientas creadas.

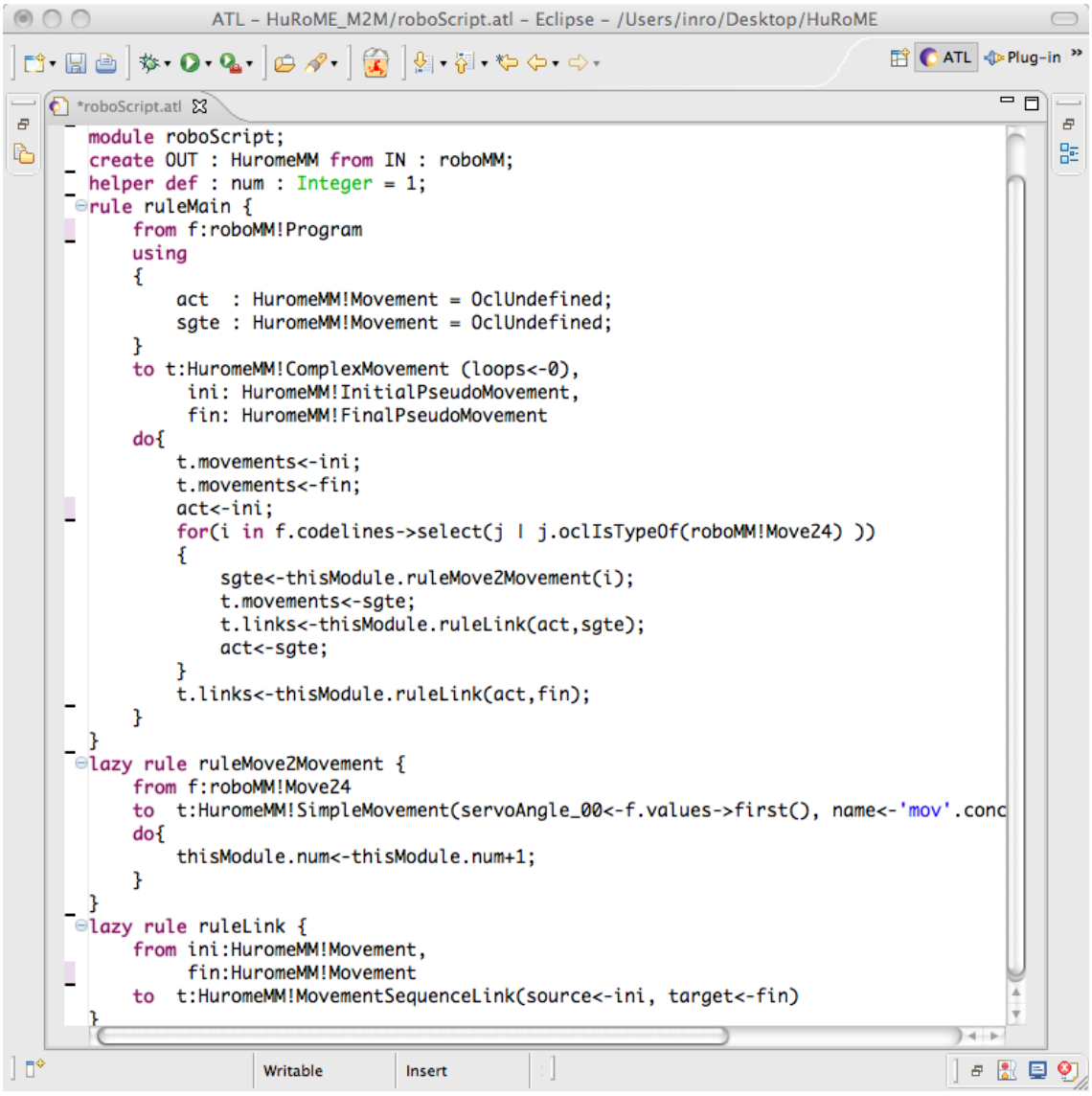


Figura 4.12.: Detalle de un fragmento de código ATL para la transformación M2M de HuRoME.

En la figura 4.12 podemos apreciar la implementación principal ATL que define la transformación M2M. Básicamente, ésta se compone de un elemento declarativo *Matched rule,* identificado como *ruleMain* en el código, que define la regla principal de la transformación, y una serie de *Lazy rules* que son reglas auxiliares imperativas invocadas desde la regla principal. El

funcionamiento es el siguiente, la regla *ruleMain* parte del elemento raíz del modelo de entrada (de tipo *Program*), elemento que contiene en su atributo *codelines* al resto del modelo, así, se obtiene el conjunto de objetos *Sentence* referenciados en codelines. Se ha podido comprobar que este conjunto aparece ordenado en función del orden de aparición en el modelo de origen. Para recorrer el conjunto se emplea la sentencia iterativa *for*, en función del tipo concreto de *Sentence* considerada se llama a la correspondiente regla secundaria destinada a crear los elementos del modelo destino.

4.6. Desarrollo de un plug-in para integrar las herramientas

A lo largo del capítulo se ha visto que el desarrollo de cada faceta de HuRoME (editor gráfico de modelos, generación de código, editor textual para RoboScript y transformación M2M) es desarrollada en proyectos Eclipse diferentes. Así pues, se tiene un conjunto de herramientas que no interaccionan, requiriendo que el usuario ejecute y gestione los ficheros de entrada y salida de los diferentes proyectos en aras de conseguir el flujo de transformaciones que produzcan el resultado final, todo ello de forma totalmente manual. Esto implica un conocimiento preciso acerca del desarrollo y uso del entorno Eclipse, algo que no encajaría con el perfil de los posibles usuarios de HuRoME. En esta sección se exponen algunas ideas dirigidas a integrar los diferentes proyectos creados en Eclipse con el objetivo de mejorar la usabilidad de HuRoME.

La arquitectura en plug-in en la que se fundamenta Eclipse permite generar código que extiende la funcionalidad de otros plug-ins existentes sin interferir en su funcionamiento. Proponemos implementar un plug-in que añada las siguientes capacidades dirigidas a la integración de las herramientas desarrolladas.

- En el editor gráfico de modelos, adición de una nueva opción en el menú para lanzar la generación de código RoboBASIC y RoboScript del un modelo **.xmi* seleccionado.
- En el editor textual, nueva opción en el menú para ejecutar la transformación y obtener un modelo de HuRoME. Ello implica, primero, la serialización del modelo xText y tras ello ejecutar la transformación ATL.

La realización de opciones de menú en la propia interfaz gráfica de los editores permitiría a un usuario dirigir todo el flujo de transformaciones de forma sencilla y enmascarando las tecnologías involucradas tras un simple clic de ratón. Aunque en la versión actual de HuRoME no se ha implementado estas facilidades, a continuación, exponemos de forma resumida el procedimiento que se realizaría para extender los menús de las herramientas creadas.

4.6.1. Extensión de los menús de los editores creados

El procedimiento para extender la funcionalidad de un plug-in existente, en este caso del editor gráfico de modelos y del editor textual RoboScript, se basa en la creación de un plug-in adicional dirigido a extender los ya existentes. Desde esta perspectiva, el código de las extensiones queda separado del código generado por los frameworks, lo que favorece la organización de las fuentes y limita la aparición de errores. Seguidamente, se ilustra el procedimiento.

1. Se crea un nuevo proyecto dedicado a desarrollar un nuevo plug-in de Eclipse (*Plug-in Project*). Entre las opciones ofrecidas en la ventana de creación, deshabilitar la opción "Generate an activator" pues no será necesario la generación código Java para el control del ciclo de vida del plug-in dado que los métodos que se desarrollarán en el plug-in serán

invocados por ocurrencia de eventos en la interfaz que deseamos extender y no requieran un hilo de ejecución permanente.

2. Abrimos el fichero *plugin.xml* asociado al proyecto creado (véase figura 4.12). La adición de nuevas opciones en un menú de la interfaz de usuario de un determinado plug-in se lleva a cabo, creando el nuevo item que deseamos añadir y definiendo el controlador que será llamado cuando ocurra una acción sobre éste. Para ello, no se necesitará programar nada, tan sólo habrá que configurar el fichero *plugin.xml*.

3. En la pestaña *Extensions* de *plugin.xml* debe añadirse una nueva extensión de tipo "*org.eclipse.ui.menus*" que permitirá concretar los detalles del nuevo item. Para ello, previamente habrá que agregar como dependencia (pentaña *Dependences*) el plug-in *org.eclipse.ui*. Dentro de la extensión añadida creamos un elemento "*menuContribution*" del que debe completarse el campo URI especificando el objetivo de la extensión, en este caso podría ser "*popup:org.eclipse.jdt.ui.PackageExplorer*" que identifica la vista del explorador de paquetes del editor en cuestión. Añadimos un nuevo elemento al anterior de tipo *Command,* éste agrupa información relativa al item como el icono asociado, etiqueta, etc, se dejarán todas las opciones en blanco. Tras el siguiente paso, cuando se cree el controlador, deberemos de configurar el campo *commandId*.

4. Añadimos una nueva extensión de tipo "*org.eclipse.ui.commands*" que permitirá definir el controlador asociado al item. Dentro de esta extensión creamos un elemento *Command* y rellenamos el campo *id* (por ejemplo, "HuRoME_EXT.MenuItem1"), este mismo identificador deberá escribirse en el campo *commandId* del paso anterior. Además, debe completarse el campo *name* con una frase que describa la acción (por ejemplo, "Opción 1 Menu"). Por último, *defautHandler* quedará configurado cuando se declare el método Java que implementa las acciones asociadas.

5. Se crea la clase Java que implementa el controlador, esta clase debe extender a *AbstractHandler*. Dentro de esta clase el método que será invocado tras pulsar sobre la respectiva opción de menú es "*execute*", será este método el que debamos programar.

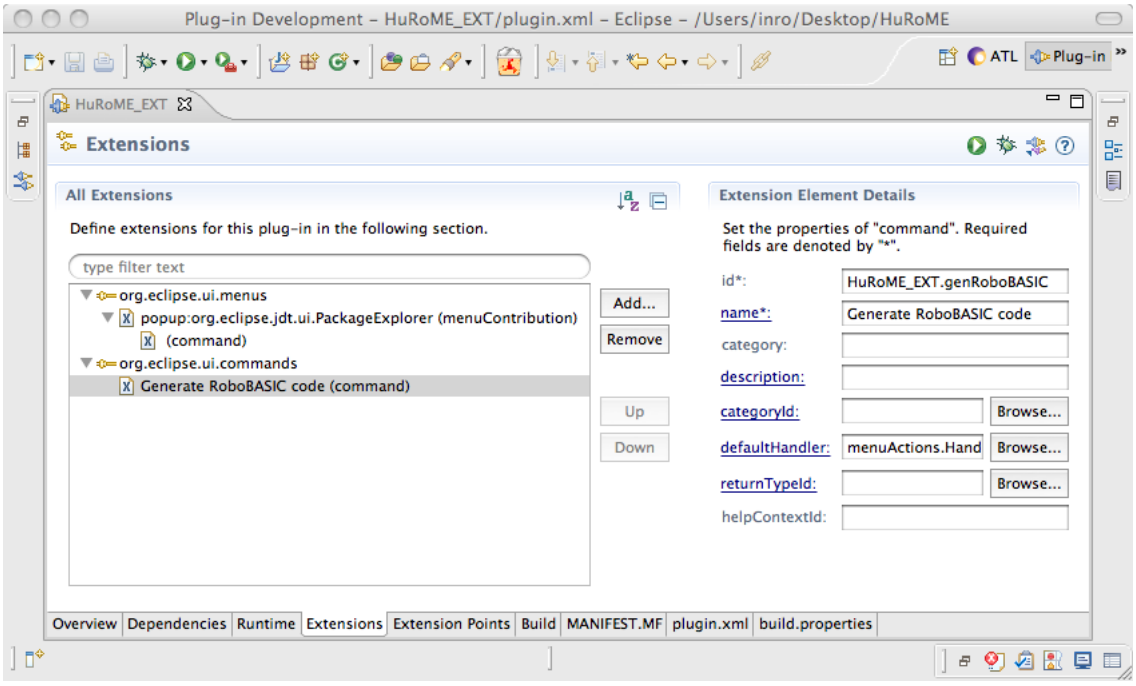


Figura 4.12.: Detalle de un fragmento de código Java dedicado a obtener la ruta del fichero seleccionado.

Debemos tener en cuenta, a la hora de escribir el código Java asociado a las acciones del menú, que habrá que añadir dependencias no sólo a los plug-in que se extienden sino también a aquellos frameworks que se utilicen para llevar a cabo las acciones (por ejemplo, será muy común añadir el plug-in *org.eclipse.emf.core* para poder acceder a los métodos del API que facilitan el manejo de modelos y meta-modelos). Por otro lado, en la figura 4.13 se muestra un fragmento de código Java destinado a obtener la ruta de un fichero seleccionado. Puede resultar muy útil cuando se crean acciones sobre tipos específicos de ficheros, así, por ejemplo, seleccionar un fichero **.xmi* y sobre su menú contextual accionar la generación de código RoboBASIC, implica que el método *execute* necesitaría obtener la ruta del modelo para invocar el motor de transformación JET.

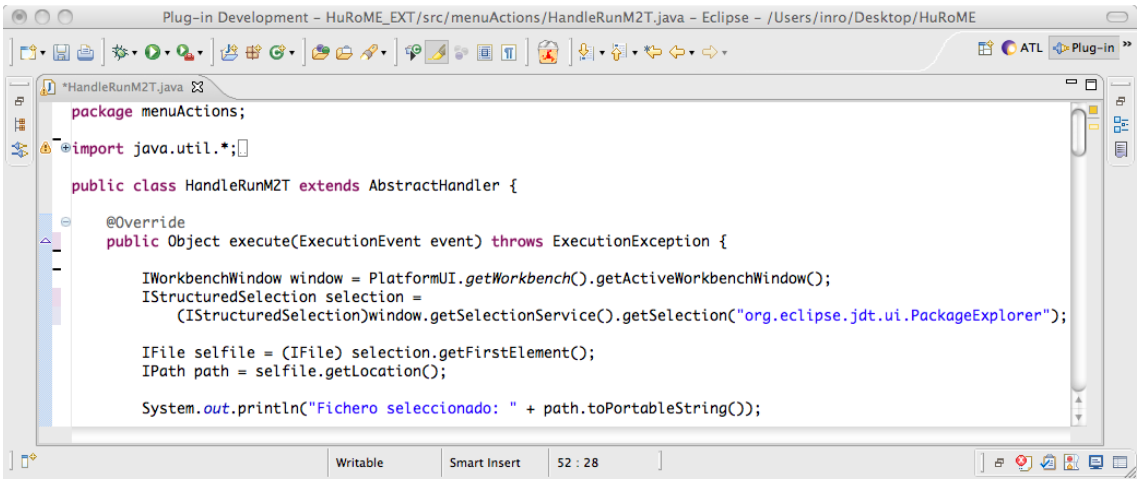


Figura 4.13.: Detalle de un fragmento de código Java dedicado a obtener la ruta del fichero seleccionado.

CAPÍTULO V

Guía de uso del entorno HuRoME

En este capítulo se describe el uso de las distintas herramientas desarrolladas anteriormente que componen HuRoME. Comentar, que en la versión actual del entorno HuRoME el usuario deberá ser capaz de lidiar con la complejidad implícita de lanzar el flujo de transformaciones de forma manual, es decir, en el parte final del capítulo anterior se expusieron algunas nociones sobre cómo añadir opciones de menú para integrar las herramientas y facilitar así su uso, sin embargo, la versión que se presenta en este proyecto no incorpora dichas mejoras, por lo que será necesario que el usuario maneje cada uno de los proyectos y conozca las peculiaridades de cada una las tecnologías empleadas.

5.1. Consideraciones iniciales

Antes de abordar en profundidad el capítulo, realizamos las siguientes consideraciones:

- La versión de Eclipse que se ha empleado para el desarrollo el proyecto ha sido *Galileo* configurado con el bundle *Eclipse Modelling Tools v.1.2*, y el plug-in *XText Antlr Runtime v.1.* Puede encontrar el correspondiente instalador en [11].
- Para ejecutar los plug-in de los editores del entorno HuRoME debemos tener la aplicación Eclipse con todos los proyectos desarrollados abiertos. El usuario debe acceder al menú *Run* y hacer clic en *Run Configurations*. Sobre la ventana que aparece, seleccionar *Eclipse Application* y pulsar *Run*. De esta forma se abrirá una nueva aplicación Eclipse con los plug-in implementados.
- Es recomendable ampliar la memoria asignada a Eclipse, para ello, en la ventana *Run Configurations* se añade las siguientes opciones en la pestaña *Arguments*, concretamente en el panel *Program arguments*: "-Xms256m -Xmx256m". Esto asigna 256MB a Eclipse, podría ajustarse esta asignación de memoria en función de las características el ordenador que se emplee.

5.2. Definición de coreografías

En esta sección introducimos el uso de HuRoME para especificar secuencias de movimientos según los conceptos del meta-modelos diseñado para tal fin (véase la figura 4.2). Así, comenzamos lanzando una segunda aplicación Eclipse para ejecutar el plug-in implementado con GMF. Una vez abierta esta nueva instancia, se crea un proyecto Java y tras ello, un nuevo modelo HuRoME (New→Other→Robonova Diagram). Aparecerá el editor gráfico de modelos listo para ser utilizado. Notar que, tras la creación de un nuevo modelo, surgen dos nuevos ficheros, uno de extensión *.robonova (modelo XMI) y otro *.robonova_diagram (información gráfica del diagrama).

5.2.1. Descripción del editor gráfico de modelos

En la figura 5.1 puede observarse el aspecto del editor gráfico de modelos, en éste, podemos diferenciar los cuatro elementos principales que lo componen.

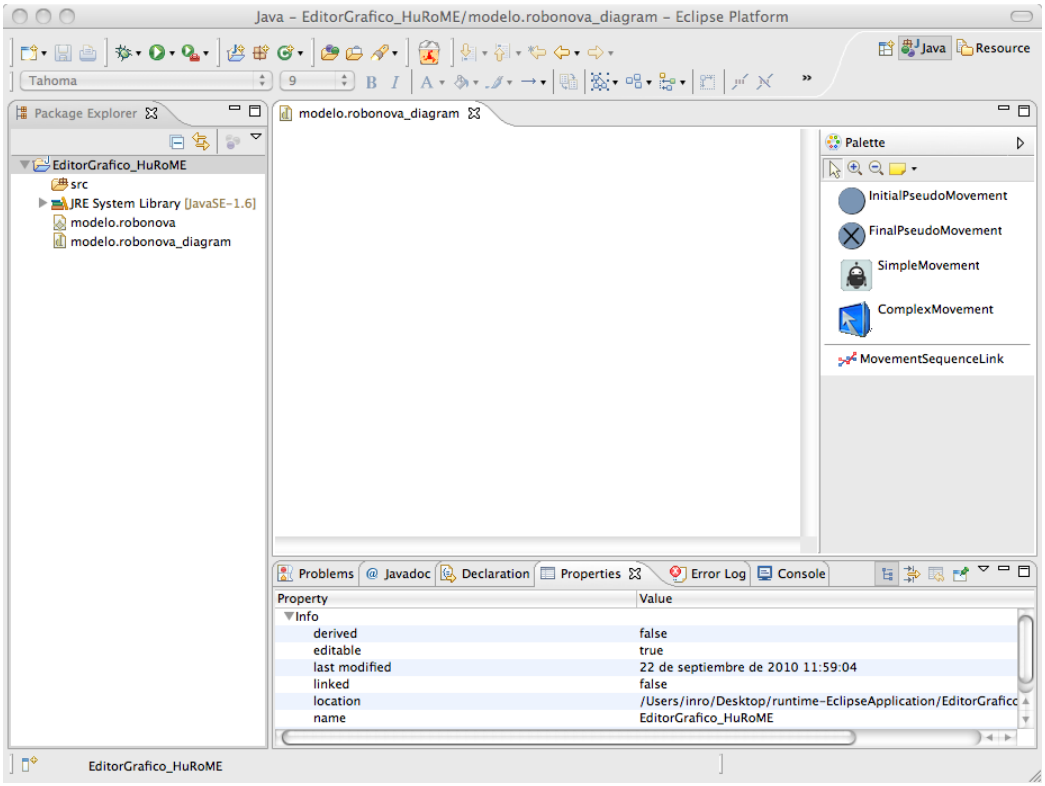


Figura 5.1.: Apariencia del editor gráfico de modelos de HuRoME.

- ***Package Explorer***. Podemos ver el explorador de paquetes en el lado izquierdo de la figura, además apreciamos los archivos asociados a un modelo HuRoME creado (modelo.robonova y modelo.robonova_diagram). El explorador permitirá gestionar la organización del proyecto, ejecutar acciones específicas sobre un fichero y tener una perspectiva general del proyecto.
- ***Panel de vistas.*** En la parte inferior de la interfaz encontramos la agrupación en pestañas de una serie de vistas que nos permitirán entre otras cosas, acceder a los atributos definidos en las meta-clases instanciadas del modelo (vista *Properties*) o registrar los errores producidos al validar el modelo (vista *Problems*).

- ***Paleta de herramientas.*** Este panel aparece en la parte derecha de la interfaz, permite seleccionar el concepto que se desea añadir al modelo. En la paleta se muestran los conceptos definidos en el meta-modelo de HuRoME junto a su representación gráfica, además, se facilita opciones adicionales como zoom, herramienta de selección y disposición de notas en el diagrama.
- ***Área de trabajo.*** La zona central de la pantalla está destinada al trazado y visualización del modelo.

5.2.2. Diseño de un modelo

La especificación de un modelo puede realizarse de forma gráfica, seleccionando los objetos a dibujar de la paleta de herramientas y trazándolos en área de trabajo o bien, de forma más rudimentaria, utilizando el *Tree Editor*, de modo que, los elementos del modelo pasan a adoptar una representación de nodos de un árbol. Ambas perspectivas son equivalentes y se mantienen sincronizadas y consistentes. Así, es posible realizar cambios en el modelo utilizando uno de los editores que tras ser salvados aparecerá en el otro. Notar que si variamos de forma gráfica el modelo, estaremos modificando el archivo *.robonova_dagram (con la nueva información gráfica del diagrama) y el *.robonova con los elementos que conforman el modelo, por lo tanto, dado que el *Tree Editor* ofrece una vista del archivo *.robonova, cuando éste cambia se ve alterada de forma directa esta vista en el editor. Sin embargo, una alteración del modelo utilizando el *Tree Editor* implica la actualización del archivo *.robonova_diagram a través, de un mecanismo implícito que genera automáticamente la representación gráfica asociada. En la figura 5.2 se muestra el mismo modelo representado en los dos editores.

Se emplee uno u otro editor para diseñar el modelo, tendremos que configurar los atributos asociados a cada elemento, para completar, por ejemplo, los valores angulares de un movimiento, para ello, podemos utilizar la vista *Properties*. De forma complementaria, según podemos apreciar en el diagrama de la figura 4.2, es posible tener acceso a algunos de los atributos desde el propio editor gráfico, por ejemplo, los nombres de los movimientos o el valor de retardo y velocidad, información dispuesta como "<*speed*, *delay*>" junto a los enlaces (*MovementSequenceLink*).

Una cuestión importante en el editor gráfico de modelos es la eliminación de elementos. Si seleccionamos un objeto y desplegamos el menú asociado haciendo clic en el botón secundario del ratón, observamos que aparecen dos opciones de borrado: *Delete from Diagram* y *Delete from Model*. La primera permite eliminar gráficamente el elemento pero no tiene efecto en el archivo *.robonova, la segunda opción, sin embargo, sí eliminaría completamente el elemento del modelo.

5.2.2.1. Especificando una coreografía

A modo de ejemplo, se propone realizar el modelo correspondiente a una coreografía sencilla. En la secuencia que planteamos el robot parte de una posición estándar, se encuentra alzado sobre sus piernas y con los brazos relajados, tras ello, se inclinará a modo de reverencia y volverá a su posición, por último, termina levantando un brazo como despedida. Notar que la coreografía estaría compuesta básicamente por cuatro movimientos simples, uno inicial para adoptar la posición estándar, la acción en la que se inclina, el retorno a la posición estándar y por último, el movimiento que alza el brazo. No obstante, proponemos agrupar los tres primeros movimientos correspondientes a la reverencia en un movimiento complejo, dado que conforman

por sí mismos una unidad funcional con sentido propio (sub-secuencia o sub-coreografía). Así, también debemos contar con los enlaces pues, a parte usarse para definir la secuencia de movimientos, permiten configurar retardos y velocidades de ejecución de los movimientos. La figura 5.3 muestra el modelo implementado.

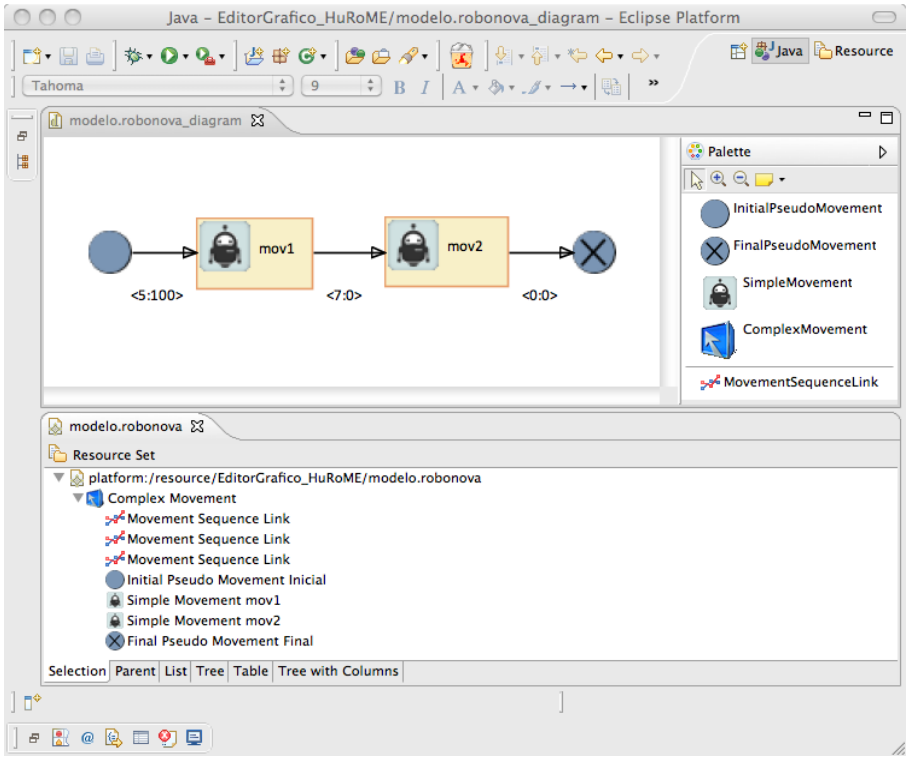


Figura 5.2.: Sincronización entre el editor gráfico y el *Tree Editor*.

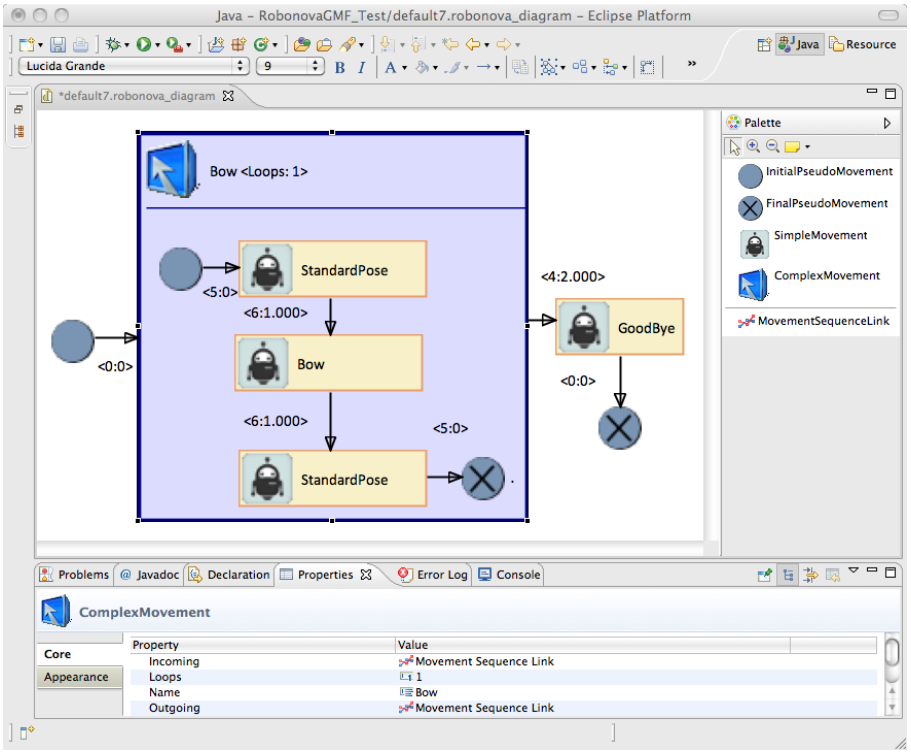


Figura 5.3.: Representación de la coreografía de ejemplo.

En la figura 5.3 pueden apreciarse los retardos y velocidades configuradas, así, por ejemplo, la secuencia comienza con la transición a la postura estándar (StandardPose) a una velocidad de 5, tras 1000ms de retardo el robot se inclina (Bow) a una velocidad de 6 y 1000ms después vuelve a la postura estándar, espera 2000ms y alza el brazo (GoodBye) a velocidad 4. Podemos ver también, que el movimiento complejo Bow se repite únicamente una vez (loops = 1), podría haberse configurado un número determinado de repeticiones variando el valor loops. Por supuesto, aunque no quede visualizado en la figura, se habrían introducido, en la vista *Properties*, los valores angulares que definen cada movimiento simple (atributos *Servo Angle 00,… ,Servo Angle 21*).

5.2.3. Validación del modelo

Cuando se abordó el desarrollo de HuRoME se escribió acerca de la implementación automática de una función de validación dirigida a detectar una violación de las restricciones OCL definidas o de las reglas sintácticas implícitamente recogidas en el meta-modelo. El validador puede adoptar dos posibles comportamientos en función de cómo fue configurada su propiedad *USE IN LIVE MODE*.

- Si la propiedad vale *true*, se chequea de forma continua el modelo durante su diseño, es decir, a la misma vez que un usuario va dibujando los distintos elementos del modelo, se está verificando el mismo en busca de errores, así, cuando se detecta una violación de las reglas, el editor impedirá realizar la operación de trazado (no deja dibujar configuraciones sintácticamente incorrectas).
- Si la propiedad vale *false*, la validación se realiza sólo cuando el usuario selecciona la opción *Validate* en el menú. Los elementos del modelo causantes del quebrantamiento de una o más restricciones son marcados con un símbolo de error y registrados en la vista *Problems*. La figura 4.4 ilustra la validación de un modelo, en concreto, se trataría de un incumplimiento de la regla OCL que impide el auto-referenciado.

HuRoME adopta una política de validación accionada de forma manual por el usuario tras el diseño del modelo. Debemos tener en cuenta que no siempre puede aplicarse el tipo de validación “en vivo” ya que podría bloquear el desarrollo. Por ejemplo, supongamos que se ha definido una regla OCL que dispone que el identificador de un elemento debe ser unívoco y no nulo, en esta situación podría ser inviable usar este tipo de validación, puesto que la herramienta no permitiría en ningún momento dibujar nada, ya que cuando se crea un objeto su identificador es nulo hasta que el usuario no lo configura.

5.3. Generación de código

Para lanzar la generación automática de código RoboBASIC o RoboScript desde un modelo creado y validado, se siguen los siguientes pasos:

1. Copiamos el modelo (archivo **.robonova*) y lo movemos a la carpeta llamada *model* localizada en el proyecto JET (primera instancia de la aplicación Eclipse).
2. Abrimos la ventana *Run Configurations* (seleccionando Run→Run Configurations). Véase la figura 5.5. En esta ventana se crea una nueva configuración haciendo doble clic en *JET Transformation* de la lista (sólo en el caso de que no exista ya una instancia creada) y escribimos la ruta al modelo de entrada de la transformación. Finalmente pulsamos *Run*.

3. Aparecerá un fichero de salida con el código en la raíz del proyecto. Recordad que la configuración de la ruta de salida se realiza en el *main.jet*.

Continuando con el ejemplo iniciado en la sección anterior, la figura 5.6 muestra los resultados de la transformación a código del modelo diseñado.

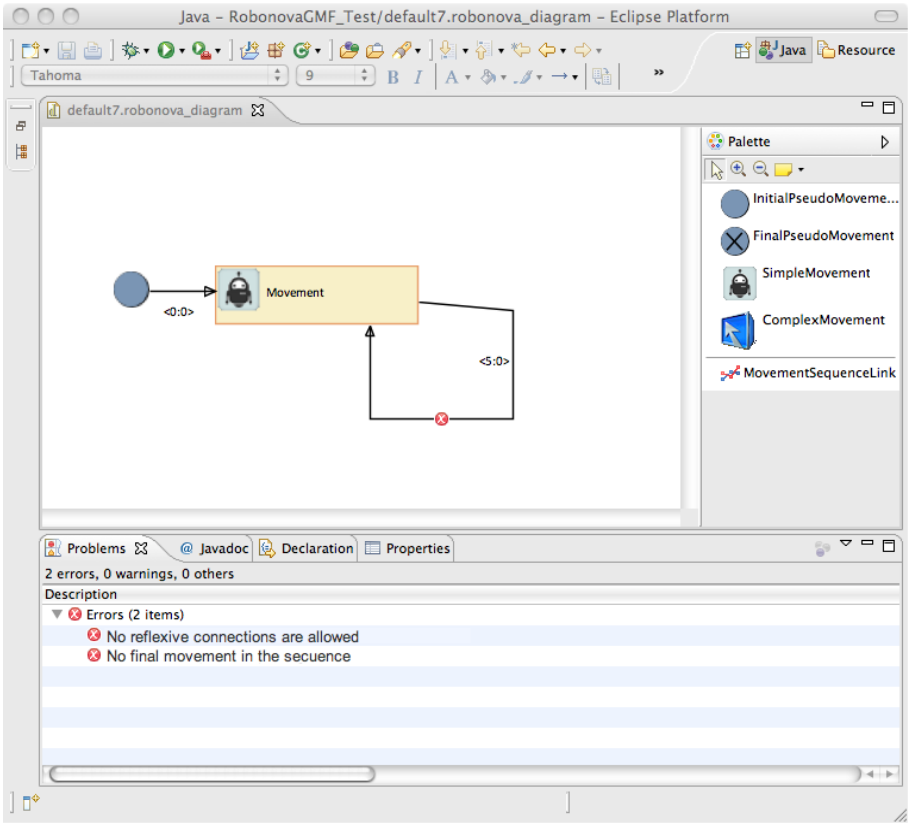


Figura 5.4.: Errores de validación detectados por la herramienta.

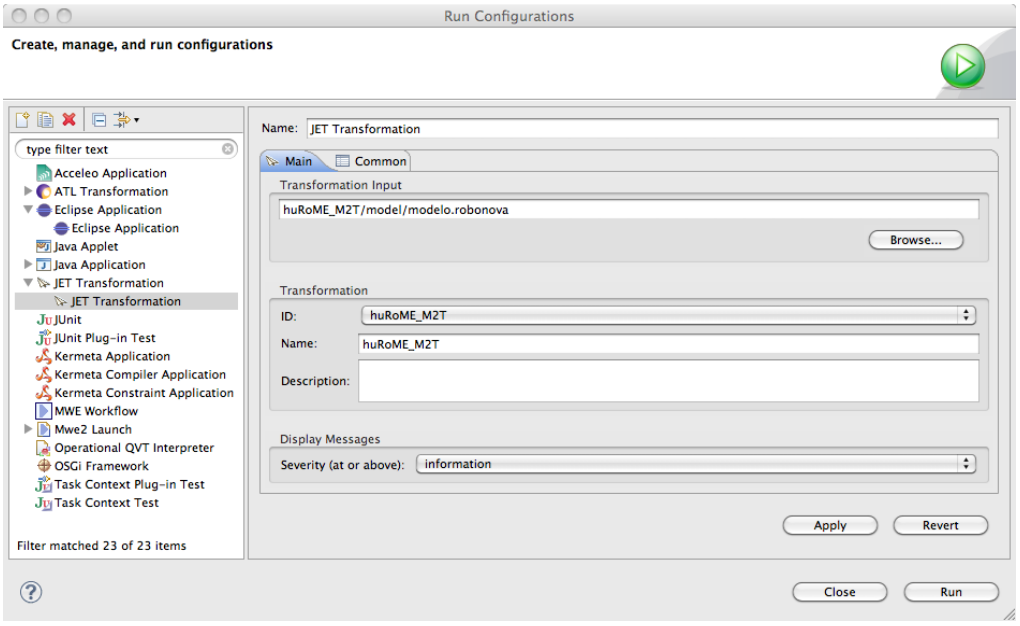


Figura 5.5.: Ventana *Run Configurations* con la transformación JET seleccionada.

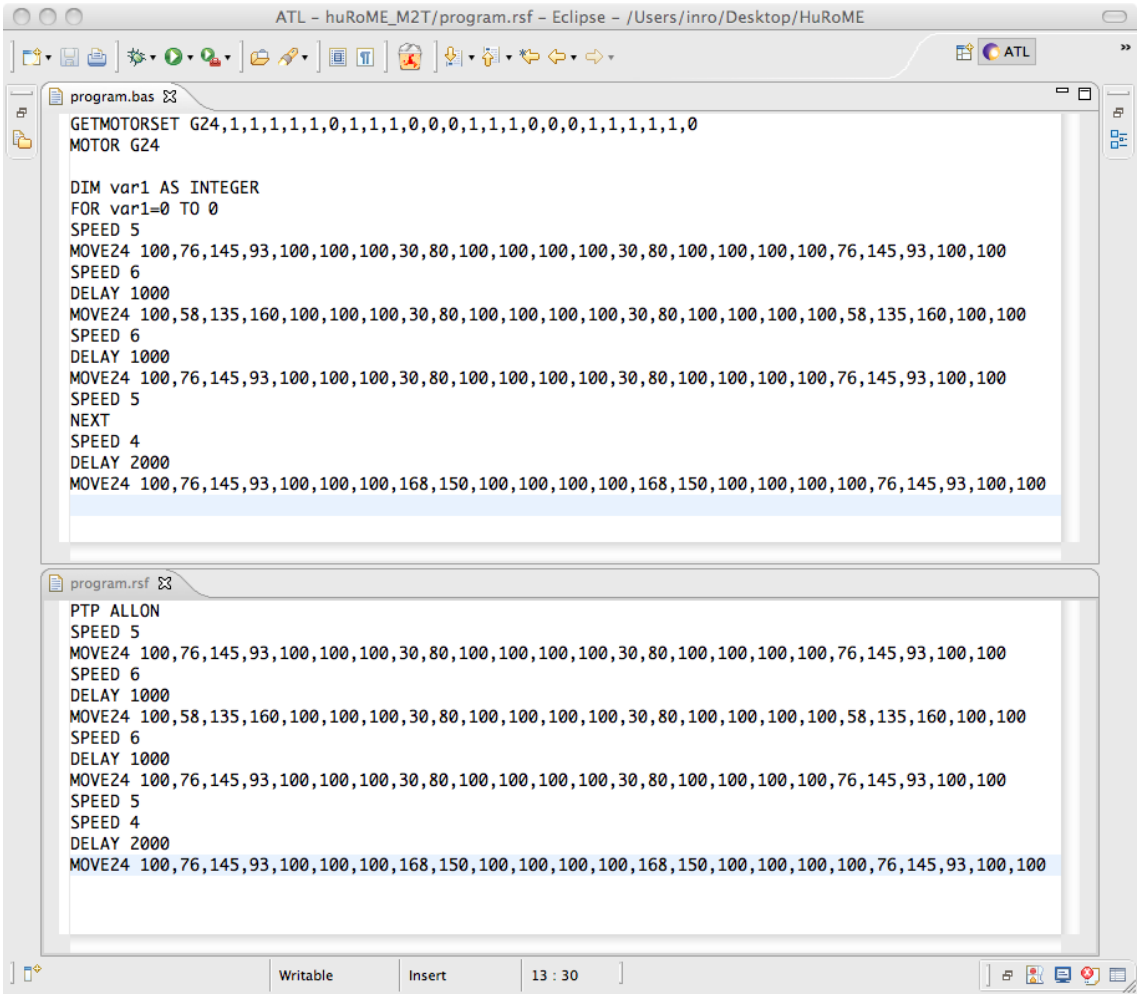


Figura 5.6.: Código generado. La parte superior corresponde a RoboBASIC, la parte inferior a RoboScript.

5.4. Obtención de modelos desde código

De forma similar a lo que se hizo con el editor gráfico de modelos, se deberá arrancar una segunda aplicación Eclipse para ejecutar el plug-in correspondiente al editor textual implementado con xText. Una vez abierta esta nueva instancia, se crea un proyecto General y tras ello, un nuevo archivo con extensión **.rsf*. Esta extensión será reconocida y, por lo tanto, aparecerá el editor textual de código RoboScript.

5.4.1. Descripción del editor textual de modelos

La figura 5.7 muestra el editor textual diseñado para realizar el proceso de reingeniería desde código RoboScript. Podemos observar las diferentes partes que compondrían este editor. En primer lugar, al igual que el editor gráfico de modelos, aparecería el *Package Explorer* y un conjunto de vistas distribuidas en pestañas, sin embargo, en la parte central se hallaría el editor, propiamente dicho. Puede apreciarse que las palabras clave especificadas en la gramática del lenguaje son reconocidas y resaltadas. Además, la aplicación permite realizar un chequeo sintáctico a la vez que se escribe el código. Notar que la sentencia "*Unknown_sentence*" no es reconocida por el validador y quedaría marcada con un símbolo de error y registrada en la vista *Problems*.

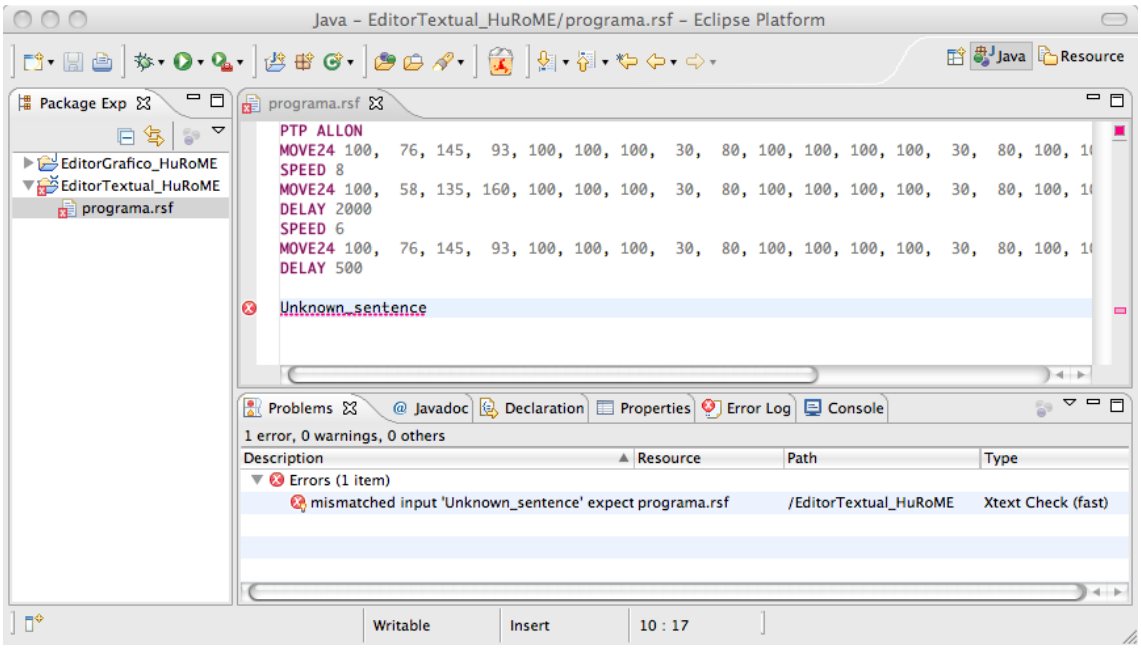


Figura 5.7.: Editor textual de HuRoME generado con xText.

El código que aparece en la figura 5.7 (exceptuando la última línea) ha sido copiado y pegado en el editor de forma directa. Se corresponde con uno de los programas demostrativos que incorpora el robot Robonova, en concreto, se trata de una versión de la reverencia, anteriormente empleada como parte de un ejemplo, compuesta por tres movimientos: una posición estándar, inclinación y vuelta a la posición de partida. Como se comentó en el capítulo anterior, este código realmente representa un modelo conforme al meta-modelo que xText infiere de la gramática del lenguaje. Así, a medida que se escribe el código en el editor, una representación del modelo asociado es completado en memoria. Para poder utilizar este modelo en el flujo de transformaciones de HuRoME es necesario obtener un archivo con formato XMI.

5.4.3. Serialización del modelo

Para obtener el modelo serializado en formato XMI es necesario ejecutar el código de la figura 4.11. Dado que es posible abordar la serialización empleando dos tecnologías diferentes: método Java o workflow. A continuación, se detalla el procedimiento para cada una de éstas.

- ***Método Java.*** Se trata de una aplicación Java compuesta por una única clase y un único método (main), desde éste se realizan las llamadas a las funcionalidades de los frameworks que permiten transformar el código RoboScript a un modelo en memoria y tras ello, escribirlo en un archivo *.xmi. Así, el proceso se basa en la lectura del fichero de código *.rsf para obtener el modelo equivalente *.xmi, por ello, previamente debe configurarse las rutas de entrada y salida en el propio código Java. La ejecución es sencilla, basta con seleccionar el fichero .java correspondiente, y acceder a la opción del menú contextual *Run As→Java Application*.
- ***Workflow***. Aunque la tecnología es diferente a la anterior, este mecanismo también permite realizar llamadas a funcionalidades que ofrecen los frameworks de DSDM. Su ejecución es igual que en el caso anterior, a excepción, de que debe seleccionarse en el menú la opción *MWE2 Workflow* en lugar de *Java Application*. En este caso es conveniente revisar la ruta de entrada y salida (escrita en el propio código) para conocer la ubicación donde debe existir el fichero *.rsf y donde encontraremos el fichero de salida *.xmi.

Debemos comentar que realmente el proceso de serialización tiene como entrada el código RoboScript, es decir, actúa como un parser entre código y modelo. Así pues, se podría prescindir del editor textual en el caso de no necesitar de un medio para escribir y verificar la sintaxis. No obstante, debe recordarse que este editor fue generado de forma automática, rápidamente y casi sin esfuerzo (comparado con el trabajo que hubiera conllevado realizar esta codificación manualmente). Además, aunque no se llegue a utilizar el editor como tal, el código generado sobre el que se sustenta el editor resulta fundamental para la implementación de los métodos de serialización. En la figura 5.8 puede verse el modelo obtenido desde código RoboScript de la figura 5.7, este modelo sería conforme al meta-modelo generado por xText (véase figura 4.10).

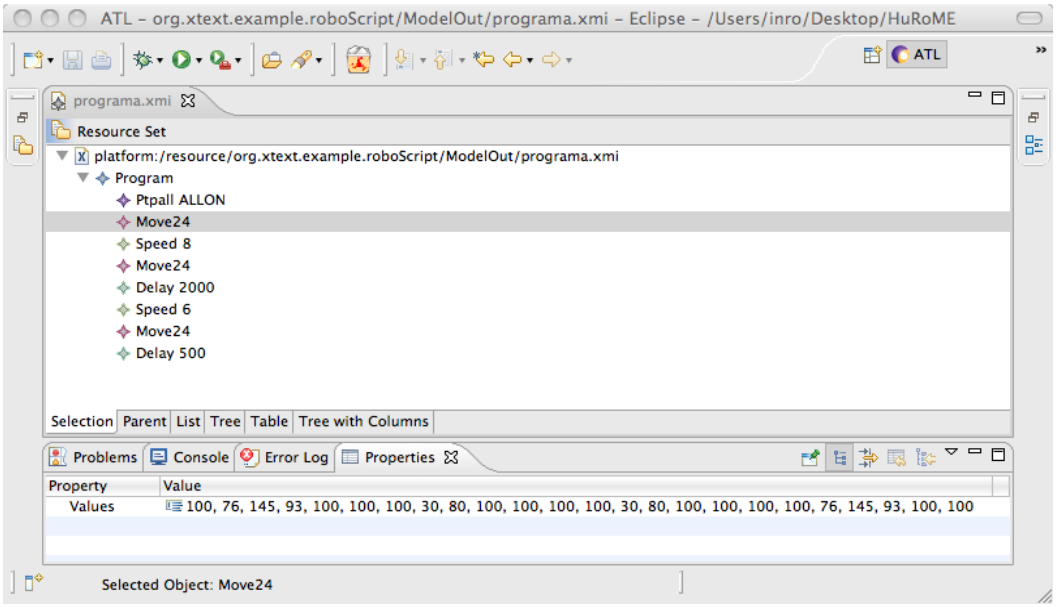


Figura 5.8.: Modelo tras el proceso de serialización.

5.4.4. Ejecución de la transformación modelo-a-modelo

Para lanzar la transformación ATL para convertir modelos expresados conforme al meta-modelo de xText a modelos de coreografías de HuRoME, se siguen los siguientes pasos:

1. Abrimos la ventana *Run Configurations* (seleccionando Run→Run Configurations). Véase la figura 5.5. En esta ventana se crea una nueva configuración haciendo doble clic en *ATL Transformation* de la lista (sólo en el caso de que no exista ya una instancia creada) y revisamos o escribimos (si es la primera vez) la ruta a los meta-modelos y modelos de entrada y salida (véase como ejemplo la figura 5.9). Finalmente pulsamos *Run*.
2. Se creará un nuevo modelo con extensión *.robonova que podrá abrirse con el *Tree Editor*.
3. Para conseguir una representación gráfica del modelo y poder abrirlo utilizando el editor GMF, debemos copiar el fichero *.robonova en el proyecto Java que fue creado para albergar el plug-in del editor gráfico. Tras ello, seleccionar el fichero y desplegar su menú contextual haciendo clic sobre la opción “Inicialize robonova_diagram diagram file”, ésta generará forma automática el fichero *.robonova_diagram. En la figura 5.10 puede verse el modelo del ejemplo en sus distintas representaciones.

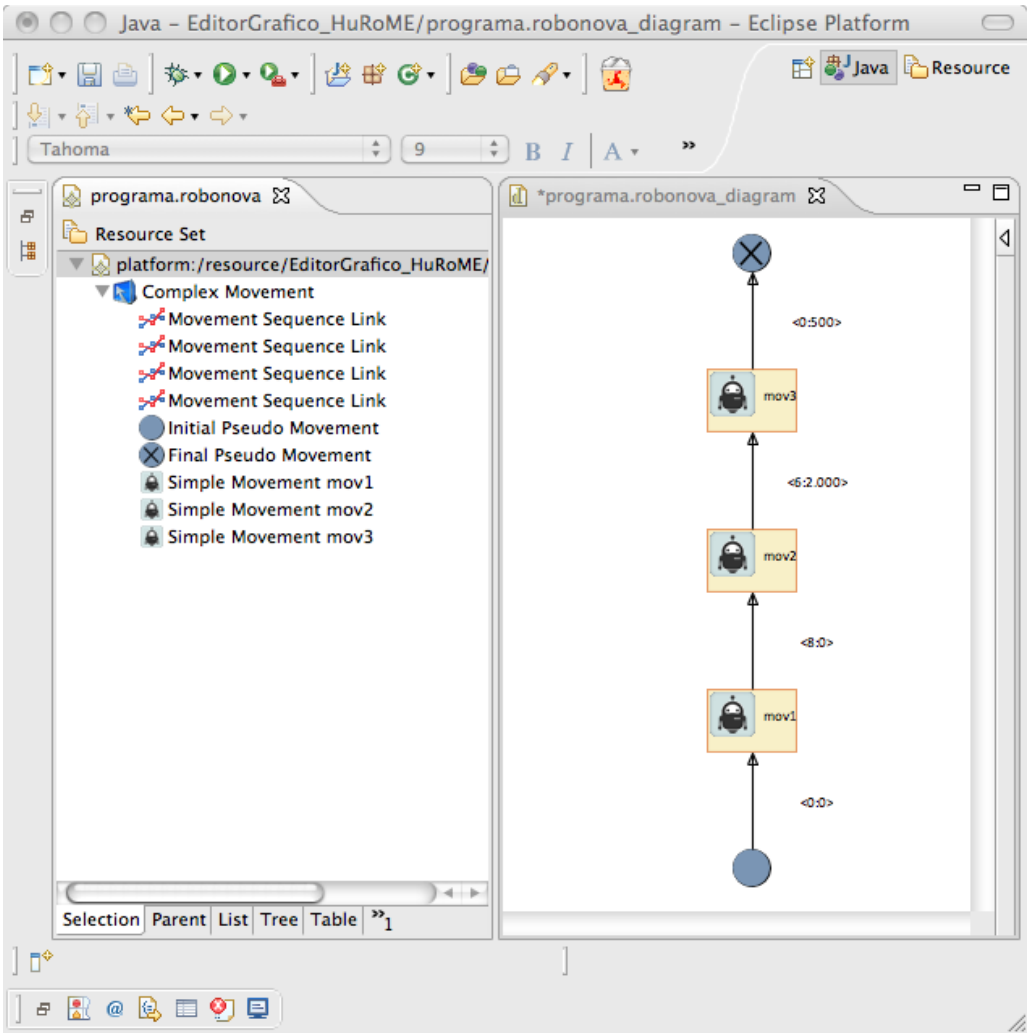


Figura 5.10.: Modelo final tras la transformación ATL.

CAPÍTULO VI

Conclusiones y Líneas de Trabajos Futuras

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

6.1. Conclusiones

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

- ✓ Se desarrollado un lenguaje de modelado dirigido a crear secuencias de movimientos en un robot Robonova. En base a éste, se ha implementado una herramienta de modelado gráfico dirigida a facilitar la especificación y validación de modelos. Se han desarrollado transformaciones de generación automática de código RoboBASIC y RoboScript. Por último, una herramienta de soporte para realizar el proceso de reingeniería desde código RoboScript existente. Al conjunto de todas estas herramientas se le ha llamado HuRoME.
- ✓ El desarrollo de HuRoME para el robot Robonova no tienen impacto alguno en el panorama de la robótica actual. Así, este trabajo surge como aproximación que demuestra la aplicabilidad de un paradigma software de gran potencia y actualmente en plena ebullición, como es el DSDM, y por otro, la adopción de perspectivas novedosas en el campo tradicional de la robótica. Esta visión sí podría repercutir positivamente causando gran impacto sobre los procesos productivos actuales.
- ✓ Se eleva el nivel de abstracción del desarrollo software, de forma que el usuario sólo ha de ser experto en su ámbito de aplicación obviando los conocimientos técnicos involucrados y ajenos a dicho ámbito.

- ✓ La herramienta de modelado gráfico desarrollada permite de forma amigable y sencilla generar los modelos de movimiento deseados y el obtener el código que ha de ser cargado en el robot.
- ✓ Dado los puntos anteriores, HuRoME podría incidir positivamente en el ámbito educativo por la sencillez y facilidad de avituallamiento, para personas no expertas, de un entorno gráfico basado en la realización de diagramas.
- ✓ Según la experiencia con HuRoME, el DSDM acorta los tiempos de desarrollo del software, permitiendo generar código para distintas plataformas y lenguajes de programación partiendo de los mismos modelos de alto nivel de abstracción, por lo que el desarrollo de las aplicaciones queda simplificado ya que sólo se exponen al diseñador los aspectos relevantes de la aplicación, ocultando los aspectos de implementación de la misma sin dejar de ser lo suficientemente detallado a medida que se baja el nivel de abstracción.
- ✓ La reingeniería de código existente favorece la introducción del paradigma de DSDM en el ámbito dominio específico de aplicación. Favorece el mantenimiento del sistema y su extensibilidad.
- ✓ Destacamos la potencia y cohesión del entorno Eclipse con los plug-ins de DSDM. Eclipse facilita un marco de trabajo en el que realizar todo el proceso de creación de la herramienta de modelado del dominio de interés, usando un entorno visual. Así, el único código, como tal, que ha sido necesario introducir es el relativo a las transformaciones. Tras esto, el mismo framework codifica automáticamente en lenguaje Java todo el trabajo realizado, teniendo finalmente una implementación, lista para usar, del editor gráfico de modelado con un validador de modelos, las respectivas transformaciones y el editor textual para el proceso de reingeniería. Comentar, que gracias a los frameworks de DSDM, las herramientas para el robot Robonova fueron implementadas en unas pocas semanas.
- ✗ Como punto negativo de los plug-ins de DSDM en Eclipse debemos destacar la gran dificultad para hacerse con el dominio de estas herramientas dada la escasa documentación existente, con partes incompletas y bastante confusas. En este sentido, debemos hacer una mención especial a xText, dado que resulta extremadamente sencillo generar un editor textual, pero en contra ha sido costoso en tiempo y esfuerzo encontrar implementar un mecanismo de serialización de los modelos.
- ✗ En la versión actual de HuRoME, no se establece una equivalencia entre los conceptos del meta-modelo y todas las posibles sentencias del lenguaje RoboBASIC, por lo tanto, perdemos flexibilidad de expresión. Por ejemplo, actualmente, no se modelan los sensores u otros elementos más avanzados.
- ✗ En la versión actual de HuRoME todo el flujo de transformaciones es dirigido de forma manual por el usuario a través del manejo directo de los proyectos, con el consiguiente trasiego de ficheros y la necesidad de conocer algunos detalles del entorno Eclipse. Ello dificulta en gran medida la usabilidad de HuRoME auque, tal y como se abordó al final del capítulo 4, existen mecanismos para integrar todas las herramientas desarrolladas.
- ✓ Por último, destacamos las publicaciones [44-46] que han sido realizadas a partir de los resultados del presente proyecto.

6.2. Líneas de Trabajo Futuras

En este apartado se incluyen algunas posibles extensiones y mejoras que consideramos que sería interesante abordar de cara al desarrollo de futuras versiones de HuRoME.

- Definición de librerías de movimientos para aportar el concepto de reutilización y facilitar la tarea de diseño. Esta perspectiva, admitiría por ejemplo, que una vez definido un movimiento de tipo *ComplexMovement*, pueda ser instanciado múltiples veces en distintos modelos sin tener que reeditar en cada momento dicho movimiento.

- Ampliación de la herramienta de modelado para la inclusión de mecanismos de control de flujo, como sentencias condicionales, etc.

- Integración del flujo de transformaciones y las facilidades de HuRoME en un mismo entorno gráfico accesibles, por ejemplo, desde el menú.

- Inclusión del concepto de concurrencia en el diagrama de HuRoME, es decir, poder expresar que dos movimientos se realicen a la vez, por ejemplo, uno para agacharse mientras el otro indica que levante un brazo.

- Motivado por el punto anterior. Simulación de la dinámica y ciertas propiedades de los modelos diseñados. Por ejemplo, si se incluye concurrencia de movimientos, poder verificar que todos los movimientos ejecutados, en un instante, de forma paralela son consistentes y no contradictorios.

- En última instancia, quizás con otras plataformas robóticas, pueden ser introducidos aspectos relativos a ejecución de modelos, implicando el desarrollo de interpretes de modelos específicos para la plataforma considerada.

Bibliografía

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

[2] Schmidt, D., *Model-Driven Engineering*. IEEE Computer. 39(2). 2006.

[3] T. Strasser, M. Rooker, I. Hegny, M.Wenger, A. Zoitl, L. Ferrarini, A. Dede 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 (INDIN'09), June 23-26, 2009, Cardiff, Wales, United Kingdom.

[4] MEDEIA: Model-driven Embedded system Design Environment for the Industrial Automation sector, http://www.medeia.eu/

[5] QUASIMODO: Quantitative system properties in model-driven design of embedded systems, http://www.quasimodo.aau.dk/

[6] C. Vicente-Chicote, F. Losilla, B. Álvarez, A. Iborra, P. Sánchez, *Applying MDE to the Development of Flexible and Reusable Wireless Sensor Networks*, International Journal of Cooperative Information Systems, Special Issue: *Software Architecture — Towards the Software Engineering Core*, Vol. 16, No. 3/4, pp. 393 - 412, 2007.

[7] BRICS: Best Practice in Robotics, http://www.best-of-robotics.org/

[8] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, B. Álvarez, *V3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development,* Journal of Software Engineering for Robotics, Vol 1, No 1 (2010).

[9] Bruyninckx, H. (2008). *Robotics software: the future should be open*. IEEE Robotics & Automation Magazine, 15(1), 9-11.

[10] Robonova 1, HITEC Robotics, http://www.robonova.com/

[11] Eclipse Modeling Project, http://www.eclipse.org/modeling/

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

[13] The Eclipse Graphical Modeling Framework (GMF), http://www.eclipse.org/modeling/gmf/

[14] D. Grunberg, R. Ellenberg, T. Kim, P. Oh, *Creating an Autonomous Dancing Robot*, Proceedings of the 2009 International Conference on Hybrid Information Technology, 2009, pp. 221-227, Daejeon, korea.

[15] Robotic Intelligence Laboratory, Universidad Jaume I, http://www.robot.uji.es/lab/plone/

[16] RoboBASIC Command Instruction Manual v2.10, http://www.hitecrobotics.com

[17] Datasheet Atmel ATmega128, http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

[18] SDK Robonova, http://www.robobasic.com

[19] Grady Booch, James Rumbaugh, Ivan Jacobson. *Unified Modeling Language User Guide*. 2nd Edition. Addison-Wesley, 2005.

[20] Jean Bézivin, Olivier Gerbé. *Towards a Precise Definition of the OMG/MDA Framework*. En Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE'01), Noviembre 2001.

[21] Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise. *MDA Distilled. Principles of Model-Driven Architecture*. Addison-Wesley, 2004.

[22] OMG. *Model Driven Architecture.* Document ormsc/2001-07-01, Julio 2001. Accesible desde www.omg.org/mda

[23] DSL Tools. http://msdn.microsoft.com/en-us/library/bb126235.aspx

[24] Meta Edit. http://www.metacase.com/

[25] Eclipse. http://www.eclipse.org/

[26] OMG. *MDA Guide Version 1.0.1. Document omg/03-06-01*, Junio 2003. Accesible desde http://www.omg.org/docs/omg/03-06-01.pdf

[27] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*, versión 2.0. *Document formal/06-01-01*, Enero 2006. Accesible desde http://www.omg.org/docs/formal/06-01-01.pdf

[28] OMG. "Unified Modeling Language: Superstructure", versión 2.1.2. *Document formal/2007-11-02*, Noviembre 2007. Accesible desde http://www.omg.org/docs/formal/07-11-02.pdf

[29] Alan W. Brown. "Model driven architecture: Principles and practice." En *Journal of Software and System Modeling (SoSym)*, Vol. 3, No. 4, pp. 314-327, Diciembre 2004.

[30] Ian Graham, Brian Henderson-Sellers, Houman Younessi. *The OPEN Process Specification*. Addison Wesley, 1997.

[31] Emfatic, www.alphaworks.ibm.com/tech/emfatic

[32] TOPCASED, www.topcased.org

[33] C. Vicente-Chicote,D. Alonso, *Herramientas Eclipse para el Desarrollo de Software Dirigido por Modelos*, XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2009), 8-11 September 2009, San Sebastián (Spain).

[34] ANTLR, ANother Tool for Language Recognition, http://www.antlr.org/

[35] xText, http://www.eclipse.org/Xtext/

[36] Eclipse Modeling Framework (EMF), www.eclipse.org/emf/

[37] ATLAS Transformation Language (ATL), http://www.eclipse.org/atl/

[38] SmartQVT, http://smartqvt.elibel.tm.fr/

[39] Xpand, http://www.eclipse.org/modeling/m2t/?project=xpand

[40] MediniQVT, http://projects.ikv.de/qvt/

[41] QVT Operational Mapping Language (QVTo), www.eclipse.org/m2m/qvto/doc

[42] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, *EMF: Eclipse Modeling Framework.* Addison-Wesley Professional, 2nd Edition, 2009, ISBN: 0-321-33188-5.

[43] XML Path Language (XPath), W3C Recommendation 16 November 1999, http://www.w3.org/TR/xpath/

[44] J.F. Inglés-Romero, C. Vicente-Chicote, D. Alonso, *Entorno para Coreografiar Movimientos en un Robot Humanoide*, III Jornadas de Jóvenes Investigadores de la Universidad Politécnica de Cartagena, Cartagena (Spain), May 2010. ISSN 1888 8356.

[45] J.F. Inglés-Romero, C. Vicente-Chicote, D. Alonso, *Desarrollo de Software Aplicado a la Robótica: Un Caso Práctico*, I Jornadas de Jóvenes Investigadores de la Universidad de Extremadura, Cáceres (Spain), April 2010. ISBN 978-84-693-1707-5.

[46] J.F. Inglés-Romero, C. Vicente-Chicote, D. Alonso, *HuRoME: Entorno para Modelado de Coreografías y Modernización de Código para un Robot Humanoide*, XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD'10), Valencia (Spain), September 2010. ISBN 978-84-92812-51-6.