UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Departamento de Tecnologías de la Información y las
Comunicaciones

**DESARROLLO DE APLICACIONES PARA REDES DE SENSORES
INALÁMBRICAS: UN ENFOQUE DIRIGIDO POR MODELOS**

Autor

D. Fernando Losilla López
Ingeniero en Telecomunicación

Directores

Dr. D. Pedro Sánchez Palma
Dra. D^a. Bárbara Álvarez Torres

2009

Desarrollo de Aplicaciones para Redes de Sensores Inalámbricas: Un Enfoque Dirigido por Modelos

Autor: Fernando Losilla López

Directores: Pedro Sánchez Palma
Bárbara Álvarez Torres

Contacto:

Copyright 2009 © Fernando Losilla López

Agradecimientos

En primer lugar quiero agradecer a mis directores, Pedro Sánchez Palma y Bárbara Álvarez Torres, su esfuerzo y paciencia conmigo. Sin ellos, este trabajo no sería lo que es.

Al grupo de investigación División de Sistemas e Ingeniería Informática (DSIE), con cuyos miembros he investigado durante estos años y que siempre han estado dispuestos a echarme una mano cuando lo he necesitado. A Andrés Iborra le agradezco el haberme dado la oportunidad de integrarme en este grupo. Además quiero reconocer la ayuda prestada por Cristina Vicente y Diego Alonso, de los cuales he podido aprender mucho, así como dar las gracias también a Manuel Jiménez y a Francisco Ortiz.

A los miembros del Departamento de Tecnologías de la Información y las Comunicaciones con quienes tengo el placer de trabajar. La relación como profesor que tengo con este departamento me ha permitido poder dedicarme a esta Tesis. Muy en especial se lo tengo que agradecer al Grupo de Ingeniería Telemática (GIT) y a Joan García por pensar en mí para formar parte de este grupo.

No puedo olvidarme tampoco de que si he llegado hasta aquí se lo debo en gran medida al Grupo de Electromagnetismo Aplicado a las Comunicaciones (GEAT), concretamente a Fernando Quesada y a Alejandro Álvarez. El haber pertenecido a este grupo me ha permitido llegar a formar parte de esta Universidad.

Finalmente me gustaría agradecer a mi familia y amigos el apoyo que durante todo este tiempo he recibido de ellos.

A mis padres, que tanto me han dado

Resumen

El desarrollo de software para redes de sensores inalámbricas implica que las personas que lo lleven a cabo dispongan de unos determinados conocimientos sobre esta tecnología. Como consecuencia, se restringe el conjunto de desarrolladores potenciales a sólo los programadores que se hayan formado adecuadamente para este fin. Esto impide que las personas que hacen uso de las redes de sensores como medio auxiliar para la realización de estudios científicos o relacionados con la productividad de un proceso (éstos suelen tener unos conocimientos más enfocados hacia áreas como por ejemplo la biología, la geología o la agricultura) puedan desarrollar sus propias aplicaciones software. La propuesta que esta Tesis hace al respecto consiste en un nuevo método de desarrollo de software para redes de sensores, basado en el Desarrollo de Software Dirigido por Modelos (DSDM), el cual permite la descripción de aplicaciones mediante conceptos sencillos y la generación de código ejecutable a partir de las descripciones. Gracias a esto, para el desarrollo de software para redes de sensores, no sólo se elimina la necesidad de dar formación adicional al personal implicado sino que además se simplifica considerablemente su elaboración.

El DSDM, también conocido como MDE (*Model Driven Engineerieng*), se basa en el uso de modelos como pieza fundamental de todo el proceso de desarrollo de software. Haciendo uso de él, la descripción de una aplicación realizada a alto nivel de abstracción genera un modelo que se va transformando sucesivamente a modelos con un nivel de abstracción menor. Cuando finalmente se obtiene un modelo con un nivel de abstracción similar al de una plataforma de ejecución concreta (por ejemplo un sistema operativo con su propio lenguaje de programación) se ejecuta una nueva transformación que, a partir de cada uno de los elementos del modelo, obtiene código ejecutable por la plataforma.

En esta Tesis se ha implementado una herramienta que lleva a cabo este enfoque. Cuenta con un lenguaje gráfico que permite modelar aplicaciones basadas en redes de sensores, así como con niveles intermedios a los que automáticamente se transforman las descripciones de las aplicaciones. El código se genera para el sistema operativo TinyOS haciendo uso de su lenguaje de programación nesC, para el que se pueden obtener aplicaciones que permiten desde la monitorización de datos (y respuesta ante determinadas situaciones) hasta otras como por ejemplo seguimiento de objetos.

Abstract

Developing software for wireless sensor networks is a task that requires certain technical knowledge about this technology. As a consequence, the potential pool of developers is constrained to those programmers who have the proper formation. This fact prevents people making use of sensor networks as an auxiliary tool for their scientific or production-related studies (who typically have a formation focused on areas such as biology, geology or agriculture) from being able to develop their own software applications. To this regard, the proposal of this PhD Thesis consists in a new software development method for wireless sensor networks, based on MDE (*Model Driven Engineering*), which enables the description of applications by using simple concepts and the generation of executable code from these descriptions. With this approach, there is no need of acquiring additional technical skills and, also, the development of applications is considerably simplified.

Model Driven Engineering is based on the use of models as the key artefact in the whole software development process. By using MDE, the high-level description of an application generates a model which is successively transformed to models with a lower abstraction level. When a model with an abstraction level equal to a concrete execution platform (for example an operating system with its own programming language) is finally achieved, a new transformation is run which, for each of the elements in this last model, obtains executable code for the target platform.

Among the results of this PhD Thesis, a tool following this approach for sensor networks software development has been implemented. This tool disposes of a graphical language for describing sensor network based applications, as also of several intermediate levels of models to which the descriptions are automatically transformed. It generates code for the TinyOS operating system in the form of programs written with its programming language nesC. These programs may perform tasks varying from data acquisition and monitoring (as also the execution of responses to certain situations) to some others such as object tracking.

Indice de figuras.....	<i>i</i>
1. Introducción.....	<i>1</i>
1.1. Motivación	2
1.2. Marco de desarrollo de la Tesis	4
1.3. Objetivos	4
1.4. Estructura de la Tesis	6
2. Redes de Sensores Inalámbricas.....	<i>9</i>
2.1. Introducción a las Redes de Sensores Inalámbricas	10
2.1.1. Nodos sensores: Motes.....	12
2.1.2. Campos de aplicación	14
2.1.3. Objetivos de diseño de una WSN.....	16
2.1.4. Evolución de las redes de sensores	18
2.2. Desarrollo de software para redes de sensores	18
2.2.1. TinyOS.....	20
2.2.1.1. Lenguaje nesC.....	21
2.2.2. Otros sistemas operativos para WSN	22
2.2.3. Deficiencias del proceso tradicional de desarrollo de software.	23
2.2.4. Otras alternativas.....	24
2.3. Conclusiones y aportación a la Tesis	24
3. Desarrollo de Software Dirigido por Modelos.....	<i>25</i>
3.1. Motivación	26
3.2. Modelado de sistemas	27
3.2.1. Modelos como mecanismo de abstracción.....	27
3.2.2. Los metamodelos en el desarrollo de software	29
3.3. Model Driven Engineering	31
3.3.1. MDA	32
3.3.1.1. Estándares MDA.....	36
3.3.2. Modelos y transformaciones en MDE.....	38
3.3.2.1. Modelos	38
3.3.2.2. Transformaciones.....	40
3.3.3. Aseguramiento de la calidad (<i>Quality Assurance</i>)	45
3.3.4. Aproximaciones a MDE.....	45
3.3.4.1. Domain Specific Modeling	47
3.3.4.2. Model Integrated Computing	47
3.3.4.3. Software Factories	48
3.3.5. Entornos de desarrollo	48
3.3.5.1. Eclipse Modeling Framework.....	49
3.3.6. Aplicación de MDE a sistemas reactivos	50
3.3.6.1. Aplicación a las redes de sensores	52
3.4. Conclusiones y aportación a la Tesis	53
4. Definición de un Lenguaje Específico de Dominio	<i>55</i>
4.1. Lenguajes Específicos de Dominio (DSL)	56
4.1.1. Ventajas e inconvenientes de los DSL	58
4.1.2. Terminología y elementos de un DSL.....	59
4.1.3. Clasificación de los DSL.....	60
4.1.4. Definición de un DSL	63

4.2.	Lenguaje Específico de Dominio para WSN	67
4.2.1.	Tecnología empleada	69
4.2.2.	Sintaxis abstracta	72
4.2.3.	Sintaxis concreta	82
4.3.	Conclusiones y aportación a la Tesis	84
5.	<i>Desarrollo de una herramienta MDE para la construcción de aplicaciones WSN</i>	85
5.1.	Introducción	86
5.2.	Modelos empleados	89
5.2.1.	Nivel PIM – WSN	89
5.2.2.	Nivel PIM – V3-CM	91
5.2.3.	Nivel PSM – nesC	93
5.3.	Transformaciones entre modelos	95
5.3.1.	Transformación PIM WSN – V3-CM	101
5.3.2.	Transformación V3-CM a nesC	105
5.3.3.	Ejemplo de transformaciones M2M	107
5.4.	Transformación a código	109
5.5.	Conclusiones y aportación a la Tesis	111
6.	<i>Uso de la herramienta MoteGen</i>	113
6.1.	Requisitos de la herramienta	114
6.2.	Uso de la herramienta	114
6.3.	Consideraciones de uso	117
6.4.	Metodología de desarrollo	118
6.5.	Ejemplo práctico	120
6.5.1.	Código generado	123
6.6.	Conclusiones y aportación a la Tesis	130
7.	<i>Conclusiones y Líneas Futuras</i>	131
7.1.	Conclusiones	132
7.2.	Aportaciones	133
7.3.	Divulgación de resultados	135
7.4.	Trabajos futuros	137
	<i>Bibliografía</i>	139
	<i>Anexo I. Instalación de la herramienta MoteGen</i>	147
	<i>Anexo II. Generación de código para redes de sensores mediante MetaEdit+</i>	149
	<i>Anexo III. Transformación M2M a PIM V3-CM</i>	155
	<i>Anexo IV. Transformación M2M a PSM-nesC</i>	193
	<i>Anexo V. Transformación M2T a código</i>	219

Indice de figuras

Figura 2-1. Despliegue típico de una WSN	11
Figura 2-2. Componentes de un mote	12
Figura 2-3. Familia de motes MICA	13
Figura 2-4. Mote telos	13
Figura 2-5. Nodo móvil para monitorización de pacientes en un hospital	15
Figura 2-6. Interconexión de componentes en nesC	22
Figura 3-1. Descripción de una aplicación con herramienta CASE (<i>Rhapsody</i>)	26
Figura 3-2. Uso histórico de los modelos: mapas	28
Figura 3-3. Esto no es una pipa	29
Figura 3-4. Modelo de un modelo	29
Figura 3-5. Relaciones entre sistema, modelo y metamodelo en un mapa	30
Figura 3-6. Relaciones entre (meta)modelos y (meta)lenguajes de modelado	31
Figura 3-7. Transformación PIM – PSM	34
Figura 3-8. Relatividad de roles en las transformaciones MDA	35
Figura 3-9. Ejemplo de uso de la arquitectura de cuatro capas de MDA	35
Figura 3-10. Niveles de abstracción de MDA en la arquitectura de cuatro capas	36
Figura 3-11. Transformación entre modelos	41
Figura 3-12. Participantes y flujo en una aproximación MDE genérica	47
Figura 3-13. Modelo de iluminación con el DSL para domótica	51
Figura 3-14. V3-CM. Control de un robot	51
Figura 3-15. Herramienta GRATIS II para redes de sensores	52
Figura 3-16. Modelo del entorno de ejecución de dos aplicaciones con ScatterFactory	253
Figura 4-1. DSL basado en conceptos del experto del dominio	62
Figura 4-2. Lenguaje basado en la generación de la salida	63
Figura 4-3. Lenguaje basado en el aspecto del sistema	64
Figura 4-4. Aplicación de ejemplo desarrollada con el DSL para WSN	68
Figura 4-5. Proceso de desarrollo con GMF	71
Figura 4-6. Metamodelo del DSL	78
Figura 5-1. Propuesta de desarrollo WSAN	87

Figura 5-2. Uso de V3-CM como nivel intermedio	88
Figura 5-3. Metamodelo PIM-WSN	90
Figura 5-4. Metamodelo V3-CM	92
Figura 5-5. Metamodelo nesC	94
Figura 5-6. Código para los enlaces de datos de tipo <i>SameVar</i>	96
Figura 5-7. Código para los enlaces de datos de tipo <i>Copy</i>	96
Figura 5-8. Uso de marcas en las transformaciones MDA	98
Figura 5-9. Uso de las marcas en las transformaciones de la herramienta <i>MoteGen</i>	99
Figura 5-10. Regla AGG	101
Figura 5-11. Arquitectura V3-CM generada para cada grupo de nodos	102
Figura 5-12. Ejemplo de transformaciones M2M en la herramienta <i>MoteGen</i>	108
Figura 5-13. Fragmento del lenguaje MOFScript	109
Figura 6-1. Áreas de la herramienta <i>MoteGen</i>	115
Figura 6-2. Validación de una aplicación	116
Figura 6-3. Aplicación para localización de personas descrita con la herramienta <i>MoteGen</i>	122
Figura 6-4 Archivo makefile para los nodos móviles	123
Figura 6-5. Configuración de primer nivel de los nodos móviles	124
Figura 6-6. Configuración de segundo nivel de los nodos móviles	124
Figura 6-7. Coordinador de los nodos móviles	127
Figura 6-8. makefile para los nodos beacon	127
Figura 6-9. Configuración de primer nivel para los nodos beacon	127
Figura 6-10. Configuración de segundo nivel para los nodos beacon	128
Figura 6-11. Controlador de los nodos beacon	129

Capítulo 1

Introducción

Este capítulo describe en primer lugar la motivación que ha llevado al desarrollo de este trabajo de Tesis incluyendo una breve descripción de las tecnologías emergentes que están implicadas en él. A continuación se enumeran los objetivos marcados para la realización de la Tesis y finalmente se presenta la estructura del resto de esta memoria.

1.1. Motivación

Las redes de sensores inalámbricas (*Wireless Sensor Networks*, WSN) constituyen una tecnología de adquisición de datos que recientemente está atrayendo gran interés gracias a sus posibilidades, siendo aplicadas en numerosos ámbitos científicos e industriales para la realización de estudios y control de procesos. El uso de comunicaciones inalámbricas permite que los dispositivos sensores que las forman sean emplazados fácilmente sobre el terreno. Esto, unido al bajo coste de cada uno de ellos, permite el despliegue de un gran número de dispositivos que actúan como nodos de la red.

A raíz de su bajo coste, los dispositivos que forman la red cuentan con recursos limitados. Como consecuencia, los programas que ejecutan son sencillos y están sujetos a restricciones de acceso a los recursos hardware de los nodos, que en gran medida son manejadas por sistemas operativos específicos para redes de sensores como TinyOS. Sin embargo, a pesar del uso de éstos, el desarrollo de software para redes de sensores no es un proceso sencillo. En primer lugar requiere que los desarrolladores sean programadores con unos determinados conocimientos y habilidades que lleva un tiempo adquirir. Esto impide que las personas principalmente interesadas en usar las redes de sensores, que suelen tener una formación especializada en áreas como la biología, geología o la agricultura, sean capaces de desarrollar y mantener las aplicaciones WSN que les permiten el estudio del entorno. En segundo lugar, aún teniendo estos conocimientos, el desarrollo de aplicaciones para redes de sensores y su depuración conlleva cierto tiempo.

A pesar de que se han producido muchos avances en las distintas áreas tecnológicas relacionadas con las redes de sensores, pocos de ellos se han centrado en la simplificación del proceso de desarrollo. Lo más cercano a esto que se ha hecho ha sido la creación de nuevas tecnologías middleware que facilitan la construcción de aplicaciones complejas, pero que aún requieren del uso de lenguajes textuales que no resultan fáciles de usar.

El uso de lenguajes gráficos e intuitivos sería de gran ayuda a este respecto, reduciendo no sólo el tiempo requerido para desarrollar una aplicación sino también el empleado en aprender la tecnología que permite este desarrollo. Su empleo no es algo nuevo. Ya en la década de los ochenta aparecieron las primeras herramientas CASE (*Computer-Aided Software Engineering*) destinadas a aumentar la productividad en el desarrollo de software, las cuales, en algunos casos, podían llegar a generar parte del código de la implementación a partir de un diseño. Pero estas herramientas no llegaron a alcanzar el éxito esperado ya que hacían uso de conceptos demasiado genéricos para representar aplicaciones en dominios concretos (por ejemplo medicina, automatización, militar, etc.) y las tecnologías subyacentes no estaban lo suficientemente avanzadas como para generar software de calidad a partir de las descripciones proporcionadas.

Sin embargo, la relativamente reciente aparición del Desarrollo de Software Dirigido por Modelos (DSDM), conocido también como MDE (*Model Driven Engineering*) [Kent 02], proporciona tecnologías y técnicas que posibilitan la construcción automatizada de aplicaciones a partir de descripciones gráficas. Este enfoque se caracteriza principalmente porque los modelos son el principal artefacto usado para el desarrollo de software y no quedan limitados a fines de documentación. Haciendo uso de MDE una aplicación puede ser descrita mediante un lenguaje gráfico de modelado y tras una serie de transformaciones sucesivas generar código ejecutable. Si además este lenguaje está formado por conceptos relativos a un dominio de aplicación concreto, en lugar de basarse en conceptos genéricos válidos para un mayor número de dominios, la productividad aumenta al ser más sencilla la construcción de aplicaciones. En este caso el lenguaje se conoce como lenguaje específico de dominio (*Domain Specific Language, DSL*).

Adicionalmente, el uso de un enfoque MDE ofrece otras ventajas, pues no sólo permite la generación de código ejecutable sino también la generación de otras estructuras de datos. Así, es posible obtener documentación de las aplicaciones y casos de uso a partir de sus descripciones que permitan probarlas. Además se pueden ejecutar otros tipos de actividades sobre las aplicaciones, por ejemplo comprobar que están bien formadas o que cumplen determinadas propiedades, o bien realizar simulaciones (visuales) de las aplicaciones mediante la animación de sus modelos.

La generación de código con un proceso de desarrollo de este tipo generalmente implica el uso de modelos situados en diferentes niveles de abstracción entre los que se ejecutan las transformaciones. Esto permite que las aplicaciones se describan de forma sencilla mediante modelos con un elevado nivel de abstracción y se conviertan a modelos menos abstractos y más relacionados con una tecnología de implementación más compleja. Este principio es el que sigue el estándar MDA (*Model Driven Architecture*) del OMG (*Object Management Group*), que se encuadra dentro de las tecnologías MDE, ofreciendo una serie de niveles de abstracción y de estándares para la realización del proceso de generación de código.

Las redes de sensores inalámbricas se pueden beneficiar del uso de MDE y en concreto de MDA, permitiendo la construcción de herramientas destinadas a la especificación y generación de aplicaciones WSN. Sin embargo no existen iniciativas MDE que hayan logrado esto plenamente. Las más destacables se limitan a la generación de esqueletos de código que representan la arquitectura de las aplicaciones modeladas [Volgyesi 02][Cheong 05][Saad 08], pero en ninguno de los casos permiten modelar el comportamiento de las aplicaciones.

Por este motivo se plantea esta Tesis, con el objetivo de aplicar estas técnicas de la Ingeniería del Software al desarrollo de aplicaciones software para redes de sensores inalámbricas.

1.2. Marco de desarrollo de la Tesis

La Tesis que aquí se expone surge por el interés común de los grupos de investigación de la Universidad Politécnica de Cartagena DSIE (*División de Sistemas e Ingeniería Electrónica*) y GIT (*Grupo de Ingeniería Telemática*) por las redes de sensores inalámbricas, ya que el autor se encuentra estrechamente relacionado con ambos grupos. Concretamente se ha desarrollado como un trabajo de investigación del grupo DSIE, el cual tiene como una línea de investigación la aplicación de MDE a diversos sistemas reactivos que incluyen dominios como la visión artificial, la domótica y las redes de sensores y la robótica.

El primero de los trabajos desarrollados dentro de esta línea de investigación [Alonso 08] propuso la utilización de MDE para el dominio de los robots de servicio. Asimismo, como resultado de este trabajo se obtuvo un lenguaje de modelado orientado a componentes (V3-CM) que ha sido utilizado en esta Tesis para la descripción de modelos en un nivel intermedio de abstracción. Este lenguaje está siendo utilizado también en las aproximaciones MDE que se están realizando para el resto de dominios de aplicaciones reactivas, por lo que sirve como punto de confluencia entre todos ellos y posibilita que en un futuro se puedan integrar los artefactos desarrollados para los distintos dominios.

El trabajo de Tesis que se describe en esta memoria ha sido realizado en el marco de los proyectos MEDWSA (*Marco conceptual y tecnológico para el desarrollo de software de sistemas reactivos*) de la Comisión Interministerial de Ciencia y Tecnología (CICYT TIN2006-15175-C05-02), perteneciente al proyecto coordinado META (*Models, Environments, Transformations and Applications*) y CON-PARTE-1 (*CONtribución a los nuevos PARadigmas y TEcnologías de red para las comunicaciones del mañana*, TEC2007-67966-C03-01) perteneciente al proyecto coordinado CON_PARTE. Dentro del primero de estos proyectos se encuadran todas las iniciativas MDE que el DSIE está aplicando a los diversos dominios de sistemas reactivos.

1.3. Objetivos

El objetivo principal de este trabajo es pues la aplicación de MDE a las redes de sensores inalámbricas, adaptando el enfoque MDA a este dominio y desarrollando una herramienta que dé soporte a todo el proceso de generación automática de código. Como plataforma de ejecución se ha seleccionado TinyOS [Hill 00] para la que se generará código en su lenguaje de programación orientado a componentes nesC. Para la consecución de este objetivo se propone una serie de tareas que deben llevarse a cabo:

- Realizar un estudio de las redes de sensores que permitirá conocer las diferentes alternativas tecnológicas existentes en ellas así como los principales conceptos que condicionan su funcionamiento.

- Estudiar las técnicas y herramientas de desarrollo MDE, en particular MDA. Este estudio servirá para seleccionar las tecnologías empleadas en el desarrollo de una herramienta como entornos de desarrollo, lenguajes y demás estándares utilizados.
- Definir un lenguaje específico de dominio (*Domain Specific Language, DSL*) para redes de sensores basado en la representación gráfica de sus principales conceptos. Este lenguaje debe ser más simple de usar y aprender que los lenguajes textuales actualmente usados para desarrollar software para este dominio de aplicaciones.
- Establecer un proceso de generación de código acorde al estándar MDA, adaptando sus conceptos y estableciendo una serie de niveles de abstracción para los modelos que se adecúe a las necesidades de la herramienta que se pretende desarrollar:
 - Las aplicaciones deben construirse con el DSL anterior y sus descripciones serán el punto de partida para el desarrollo. Por tanto, este DSL dará lugar a los modelos de mayor nivel de abstracción usados por MDA.
 - Se deben tener modelos con un nivel intermedio de abstracción contruidos con el lenguaje de modelado orientado a componentes V3-CM desarrollado en el DSIE. Esto permitirá la integración entre distintos dominios de aplicaciones reactivas.
 - Se tendrá un último nivel de abstracción con modelos que correspondientes a la plataforma de ejecución nesC que representarán los conceptos de este lenguaje.
 - Se deben establecer correspondencias entre los distintos niveles de abstracción que permitan transformar automáticamente sus modelos. Igualmente se deben encontrar mecanismos para llevar a cabo estas transformaciones.
 - A partir de los modelos correspondientes a la plataforma de ejecución se debe generar código que pueda ejecutarse. Por lo tanto, se deben establecer correspondencias entre los elementos de estos modelos y estructuras de código nesC.
- Construir una herramienta que soporte la construcción del proceso anterior de generación de código, implementando cada uno de sus niveles y transformaciones.
- Elaborar una metodología de desarrollo que permita, de forma guiada, la construcción de aplicaciones para redes de sensores inalámbricas.

- Finalmente, se debe validar tanto el enfoque como la herramienta desarrollados mediante la construcción de una aplicación práctica con la misma y la comprobación de su funcionamiento sobre la plataforma de ejecución seleccionada.

1.4. Estructura de la Tesis

Los capítulos y anexos de esta memoria de Tesis se organizan de la siguiente manera:

Capítulo 1. Introducción

En este capítulo se introduce la motivación que ha llevado al desarrollo de esta Tesis así como los objetivos que se pretenden cumplir durante su desarrollo.

Capítulo 2. Redes de Sensores Inalámbricas

En este capítulo se da una introducción a las redes de sensores, para las que el desarrollo de aplicaciones constituye el objetivo de la Tesis. En primer lugar se da una introducción a las características y elementos típicos más destacados de una red de sensores para después centrarse en el desarrollo de software para estas redes.

Capítulo 3. Desarrollo de software basado en modelos

Este capítulo cubre el desarrollo de software dirigido por modelos (MDE) que es la pieza central en esta Tesis. Para su entendimiento, en primer lugar, se exponen los conceptos básicos del modelado de sistemas software y posteriormente se explican los principios sobre los que se sustenta MDE, incluyendo tecnologías y aproximaciones típicamente empleadas en este enfoque. De entre las aproximaciones expuestas destaca MDA a la que se le da especial atención. Finalmente, se aborda la aplicación de MDE a sistemas reactivos y más específicamente a las redes de sensores.

Capítulo 4. Definición de un lenguaje específico de dominio

En este capítulo se define un lenguaje específico de dominio para redes de sensores. Un lenguaje de este tipo es fundamental en un enfoque de desarrollo basado en el uso de modelos pues permite elevar el nivel de abstracción con el que se describen las aplicaciones. A lo largo del capítulo se describen estos lenguajes, incidiendo en sus ventajas e inconvenientes, clasificaciones, así como el proceso que se sigue para su definición. Posteriormente se explica el lenguaje desarrollado para el dominio de las redes de sensores incluyendo detalles sobre su implementación.

Capítulo 5. Desarrollo de una herramienta MDE para la construcción de aplicaciones WSN

Este capítulo detalla la propuesta que se ha elaborado para llevar a cabo el proceso de transformación desde las descripciones realizadas con el DSL a código ejecutable y que ha sido implementado por la herramienta desarrollada. Esta propuesta se basa en MDA y en el uso de modelos en tres niveles de abstracción. A lo largo del capítulo se detalla cómo se han creado los distintos niveles, exponiendo sus correspondientes metamodelos, y cómo se han realizado las transformaciones entre ellos, comentando además las estrategias y decisiones que han sido tomadas en la elaboración y aplicación de éstas.

Capítulo 6. Uso de la herramienta *MoteGen*

En este capítulo se presenta la herramienta que se ha desarrollado incidiendo en aspectos relacionados con su utilización como las funciones que ofrece o las consideraciones de uso. Además, se propone una metodología para el desarrollo de aplicaciones con la herramienta y finalmente se desarrolla una aplicación práctica con ella para mostrar su utilidad.

Capítulo 7. Conclusiones y Trabajos Futuros

En este capítulo se presentan las conclusiones obtenidas tras la realización de la Tesis así como los resultados que, hasta el momento, se han obtenido de ella. A continuación se enumeran las principales líneas de investigación que se abren a raíz de la realización de este trabajo y que permitirán su mejora y ampliación.

Anexo I. Instalación de la herramienta *MoteGen*

Este anexo detalla la instalación de la herramienta desarrollada. Esto implica en primer lugar la instalación de todos los *plug-ins* que la herramienta necesita. Para ello se dan ubicaciones desde las cuales el entorno de desarrollo *Eclipse* puede obtenerlas automáticamente. Posteriormente se ofrecen instrucciones para instalar cada uno de los proyectos de los que se compone la herramienta y las librerías de componentes usadas.

Anexo II. Generación de código para WSN mediante *MetaEdit+*

En este anexo se comenta una aproximación MDE alternativa a la propuesta en esta Tesis para el desarrollo de software para redes de sensores. Para ello se emplea la herramienta *MetaEdit+* para la que se han creado un DSL y un script de transformación que permiten la generación de aplicaciones. Este enfoque, aunque es más sencillo que el empleado en esta Tesis, no ofrece todas las posibilidades que la herramienta *MoteGen* ofrece.

Anexo III. Transformación M2M a PIM V3-CM

Este anexo recoge las reglas de transformación, definidas con el lenguaje de grafos AGG, que se emplean en la transformación desde los modelos descritos con el DSL para redes de sensores a modelos V3-CM, el lenguaje de modelado de propósito general definido en el DSIE.

Anexo IV. Transformación M2M a PSM-nesC

Este anexo recoge las reglas de transformación empleadas en la transformación desde los modelos descritos con V3-CM a modelos PSM para nesC. Al igual que las reglas que transforman a modelos PIM correspondientes a V3-CM también se han implementado con AGG.

Anexo V. Transformación M2T a código

Este anexo contiene el script que genera código nesC desde los modelos PSM-nesC. Para su definición se ha empleado el lenguaje de transformación *MOFScript*.

Capítulo 2

Redes de Sensores Inalámbricas

Las redes de sensores constituyen una tecnología de adquisición de datos escalable y de bajo coste. Éste capítulo ofrece una visión global de esta tecnología para la cual el desarrollo de software es el propósito fundamental de esta Tesis.

En primer lugar se describen las características básicas de las redes de sensores que permiten su comprensión y que condicionarán la programación de sus aplicaciones. Posteriormente se aborda el desarrollo de software para esta tecnología. Éste típicamente se realiza con el sistema operativo TinyOS y su lenguaje de programación nesC por lo que se incide especialmente en ellos, que además son los usados a lo largo de esta Tesis como plataforma de ejecución.

2.1. Introducción a las Redes de Sensores Inalámbricas

La recolección de datos es una tarea indispensable en un gran número de aplicaciones. Ejemplos son las aplicaciones ambientales para el estudio de las condiciones de un terreno o las militares para control de equipamiento o tropas. En el pasado, tanto operadores humanos como equipos sensores unidos a una central mediante cableado realizaban la recogida de datos. Las dos opciones presentaban problemas: las mediciones realizadas por operadores humanos eran costosas en tiempo, lo que restringía la cantidad de ubicaciones en las que tomar medidas. Por otro lado, el uso de sensores cableados obligaba al despliegue de grandes cantidades de cable e impedía la toma de medidas en lugares de difícil acceso. Con los avances realizados en los campos de la electrónica y las comunicaciones inalámbricas se pudieron solventar estos problemas, abaratando el uso de dispositivos inalámbricos para adquisición de datos, lo que ha permitido el auge de las redes de sensores inalámbricas (*Wireless Sensor Network*, WSN). Esta tecnología, reconocida en un informe del MIT como una de las diez tecnologías emergentes que cambiarán el mundo [Huang 03], constituye una elección muy oportuna para los casos en los que se deben recoger datos de múltiples ubicaciones. Su uso permite el despliegue de un gran número de dispositivos sensores de bajo coste que forman una red inalámbrica robusta, escalable y adaptable a los cambios en el entorno o en su topología.

Los dispositivos sensores, normalmente emplazados manualmente en el terreno, constituyen los nodos de una red encargada de leer magnitudes como temperatura, humedad, presión, velocidad, luminosidad, ruido, presencia de objetos, compuestos químicos, etc. Estos datos, tras ser procesados si es necesario, son transmitidos por medio de comunicaciones inalámbricas hasta un observador externo a la red que se encargará de su análisis como se puede observar en la Figura 2-1. En ella se muestra un despliegue típico de una red de sensores donde una serie de nodos toman mediciones del terreno y las mandan a uno o varios nodos frontera, denominados sumideros, que se encargan de extraer la información de la red y retransmitirla hacia el exterior, normalmente a un computador.

Aunque ya se disponía de sensores con capacidad de comunicación inalámbrica desde hace algunas décadas, ha sido en los últimos años cuando se han reducido substancialmente los costes y consumos de potencia de éstos gracias a los avances realizados en distintas áreas tecnológicas. En las WSN se aplican estos avances, pero además los dispositivos que las forman hacen uso de enlaces inalámbricos de corta distancia para reducir aún más el consumo de potencia. A consecuencia de esto, normalmente un nodo sensor no es capaz de alcanzar al nodo sumidero por encontrarse a una distancia mayor que el radio de acción de su subsistema de comunicaciones. Por lo tanto, los nodos sensores deben organizarse en una red de comunicaciones que permita que la información fluya desde cualquier nodo hacia el nodo sumidero pasando por una serie de nodos intermedios.

Figura 2-1. Despliegue típico de una WSN

Además de las directamente relacionadas con el uso de las comunicaciones inalámbricas, las principales ventajas de las redes de sensores en comparación con otras tecnologías son el bajo coste económico de los dispositivos que las forman y su pequeño tamaño. Sin embargo, como contrapartida, estos dispositivos tienen unas prestaciones limitadas, lo que condiciona en gran medida el software que se diseña para este tipo de redes. En primer lugar, no disponen de una elevada capacidad de procesamiento, por lo que se limitan a ejecutar algoritmos computacionalmente sencillos y con un bajo uso de memoria RAM. Pero la principal consideración que imponen al desarrollar software para esta tecnología es el ahorro de energía. Esto se debe a que tras desplegar los nodos probablemente no se vuelva a tener acceso físico a ellos y no se podrá reponer sus baterías. Bajo esta situación, la capacidad que tengan los dispositivos para economizar el uso de éstas resultará muy importante, pues el tiempo de la vida de la WSN vendrá determinado por el tiempo que cada uno de sus nodos tarde en agotar sus recursos energéticos. Las medidas que se tomen para reducir este consumo resultarán muy importantes, y, para un funcionamiento típico de unos pocos años, deben ser tenidas muy en cuenta.

Pero no todos los dispositivos de una WSN están necesariamente tan restringidos. Los avances que se han producido durante su relativamente corta existencia permiten que puedan realizar más funciones que la mera recolección de datos. En esta línea es posible encontrar nuevos dispositivos equipados con actuadores que permiten la interacción con el medio y que pueden abarcar desde simples interruptores hasta controladores de robots más complejos. Cuando dispositivos de este tipo están presentes en una red nos podemos referir a ella como una red de sensores y actuadores inalámbrica (*Wireless Sensor and Actor Network*, WSAN). Este tipo particular de redes con elementos heterogéneos (desde el punto de vista del hardware) pueden ofrecer ventajas adicionales a la capacidad de actuación que radican en el hecho de que algunos elementos de estas redes no están tan limitados en cuanto a prestaciones como lo está un nodo sensor. De esta forma, se puede organizar la red para que sobre estos elementos recaigan las tareas de procesamiento más complejas así como otras tareas de comunicaciones que impliquen un mayor consumo de energía, aligerando así la carga de trabajo de otros nodos más limitados.

2.1.1. Nodos sensores: Motes

Los nodos sensores, más comúnmente conocidos como motes, son la pieza básica de la red de sensores. Se trata de dispositivos capaces de realizar medidas acerca de un determinado fenómeno y propagarlas vía radio. Sus principales características, ya comentadas, son su bajo tamaño, coste y consumo de potencia. Disponen de capacidad de procesamiento, comunicación inalámbrica así como de expansión a través de puertos de entrada / salida de su microprocesador.

El nombre de mote procede del inglés, donde su significado es mota de polvo. Esta denominación fue usada por primera vez en el Proyecto *Smartdust* [Warneke 01], concebido en 1998 con la intención de desarrollar nodos sensores de tamaño muy pequeño (en torno a 1 mm³) que en conjunto formarían un polvo inteligente que permitiría estudiar el entorno.

Figura 2-2. Componentes de un mote

Un mote está formado por los subsistemas hardware que se muestran en la Figura 2-2 [Akyildiz 02]. Existen cuatro subsistemas que siempre se encuentran presentes en un mote. El primero de ellos, la unidad de procesamiento, se encarga de coordinar al resto de subsistemas, disparando las tareas de recogida de datos y comunicaciones con otros nodos, así como de procesar los datos tomados. La unidad sensora toma los datos del entorno a través de sus sensores y transforma las medidas analógicas tomadas en medidas digitales que el microprocesador puede manejar. El transceptor se encarga de las comunicaciones con el resto de nodos. Por último, nos encontramos con la unidad de alimentación que suministra energía al resto de subsistemas. Además, existen otros subsistemas que pueden existir opcionalmente en los motes como podrían ser movilizadores (si el mote necesita desplazarse), el subsistema de localización, o un generador de potencia (por ejemplo una célula solar).

Los motes son dispositivos limitados en cuanto a sus prestaciones. Su CPU funciona a baja frecuencia, disponen de pocos kilobytes de memoria RAM y tienen una fuente de alimentación que deben conservar durante el mayor tiempo posible. Debido a esto, las funciones que realizan los nodos se limitan a la captura y envío de datos, con algunas tareas de procesamiento destinadas a disminuir la cantidad de información que se transmite o a obtener nueva información, ya sea a partir de las distintas medidas que se

almacenan en un nodo o a partir de un procesamiento distribuido en varios nodos. El almacenamiento en memoria no volátil está menos restringido ya que disponen de una memoria flash del orden de cientos de KBytes, lo que permite el almacenamiento de gran cantidad de datos, que no tienen porqué ser inmediatamente transmitidos por la radio. Además, los motes cuentan con puertos de entrada/salida que permiten tanto el conexionado de subsistemas adicionales como la comunicación con un PC o la programación del mote.

Actualmente existen motes comerciales que se pueden adquirir desde precios algo inferiores a los cien euros. Durante cierto tiempo los motes más populares fueron los MICAs, desarrollados en la Universidad de Berkeley [Berkeley]. En la Figura 2-3 se pueden ver dos motes de esta familia: los MICA2 [Crossbow], ampliamente usados para investigación y los MICA2dot, más reducidos, del tamaño de una pila botón.

Figura 2-3. Familia de motes MICA

Con posterioridad a los MICAs aparecieron en el mercado los motes *telos*, también desarrollados en la Universidad de Berkeley y comercializados por distintas empresas bajo dos marcas: *telos* [Crossbow] y *tmote Sky* [Berkeley] (véase Figura 2-4). Estos motes tienen un coste más reducido y un menor consumo de potencia, en gran parte por el uso del estándar *ZigBee* [ZigBee] para las comunicaciones, por lo que se están convirtiendo en los más usados actualmente.

Figura 2-4. Mote telos

El mote *Imote2*, diseñado por Intel Corporation [Intel], ha aparecido recientemente con unas prestaciones más avanzadas que los anteriores y unas dimensiones reducidas. Sin embargo, su precio actualmente también es notablemente superior. En la tabla 1 se puede encontrar una comparativa entre los motes anteriores.

	MICA2	MICA2dot	Telos - rev. b	Imote2
Frecuencia CPU	7.37 MHz	4 MHz	8 MHz	13 – 416 MHz
memoria FLASH	512 KB	512 KB	1024 KB	32 MB
memoria RAM	4 KB	4 KB	10 KB	32 MB
Tasa de transmisión (radio)	38.4 Kbps	38.4 Kbps	250 Kbps	250 Kbps
Dimensiones (mm)	58 x 32 x 7	25 x 6	65 x 31 x 6	36 x 48 x 9
Batería	AA x 2	pila botón	AA x 2	AAA x 3 / ión-Litio recargable
Otros	I2C, SPI, DIO, UART	I2C, SPI, DIO, UART	I2C, SPI, USB, DIO, UART	I2C, SPI, mini-B USB, DIO, I2S...

Tabla 1. Comparativa de motes comerciales

2.1.2. Campos de aplicación

Las redes de sensores son objeto de estudio por una parte importante de la comunidad investigadora. Gracias al esfuerzo por parte de ésta se están encontrando un gran número de campos de aplicación para estas redes. Además, están despertando el interés de empresas dispuestas a comercializar esta tecnología, lo que junto con sus prestaciones augura un gran futuro para estas redes. Las características que se pueden conseguir con estas redes como robustez, gran escala y bajo coste hacen que ya se puedan encontrar aplicadas en gran cantidad de aplicaciones reales [Römer 04] que abarcan muy diversos ámbitos como a continuación se verá.

En el campo sanitario hay experiencias como *CodeBlue* [Lorincz 04] donde nodos fijos se despliegan por la estructura de un hospital y nodos móviles se colocan sujetos a pacientes, doctores y otros trabajadores. Con este despliegue se puede controlar la localización de cualquier persona, las constantes vitales de los pacientes e incluso la medicación que se les ha administrado.

Dentro del campo militar, las redes de sensores son una elección interesante debido a la robustez que se puede conseguir con ellas. Dado que en este tipo de aplicaciones es frecuente la caída de los nodos, la redundancia en las comunicaciones en una red WSN minimiza este problema y apenas se ve reflejado en el funcionamiento de la aplicación. Algunos ejemplos de aplicación al campo militar son la detección de ataques nucleares, biológicos y químicos, la monitorización del equipamiento y munición así como aplicaciones de reconocimiento del terreno.

Figura 2-5. Nodo móvil para monitorización de pacientes en un hospital [Lorincz 04]

Otras aplicaciones ampliamente documentadas son las aplicaciones ambientales. En estas aplicaciones la dispersión de nodos por el medio permite analizar las condiciones atmosféricas y del terreno, realizar seguimiento de seres vivos [Juang 02], estudiar fenómenos como volcanes [Werner-Allen 05], terremotos y prevenir otros desastres naturales como incendios e inundaciones. A modo de ejemplo se cita el caso de estudio realizado en Great Duck Island (USA) [Mainwaring 02] donde se despliega una red WSN para la monitorización de la actividad de aves. La red permite la observación del comportamiento de las aves sin ser molestadas por la presencia de humanos. Simultáneamente los biólogos recogen en tiempo real información relativa a niveles de humedad, presión, temperatura y luminosidad en los nidos de los animales. Esta información es recogida por los sensores de los nodos los cuales la reenvían a un nodo central donde se procesa convenientemente.

Dentro del hogar, gracias a la tecnología inalámbrica, se pueden crear espacios inteligentes. Es posible incorporar los nodos a aparatos domésticos como hornos, frigoríficos y reproductores multimedia sin necesidad de cableado adicional. Estos nodos formarían una red que podría ser accedida por los usuarios finales vía Internet o vía satélite. Además, los propios nodos podrían publicar información sobre los servicios disponibles (impresión, fax, etc.) de forma que otros nodos puedan hacer uso de ellos.

La ingeniería civil también se beneficia de esta tecnología que se puede emplear para vigilar el estado de estructuras construidas por el hombre como puentes [Marrón 05] y edificios. Los defectos de éstos se pueden detectar gracias a las vibraciones que generan, las cuales pueden ser recogidas por los nodos de la red. Además, en el campo industrial se pueden encontrar aplicaciones que se benefician del uso de las redes de sensores, al usarse en el control y automatización de numerosos procesos de fabricación.

Por último, otra posibilidad para las redes de sensores es el desarrollo de productos comerciales que se basen en ellas. Por ejemplo, se podrían crear juguetes que cooperen entre ellos, nodos sensores que se incorporen a prendas de vestir [David 05], sistemas para la detección de robos de coches, gestión de inventario, etc.

Como se puede deducir de un estudio de cada una de las posibles aplicaciones citadas su estructura es muy similar, quedando las diferencias estructurales de su

software reducidas a unos pocos subsistemas. Respecto a las comunicaciones, no todas las redes gestionan su topología de la misma forma y siguiendo los mismos algoritmos, si bien se encuentran pocas variantes en la forma de hacerlo. En la mayoría de los casos las redes que se forman tienen sus nodos desplegados en posiciones que permanecen fijas. En casos especiales se contempla la movilidad de los nodos que se encargan de encaminar los paquetes. Finalmente, existen otras situaciones en las que se permiten dispositivos móviles que no encaminan sino que simplemente introducen tráfico en la red. A parte de esto, las principales diferencias vienen impuestas por los dispositivos hardware especiales que cada red gestione o por la capacidad de los nodos para la toma de decisiones o la actuación sobre el entorno.

2.1.3. Objetivos de diseño de una WSN

Para que una red pueda funcionar de acuerdo con las características y restricciones que antes se han mencionado, surgen una serie de retos que los motes y la aplicación que ejecutan deben resolver, como se detalla en los siguientes apartados.

La **eficiencia energética** es sin duda el objetivo más importante para las redes de sensores, pues determina el tiempo de vida de la red. Para ahorrar energía se debe limitar el uso de los subsistemas que componen el mote, en especial el uso de la radio inalámbrica, responsable de una parte importante del gasto energético. Se debe reducir lo máximo posible el número de mensajes que se mandan por radio, lo que se puede conseguir de diversas formas. Una primera medida es ajustar la frecuencia de transmisión de datos por radio al valor más bajo que la aplicación permita. Adicionalmente, se puede reducir aún más la cantidad de transmisiones aplicando técnicas de filtrado y agregación de datos. Mediante el filtrado de datos se puede evitar mandar por radio determinados datos intrascendentes que no cumplan una determinada condición, por ejemplo una aplicación puede ignorar todas las lecturas de temperatura inferiores a cuarenta grados. La agregación de datos permite unir varios flujos de datos en uno solo. De esta manera, si varios nodos para comunicarse con el sumidero comparten algún nodo en su ruta hacia él, estos nodos intermedios retransmitirán un único mensaje con los datos de todos los nodos generadores de información en lugar de un mensaje para cada uno de los nodos.

Una reducción en la potencia de transmisión, controlable desde la propia aplicación, también conlleva un ahorro de energía. Reducir la potencia de transmisión implica reducir el alcance de la radio y, en muchos casos, nodos que tenían visión directa entre ellos dejan de tenerla y deben usar nodos que actúen de intermediarios para intercambiar sus mensajes. Así ocurre en las redes *multihop*, en las que los nodos se organizan en una estructura de comunicaciones en la que los mensajes transmitidos por cada nodo deben realizar varios saltos intermedios antes de llegar al sumidero. A pesar de que con estas redes el número de mensajes transmitidos es mayor se consigue ahorrar energía ya que la potencia que se consume para cada mensaje es mucho menor.

No sólo la transmisión de mensajes consume potencia sino que también la recepción de éstos lo hace. Simplemente por tener la radio en modo de recepción de

mensajes, en espera de que éstos lleguen, se está consumiendo energía. Para evitar derrochar energía lo que se hace es apagar y encender periódicamente la radio, definiendo su ciclo de trabajo. Las radios de todos los nodos deben estar sincronizadas y durante el periodo en el que se encuentren encendidas los nodos deben transmitir sus mensajes. Este parámetro, al igual que la potencia de transmisión, es configurable desde la aplicación.

Además de la radio, otros componentes consumen energía de la batería de los motes, pero de entre todos ellos el consumo más destacable es el de la CPU. La potencia que necesita es menor que la que necesita la radio cuando se encuentra activa, pero si no se toma ninguna medida y la CPU está siempre activa la cantidad de energía que se consume es elevada. Como consecuencia se debe dormir la CPU cuando sea posible. Esto es algo de lo que se encargan los sistemas operativos para redes de sensores, bien durmiendo la CPU cuando no quedan tareas pendientes o bien a petición de la aplicación, despertándola en ambos casos tras recibir alguna interrupción hardware.

Los nodos de las redes de sensores deben tener también capacidad de **autoorganización** para establecer una red de comunicaciones con la que cada nodo sepa cómo hacer llegar la información que obtienen a su destino. Esto implica que los nodos conozcan a sus vecinos, pero normalmente esto resulta inviable por la forma de realizar el despliegue (por ejemplo, si son lanzados desde un avión) o por el gran número de nodos desplegados, que obligaría a cargar previamente en cada nodo información relativa sus vecinos. Si los nodos son capaces de autoorganizarse se puede cargar el mismo programa en todos ellos y que automáticamente obtengan toda la información que necesitan de sus vecinos para formar una red.

Un requisito indispensable y obvio es que la red funcione perfectamente en todo momento. Pero existen situaciones en que la red es más propensa a fallar: cuando se amplía o cuando alguno de sus nodos falla. Estas situaciones suelen presentarse durante el ciclo de vida de una red pues es frecuente que la red crezca, ya que a veces su despliegue se realiza en varias fases. Igualmente, los nodos pueden fallar al haber condiciones externas desfavorables, al agotarse sus baterías o ser víctimas de sabotajes o robo. Por lo tanto, el diseño de una red debe ser **escalable** para permitir su crecimiento y también debe ofrecer **tolerancia a fallos**, impidiendo que el fallo de uno de sus nodos provoque el mal funcionamiento del sistema. A la hora de implementar la aplicación que se ejecuta en los nodos, esto se traduce en la necesidad de dotarla de mecanismos para que pueda adaptar los enlaces de comunicaciones entre nodos a las nuevas situaciones que se van presentando.

Dependiendo de la aplicación que se realice se pueden presentar otros requisitos, como **seguridad**, impidiendo el acceso a los datos por personas ajenas a la red y la introducción de mensajes malintencionados en ésta, o **tiempo real**, para aplicaciones donde el retraso con que se reciben los eventos es crítico, si bien las redes de sensores no constituyen la elección más acertada en este último caso.

2.1.4. Evolución de las redes de sensores

Aún no se ha conseguido alcanzar las dimensiones y costes proyectados para los elementos de las redes de sensores. Los motes comerciales más pequeños actualmente desarrollados tienen dimensiones del orden de 2 centímetros de lado. Existen prototipos de un tamaño menor, de 100 milímetros cúbicos, pero no han llegado a ser funcionales. Sin embargo se espera que se reduzcan significativamente los tamaños y costes de los motes, así como que se produzcan avances en los mecanismos de obtención de energía, en las capacidades de los motes que posibilitaran el éxito de las redes de sensores y actuadores inalámbricas.

Las expectativas asociadas a las redes de sensores inalámbricas son altas [Huang 03]. Según dijo Kris Pister, profesor de la Universidad de Berkeley, fundador de la empresa *Dust Inc.* y experto en miniaturización de sensores, “en el año 2010 los sensores MEMS (*Micro-Electro-Mechanical Systems*) estarán en todas partes, y medirán virtualmente cualquier cosa. Obtendrán energía de la luz solar, las vibraciones, los gradientes de temperatura, emisiones RF de fondo. Los motes serán inmortales, completamente autocontenidos, un único chip con sensorización, comunicación y suministro energético incorporados. Enteramente en estado sólido, sin proceso de decaimiento, pueden sobrevivir a la raza humana”. Estas predicciones fueron realizadas en el año 2000 junto con otras muchas [Pister 00] sobre las funcionalidades, costes y aceptación de las redes de sensores que a día de hoy aún no se han cumplido. Sin embargo sí se han realizado numerosos avances que permiten pensar que algún día esta visión sobre las redes de sensores puede hacerse realidad.

2.2. Desarrollo de software para redes de sensores

Tener que trabajar directamente con los recursos hardware de los motes convierte la programación de una aplicación en una dura tarea. Para simplificar esto se está recurriendo a sistemas operativos que aíslen al programador de las particularidades del hardware. Estos sistemas operativos constituyen el núcleo en torno al cual se van a desarrollar aplicaciones, ofreciendo unos servicios básicos que serán ampliados por otros servicios y herramientas adicionales. Entre estas ampliaciones podemos encontrar *middleware* distribuidos sobre los que implementar las aplicaciones software, como por ejemplo *TinyDB* [Madden 05] que trabaja con la red de sensores como si de una base de datos se tratara (usando sentencias SQL).

Puesto que los sistemas operativos constituyen la pieza sobre la que se va a construir el software para redes de sensores, conviene realizar una serie de consideraciones sobre su diseño. Principalmente, dadas las características y limitaciones de estas redes, los sistemas operativos que vayan a trabajar con ellas deben tener en cuenta dos aspectos muy importantes: la gestión de la potencia y la escasez de recursos hardware disponibles. Respecto al primero de éstos, cada sistema operativo tiene sus propias técnicas, pero todas coinciden en desactivar los subsistemas hardware que no están siendo utilizados, en concreto la radio y la CPU. En cuanto al

segundo aspecto, se deben reducir las necesidades que tanto de memoria RAM, como de memoria de programa tenga el sistema desarrollado. De esta forma se permite que tanto sistemas operativos como aplicaciones puedan ser usados en motes con capacidades muy limitadas.

La imposición de un bajo consumo por parte de los motes ha popularizado el desarrollo de sistemas operativos dirigidos por eventos (aunque no todos siguen esta aproximación). Con esta filosofía el procesado de los datos por la CPU comienza en el momento que se genera un evento, que es gestionado por su correspondiente manejador. El código generado mediante esta aproximación será asíncrono, lo que para redes de sensores permite lograr una mayor eficiencia energética. Gracias a esta asincronía, tras pedir un dato, una aplicación no queda bloqueada a la espera de que el dato esté disponible. De esta forma es posible dormir la CPU mientras se está esperando el dato para despertarla una vez se obtenga. Como contrapartida la programación de aplicaciones es algo más compleja.

Un buen sistema operativo para redes de sensores además debe contar con otras características. Dada la constante evolución de esta tecnología, un sistema operativo debe ser extensible y adaptable. La extensibilidad permite que las aplicaciones desarrolladas puedan ir contando con los últimos avances tecnológicos a medida que se vayan incorporando al sistema operativo. La flexibilidad permite que las aplicaciones se adapten correctamente a los requisitos que se les exigen. De igual manera el sistema operativo también debe adaptarse y manejar con facilidad los cambios en el hardware, ya que la variedad de dispositivos existentes es enorme y sigue creciendo. Además, como explica Hill [Hill 00], la aplicación debe poder controlar directamente todos estos recursos hardware y permitir que exista interacción entre las distintas capas que la forman.

Otra característica deseable para un nodo sensor es la posibilidad de reprogramarlo sin tener que extraerlo de la red. La forma en que esto se hace difiere entre sistemas operativos. Algunos no permiten esta característica, otros como TinyOS pueden hacerlo pero se debe volcar el programa entero al mote para después reiniciarlo con el nuevo código. Por último, algunos sistemas operativos [Han 05][Bhatti 05][Dunkels 04] pueden cargar dinámicamente módulos aislados. Esto permite la adición de funcionalidades a los nodos sin interrumpir su funcionamiento.

Cada sistema operativo tomará decisiones que mejorarán unos aspectos del sistema en detrimento de otros. Generalmente, la adición de nuevos servicios a los nodos suele acarrear una mayor complejidad del sistema operativo que desborda las capacidades de muchos motes. Los principales ejemplos de esto son la gestión dinámica y protección de memoria. Por lo tanto, muchos sistemas operativos están optando por no contar con ellos para reducir los recursos que consumen. Otro caso parecido es la gestión de procesos. Puesto que el cambio de contexto entre dos procesos consume muchos ciclos del procesador se tiende a simplificar este aspecto lo más posible.

Por todo lo citado en los párrafos anteriores, las alternativas tradicionales usadas en otros sistemas empujados no son válidas para las redes de sensores. Sistemas operativos como *WinCE*, *VxWorks*, *PalmOS*, *QNX* y *pOSEK* no cumplen con los requisitos anteriores. Se basan en micro-kernels que permiten la adición o sustracción de funcionalidad según se requiera, por lo que no permiten comunicación directa con los recursos hardware. El entorno de ejecución es similar al de un PC, con cambios de contexto entre procesos excesivamente pesados. Algunos de estos sistemas operativos, además, incorporan mecanismos de protección de memoria que complican en exceso el sistema. Éstos son útiles para una PDA, un teléfono móvil o un PC empujado pero no para un nodo sensor. Otras alternativas como plataformas *middleware* distribuidas existentes (CORBA, DCOM o RMI) tampoco ofrecen una buena solución por su consumo excesivo de memoria. Se están realizando implementaciones ligeras de CORBA para redes de sensores, sin embargo, tampoco resultan apropiadas por los problemas de bloqueo que ocasiona el mecanismo síncrono de pregunta y respuesta de CORBA.

2.2.1. TinyOS

A continuación se describe resumidamente el sistema operativo TinyOS [Hill 00] y su lenguaje de programación nesC, que es una extensión de C orientado a componentes, ya que son los más usados para el desarrollo de aplicaciones con redes de sensores y han sido los elegidos para dar soporte a esta Tesis.

TinyOS es un sistema operativo desarrollado en el año 2000 por la Universidad de California, Berkeley [Berkeley]. Está pensado para trabajar sobre motes con recursos limitados, por lo que destaca el eficiente uso que hace de ellos y su bajo consumo de energía. Esto se consigue desactivando los recursos mientras no están siendo utilizados y activándolos ante la ocurrencia de ciertos eventos. Por lo tanto resulta importante contar con aplicaciones tan simples como sea posible para permitir que la CPU esté desactivada durante un periodo de tiempo mayor. Como consecuencia de esto TinyOS no implementa determinadas funciones complejas como la gestión de memoria o el uso de múltiples procesos, lo que además reduce las necesidades de memoria RAM y memoria de programa de las aplicaciones. En definitiva, TinyOS es un sistema operativo muy ligero y sencillo.

El lenguaje de programación nesC [Gay 03] de TinyOS, favorece la reutilización de código gracias a su orientación a componentes. De esta forma se permite la definición de componentes reutilizables y su interconexión. Esto resulta muy útil para estas redes si se tienen en cuenta la cantidad de nuevos algoritmos y mecanismos de encaminamiento, sincronización y agregación de datos que constantemente están apareciendo. Para la adopción de uno de estos nuevos mecanismos basta con la sustitución de un componente en la aplicación por otro más avanzado que los implemente. Esta modularidad también se extiende al campo del hardware, ya que permite el fácil reemplazo y adición de componentes hardware como placas de sensores.

A este respecto, paulatinamente van apareciendo nuevos componentes que completan las funcionalidades que el núcleo de TinyOS no provee y que pueden ser fácilmente integrados en las aplicaciones nesC. Por ejemplo, ya es posible el uso de componentes para cifrar los datos o para permitir la reprogramación vía radio de todo el código ejecutado en el mote. De igual manera el número de distintos tipos de mote (plataformas hardware) soportados por TinyOS es bastante elevado. Estas características y el hecho de que estuviera disponible bastante antes que otros sistemas operativos, han hecho que TinyOS se convierta en el sistema operativo más extendido para redes de sensores.

2.2.1.1. Lenguaje nesC

Como se dijo, el lenguaje nesC [Gay 03] es una extensión C orientada a componentes. Sus principales características son:

- **Separación entre construcción y composición.** Los programas se componen de componentes, los cuales se interconectan para formar programas. Existen dos tipos distintos de componentes: **módulos** y **configuraciones**. Los módulos implementan una especificación de componente mediante código C encargándose de aspectos computacionales. En cambio las configuraciones se encargan de la especificación de interconexiones entre otros componentes.
- **Especificación del comportamiento de componentes en función de interfaces.** Un componente se relaciona con otros componentes mediante las interfaces que los conectan con ellos. Los componentes para ello disponen de **instancias** de las interfaces, bien **provistas**, por las cuales ofrecen funcionalidad a otros componentes, o bien **usadas**, que indican que requieren funcionalidades de otros componentes.

Las interfaces están compuestas por dos tipos de funciones, **comandos** y **eventos**. Los comandos deben ser implementados por el componente que provee una funcionalidad (e invocados por el componente que la usa), mientras que los eventos son implementados por el componente que usa una funcionalidad (y señalizados por el componente que la provee). Las funcionalidades o servicios que ofrecen los interfaces pueden ser accedidas de forma síncrona o asíncrona. Los servicios ofrecidos de forma síncrona son ejecutados dentro de la implementación de un comando. Los servicios asíncronos comienzan su ejecución tras finalizar la del comando que los provee. Una vez realizado el servicio, si es necesario realizar una notificación al componente que solicitó el servicio, se señala el evento correspondiente.

- **Concurrencia mediante tareas y manejadores de interrupciones.** Los procesados menos críticos en tiempo se ejecutan en tareas, las cuales se van introduciendo en una cola FIFO y son ejecutadas cuando la CPU queda

libre. En cambio las interrupciones son atendidas inmediatamente por sus manejadores (eventos).

En la Figura 2-6 se puede ver lo que se acaba de comentar para el caso de la ejecución asíncrona de un servicio en un componente. En ella, el componente A solicita un servicio al componente B mediante la invocación del comando C1 de una instancia de la interfaz I1 (usa la interfaz I1). Al estar el componente B conectado a dicha instancia de interfaz, éste ejecuta el código asociado al comando. El comando entonces pone una tarea T en la cola de tareas del nodo y devuelve el control al componente A. Más adelante, cuando la CPU del nodo esté libre, se ejecutará la tarea T (y por tanto el servicio). Una vez haya finalizado la ejecución del servicio la tarea señalará un evento E1 a la interfaz I1, el cual recibirá el componente A (y ejecutará su manejador correspondiente).

Figura 2-6. Interconexión de componentes en nesC

2.2.2. Otros sistemas operativos para WSN

Posteriormente a la aparición de TinyOS se han desarrollado otros sistemas operativos que, al menos de momento, no han conseguido desbancarlo de su posición dominante. Los primeros motes existentes hubieran tenido problemas para usar estos sistemas. Sin embargo, motes más recientes como los *MICA2* o los *telos* pueden ejecutar sin problemas la mayoría de ellos.

Ejemplos de estos sistemas operativos son *Contiki*, *SOS* y *MANTIS OS*. El primero de ellos, *Contiki* [Dunkels 04], destaca por el soporte que da a la pila de protocolos TCP/IP. En cambio en *SOS* (*Save Our Sensors*) [Han 05] uno de principales objetivos es permitir la reprogramación dinámica (total o parcial) del código que ejecutan los sensores mediante la carga y descarga de módulos sobre el kernel del sistema operativo. *MANTIS OS* [Bhatti 05], por el contrario, no permite esta reprogramación dinámica pero ofrece otras características como una planificación de

tareas basada en prioridades y con garantías de tiempo real.

Como se ha visto cada sistema operativo ofrece unas prestaciones distintas, pero no por ofrecer más prestaciones un sistema operativo es mejor ya que éstas suelen acarrear un aumento en los requisitos hardware de los nodos y una menor eficiencia en su funcionamiento. En función de los requisitos de la aplicación habrá que determinar qué sistema operativo es el más apropiado.

2.2.3. Deficiencias del proceso tradicional de desarrollo de software.

El principal problema que se afronta al desarrollar aplicaciones para redes de sensores es la complejidad de su programación, que requiere que usuarios con ciertas habilidades al respecto se encarguen del desarrollo. Además, se encontrarán con la dificultad añadida que conlleva el aprendizaje de esta tecnología ya que su programación incorpora ciertos elementos como el manejo de la asincronía de los eventos y con el que la familiarización requiere cierto tiempo. Sin embargo muchas veces los usuarios de las WSN no son programadores sino científicos o empleados cuyo campo de trabajo no son las WSN e intentarán dedicar a éstas el menor esfuerzo posible. La complejidad que puede llegar a entrañar el desarrollo de una aplicación puede hacer a los programadores inexpertos desistir de desarrollar aplicaciones propias para sus propios fines y obligarles a usar aplicaciones comerciales ya existentes que pueden no adaptarse a sus necesidades.

Por otro lado, aunque los sistemas operativos para WSN constituyen entornos de desarrollo bastante completos que incluyen herramientas para distintos fines, como la realización de simulaciones y la optimización y detección de errores, estas tareas se llevan a cabo en fases muy tardías del proceso desarrollo y no es fácil determinar en qué parte del código se cometió un error.

Un tercer problema es la dependencia del software desarrollado respecto de la plataforma para la que se creó. Esto es totalmente cierto para el caso de las plataformas software (sistemas operativos y demás entornos de desarrollo) en las que el código de sus aplicaciones no podrá ser reutilizado en otras plataformas (al menos sin ser adaptado). En el caso de las plataformas hardware (los motes) no siempre es cierto pues los sistemas operativos proveen mecanismos para generar imágenes ejecutables en distintas plataformas hardware, si bien en ciertos casos el código necesita de pequeñas adaptaciones.

Por todo esto se hace necesario seguir un nuevo proceso de desarrollo que solvante esta problemática, como se comenta en el capítulo 3, en el que se describe la aproximación MDE (*Model Driven Engineering*) para el desarrollo de software. Esta aproximación pretende, mediante el uso de modelos, simplificar y mejorar el desarrollo de aplicaciones.

2.2.4. Otras alternativas

Existen otras alternativas para programar las redes de sensores que hacen uso de conceptos de más alto nivel (en comparación con los lenguajes de programación estructurada) para programación de aplicaciones. En su mayor parte implican el uso de tecnologías *middleware* distribuidas que proporcionan abstracciones para simplificar el intercambio de datos entre nodos. Así, a la ya comentada *TinyDB* que ve la red de sensores en una base de datos, se unen otras como *Regiment* [Newton 07] que ofrecen una serie de primitivas para determinar los vecinos de un nodo e intercambiar datos entre ellos. Sin embargo estas aproximaciones aún siguen empleando lenguajes textuales complicados de usar.

Más en la línea de esta Tesis, y ofreciendo ya notaciones visuales, se encuentran otras alternativas que aplican un enfoque basado en el uso de modelos. Ejemplos significativos son *GRATIS* [Volgyesi 02], *Viptos* [Cheong 05], *ATaG* [Bakshi 05] y *Scatterplug* [Saad 08], que serán explicados en el capítulo 3. Estas herramientas permiten la construcción de aplicaciones mediante el ensamblaje de componentes o librerías ya existentes, pero ninguna de ellas permite modelar completamente el comportamiento de una red de sensores. No alcanzan por esto una generación de la totalidad de la implementación, lo que implica que para el desarrollo de nuevas aplicaciones hay que introducir una cantidad importante de código. Sólo *ATaG* modela parcialmente el comportamiento de los nodos, ofreciendo un nivel de abstracción superior al de las otras herramientas. Sin embargo la generación de código está automatizada sólo en una pequeña parte, ya que el usuario debe introducir el código correspondiente a cada una de las tareas que ejecuta una aplicación. Además, como consecuencia, se hace necesario tener conocimientos sobre la programación de la plataforma de implementación usada.

2.3. Conclusiones y aportación a la Tesis

A lo largo de este capítulo se han descrito las redes de sensores y sus principales características y aplicaciones. Se ha hecho especial hincapié en el desarrollo de software para esta tecnología ya que ese es el propósito de esta Tesis. A consecuencia se han comentado tecnologías *middleware* y sistemas operativos típicamente empleados en la programación de las redes de sensores. De entre todos éstos se ha dedicado mayor atención *TinyOS* y a su lenguaje de programación *nesC* que son los empleados durante el desarrollo de la Tesis.

En vista de las deficiencias expuestas a lo largo del capítulo del proceso de desarrollo existente para redes de sensores se justifica la necesidad de una aproximación al desarrollo de software MDE como la que se desarrolla durante esta Tesis.

Capítulo 3

Desarrollo de Software Dirigido por Modelos

Este capítulo cubre el Desarrollo de Software Dirigido por Modelos (MDE) que es la pieza central en esta Tesis. Para su entendimiento, en primer lugar se exponen los conceptos básicos del modelado de sistemas software y, posteriormente, se explican los principios sobre los que se sustenta MDE, incluyendo tecnologías y aproximaciones típicamente empleadas en este enfoque. De entre las aproximaciones expuestas destaca MDA a la que se le da especial atención. Finalmente, se aborda la aplicación de MDE a sistemas reactivos y más específicamente a las redes de sensores.

3.1. Motivación

A pesar de los avances realizados tanto en lenguajes de programación como en sistemas operativos y plataformas *middleware*, la programación de los sistemas software actuales es, en muchos casos, un proceso complejo. Esto se debe al gran número de artefactos y relaciones entre ellos que hay que manejar y a la distancia conceptual que existe desde el problema que se plantea a la solución que se emplea para resolverlo. Si además se tienen en cuenta los numerosos cambios a los que está sujeto un sistema software, ya sean relativos a sus requisitos o a sus plataformas de implementación, la complejidad aumenta pues es necesario desarrollar artefactos software con capacidades de evolución y reutilización. Aunque la orientación a objetos ha conseguido logros interesantes a este respecto, no ha llegado a cumplir estos objetivos con los niveles de satisfacción que de ella se esperaba [Bézivin 01].

Para simplificar el desarrollo de software ya en el pasado se persiguieron métodos que permitieran generar código a partir de descripciones sencillas de las aplicaciones. Un ejemplo fueron las herramientas CASE (*Computer Aided Software Engineering*) que comenzaron a aparecer en los años 80, las cuales interconectaban conceptos gráficos genéricos para la especificación del comportamiento de las aplicaciones (véase Figura 3-1). Sin embargo tuvieron poca aceptación en el entorno industrial. No simplificaban en exceso el desarrollo de aplicaciones pues aún requerían ciertos conocimientos de programación y técnicas de modelado por parte de los desarrolladores. Además, proporcionaban conceptos que no se adecuaban a gran cantidad de dominios de aplicación y cuya transformación a un código usable por los sistemas operativos de la época era demasiado compleja para las herramientas entonces existentes [Schmidt 06].

Figura 3-1. Descripción de una aplicación con herramienta CASE (*Rhapsody*)

Una aproximación reciente considera el uso de modelos para el desarrollo de software, no utilizándolos solamente para fines de documentación sino que, además, son el principal artefacto involucrado en el desarrollo de software.

Los modelos pueden representar a un sistema de forma sencilla, centrándose en la información de éste que es de relevancia para el desarrollo de software e ignorando el resto. A la vez ofrecen una ventaja muy importante ya que pueden ser interpretados por máquinas, lo que posibilita el desarrollo de herramientas que automaticen el proceso de desarrollo de software. Como consecuencia se puede llegar a la generación automática de código, la cual recientemente es más viable debido a la existencia de plataformas tecnológicas más avanzadas, como sistemas operativos, *middlewares* distribuidos y lenguajes que pueden hacer uso de gran cantidad de librerías de funciones. Estas plataformas actuales reducen la complejidad del código que se debe generar automáticamente al encargarse del manejo de aspectos no funcionales del sistema como seguridad, tolerancia a fallos, gestión de recursos distribuidos y descubrimiento de servicios o nodos, entre otras, por lo que evitan tener que programar transformaciones a código complicadas para garantizarlos. El uso de aproximaciones basadas en el uso de modelos está teniendo un impacto enorme en el mundo académico en los últimos años aunque apenas se encuentra aplicado en el ámbito industrial.

3.2. Modelado de sistemas

3.2.1. Modelos como mecanismo de abstracción

Desde los orígenes de la programación la abstracción ha sido una pieza clave en el desarrollo de software. Inicialmente la programación se realizaba mediante código máquina, a través de secuencias de bits que los ordenadores ejecutaban. En este caso la abstracción era nula y la distancia con respecto al razonamiento humano era muy grande, por lo que el desarrollo de software era realmente costoso. Como consecuencia se desarrolló el lenguaje ensamblador como resultado de un primer proceso de abstracción que asignaba a cada una de las distintas secuencias de bits comprensibles por una máquina una representación textual más comprensible por una persona. Posteriormente aparecieron las macroinstrucciones, que a su vez abstraían de las propias instrucciones del lenguaje ensamblador. En un siguiente paso, con la introducción de la programación estructurada a principios de los años 60, aparecieron una serie de lenguajes que daban lugar a programas no sólo eficientes sino que además facilitaban su lectura y comprensión. El teorema del programa estructurado [Bohm 66] establecía que cualquier programa construido con éstos podía construirse en función de tres estructuras de control: secuencias, condicionales e iteraciones. Esto representaba un avance frente a la programación en ensamblador, sin embargo este tipo de programación aún requería de esfuerzo y ciertas habilidades por parte de los programadores. En el año 65 aparecía el lenguaje Simula I, en respuesta a las dificultades que Dahl y Nygaard habían tenido previamente en el desarrollo de un simulador para reactores nucleares. Éste fue el primer lenguaje que empleó versiones

primitivas de los conceptos propios de la programación orientada a objetos como herencia y vinculación dinámica que más tarde serían incorporados a otros lenguajes en los 70 y que se harían muy populares en los 80. Estos lenguajes orientados a objetos, con conceptos de más alto nivel, hacían los programas más fáciles de escribir y mantener.

Sin embargo todos estos avances han ido enfocados al dominio de la solución, es decir, a las propias tecnologías de programación. Para lograr un mayor nivel de abstracción es necesario centrarse en el dominio del problema, buscando abstracciones que describan conceptos de dominios como por ejemplo las telecomunicaciones, la industria del automóvil o la espacial. Centrándose en este dominio se consigue una programación de aplicaciones más simple, que se asemeja más a los conocimientos y a la forma de razonar de los profesionales de cada una de estas áreas.

Para apuntar al dominio del problema se puede hacer uso de los modelos. Bézivin [Bézivin 01] da una definición muy sencilla de modelo según la cual:

“Un modelo es una simplificación de un sistema con un objetivo en mente.”

De acuerdo con la definición anterior se puede ver cómo el uso de modelos no es algo nuevo ni restringido al desarrollo de software, sino que se está llevando a cabo durante un periodo de tiempo muy extenso para muy distintos fines. Durante siglos éstos han sido usados para la construcción de diversos sistemas, como por ejemplo los planos de un edificio. También se han realizado modelos para el estudio de propiedades de los sistemas. Como ejemplos se pueden citar los diversos modelos del cuerpo humano (locomotor, respiratorio, circulatorio, digestivo, etc.) o modelos de territorios como son los mapas. En la Figura 3-2 se puede ver un ejemplo.

Figura 3-2. Uso histórico de los modelos: mapas

Las características que se deben encontrar en un modelo han sido citadas por Selic [Selic 03], que identifica cinco de ellas como fundamentales. La principal es la abstracción, por la cual todas las propiedades de un sistema que no son de interés no

deben aparecer en su modelo, permitiendo así gestionar la complejidad del sistema de manera eficiente. También destaca que los modelos deben ser comprensibles para simplificar su uso, precisos para dar una interpretación real de las propiedades del sistema, predictivos, permitiendo averiguar mediante análisis propiedades no obvias del sistema y, por último, económicos, siendo más asequible la construcción y uso del modelo que la del propio sistema.

Una característica importante de los modelos es la simplicidad frente a los sistemas que representan. Este último punto ha quedado constatado en los relatos contados por diversos escritores. Jorge Luis Borges y Umberto Eco contaban historias en las que los cartógrafos de un reino, al no estar contentos con los mapas de tamaños desmesurados existentes, exigían la construcción de mapas aún mayores, de escala 1:1, que cubrían todo el país. El tamaño de estos mapas finalmente llevaba a su abandono. Lewis Carroll, en una historia similar, al ocultar el mapa la luz del Sol, proponía el uso del propio país como mapa. Los modelos persiguen la simplificación de los sistemas, en caso de no ser así, su uso no está justificado.

3.2.2. Los metamodelos en el desarrollo de software

Meta es un prefijo griego que significa “más allá” o “de nivel superior”. Por lo tanto un metamodelo es un modelo de nivel superior. Frecuentemente se suele decir que un metamodelo es un modelo de otro modelo, sin embargo esta afirmación no es cierta. Bézivin utiliza el ejemplo del cuadro de Magritte mostrado en la Figura 3-3 para explicar esto. Este cuadro además es un ejemplo perfecto para dejar clara la diferencia entre un modelo y el sistema al que representa. En él aparece dibujada una pipa bajo la cual se puede leer “*Ceci n'est pas une pipe*” (esto no es una pipa). En efecto no es una pipa, sino una representación de una pipa o un modelo de una pipa (este modelo puede usarse para estudiar propiedades de la pipa, sin embargo nunca podrá usarse para fumar tabaco). Según esto, el sistema es una pipa y el modelo es un cuadro con el dibujo de pipa. En este caso un modelo de un modelo podría ser el cuadro del cuadro de una pipa (ver Figura 3-4).

Figura 3-3. Esto no es una pipa

Figura 3-4. Modelo de un modelo

Una definición más apropiada dice que un metamodelo es un modelo que define el lenguaje para expresar un modelo [MOF 06]. Un metamodelo define los conceptos de un lenguaje y las relaciones que entre ellos se establecen. En el caso concreto de un mapa, el metamodelo define el lenguaje que se usa para definir el mapa, es decir, el metamodelo es su leyenda. Si el mapa es un plano de líneas de autobús esta leyenda determina los distintos tipos de elementos que se pueden usar para construirlo, como el uso de líneas de distintos colores para las distintas líneas de autobuses o de símbolos especiales para situar diversos puntos de interés como hospitales o estaciones de autobuses y trenes.

Al igual que la relación que se establece entre un sistema y su modelo es de representación (el modelo representa al sistema), al introducir los metamodelos se establece una nueva relación de conformidad entre el modelo y el metamodelo: un modelo es conforme a un metamodelo. Esta relación indica que un metamodelo ha sido creado de acuerdo con los conceptos y reglas definidos en el metamodelo. En la Figura 3-5 se muestran estas relaciones.

Figura 3-5. Relaciones entre sistema, modelo y metamodelo en un mapa

Vistas las relaciones existentes entre sistemas, modelos y metamodelos sólo falta por ver las nuevas relaciones que se obtienen al introducir los lenguajes (y metalenguajes). En la definición que se dio de metamodelo se puede ver una de ellas: un metamodelo es un modelo que define un lenguaje para expresar un modelo. Por lo tanto entre metamodelo y lenguaje se establece una relación de definición. A su vez, un lenguaje de modelado se define como un conjunto de modelos [Favre 04], en particular el conjunto formado por todos los modelos que ese lenguaje puede formar, los cuales pertenecen a él. Según esto la relación entre lenguaje y modelo es de pertenencia, al igual que lo es entre metalenguaje y metamodelo. El conjunto de todas las relaciones se puede ver en la Figura 3-6.

Figura 3-6. Relaciones entre (meta)modelos y (meta)lenguajes de modelado

La importancia de los metamodelos en el desarrollo de software radica en el hecho de que gracias a ellos los modelos se pueden especificar formalmente. Una especificación formal es susceptible de ser interpretada por una máquina, lo cual permite el desarrollo de herramientas que la manipulen y automaticen el desarrollo de software.

3.3. Model Driven Engineering

En un enfoque de desarrollo de software MDE (*Model Driven Engineering*) [Kent 02], también conocido por otros diversos nombres como DSDM (Desarrollo de Software Dirigido por Modelos) o MDD (*Model Driven Development*), los modelos constituyen el principal artefacto en el proceso de desarrollo. Si antes primordialmente se usaban para fines de documentación, ahora son una parte fundamental en el desarrollo de código.

El uso de un enfoque de este tipo pretende facilitar el desarrollo de software, simplificando la especificación de aplicaciones gracias a la posibilidad de abstracción que ofrecen los modelos y generando una implementación ejecutable a partir de esta especificación. Para poder llevar esto a cabo cobran gran importancia las transformaciones, ya sean entre modelos a distinto nivel de abstracción o entre modelos y código. Además, MDE pretende gestionar el desarrollo de aplicaciones complejas permitiendo la especificación por separado de modelos correspondientes a las distintas vistas de un mismo sistema, que después serán integradas y transformadas.

Según Atkinson y Kühne [Atkinson 02] otro propósito fundamental de MDE es la mejora de la productividad, dotando de más funcionalidad a los artefactos desarrollados, de un mayor tiempo de vida y una mejor adaptación ante cambios. Respecto a esto último Atkinson y Kühne citan cuatro fuentes de cambios y de líneas de actuación para gestionarlos, que constituyen requisitos deseables en MDE:

- **Cambios de personal.** Las fluctuaciones de personal son frecuentes en una empresa y provocan una pérdida de conocimiento. La posibilidad de representar este conocimiento mediante modelos concisos, adaptables y fáciles de entender palia los efectos de este cambio ya que el nuevo personal puede adquirir este conocimiento con más facilidad.
- **Cambios en los requisitos.** Estos cambios son muy frecuentes en el software actual. El impacto de estos cambios debe ser mínimo, tanto en términos de esfuerzo de desarrollo como en términos de interrupción del servicio. Esto implica dar soporte a la adición dinámica de tipos en tiempo de ejecución.
- **Cambios en las plataformas de desarrollo.** Las plataformas que se usan para soportar el proceso de desarrollo en un enfoque MDE también cambian constantemente. Además generan artefactos dependientes de ellas que quedan obsoletos al cambiar las plataformas. Para solucionarlo los artefactos deberían estar desacoplados de las plataformas que los generan. Por tanto un requisito necesario es el almacenamiento de datos en formatos que puedan ser usados por otras herramientas, ofreciendo así interoperabilidad entre ellas.
- **Cambios en las plataformas de implementación.** Debido a la evolución de las plataformas existentes y a la aparición de nuevas se debe proteger a los artefactos desarrollados de los cambios en las plataformas. Esto se consigue desarrollando artefactos independientes de la plataforma, con lo que se asegura que seguirán siendo válidas aún cuando éstas cambien, y transformándolos, de forma automatizada, a artefactos dependientes de la plataforma.

A continuación se describe el estándar MDA por su alta importancia dentro de MDE.

3.3.1. MDA

MDA se puede considerar como la visión particular del OMG (*Object Management Group*) sobre MDE. Aunque MDE ofrece una aproximación más amplia al desarrollo basado en modelos, MDA propone muchos de los conceptos que son usados en MDE.

MDA [MDA 03] define una arquitectura de desarrollo formada por diversos niveles de abstracción y estándares para la puesta en marcha de un enfoque de Desarrollo de Software Dirigido por Modelos. El concepto básico es la separación entre el funcionamiento de una aplicación y los detalles de cómo éste se implementa en una plataforma concreta. De acuerdo con el OMG, MDA permite que un mismo modelo especifique funcionalidad que puede ser implementada en diversas plataformas mediante sus correspondientes transformaciones. De la misma forma, MDA persigue la estandarización de las tecnologías empleadas en el proceso de desarrollo para permitir interoperabilidad entre las distintas herramientas y aplicaciones que toman parte en el desarrollo de software.

Según este estándar los modelos pueden ser clasificados en función de su nivel de abstracción definiendo tres tipos de modelos:

- **Computation Independent Model (CIM).** Un modelo CIM describe una aplicación en función de conceptos independientes de la computación, especificando los requisitos del sistema. Estos modelos son independientes de la forma en la que el sistema posteriormente será implementado. De hecho los modelos CIM son principalmente usados por los expertos del dominio o consultores de negocio que no tienen por qué tener conocimiento de cómo los modelos y otros artefactos del proceso de desarrollo se usan.
- **Platform Independent Model (PIM).** Estos modelos se centran en describir el funcionamiento de una aplicación mediante un enfoque computacional pero sin emplear conceptos relacionados con ninguna plataforma de ejecución concreta (según MDA una plataforma es un conjunto de subsistemas y tecnologías que ofrecen un conjunto coherente de funcionalidad a través de interfaces y patrones de uso). Por lo tanto, una especificación es válida para más de una plataforma. Para describir un modelo PIM se puede usar un lenguaje de modelado de propósito general o un lenguaje específico del área en la que el sistema se usará.
- **Platform Specific Model (PSM).** Un modelo PSM describe un sistema desde el punto de vista de una plataforma concreta. Este tipo de modelos combina la especificación de la aplicación realizada en los modelos PIM con detalles relativos a cómo ésta se implementa en una plataforma concreta. Son por tanto válidos para una única plataforma, con la cual deben tener una relación directa que permita la generación de código desde ellos.

El proceso que MDA propone para el desarrollo de aplicaciones para una plataforma concreta típicamente consiste en la definición de un modelo PIM, independiente de la plataforma de ejecución, para ser transformado a un modelo PSM de la plataforma en cuestión, y, por último, ser transformado a código (o a otras estructuras de datos). La Figura 3-7 muestra la transformación de un modelo PIM a un modelo PSM. Además en ella se puede observar cómo previamente a la definición de estos modelos se deben crear los metamodelos que definen a sus dos lenguajes de modelado.

Las transformaciones que se ejecutan idealmente deben estar automatizadas y ser ejecutadas por herramientas, aunque en muchos casos debe haber cierta intervención humana en el proceso. El uso de los modelos CIM y la automatización de sus transformaciones apenas está descrito por la especificación de MDA, no quedando del todo claro qué se entiende como modelo CIM. En general se considera que estos modelos se usan sólo para especificar requisitos y obtener, de forma manual, los modelos PIM, aunque no se rechaza que esta transformación pueda ser automatizada.

Figura 3-7. Transformación PIM – PSM [MDA 03]

A pesar de que MDA define tres tipos de modelos según su abstracción no tienen porqué existir tres niveles de abstracción para clasificar a los modelos sino que el número puede ser mayor. Esto se debe a la posibilidad de tener modelos PIM con distintos niveles de abstracción, al igual que puede ocurrir con los modelos PSM. Las transformaciones se ejecutarían entre los distintos niveles PIM primero y entre los distintos niveles PSM después, según el nivel de abstracción de los modelos. En este caso los roles PIM y PSM son relativos, pudiendo ser un mismo modelo considerado como PIM o PSM en función de su nivel de abstracción relativo a otros modelos. Un mismo modelo se puede considerar como PIM si se le compara con modelos de menor nivel de abstracción (sería un PSM respecto a él) o PSM si le compara con otro modelo de mayor abstracción. Un ejemplo se puede ver en la Figura 3-8 donde un modelo de alto nivel de abstracción se quiere transformar, primero a un modelo basado en componentes genéricos y, posteriormente, a un modelo de componentes CORBA más concreto. En la primera transformación el modelo más abstracto tendrá el rol de PIM y se creará un modelo de componentes que asumirá el rol de PSM. Sin embargo, este último modelo en la siguiente transformación asumirá el rol de PIM.

Para expresar los modelos hace falta un lenguaje de modelado definido por un metamodelo. De la misma forma, para expresar este metamodelo es necesario otro lenguaje y su correspondiente metamodelo, conocidos como metalenguaje y meta-meta-modelo. Y así sucesivamente se expresaría cada meta^n -modelo usando meta^n -lenguajes definidos por meta^{n+1} -modelos. Para cerrar este bucle infinito, el metamodelo de nivel más alto ($n+1$) debe describirse usando el lenguaje que él mismo define. En MDA se usa una arquitectura de cuatro capas ($n=2$, con niveles M3, M2, M1, M0). En la capa más alta (M3) se encuentra el metamodelo de MOF (estándar del OMG para meta-meta-modelado), en la capa M2 se encuentran los metamodelos que definen a los modelos de la capa M1 y en la capa M0 se encuentra el mundo real con instancias de los modelos en M1 obtenidas en tiempo de ejecución (dependiendo de cuál sea el modelo éstas podrán ser estructuras de datos, un programa, la ejecución de un

programa, etc.). En la Figura 3-9 se muestra un ejemplo donde se puede observar cómo cada capa es definida haciendo uso de los conceptos de la capa de nivel superior.

Figura 3-8. Relatividad de roles en las transformaciones MDA

Figura 3-9. Ejemplo de uso de la arquitectura de cuatro capas de MDA [UML 07]

En el caso de MDA estas capas además se usan para modelar mediante niveles CIM, PIM y PSM, los cuales tienen sus metamodelos en la capa M2 y los modelos definidos con ellos en la capa M1. En la Figura 3-10 se puede ver cómo los niveles de abstracción de MDA encajan en la arquitectura de modelado de cuatro capas.

Figura 3-10. Niveles de abstracción de MDA en la arquitectura de cuatro capas

3.3.1.1. Estándares MDA

Una de los propósitos de MDA es definir un *framework* que integre distintos estándares del OMG y proporcione interoperabilidad entre las herramientas MDA. Sus principales estándares se describen a continuación:

- **Meta-Object Facility (MOF)**

El estándar MOF del OMG provee un *framework* y un conjunto de metaservicios para permitir el desarrollo y la interoperabilidad de modelos y sistemas guiados por metadatos [MOF 06]. Para esto MOF define un lenguaje común para definir lenguajes de modelado e intercambiar modelos expresados en esos lenguajes. Este lenguaje se compone de cinco conceptos básicos de la orientación a objetos, usados para especificar la sintaxis abstracta de los lenguajes de modelado: clases, asociaciones entre clases, tipos de datos, paquetes y restricciones. Desafortunadamente MOF no da facilidades para la especificación de la sintaxis concreta de los lenguajes. Ésta puede realizarse de diversas maneras, dependiendo de la herramienta que se use en la implementación de MOF, siendo por lo tanto la sintaxis concreta dependiente de la herramienta.

Ejemplos de lenguajes creados con MOF son UML, CWM (*Common Warehouse Metamodel*) o el mismo MOF, que se define a sí mismo, como se vio anteriormente al situarlo en la capa M3 de MDA. Con estos lenguajes se pueden

desarrollar herramientas de modelado y desarrollo, sistemas de almacenamiento de datos y almacenes de meta-datos, entre otros.

El hecho de construir los distintos metamodelos (M2) conformes a un mismo meta-meta-modelo (M3, MOF) proporciona interoperabilidad entre ellos. Gracias a esto se facilita el desarrollo de transformaciones entre los modelos. De la misma forma MOF también provee conceptos para la transformación desde éstos a diversas plataformas específicas. Ejemplos son la transformación de MOF a IDL (*Interface Definition Language*) usado para definir interfaces en CORBA, a XML mediante el estándar XMI y a Java mediante el estándar JMI.

- **Unified Modeling Language 2 (UML 2)**

UML es un lenguaje visual para especificar, construir y documentar sistemas [UML 07]. Los conceptos y relaciones del lenguaje se encuentran definidos con MOF y disponen de una semántica bien definida y una representación fácilmente comprensible. Uno de sus principales propósitos es permitir la interoperabilidad entre herramientas, por lo que al igual que ocurre con MOF, sus modelos se pueden exportar gracias al estándar XML.

El lenguaje UML es muy amplio y cubre un conjunto muy extenso de dominios de aplicaciones. Esto hace que muchas de sus capacidades de modelado no sean siempre útiles en todos los dominios, por lo que dispone de mecanismos para seleccionar las partes del lenguaje que pueden ser de interés. Además, permite la extensión del lenguaje mediante perfiles UML para poder adaptarlo a conceptos de diversas plataformas de implementación o dominios.

En la arquitectura MDA, el lenguaje UML desempeña un papel muy importante pues el OMG intenta que sea el estándar usado para la descripción de los modelos PIM y PSM, si bien se considera que también se pueden usar otros lenguajes definidos con MOF.

En contra de UML hay que decir que es un lenguaje muy complejo en el que, a pesar de sus posibilidades de personalización, la descripción de aplicaciones en dominios específicos no es tan sencilla que puede resultar mediante el uso de lenguajes específicos del dominio (*Domain Specific Language*, DSL – concepto explicado en el capítulo siguiente). Igualmente tampoco se adapta al dominio en cuestión como éstos.

- **XML Metadata Interchange (XMI)**

XMI [XMI 05] es un estándar desarrollado por la OMG para conseguir interoperabilidad entre herramientas. Define un mecanismo para serializar (codificar un objeto mediante una secuencia de bits con el fin de ser transmitido o almacenado) modelos y metamodelos e intercambiarlos entre distintas aplicaciones. Inicialmente se pensó para el intercambio de modelos UML pero

también puede ser usado para intercambiar otros metamodelos no UML definidos con MOF.

XMI hace uso del lenguaje de marcas XML para almacenar los modelos, transformando los metamodelos definidos con MOF a *XML Schema* y los modelos a documentos XML.

Para finalizar con MDA, resta comentar algunas de las deficiencias que MDA presenta [Muñoz 04]. En primer lugar, las descripciones del estándar carecen en muchas ocasiones de precisión. Si se consultan las especificaciones de MDA [MDA 03] se puede observar que muchos de los conceptos y procesos que se definen no están claramente definidos o no cuentan con el nivel suficiente de detalle, por lo que pueden ser interpretados de diversas formas. Esto se hace patente consultando la gran cantidad de información existente acerca de MDA la cual, dependiendo de la fuente a la que se acuda, puede presentar variaciones. Por otro lado, ciertos aspectos relativos a la calidad del software generado, como por ejemplo Verificación y Validación (V&V), apenas están cubiertos por MDA, como tampoco lo está el uso de clasificaciones adicionales de los modelos (aparte de CIM, PIM y PSM).

3.3.2. Modelos y transformaciones en MDE

3.3.2.1. Modelos

El estándar MDA define, entre otras cosas, una serie de conceptos y guías para desarrollar herramientas de desarrollo de software basadas en modelos. En cuanto a los modelos, se consideran principalmente como una forma de abstraerse de detalles tecnológicos, clasificándolos según tres perspectivas CIM, PIM y PSM de acuerdo con su nivel de abstracción.

MDE [Kent 02] ofrece una visión más amplia ya que combina proceso y análisis con arquitectura, considerando que MDA se centra excesivamente en las categorías PIM y PSM y sus transformaciones y olvida muchos otros aspectos relevantes. MDE tiene en cuenta la existencia de más de un criterio para la clasificación de los modelos frente al único criterio de abstracción tecnológica de MDA (MDA admite la posibilidad de que existan más criterios pero se centra en el criterio de abstracción).

Kent considera una serie de criterios adicionales para clasificar los modelos que dan lugar a un mayor número de perspectivas que CIM, PIM y PSM. Según esto se puede considerar como que cada uno de estos criterios formaría una dimensión en un sistema n-dimensional y las perspectivas corresponderían con cada una de las intersecciones entre las distintas dimensiones. Además, considera que los criterios de clasificación de modelos se pueden agrupar a su vez en tres grupos principales. El primero estaría formado por un único criterio que, al igual que ocurre con MDA, correspondería con la abstracción respecto de la tecnología de implementación. El segundo grupo se compondría de criterios existentes en el área de desarrollo de software orientado a aspectos. En este grupo se pueden encontrar criterios como el

tema (*subject area*), que puede cubrir áreas desde el trato con los clientes hasta el procesamiento de órdenes o criterios como el aspecto, encargado del control de la concurrencia y la distribución. Una característica que destaca de este segundo grupo de criterios es que comúnmente no interesa definir transformaciones entre los modelos que ocupan distintas posiciones dentro de una misma dimensión (o criterio), sino que es más interesante definir transformaciones que tomen a todos los modelos de una dimensión como origen y como destino tengan a un único modelo integrador. Por último, en un tercer grupo, se clasifican criterios relacionados con aspectos de gestión como autoría, versión, localización y participantes en los procesos modelados.

MDE también cubre gran cantidad de aspectos relacionados con la calidad del software generado. Para esto se dispone de mecanismos, como por ejemplo validación y verificación de modelos, que serán comentados brevemente más adelante. En general éstos ayudan a mejorar la calidad de los modelos y artefactos generados a partir de ellos. En este sentido también se pueden hacer consideraciones respecto a la creación de modelos para asegurar su calidad. La calidad de los modelos empleados es muy importante, no sólo porque afecta directamente al código generado sino porque sin ella no se puede llegar a aplicar correctamente un enfoque MDE. Solheim y Neple [Solheim 06] proponen una serie de condiciones que los modelos empleados deben cumplir para asegurar, por un lado la transformabilidad de los modelos, y, por otro, la mantenibilidad de éstos y de otros artefactos generados.

Respecto a la transformabilidad se deben exigir las siguientes características a los modelos:

- **Compleción.** Todos los conceptos que se representan en los modelos deben ser de relevancia en el dominio.
- **Gramaticalidad.** El modelo debe ser conforme con la definición de su lenguaje.
- **Precisión.** El modelo debe ser suficientemente preciso y detallado para llevar a cabo una transformación automática a partir de él.
- **Relevancia.** El modelo debe contener solamente las declaraciones necesarias para una transformación particular.

Para asegurar mantenibilidad del software las características propuestas son las siguientes:

- **Trazabilidad.** Los elementos de los modelos deben poder ser trazables hacia los elementos de los modelos que los generaron y hacia los elementos de los modelos que generan. Es decir, al aplicar una transformación desde un modelo origen a un modelo destino se deben crear relaciones entre los elementos de ambos modelos indicando cómo se generan los elementos del nuevo modelo.

- **Diseño correcto.** El modelo debe tener un diseño ordenado, comprensible por humanos y transformable a un resultado ordenado y comprensible.

Las características anteriores, a excepción de la trazabilidad, están relacionadas con modelos aislados (sin considerar sus relaciones con otros modelos). Pero en una metodología MDE además hay que tener en cuenta que existen distintos modelos de un mismo sistema ofreciendo diversos niveles de abstracción o vistas del sistema. Esto implica una serie de consideraciones adicionales entre las relaciones que se deben presentar entre estos modelos. Para Mellor [Mellor 02] estos requisitos son:

- **Consistencia.** Los modelos empleados para las distintas vistas de un sistema deben ser semántica y sintácticamente compatibles entre ellos.
- **Transformación y evolución.** Un modelo debe ser semánticamente consistente con los modelos que a partir de él se obtienen.
- **Integración.** Los modelos de diferentes vistas deben ser integrados como uno solo antes de proceder a la producción de código.

3.3.2.2. Transformaciones

De acuerdo con Kleppe et al. [Kleppe 03] se puede definir el concepto de transformación entre modelos como sigue:

“Una transformación es la generación automática de un modelo destino a partir de un modelo origen, de acuerdo con una definición de la transformación. Una definición de la transformación es el conjunto de reglas de transformación que juntas describen cómo un modelo en un lenguaje origen puede ser transformado en un modelo en el lenguaje destino. Una regla de transformación es la descripción de cómo una o más construcciones en el lenguaje origen pueden ser transformadas en una o más construcciones en el lenguaje destino”.

Ésta es una definición muy interesante de transformación y de los elementos que toman parte en ella (transformación, definición de la transformación y regla de transformación). Aunque implica que las transformaciones son siempre automáticas, lo cual, según otros autores (o como el propio autor comenta más adelante en el mismo trabajo), no siempre es cierto, y también se consideran transformaciones no automatizadas o semi-automáticas.

Además, no necesariamente las transformaciones toman un modelo de origen y otro de destino, sino que es posible tomar tanto múltiples modelos de entrada como múltiples modelos de salida. Este tipo de transformaciones permiten realizar operaciones como fusión de modelos en un único modelo o generación de múltiples modelos a partir de uno sólo.

Las transformaciones se basan en el uso de metamodelos. Una definición de transformación está formada por reglas que relacionan los metamodelos de origen y de destino. Ésta tiene su propio metamodelo y las transformaciones entre dos modelos,

son a su vez modelos conformes a dicho metamodelo como se puede apreciar en la Figura 3-11.

Figura 3-11. Transformación entre modelos

Clasificación de las transformaciones

Una clasificación muy extendida de las transformaciones distingue dos tipos de acuerdo a la naturaleza de los elementos que forman el origen y el destino:

- **Transformaciones modelo a modelo (*Model-To-Model*, **M2M**)**. Tanto el origen como el destino de la transformación son modelos. Estas transformaciones pueden ser ejecutadas por un lenguaje declarativo, imperativo o híbrido:
 - o **Declarativo**: Describen qué es lo que hay que hacer sin dar detalles de cómo se va a hacer. Un lenguaje de transformación declarativo, por ejemplo, indicaría en qué elementos de los modelos de destino se transformarán los elementos del origen, pero no permitiría especificar cuáles son los algoritmos para realizar la selección de los elementos en el origen ni la construcción de los modelos de destino. En general, un lenguaje declarativo resulta adecuado para la ejecución de transformaciones ya que resultan más intuitivas. Este tipo de lenguajes se pueden descomponer en tres aproximaciones [Mens 04]:
 - *Programación funcional*. Las transformaciones se consideran como funciones que toman una entrada formada por los modelos de origen y devuelven una salida, los modelos de salida.
 - *Programación lógica*. Estos lenguajes, como *Prolog*, se basan en la definición de objetivos que a su vez se componen de cláusulas con una condición y una secuencia de operaciones asociada. Si se

cumplen todas las cláusulas de un objetivo se considera como cumplido, si no se cumplen se deshacen todas las operaciones llevadas a cabo dentro del marco de ese objetivo.

- *Transformaciones de grafos.* Mediante grafos se pueden definir gráficamente relaciones entre objetos. Se pueden usar estas relaciones para comprobar si se cumplen ciertas condiciones que dispararían la ejecución de una regla, así como para especificar cuál sería la salida de la regla.
- **Imperativo.** Una aproximación imperativa permite definir los detalles acerca de cómo se lleva a cabo la transformación, especificando un algoritmo según el cual se realiza la transformación (no sólo se dice qué se transforma sino también cómo se transforma). Puede resultar de utilidad cuando se desea ejecutar una transformación de forma incremental o cuando se desea hacer reglas mediante la composición de otras.
- **Híbrido.** Es una combinación de los anteriores. Es deseable disponer de un lenguaje que combine las ventajas de las dos aproximaciones anteriores, con reglas de transformación intuitivas y con posibilidad de composición.
- **Transformaciones modelo a texto (*Model-To-Text*, M2T).** En este caso el origen es un modelo pero el destino es texto no sujeto a ningún metamodelo. Son útiles para la generación de código y de documentación.

Mens [Mens 04] propone otra clasificación atendiendo a los lenguajes de modelado usados para describir los modelos de origen y de destino, distinguiendo entre transformaciones endógenas y exógenas. Las primeras tienen como origen y destino modelos conformes a un mismo metamodelo mientras que las segundas los modelos de origen y destino son conformes a distintos metamodelos. En otros trabajos estas transformaciones reciben nombres distintos como traslación de modelos, refinamiento y refactorización para las transformaciones endógenas y traslación de lenguajes o simplemente traslación para las exógenas.

Las transformaciones además se pueden clasificar como verticales u horizontales. En este caso una transformación vertical transforma entre modelos con distintos niveles de abstracción (o colocados a diferente nivel de acuerdo con uno de los criterios usados para la definición de perspectivas) mientras que una transformación horizontal mantiene el mismo nivel. Según Mens no hay correlación entre las clasificaciones endógena frente a exógena y vertical frente a horizontal, así que se pueden presentar las siguientes cuatro combinaciones, conocidas mediante los nombres que a continuación se indican:

- Mismo lenguaje, mismo nivel de abstracción: **refactorización**.

- Mismo lenguaje, distinto nivel de abstracción: **refinamiento**.
- Distinto lenguaje, mismo nivel de abstracción: **migración de lenguajes**.
- Distinto lenguaje, distinto nivel de abstracción: **generación de código**.

Finalmente, como última clasificación, se pueden dividir las transformaciones según la notación empleada en su representación, que puede ser gráfica o textual.

Características deseables en las transformaciones

Como se acaba de ver las características de los lenguajes y herramientas empleados en las transformaciones son muy variables. Se puede tener desde lenguajes textuales o gráficos hasta lenguajes declarativos o imperativos, en los cuales además hay diversas opciones de implementación. Sendall y Kozaczynski [Sendall 03] evalúan la validez de cada una de estas aproximaciones y destacan una serie de requisitos deseables en los lenguajes de transformación:

- **Precondiciones.** Se debe poder definir las condiciones que se deben cumplir para que se ejecute una transformación.
- **Composición.** Es un requisito deseable el poder crear nuevas reglas de transformación a partir de otras existentes y no tener que crearlas y probarlas desde cero.
- **Forma.** Una representación gráfica es la más adecuada por la ganancia cognitiva que ofrece respecto a una notación textual. En cualquier caso, es posible tener una representación con parte gráfica y parte textual.
- **Usabilidad.** Un lenguaje debe encontrar un equilibrio entre su facilidad de comprensión, precisión, concisión y facilidad de modificación, que dependerá del propósito de las transformaciones que ejecutará y de los conocimientos y preferencias de los posibles usuarios del lenguaje. En este sentido es importante la elección entre un lenguaje declarativo, que ofrece interpretación implícita y facilita la definición de reglas, y un lenguaje imperativo que puede hacer uso de secuencias, selección e iteración para la composición de reglas. Un enfoque híbrido que combine las ventajas de los dos anteriores es en algunos casos la mejor elección.

Puesto que las transformaciones son definidas y ejecutadas por herramientas, tan importantes como los requisitos exigibles a las transformaciones son los requisitos exigibles a las herramientas que los soportan. Mens cita los siguientes como requisitos no funcionales que se aplican tanto a los lenguajes como a las herramientas que los soportan:

- **Usabilidad y utilidad.** La herramienta debe ser sencilla de utilizar, facilitando el trabajo en la medida de lo posible, intuitiva y, por supuesto, debe permitir definir y ejecutar transformaciones correctamente.
- **Concisión.** La herramienta debe ser concisa y tener las menores construcciones sintácticas posibles. Pero esto puede incrementar el trabajo en transformaciones complejas, por lo que el lenguaje de transformación debe ser más extenso e introducir construcciones adicionales para manejar esta complejidad, encontrando un balance apropiado entre la concisión y la complejidad del lenguaje.
- **Escalabilidad.** El lenguaje debe permitir la ejecución de grandes transformaciones.
- **Propiedades matemáticas.** Si el lenguaje dispone de ellas es posible emplearlas para probar ciertos aspectos teóricos de las transformaciones.
- **Estandarización.** El lenguaje debe soportar los principales estándares, como por ejemplo MOF y XML.

Mientras que como requisitos funcionales Mens cita que las herramientas y lenguajes tengan habilidad para:

- Crear, actualizar y borrar transformaciones.
- Sugerir cuándo personalizar o reusar transformaciones.
- Garantizar la corrección de las transformaciones.
- Tratar con modelos incompletos o inconsistentes.
- Agrupar, componer y descomponer transformaciones.
- Testear, validar y verificar transformaciones.
- Refactorizar transformaciones.
- Especificar transformaciones bidireccionales.
- Soportar trazabilidad y propagación de cambios.

Muchos de los requisitos anteriores son usados para enriquecer y mejorar el lenguaje y herramienta de transformación. Otros como la trazabilidad pueden ser usados para desarrollar herramientas adicionales a las de transformación de modelos que den un valor añadido. La trazabilidad en las transformaciones [Palmer 97] es esencial para asegurar que los artefactos son conformes a las especificaciones del sistema, de forma que se puedan identificar qué elementos o requisitos dan lugar a

cada uno de ellos para poder localizar errores en el diseño de las transformaciones, de la especificación de la aplicación y permitir la evolución del software.

3.3.3. Aseguramiento de la calidad (*Quality Assurance*)

Además de las ventajas antes comentadas que aporta MDE como simplificación del desarrollo de aplicaciones, gestión de la complejidad y los cambios o la abstracción de la tecnología, este enfoque ofrece otras posibilidades para asegurar la calidad en los modelos y en el software desarrollado mediante la construcción de diversas herramientas que operan sobre los modelos. El uso de modelos para automatizar los mecanismos de control de calidad del software se conoce como MDQA (*Model Driven Quality Assurance*) [Mohagheghi 07]. A continuación se describen brevemente las principales posibilidades que a este respecto permite MDE.

La Validación y Verificación (V&V) del software aseguran que el producto es el que el usuario desea (validación) y que cumple con los requisitos impuestos por él (verificación). Una ventaja de aplicar un enfoque MDE es que los procesos de V&V no tienen porqué ser ejecutados sobre el software final, si no que se pueden realizar sobre los modelos que lo describen. La validación, que debe comprobar los requisitos del usuario con el modelo, en un enfoque MDE se hace comprobando el modelo frente a distintos metamodelos y restricciones. La verificación asegura el cumplimiento de diversas propiedades deseables de los modelos. Para ello, tras ser fijadas estas propiedades, se pueden realizar operaciones sobre los modelos para asegurarse que realmente se cumplen. En un análisis de este tipo, el sistema debe ser modelado formalmente mediante procesos e iteraciones. Un verificador de modelos toma tanto el anterior modelo del comportamiento como las restricciones y, mediante la aplicación de algoritmos basados en la exploración de estados, comprueba cada una de las propiedades.

Otras muchas herramientas que se pueden emplear con MDE se encargan de tareas relacionadas con la elaboración de mecanismos para el testeo de las aplicaciones sobre la plataforma de implementación. Esto último se conoce bajo el nombre de *Model-Based Testing* (MBT) [Santos 07]. Esta técnica consiste en la generación automática de casos de prueba a partir de los modelos de las aplicaciones, que serán usados para comprobar el software final. Además, en caso de detectarse errores durante la ejecución y ser localizados en el código, se puede llegar hasta los elementos de los modelos de nivel superior que dieron origen al fallo (siguiendo las trazas de las transformaciones).

3.3.4. Aproximaciones a MDE

Actualmente, la más conocida de todas las aproximaciones al Desarrollo de Software Dirigido por Modelos es MDA, sin embargo hay algunas más. En general cualquiera de éstas tiene un funcionamiento a grandes rasgos similar, tal y como se muestra en la Figura 3-12. En ellas se encuentran una serie de participantes

involucrados en su desarrollo mediante diferentes roles [Haan 09a], varios de los cuales pueden llegar a combinarse en una misma persona:

- **Experto del dominio.** Es el encargado de recopilar información acerca del dominio para el cual se van a construir las aplicaciones.
- **Ingeniero del lenguaje.** Es el encargado de diseñar el lenguaje de alto nivel de abstracción con el que se describirán las aplicaciones. Los conocimientos exigidos a éste acerca de ingeniería de lenguajes pueden variar mucho en función de la aproximación MDE por la que se opte.
- **Especialista en transformaciones.** El especialista en transformaciones se encarga de definir las transformaciones que convierten las especificaciones de cada una de las aplicaciones en código.
- **Experto de la plataforma.** Puesto que se necesita generar código para una determinada plataforma, se hace necesaria una persona que conozca la plataforma tecnológica en cuestión. Esta persona es la responsable de dar una interpretación a los modelos asociados con la plataforma de implementación en cuestión y de desarrollar un *framework* sobre la misma del que haga uso el código transformado.

La construcción de la herramienta se divide, por un lado, en la construcción de un lenguaje para expresar de forma sencilla las aplicaciones que se van a modelar, conocido como DSL (*Domain-Specific Language*) y, por otro lado, en la implementación de las transformaciones que generarán el código. El DSL se construye conjuntamente por un experto del dominio en cuestión y un ingeniero de lenguajes haciendo uso de un metalenguaje. Las transformaciones corren a cargo de un experto en transformaciones con la ayuda de un experto en la implementación de sistemas. Una vez construida la herramienta ésta será usada por un ingeniero del dominio, que describirá las aplicaciones mediante el DSL, dando lugar a especificaciones funcionales que mediante transformaciones generarán el código de la implementación.

Las principales diferencias que se encuentran entre las distintas aproximaciones son los mecanismos empleados para la definición de DSL y transformaciones. Respecto a los DSL, algunas de las aproximaciones abordan su desarrollo mediante la especificación explícita de diversos metamodelos y representaciones, mientras que otras ofrecen un entorno más amigable donde se especifican los elementos del lenguaje y sus relaciones, sin entrar en conceptos más complejos como el uso (explícito) de metamodelos. En cuanto a las transformaciones, se presentan cambios correspondientes a los lenguajes de transformación usados y a los tipos de transformaciones empleadas (distintas combinaciones de transformaciones M2M y M2T).

Figura 3-12. Participantes y flujo en una aproximación MDE genérica [Haan 09b]

A continuación se comentan algunas de las aproximaciones que en MDE se pueden adoptar.

3.3.4.1. Domain Specific Modeling

El DSM Forum [DSM Forum] se define a sí mismo como una institución independiente dedicada a difundir el conocimiento y el *know-how* acerca del desarrollo de software mediante un enfoque DSM (*Domain Specific Modeling*). El DSM eleva el nivel de abstracción por encima de la programación mediante la especificación directa de la solución mediante conceptos del dominio y, a partir de estas especificaciones, la obtención automática del código generado. Este enfoque, en comparación con un enfoque MDA basado en el uso de modelos UML (no basados en conceptos del dominio), es entre 5 y 10 veces más rápido según este foro.

Se puede considerar que el DSM define un concepto casi tan genérico como lo es MDE, con la diferencia que sólo considera la descripción de aplicaciones mediante lenguajes específicos del dominio de las aplicaciones. Dado que el concepto DSM es muy genérico, existe un gran número de herramientas que lo permiten implementar, como *MetaEdit+* [MetaEdit], *Eclipse + EMF* [EMF], *Microsoft DSL Tools* o *GME* [GME 04], que varían en la forma en que implementan tanto los lenguajes como el motor de transformación.

3.3.4.2. Model Integrated Computing

Model Integrated Computing (MIC) [Ledezci 05] es una tecnología MDE realizada por la Universidad de Vanderbilt que se soporta por la herramienta GME (*Generic Modeling Environment*) [Ledezci 01], con la que está estrechamente relacionada. Tiene su propio entorno de metamodelado, conocido como *MetaGME*, con una arquitectura de modelado de cuatro capas. En este sentido es similar a MDA, sin embargo no se basa en MOF como hace ésta, sino que usa a UML, junto con OCL, como lenguaje de

meta-meta-modelado. Además éste se usa no sólo para construir la sintaxis abstracta de los lenguajes de modelado, sino también su sintaxis concreta.

Además del modelado de aplicaciones un enfoque MIC también cuenta con otras herramientas para realizar tareas de análisis, simulación, verificación y transformación entre modelos que permite la construcción de herramientas que funcionan en el entorno de desarrollo de GME. Por contra no es compatible con MDA ni cuenta con la interoperabilidad de la que disponen otras herramientas MDA.

3.3.4.3. Software Factories

Las Software Factories [Greenfield 04] constituyen otra aproximación radicalmente distinta a MDA para el desarrollo de Líneas de Producto Software (*Software Product Lines*, SPL), es decir, de familias de aplicaciones con características similares. En este caso el desarrollo de aplicaciones se basa en el uso de esquemas (*Schemas*) y plantillas (*Templates*). Los esquemas son modelos que describen los productos software (las aplicaciones que se quieren construir), cómo se construyen y qué relaciones hay entre los productos de una misma familia. En definitiva estos esquemas dicen cómo construir las aplicaciones haciendo uso de unos determinados recursos. Una plantilla en cambio, es una colección de los recursos anteriores (frameworks, asistentes y lenguajes específicos de dominio, etc.), y puede ser usada para configurar el entorno de desarrollo para construir aplicaciones de una SPL concreta. Simplificando, se puede considerar que el esquema constituye la receta del producto, mientras que la plantilla contiene los ingredientes.

Las *Software Factories* en comparación con MDA dan una menor importancia a la independencia de la plataforma, pero se centran más en la productividad. En esta aproximación, aunque los modelos desempeñan un papel importante no lo son todo, sino que además se debe disponer de un conjunto de recursos adicionales para poder llevar a cabo el desarrollo de software. Demir [Demir 06] da una comparativa con MDA en la que destaca que las SPL son una aproximación productiva aunque es costoso desarrollar herramientas con ellas. Una de las principales ventajas que ve en las *Software Factories* es el bajo tiempo de aprendizaje por parte de los desarrolladores de aplicaciones en comparación con MDA, aunque hay que resaltar que la comparación se realiza con un enfoque MDA basado en la descripción de las aplicaciones con UML y no con lenguajes específicos del dominio.

3.3.5. Entornos de desarrollo

Existen numerosos entornos de desarrollo que se pueden emplear para construir las herramientas que permitirán el desarrollo de aplicaciones mediante MDE. A continuación se describe el formado por el entorno Eclipse en conjunto con el *framework* EMF, que sirve como base para una gran cantidad de herramientas de modelado. Se describe este entorno debido a la gran popularidad que éste tiene para modelado y a que ha sido el elegido para la realización del presente trabajo de Tesis. Existen otros entornos y aplicaciones de desarrollo populares, como por ejemplo la aplicación

comercial *MetaEdit+* [MetaEdit] que ofrece facilidades para la construcción de lenguajes específicos de dominio y generadores de código basados en la ejecución de transformaciones M2T. La principal característica de la que presume esta aplicación es la sencillez con la que se desarrollan herramientas con ella. Otra aplicación comercial es Microsoft DSL Tools integrado en Visual Studio que permite la definición de lenguajes específicos de dominio y de plantillas para la generación de código. Volviendo al código libre está el entorno GME, que como ya se comentó, forma una parte esencial en una aproximación MIC. El uso de MIC no es compatible con MDA, y la interoperabilidad queda muy reducida, por lo que ha habido esfuerzos para hacer que GME sea compatible con MOF y MDA (aunque dejaría de tratarse de un enfoque MIC) [Emerson 05]. En cualquier caso GME carece de la cantidad de herramientas MDE con las que por ejemplo cuenta Eclipse.

3.3.5.1. Eclipse Modeling Framework

Eclipse [Eclipse] es una plataforma de desarrollo libre, inicialmente impulsada por IBM, aunque en la actualidad cuenta con el apoyo de numerosas instituciones y voluntarios. En sí mismo Eclipse es simplemente un *framework* y un conjunto de servicios para el desarrollo de *plug-ins*, lo que hace de él un entorno extensible.

De entre todos sus *plug-ins* para el desarrollo de herramientas MDE destaca EMF (*Eclipse Modeling Framework*) [EMF]. Este *plug-in* proporciona un *framework* de modelado y facilidades para la construcción de herramientas y aplicaciones basadas en un modelo estructurado de datos. EMF se encarga de las funciones básicas de modelado y permite la ejecución de un gran número de *frameworks* de modelado que se basan en él.

El *plug-in* EMF se compone de tres paquetes básicos: EMF (core) que permite la definición de modelos mediante su metamodelo *Ecore* (alineado con EMOF, un subconjunto de MOF) y el soporte en tiempo de ejecución de modelos; *EMF.Edit* que ofrece clases para el desarrollo de editores de modelos y *EMF.Codegen* que permite la generación de editores de modelos.

Los servicios de modelado que EMF ofrece son utilizados por otros *frameworks*. Para la creación de lenguajes específicos de dominio se puede hacer uso de GEF (*Graphical Editing Framework*) con el que se pueden desarrollar editores gráficos avanzados (aunque su desarrollo es complejo) o GMF que permite la obtención, a partir de una serie de modelos, de editores gráficos basados en EMF y GEF.

Bajo el proyecto M2M de Eclipse se está desarrollando un *framework* para la ejecución de transformaciones entre modelos, compuesto por tres motores de transformación: ATL (*Atlas Transformation Language*), Procedural QVT (*Query/View/Transformation*) y Declarative QVT. Adicionalmente se pueden obtener otros motores de transformación como AGG para transformaciones basadas en grafos, inicialmente una herramienta externa a Eclipse y no compatible con él, pero que cuenta con el *plug-in* EMT (*EMF Model Transformation*) que lo implementa en Eclipse.

El proyecto M2T se ocupa de las transformaciones de modelos a textos, ofreciendo estándares de transformación como JET que permite la generación de código a partir de plantillas, el estándar *Acceleo* del OMG o *Xpand*. Al igual que ocurre con las transformaciones M2M se pueden obtener otros *frameworks* como por ejemplo *MOFScript* para las transformaciones a texto.

3.3.6. Aplicación de MDE a sistemas reactivos

Aunque MDE resulta muy útil para modelar procesos de negocio y sistemas de información, su ámbito de aplicación no se limita a éstos. Esta tecnología puede ser empleada en otros tipos de sistemas de computación. Una posible clasificación de estos sistemas [Riesco 02] los divide en cuatro grupos:

- **Sistemas transformacionales.** Ofrecen una serie de salidas en función de sus entradas, su comportamiento puede compararse con el de una función en un lenguaje de programación.
- **Sistemas interactivos.** Interactúan con el entorno, siendo el control realizado por un ordenador y requiriendo de sincronización.
- **Sistemas reactivos.** Interaccionan con el entorno, reaccionando a sus estímulos pero careciendo de capacidad de sincronización.
- **Sistemas en tiempo real.** Se trata de sistemas reactivos con requisitos temporales estrictos.

Los sistemas reactivos, típicamente sistemas de control y de telecomunicaciones, además tienen la característica añadida de que la respuesta ante un mismo evento dependerá de lo que con anterioridad haya ocurrido en el sistema. Por lo tanto pueden ser modelados mediante máquinas de estados. Para llevar a cabo este tipo de modelado, el grupo DSIE (División de Sistemas e Ingeniería Electrónica) de la Universidad Politécnica de Cartagena ha desarrollado un metamodelo orientado a componentes con soporte para máquinas de estados y diagramas de actividades [Alonso 08] conocido como *V3-CM*, que se pretende usar para modelar aplicaciones pertenecientes a los dominios domótico, robótico y el de las redes de sensores inalámbricas.

En un enfoque MDE este metamodelo se sitúa en un nivel PIM por su generalidad, siendo necesarios una serie de modelos PSM para cada una de las plataformas de implementación. La productividad de un enfoque MDE aumenta si además se dispone de lenguajes específicos de dominio (DSL) para especificar las aplicaciones de cada uno de estos tres dominios. En este caso, el metamodelo PIM haría de nivel intermedio entre las especificaciones de las aplicaciones y los modelos PSM.

En el campo de la domótica se dispone de experiencia muy reciente en la aplicación de MDE [Jiménez 09]. En el trabajo en cuestión se ha desarrollado un

lenguaje específico de dominio para domótica (véase la Figura 3-13), que permite describir entornos domóticos mediante conceptos relacionados con este dominio (dispositivo, servicio, escena, etc.). Los modelos generados a partir de estos conceptos domóticos se transforman a V3-CM. Actualmente se está trabajando en las transformaciones desde estos modelos a un PSM domótico de la plataforma tecnológica KNX/EIB y en sus transformaciones a código.

Figura 3-13. Modelo de iluminación con el DSL para domótica [Jiménez 09]

En el campo de la robótica se ha propuesto la utilización de MDE [Ortiz 07], aplicando el enfoque a una familia de robots teleoperados destinados a la limpieza de cascos de buques. En la Figura 3-14 se muestra un ejemplo para el control de un robot mediante V3-CM.

Figura 3-14. V3-CM. Control de un robot

Los resultados obtenidos en estos dos dominios han sido muy significativos y reconocidos en publicaciones de prestigio [Jiménez 09] [Iborra 09].

3.3.6.1. Aplicación a las redes de sensores

Existen pocas aproximaciones que apliquen un enfoque MDE para el desarrollo para redes de sensores. La herramienta GRATIS II [Volgyesi 02] desarrollada por la Universidad de Vanderbilt usando el entorno GME y tecnología MIC es una de las primeras desarrolladas. Esta herramienta cuenta con un metamodelo del lenguaje de programación nesC empleado en el sistema operativo TinyOS, lo que permite construir modelos de aplicaciones nesC y generar código a partir de éstos. Sin embargo, por un lado el código generado sólo se encarga de la estructura del software interconectando componentes existentes (véase la Figura 3-15) y por otro lado la descripción de aplicaciones se realiza al nivel de abstracción de nesC. En cualquier caso, la herramienta resulta útil como entorno de desarrollo visual, haciendo que los programadores usen elementos gráficos en lugar de instrucciones nesC para conectar componentes. Una herramienta con características similares es proporcionada por CADENA [Ranganath 03], un *framework* de modelado para sistemas basados en componentes. Este *framework* permite definir entornos de modelado basados en componentes, entre los que se encuentra uno para nesC.

Otra herramienta basada en el modelado de redes de sensores, pero con un mayor nivel de abstracción, es ATaG (*Abstract Task Graph*) [Bakshi 05]. Esta aplicación que también ha sido desarrollada con GME, permite modelar las aplicaciones basadas en flujos de datos complejos en función de una serie de tareas y de mecanismos de control descritos por el lenguaje que define. Las descripciones generadas con este lenguaje se usan para generar un código de control, siendo necesaria por parte del usuario la introducción del código correspondiente a cada una de las tareas.

Figura 3-15. Herramienta GRATIS II para redes de sensores [Volgyesi 02]

El trabajo que se presenta en esta Tesis constituye la primera aproximación conocida que, con un enfoque de desarrollo dirigido por modelos, ofrece una solución completa y automatizada para el desarrollo de software con redes de sensores. Recientemente se han realizado otras aportaciones a las redes de sensores que hacen uso de MDE. El ejemplo más representativo lo constituye *ScatterFactory2* [Saad 08] que proporciona un entorno de desarrollo para aplicaciones basadas en redes de sensores con la plataforma hardware MSB. Este entorno principalmente se centra en la realización de pruebas y la monitorización de aplicaciones WSN, teniendo como una de sus mayores aportaciones el uso de técnicas MBT (*Model Based Testing*) para la generación de casos de uso. Dispone de un lenguaje sencillo para la especificación de entornos de ejecución de las aplicaciones (Figura 3-16) y que se usa para generar los archivos *makefile* de éstas. Sin embargo, el código de las aplicaciones propiamente dichas debe ser programado manualmente.

Figura 3-16. Modelo del entorno de ejecución de dos aplicaciones con ScatterFactory2

3.4. Conclusiones y aportación a la Tesis

El uso de modelos como principal artefacto en el desarrollo de software, como propone MDE, es relativamente reciente, y sólo en los últimos años se está popularizando su uso. MDE promete ventajas como:

- Elevar el nivel de abstracción en el desarrollo de aplicaciones mediante el uso de los conceptos propios del dominio.
- Independencia de la plataforma de implementación.
- Mecanismos para la transformación automática de modelos.
- Evolución de los artefactos software desarrollados.

La propuesta MDA del OMG reincide sobre estos conceptos, proporcionando además una serie de estándares destinados a dar interoperabilidad entre las distintas herramientas y lenguajes que forman un entorno de desarrollo de software basado en modelos.

En este capítulo se han introducido conceptos básicos relativos al modelado de sistemas, incidiendo en conceptos como modelo, meta-modelo o lenguaje de modelado, así como las relaciones que existen entre éstos. Sentadas las bases del modelado, se ha entrado a analizar las principales propuestas para el desarrollo de software como MDE y MDA, la visión del OMG sobre éste. Se han explicado los principales participantes en estas tecnologías, como son modelos y transformaciones, y se han citado una serie de características deseables en ellos. Tras esto se han comentado otros estándares MDE con filosofías opuestas a MDA y se han descrito brevemente algunas de las plataformas de desarrollo MDE más populares.

Finalmente se ha hablado de la aplicación de MDE a sistemas reactivos y más en concreto a redes de sensores. En este último caso el uso de MDE es algo innovador ya que no se dispone de herramientas que cubran un proceso de desarrollo de software integral, sino que sólo se consiguen esqueletos de código incompletos a partir de descripciones realizadas con un nivel de abstracción bajo, muy cercano al comúnmente utilizado en métodos tradicionales de programación de software.

MDE es la pieza central en el desarrollo de esta Tesis. Más en concreto, el estándar MDA ha sido usado para el desarrollo de una propuesta y una herramienta que permitan el desarrollo de software para redes de sensores. A diferencia de otras herramientas existentes, en esta ocasión se pretende lograr la generación de todo el código de la implementación.

Capítulo 4

Definición de un Lenguaje Específico de Dominio

En este capítulo se define un lenguaje específico de dominio para redes de sensores inalámbricas. Un lenguaje de este tipo es fundamental en un enfoque de desarrollo basado en el uso de modelos pues permite elevar el nivel de abstracción con el que se describen las aplicaciones. A lo largo del capítulo se describen estos lenguajes, incidiendo en sus ventajas, inconvenientes y clasificaciones, así como el proceso que se sigue para su definición. Posteriormente se explica el lenguaje desarrollado para el dominio de las redes de sensores incluyendo detalles sobre su implementación.

4.1. Lenguajes Específicos de Dominio (DSL)

Un lenguaje específico de dominio (*Domain Specific Language*, DSL) es un lenguaje de programación o especificación normalmente ejecutable que ofrece, a través de notaciones y abstracciones apropiadas, potencia expresiva enfocada, y normalmente restringida, a un dominio de problema particular [Deursen 00]. Se trata de lenguajes pequeños que ofrecen un conjunto reducido de notaciones y abstracciones más cercanas al dominio del problema que al dominio de la implementación, ofreciendo un alto nivel de abstracción para el desarrollo de aplicaciones. De esta forma consiguen que los desarrolladores de software trabajen con conceptos que comprenden y que recojan sus necesidades, disponiendo de un lenguaje que contiene únicamente estos conceptos.

Tradicionalmente han existido más aproximaciones para el desarrollo de aplicaciones específicas del dominio como el uso de librerías de subrutinas que son llamadas por las aplicaciones para realizar las tareas del dominio, o como el uso de *frameworks* que invocan a los métodos de la aplicación para realizar estas tareas. Frente a estas alternativas se encuentran los DSL que ofrecen una potencia expresiva mayor. El estudio de estos DSL es un tema que actualmente se encuentra en auge gracias al importante papel que éstos desempeñan dentro de una metodología de Desarrollo de Software Dirigido por Modelos (DSDM). Sin embargo, el uso de estos lenguajes no es algo nuevo. Ya desde los años 50 se desarrollaron los primeros DSL, entre ellos el ampliamente conocido BNF (*Backus-Naur Form*) para la especificación de formalismos sintácticos. Actualmente se puede encontrar un gran número de DSL disponibles en diversos campos. Como ejemplos se pueden citar el lenguaje de marcas HTML (*HyperText Markup Language*), VHDL (*Very High Speed Integrated Circuits Hardware Description Language*) para el diseño de hardware, o SQL (*Structured Query Language*) para el acceso a bases de datos. Con estos lenguajes se puede observar claramente la propiedad que se comentó anteriormente de elevar el nivel de abstracción al desarrollar software alejando a sus usuarios de la programación pura, hasta el punto de que muchos usuarios de algunos de estos lenguajes (especialmente HTML) no se consideran a sí mismos programadores. En la Tabla 4-1 se pueden observar estos DSL y otros lenguajes junto con su correspondiente nivel definido según [Jones 96], el cual relaciona la cantidad de instrucciones lógicas que se emplean para ejecutar una misma tarea (mayor nivel indica mayor capacidad expresiva). Para poder comparar la capacidad expresiva de los programas desarrollados se han incluido en la tabla lenguajes que claramente son de propósito general como C y Ensamblador, así como el lenguaje Fortran. Este último inicialmente fue concebido como un DSL para realizar cálculos matemáticos pero ha ido evolucionando hasta convertirse en un lenguaje de propósito general.

En comparación con los Lenguajes de Propósito General (*General Purpose Languages*, GPL), que ofrecen una aproximación genérica al problema que se quiere resolver, los DSL ofrecen una solución específica. Bajo la aproximación genérica, la

programación de una aplicación puede ser una tarea costosa con la que se obtiene una solución que puede no ser la óptima. Sin embargo, a partir de estos lenguajes genéricos se pueden desarrollar gran cantidad de aplicaciones muy variadas gracias al uso de librerías que les permiten llevar a cabo tareas relativas a dominios concretos, como por ejemplo el acceso a bases de datos o la creación de interfaces gráficas. Por contra, una solución específica como la que ofrecen los DSL resulta más apropiada, si bien solamente es aplicable a un pequeño conjunto de aplicaciones pertenecientes a una misma familia.

LENGUAJE	NIVEL
C	2.50
Ensamblador	1
Excel 5	57
FORTRAN 95	4.50
HTML 2.0	20
SQL	25
VHDL	17

Tabla 4-1. Lenguajes de programación y sus niveles.

Otra alternativa al uso de los DSL la constituyen los lenguajes de modelado de propósito general, especialmente UML (*Unified Modeling Language*), ampliamente extendido y soportado por gran cantidad de herramientas. Este lenguaje ha sido comúnmente usado para realizar el modelado de sistemas que se describen en función de diagramas de clases, máquinas de estados, etc. Sin embargo, no se suele emplear con el propósito de generar código en dominios específicos. El principal problema que presenta es su complejidad debida al gran número de elementos con que cuenta el lenguaje, lo que dificulta su uso. Además, intentar describir aplicaciones de un dominio específico puede resultar inapropiado por la incapacidad de expresar ciertos conceptos del dominio en UML. Aunque éste dispone de mecanismos de extensión que pueden permitir modelar estos conceptos, éstos tampoco constituyen una solución apropiada. En UML 2.0 se usan los perfiles (*UML Profiles*) para este fin, que permiten definir nuevos elementos que representan los conceptos del dominio en cuestión. Pero surgen una serie de problemas. Por un lado la capacidad de extensión está limitada por lo que ciertos dominios no pueden llegar a ser satisfactoriamente modelados. Por otra parte, nos seguimos encontrando con un lenguaje aún muy amplio con el cual expertos del dominio sin un conocimiento profundo del lenguaje no son capaces de expresar los conceptos específicos.

Los siguientes apartados describen las ventajas e inconvenientes del uso de los DSL para poder sopesar la decisión de desarrollar un nuevo lenguaje de este tipo. A continuación se definen los elementos de que se compone y cuya definición será necesaria para crear el lenguaje. Posteriormente, se da una clasificación de los DSL para poder encuadrar el DSL que se va a desarrollar para aplicaciones basadas en redes de sensores. Por último, se dan los pasos que se deben seguir para el desarrollo de un nuevo DSL y las consideraciones que se deben tener en cuenta para asegurar que éste sea de utilidad.

4.1.1. Ventajas e inconvenientes de los DSL

Diversos artículos [Deursen 00][Wile 01] enumeran las ventajas que se obtienen al utilizar un DSL para la descripción de aplicaciones. Todos ellos coinciden en que la principal ventaja es elevar el nivel de abstracción, de forma que las descripciones de las aplicaciones puedan ser realizadas por un solo experto del dominio en lugar de por un programador o un equipo de programadores. Wile [Wile 01] destaca dos características de los DSL que dan origen a las ventajas que se obtienen de su uso. En primer lugar subraya la comprensibilidad de las especificaciones realizadas con los DSL. Estas descripciones pueden ser fácilmente interpretadas por los expertos del dominio, facilitando su escritura y reduciendo el número de errores introducidos en ellas, más relacionados con los conceptos del dominio que con conceptos de programación de los GPL. Una segunda característica notable es el hecho de que las representaciones internas de las construcciones lingüísticas de los DSL sean más adecuadas para el tipo de procesado y análisis que ejecutan los analizadores, simuladores, animadores y traductores de los lenguajes que las empleadas por otros tipos de lenguajes. Esto facilita en gran medida el desarrollo de herramientas auxiliares de los anteriores tipos adaptadas al dominio, no sólo reduciendo su tiempo de desarrollo sino también aumentando su calidad.

Teniendo en cuenta lo descrito en el párrafo anterior se pueden concretar las ventajas del uso de un DSL tal y como viene recogiendo en [Deursen 00], donde además se cita una extensa bibliografía que las describe en detalle. A continuación se enumeran las anteriores ventajas de forma resumida, añadiendo alguna más de interés:

- Los DSL permiten elevar el nivel de abstracción hasta el dominio del problema, como se comentó anteriormente. De esta forma, expertos del dominio pueden entender, validar y desarrollar programas haciendo uso del lenguaje.
- Las especificaciones creadas con estos lenguajes son concisas, autodocumentadas y pueden ser reutilizadas para diferentes propósitos, no limitándose a su ejecución.
- Se mejora la productividad, fiabilidad, mantenibilidad y portabilidad de las aplicaciones.

- Los DSL promueven la conservación y reutilización del conocimiento del dominio.
- Se pueden realizar tareas de optimización y validación a nivel de dominio.
- Los programas realizados con los DSL pueden ser probados.
- Los DSL pueden ser usados para la generación de código, permitiendo que una parte importante del proceso sea automatizada.

Por el contrario, optar por el uso de un DSL también conlleva una serie de inconvenientes que se deben tener en consideración antes de abordar su desarrollo. El principal de estos problemas lo constituye el coste de diseño, implementación y mantenimiento del nuevo lenguaje, que requiere de un importante esfuerzo y de unas mayores cualificaciones técnicas que las exigidas a un programador convencional. Este coste es aún mayor si se tiene en cuenta el desconocimiento de los mecanismos para la elaboración de un DSL por parte de desarrolladores y la falta de documentación que se encuentra en muchos de los casos. Esto se suma a la complejidad de encontrar una solución que permita describir cada aplicación del dominio concreto, más la complejidad introducida al gestionar otro tipo de problemas relacionados con el lenguaje (análisis sintáctico, control de tipos, restricciones, etc.). A estos costes por parte del desarrollador del DSL hay que sumar los costes, principalmente en términos de aprendizaje, que tendrían los usuarios del lenguaje, si bien hay que matizar que el esfuerzo para aprender a usar un DSL es considerablemente menor que el empleado en el caso de un GPL.

Resumiendo los párrafos anteriores se puede decir que una metodología basada en el uso de DSL acarrea mayores costes de arranque (por su diseño e implementación) pero simplifica y reduce los costes de producción de aplicaciones en su dominio posteriormente. Por lo tanto, para que un DSL sea rentable, deben poder generarse una gran cantidad de aplicaciones con él. Cuando los costes de usar un lenguaje ya existente (debidos a su ineficiencia para expresar conceptos de un determinado dominio) sumados al periodo de tiempo durante el que se prevea que se vayan a desarrollar aplicaciones para el dominio en cuestión, excedan ampliamente a los costes de desarrollo y popularización de un nuevo lenguaje, se puede considerar que está justificado su desarrollo.

4.1.2. Terminología y elementos de un DSL

Un lenguaje está compuesto por tres elementos fundamentales [Guizzardi 02]: la sintaxis, que trata la relación formal entre los signos del lenguaje; la semántica, que relaciona los signos del lenguaje con un significado en el mundo real; y la pragmática, dedicada a la relación de los signos y los intérpretes humanos. En el caso concreto de los lenguajes de programación la sintaxis se refiere al formato de los programas de un lenguaje, la semántica a la interpretación que se haga de su sintaxis de los programas y la pragmática a los aspectos prácticos de su uso.

La sintaxis de un lenguaje de programación está relacionada con la forma de sus programas. Una primera componente de ésta, conocida como **sintaxis abstracta**, define formalmente al lenguaje, caracterizando los elementos que lo forman y las reglas según las cuales estos se pueden combinar. Para expresarla en lenguajes de programación basados en texto tradicionalmente se ha recurrido a sentencias con notación BNF (*Backus-Naur Form*), si bien en el caso de lenguajes que usan notación gráfica se suelen usar metamodelos (y restricciones) ya que éstos ofrecen capacidades adicionales para estos fines. Gracias a la sintaxis abstracta un compilador o intérprete del lenguaje puede trabajar con él, sin embargo carece de una representación manejable por personas. De esto se encarga la **sintaxis concreta**, que cubre la forma en que los signos del lenguaje serán representados, ya sea de forma gráfica (en forma de signos) o textual (mediante caracteres). La sintaxis concreta no define un lenguaje sino que está más relacionada con el compilador/intérprete del lenguaje, de forma que un mismo lenguaje puede ser representado de distintas formas en distintos compiladores pero el lenguaje seguir siendo el mismo, manteniendo la misma sintaxis abstracta.

En cuanto a la semántica hay que distinguir entre dinámica y estática. La **semántica dinámica** define lo que las expresiones válidas del lenguaje significan. En el caso de lenguajes de programación en los que sus elementos no guardan relación con conceptos del mundo real la semántica indica reglas para la compilación o interpretación de programas. Sin embargo en el caso de un DSL, donde sí hay relación con un dominio del mundo real, indica una correspondencia entre el vocabulario del lenguaje y las instancias del mundo real. La descripción de la semántica se puede realizar en lenguaje natural, lo que no da una descripción objetiva, pero sí puede resultar de utilidad. Sin embargo, una correspondencia precisa de los elementos del DSL a un lenguaje con semántica bien definida da como resultado una semántica objetiva del lenguaje en lo que se conoce como semántica traslacional, típicamente empleada en un proceso de desarrollo DSDM. Otras posibilidades para definir la semántica son el enfoque operacional, en el que los programas del DSL son ejecutados directamente por un intérprete; el enfoque denotacional, que asigna elementos del DSL a formalismos matemáticos (como las redes de Petri o la teoría de conjuntos) y el enfoque extensional, que define la semántica del DSL como extensión de la semántica de otro lenguaje existente al que añade nuevos conceptos. Por otro lado, la **semántica estática** es la que se encarga de determinar cuáles de las expresiones del lenguaje están bien formadas. Este tipo de semántica impone una serie de reglas adicionales a las que expresa la sintaxis abstracta, normalmente expresadas en un lenguaje de restricciones como OCL.

4.1.3. Clasificación de los DSL

Los DSL se pueden clasificar de acuerdo con diversos criterios, entre los que destacan: estilo de programación, tecnologías de implementación, estructura, notación utilizada y modelo computacional.

En primer lugar se puede clasificar un DSL por su estilo de programación, declarativo frente a imperativo. En un estilo declarativo el lenguaje se usa para indicar cuál es el problema pero sin mencionar cómo resolverlo. La ventaja que se obtiene al emplearlo es que los programas resultantes son cortos y fácilmente legibles. A. Deursen cita una amplia lista de trabajos realizados en el campo de los DSL [Deursen 00]. Entre ellos el estilo declarativo resulta ser el más comúnmente empleado. Por el contrario, un estilo imperativo, que especifica claramente cómo resolver el problema, es menos frecuente pero también es compatible con el uso de DSL, dando lugar a programas no tan cortos pero rápidos y especializados.

Otras dos consideraciones importantes al desarrollar un DSL son la notación (sintaxis concreta) que se utilizará y su estructura, las cuales determinan la tecnología que deberá ser empleada para desarrollar el DSL. En esta línea K. Czarnecki hace una clasificación de los DSL [Czarnecki 05] según su tecnología de implementación y su estructura de representación de la información. En ella se presentan los DSL como textuales, empotrados en otros lenguajes de programación, gráficos y, finalmente, como asistentes y GUIs interactivos. El espectro de estructuras que los DSL pueden tener cubre desde asistentes de configuración de rutinas hasta lenguajes de grafos, pasando por otros tipos basados en árboles de decisión o diagramas de propiedades. En cuanto a la representación del lenguaje es posible tanto el uso de notaciones gráficas como textuales.

Un estudio sobre diversos DSL [Tolvanen 05] presenta otra clasificación alternativa para DSL gráficos basada en los conceptos, reglas y modelo computacional que éstos emplean. Aunque el estudio se ha realizado sobre una serie de lenguajes empleados en una metodología de desarrollo DSM (*Domain Specific Modeling*), es extrapolable a otros DSL que utilizan representación gráfica. De acuerdo con esta nueva clasificación se pueden agrupar los DSL según cuatro aproximaciones empleadas para su desarrollo. Estas aproximaciones, que no son ortogonales entre sí (por lo que un mismo DSL puede encuadrarse en varias de ellas) son las siguientes: 1) conceptos del desarrollador o del experto del dominio; 2) generación de la salida; 3) aspecto (look & feel) del sistema construido y 4) espacio de la variabilidad. En los siguientes párrafos se describe esta categorización.

En el primer grupo nos encontramos a los lenguajes basados en los conceptos del experto del dominio o conceptos del desarrollador. En la Figura 4-1 se puede ver un ejemplo de este tipo de lenguajes, en este caso aplicado al campo de los seguros. Se puede comprobar cómo los elementos del DSL están directamente relacionados con conceptos de este dominio como accidentes, robos, etc. La principal ventaja de este tipo de DSL es que elevan el nivel de abstracción por encima de los conceptos de la programación. Como consecuencia, el desarrollo de aplicaciones con lenguajes de este tipo puede ser llevado a cabo por usuarios sin ningún conocimiento de programación.

En la segunda aproximación, los lenguajes basados en la generación de la salida se definen en función de las estructuras de código que se van a generar partiendo de las descripciones con el DSL, de forma que los conceptos del DSL tienen una

correspondencia directa con cada una de estas estructuras. Un ejemplo de este tipo de lenguajes es CPL (*Call Processing Language*) [Lennox 04], usado para el control de servicios de telefonía a través de Internet, que se muestra en la Figura 4-2. La salida de este lenguaje es un archivo XML cuyo formato de datos da origen a los conceptos del DSL, siendo la representación gráfica con el DSL y la representación con XML equivalentes en cuanto a la información que contienen. La mayor dificultad para desarrollar los DSL de este tipo se encuentra en localizar conceptos apropiados para modelar el comportamiento y la lógica basados en reglas del dominio, cosa que se puede conseguir cuando las plataformas subyacentes proveen servicios a los cuales los modelos pueden ser transformados. Desafortunadamente, un DSL que se encuentra tan cercano a la salida que se va a obtener de él no eleva excesivamente el nivel de abstracción, pero sin embargo sí que permite un diseño guiado con prevención y detección temprana de errores además de facilitar la generación de código ejecutable.

Figura 4-1. DSL basado en conceptos del experto del dominio [Tolvanen 05]

El tercer enfoque, un lenguaje basado en el aspecto del sistema, se puede aplicar a productos cuyo diseño puede ser entendido viendo, tocando u oyendo, como por ejemplo un teléfono móvil. Las piezas de las que se componen estos lenguajes son principalmente elementos de un interfaz gráfico de usuario como botones, etiquetas o listas interconectados entre ellos y dependiendo del estado del programa se mostrará el que corresponda. En este tipo de lenguajes, la identificación y definición de las construcciones del lenguaje es sencilla, por lo que puede ser realizada por consultores externos con sólo una pequeña ayuda de los expertos del sistema. La mayor dificultad surge al definir los elementos del lenguaje que no se corresponden con el aspecto del

sistema. En la Figura 4-3 se muestra un ejemplo de un lenguaje para diseñar aplicaciones para teléfonos móviles.

El último tipo de lenguajes de acuerdo con esta clasificación son los basados en el espacio de la variabilidad, típicamente usados para el desarrollo de familias de productos. En este caso los conceptos del lenguaje modelan la variabilidad presente en las aplicaciones del dominio. Para llevar a cabo este enfoque se deben conocer las características comunes del dominio y las características diferenciadoras. Las descripciones realizadas por los usuarios del DSL se centrarán en seleccionar las distintas variables que diferencian unos productos de otros. Los servicios comunes suelen ser provistos por una plataforma tecnológica concreta (a nivel de implementación) y las especificaciones producidas con el DSL son normalmente usadas para llamar a los servicios de la plataforma, aunque en algún caso pueden ser usadas para generar código. El principal problema a la hora de desarrollar un DSL de este tipo es la dificultad para predecir las futuras variantes que pueden presentarse en las aplicaciones, especialmente si el lenguaje trabaja con variabilidad del comportamiento de las aplicaciones y no sólo variabilidad estática (configuración de componentes).

Figura 4-2. Lenguaje basado en la generación de la salida [Tolvanen 05]

4.1.4. Definición de un DSL

La definición de un DSL es una tarea complicada que requiere de un gran esfuerzo así como de un gran conocimiento tanto acerca del dominio de aplicación

como del desarrollo de lenguajes. Así que, antes de decidirse por el desarrollo de un DSL, es conveniente realizar un estudio sobre la viabilidad de esta opción, que demuestre que el esfuerzo adicional que se va a emplear en el desarrollo del lenguaje va a ser compensado por las ventajas que éste ofrecerá. De acuerdo con Mernik [Mernik 05] este estudio se consideraría como la primera fase en el desarrollo de un DSL, conocida como la fase de decisión. En ésta se deben enfrentar de un lado todas las dificultades y contrariedades asociadas al desarrollo de un DSL (como costes económicos y temporales) y la existencia de otros DSL para el dominio en cuestión. De otro lado se pondrán los beneficios que se espera obtener de su uso. Para evaluar estos beneficios se deben buscar una serie de patrones en el dominio que previamente han demostrado ser resueltos eficientemente por DSL en otros dominios. Ejemplos de patrones potencialmente ventajosos para un DSL son la ejecución de operaciones de análisis, verificación, optimización, paralelización, transformación, representación de datos y la realización de tareas mecánicas o que impliquen interacción con el usuario. En general, cualquier propiedad de los DSL que permita simplificar el desarrollo de aplicaciones o rebajar los conocimientos exigidos a los usuarios potenciales del DSL debe ser considerada en esta fase.

Figura 4-3. Lenguaje basado en el aspecto del sistema [Tolvanen 05]

Mernik propone tres fases adicionales en el desarrollo de un DSL y busca patrones comúnmente empleados en ellas. Estas fases son la de análisis, diseño e implementación del lenguaje. En la fase de análisis se recopila la información del

dominio tomando como entradas documentos técnicos, el conocimiento de los expertos del dominio, código fuente existente y estudios de los usuarios del dominio. La salida de esta fase dependerá de la forma en la que sea abordada, normalmente de manera informal aunque también se pueden usar metodologías formales de análisis como FODA [Kang 90] o FAST [Weiss 95]. Con éstas se puede obtener un modelo del dominio formado por una definición y terminología del dominio, una descripción de los conceptos del mismo y modelos de características del dominio. Este modelo del dominio serviría a su vez como entrada para la fase de diseño, que también puede afrontarse de manera formal e informal, optando por desarrollar un DSL desde cero o bien por extender a otro lenguaje existente. Finalmente, en la fase de implementación se distinguen diversas alternativas como el uso de compiladores, intérpretes, preprocesadores, extensión de lenguajes existentes o de compiladores.

Aunque la clasificación propuesta por Mernik encuentra un gran número de patrones presentes en el desarrollo de un DSL no propone unas líneas claras a seguir para la definición de un DSL. En cambio Tolvanen [Tolvanen 06] propone una serie de pasos sencillos para el desarrollo de DSL. Este último autor considera el desarrollo de un DSL como un proceso incremental que se repite refinando el lenguaje mediante cuatro fases:

1. Identificación de abstracciones y relaciones entre ellas.
2. Especificación de los conceptos del lenguaje y sus reglas.
3. Elaboración de la representación visual del lenguaje.
4. Creación de las herramientas.

Comparando ambas propuestas se puede ver la coincidencia de la fase 1 con la fase de análisis, las fases 2 y 3 con la de diseño y la fase 4 con la de implementación. A la hora de ejecutar estos pasos, especialmente en la segunda fase, se deben tener en cuenta una serie de consideraciones de diseño. Christensen [Christensen 03] recomienda evitar todas las decisiones que puedan restringir el uso del lenguaje, como por ejemplo puede ser ligar el DSL a una tecnología concreta de implementación (obligaría a los usuarios del DSL a aprenderla) o que haga necesarias más herramientas externas para funcionar. De igual forma siempre hay que tener en cuenta quiénes serán los usuarios del lenguaje y cuáles son sus conocimientos y habilidades para poder reducir los tiempos de formación con el DSL y los empleados en el desarrollo de aplicaciones.

Siguiendo esta misma línea, otro trabajo de S. Kolovos [Kolovos 06] sigue profundizando en el tema de los requisitos de diseño. En él se menciona que cualquier DSL debe contar con una serie de requisitos obligatorios más algún requisito adicional. Señala en primer lugar a los conceptos del lenguaje, que necesariamente deben corresponderse con los conceptos importantes del dominio, con una relación de uno a uno, es decir, cada concepto del lenguaje representa sólo a un concepto del dominio.

Por otro lado, el conjunto del lenguaje debe ser simple, pero también debe permitir mejorar la calidad de los sistemas generados. En este sentido características como fiabilidad, seguridad o rendimiento (entre otras) son las que se considera que mejoran la calidad de los productos. De igual manera, cita como requisitos fundamentales la longevidad del DSL, de forma que un gran número de aplicaciones desarrolladas con él compense el esfuerzo de su desarrollo, y el soporte del lenguaje por herramientas así como el que se facilite todo lo posible la integración con otras herramientas existentes. Otros requisitos que considera opcionales en un DSL, pero en cualquier caso deseables, son la escalabilidad del lenguaje y otra serie de requisitos como comprensibilidad, economía del espacio o accesibilidad que se engloban, de forma más genérica, en el requisito de usabilidad.

Una vez diseñado un DSL, debe ser implementado mediante la creación de un entorno de desarrollo formado por compiladores, editores, traductores y depuradores, entre otras herramientas. La dificultad de esta tarea ha sido tradicionalmente un grave inconveniente que muy frecuentemente ha hecho renunciar al desarrollo de nuevos DSL. Afortunadamente, en los últimos años han aparecido nuevas tecnologías que simplifican considerablemente el desarrollo de DSL y permiten su integración dentro de un entorno MDE. Un estudio reciente [Amyot 06] realiza una comparativa entre algunas de estas tecnologías. De todas las tecnologías analizadas destaca Eclipse, junto con sus *frameworks* EMF y GEF, como la alternativa más potente a la que el estudio anterior señala como la mejor para el desarrollo de editores serios y de potencia industrial. Cuenta con el respaldo del entorno Eclipse, lo que añade gran cantidad de funcionalidad a su editor del lenguaje y ofrece la posibilidad de integrar el lenguaje con un gran número de herramientas externas. Como contrapartida, esta tecnología es la más compleja de utilizar de entre todas las analizadas. El hecho de que recientemente haya aparecido el *framework* GMF (*Graphical Modeling Framework*) para la generación de editores gráficos basados en EMF y GEF rebaja en gran medida esta complejidad, si bien su lanzamiento aún es muy reciente y la documentación no es todo lo completa que debería.

Otra tecnología analizada es GME [GME 04] (*Generic Modeling Environment*), no tan potente como la anterior en cuanto al editor y la capacidad de modelar cualquier concepto gráficamente pero más sencilla de utilizar, lo que la hace adecuada para el prototipado sencillo de editores de modelado. Frente a la opción que ofrece Eclipse ésta tiene otra importante desventaja, y es la baja integración con otras herramientas que ofrece. El mismo trabajo también cita dos entornos de modelado que permite el desarrollo de DSL basados en UML 2.0: *Telelogic Tau G2* y *Rational Software Architect* (RSA). En ambos casos destaca la poca flexibilidad que los entornos de desarrollo ofrecen para representar los conceptos del dominio, mayor incluso en el segundo de ellos.

Las tecnologías analizadas en el citado trabajo han sido elegidas por su popularidad o su potencial técnico, pero existen otras alternativas muy extendidas como por ejemplo *Microsoft Tools for Domain Specific Languages* [DSLTools], basado en

Factorías de Software y la herramienta *MetaEdit+* [MetaEdit] que sigue una metodología de desarrollo DSM (*Domain-specific Modeling*). Éstas no siguen un enfoque MDA ya que no permiten implementar, al menos de manera directa, las transformaciones entre modelos. Sin embargo permiten realizar la implementación de un DSL y de sus transformaciones a código de forma mucho más rápida, hasta en un periodo de tiempo treinta veces menor, según los fabricantes de esta última herramienta.

4.2. *Lenguaje Específico de Dominio para WSN*

A continuación se describe el DSL que se ha desarrollado para la descripción de aplicaciones basadas en redes de sensores (WSN). Como se ha comentado en otros capítulos, este lenguaje forma el punto de partida de un proceso de desarrollo más amplio que lleva desde la definición con el DSL hasta la generación de código ejecutable. Siguiendo una metodología DSDM las descripciones realizadas con el DSL se transformarán en primer lugar al lenguaje PIM V3-CM (brevemente introducido en el capítulo 1 y que se verá más detenidamente en el capítulo 5). Desde éste se transformarán a modelos específicos de la plataforma TinyOS – nesC y finalmente se generará código ejecutable.

Con la creación de un DSL para WSN se pretenden conseguir las ventajas descritas anteriormente (véase el apartado 4.1.1). En especial se pretende simplificar el desarrollo de aplicaciones WSN gracias al uso de una notación gráfica y a la utilización de conceptos del dominio fácilmente comprensibles. Estos conceptos, como son temporizadores o el envío y recepción de mensajes, tienen un significado intuitivo y un uso sencillo que facilita la asimilación del lenguaje. Al disminuir la complejidad que requiere el desarrollo de una aplicación se rebajan las exigencias requeridas a los desarrolladores. Por lo tanto se consigue que éstos no tengan por qué ser programadores expertos a la vez que también disminuye el tiempo empleado en crear cada aplicación.

En la Figura 4-4 se puede ver una aplicación simple modelada con el DSL propuesto. En esta aplicación un grupo de nodos (identificado en la figura como *Monitoring_NodeGroup*), formado por entre uno y cien nodos, realiza cada segundo una medición de la luz que incide sobre ellos y la envía a un segundo nodo (*Receiver_NodeGroup*) conectado a un PC al que reenvía la lectura. Dado que representar gráficamente toda la información en el diagrama lo sobrecargaría en exceso, algunos de los elementos tienen información adicional que se almacena de forma externa al diagrama. Esta información aparece por defecto en la parte inferior de la pantalla en el área de propiedades (*Property*) al seleccionar el elemento en cuestión. Otra de las áreas de interés es la paleta de herramientas situada en la parte derecha. En la parte superior aparece la barra de menú y en la parte izquierda aparece un navegador de archivos que no se observa en la figura ya que se oculta automáticamente cuando es necesario disponer de más espacio para el área de edición gráfica.

Para hacerse una idea de las ventajas de usar este DSL se puede comparar esta aplicación con la que se obtendría implementándola directamente en nesC. Usando el conjunto TinyOS – nesC se deberían introducir 154 líneas de código, además de aprender a usar el lenguaje nesC, lo que no es sencillo. Debido a la sencillez de la aplicación esto resulta excesivo. En cambio, haciendo uso del DSL hay que crear 17 nodos y 7 enlaces. Además habrá que establecer el valor de 38 propiedades, 27 de ellas editables de forma gráfica sobre el diagrama y 11 seleccionables mediante listas desplegables de valores en el área de propiedades. En total se han empleado cinco minutos en la descripción de la aplicación.

Figura 4-4. Aplicación de ejemplo desarrollada con el DSL para WSN

El lenguaje tiene un enfoque predominantemente imperativo ya que una parte importante del mismo se encarga de describir las actividades ejecutadas en los nodos. La especificación gráfica de las secuencias de actividades simplifica notablemente el desarrollo de aplicaciones frente a un enfoque tradicional como el de TinyOS, además de tener transformaciones sencillas a los diagramas de actividad de V3-CM. Otra parte del lenguaje que describe las comunicaciones, muy importantes en WSN, tiene un estilo declarativo. Los componentes de este conjunto del DSL se encargan de seleccionar las características de las comunicaciones y de determinar entre qué nodos fluirá la información. La información que proporcionan será usada para seleccionar y configurar los componentes (a nivel de código) que se ejecutarán en los nodos.

En el apartado 4.1.3 se clasificaba a los DSL según la aproximación tomada en su desarrollo como: 1) conceptos del desarrollador o del experto del dominio; 2) generación de la salida; 3) aspecto (*look & feel*) del sistema construido y 4) espacio de la variabilidad. De acuerdo con esta clasificación el DSL pertenecería más claramente al segundo y primer tipo, por este orden. El DSL se considera basado en la generación de la salida ya que los conceptos que lo forman tienen una relación directa con los estructuras de código de las WSN. Ha sido de utilidad contar con el entorno TinyOS que ofrece una plataforma que implementa los conceptos del DSL. Al igual que lo hace TinyOS, otros entornos de desarrollo ofrecen conceptos y estructuras equivalentes, por lo que el DSL no queda ligado a este entorno en concreto y se posibilita la futura implementación en otras plataformas. Igualmente el DSL se puede considerar como basado en conceptos del dominio. Determinar el conjunto de conceptos del dominio WSN no es algo totalmente objetivo y puede variar según la aproximación con que se observen. Sin embargo, queda claro que conceptos como comunicaciones, temporización o adquisición de datos sí son relevantes y es en estos conceptos en los que se basan las propuestas de desarrollo para WSN, incluido el DSL implementado.

Aparte de caer bajo estas dos clasificaciones, la parte declarativa del lenguaje se puede considerar que además pertenece al cuarto tipo, el espacio de la variabilidad. Una serie de elementos del DSL se usan para modelar la variabilidad estática de las comunicaciones. Es decir, como se introdujo anteriormente, analiza las características diferenciadoras presentes en una red de comunicaciones y las utiliza para configurar la aplicación final. No se optó por modelar la variabilidad dinámica por la complejidad que esto hubiera introducido y por contar con un lenguaje V3-CM al que las transformaciones de una secuencia de actividades son más sencillas.

El resto de apartados describen el DSL realizado, comenzando por la plataforma tecnológica empleada para su desarrollo y siguiendo por los elementos que definen el lenguaje. Entre éstos se incluye la sintaxis abstracta con su correspondiente metamodelo y la sintaxis concreta con las representaciones gráficas de cada elemento del lenguaje. Una especificación rigurosa de la semántica del lenguaje no se da en este capítulo pero puede extraerse de la lectura de los capítulos posteriores, en los cuales se explica cómo las descripciones del DSL son transformadas a V3-CM (semántica traslacional). Sin embargo, en este capítulo, con la explicación de los conceptos del DSL y su metamodelo (contenidas en el apartado de sintaxis abstracta) se da una definición informal de la semántica.

4.2.1. Tecnología empleada

Para la implementación del lenguaje se ha elegido el entorno de desarrollo Eclipse junto con los *frameworks* EMF, GEF y GMF. EMF (*Eclipse Modeling Framework*) es un *framework* de modelado para construir aplicaciones basadas en un modelo estructurado de datos que da soporte a la generación de código [EMF]. Incluye un metamodelo llamado *ecore* que es un subconjunto de MOF, por lo que puede leer y escribir serializaciones de MOF (concretamente de su subconjunto EMOF). Esto

permite el intercambio de datos entre aplicaciones y facilita su uso dentro de un enfoque de desarrollo MDE. Con EMF se pueden definir metamodelos, para lo que se dispone tanto de un editor gráfico como de un editor reflexivo (*tree editor*) que han sido usados para crear el metamodelo que da soporte a la sintaxis abstracta del DSL.

Para trabajar con instancias de los metamodelos creados, EMF permite la generación de una serie de clases Java que permiten desde crear y editar modelos hasta la ejecución de una interfaz gráfica sencilla que hace uso de ellas. Sin embargo este editor es demasiado simple y la construcción de modelos con él no resulta ni intuitiva ni sencilla. Para desarrollar editores más potentes se desarrolló GEF (*Graphical Editing Framework*) [GEF] que provee una colección de herramientas para representación gráfica y edición de modelos las cuales permiten construir prácticamente cualquier aplicación. El desarrollo de aplicaciones en general, o de editores más concretamente, se hace mediante la creación de clases Java que realizan invocaciones a las librerías proporcionadas por GEF y EMF. Desafortunadamente el uso de GEF es bastante complejo.

GMF (*Generic Modeling Framework*) [GMF] pretende simplificar el desarrollo de editores de modelos. Aborda el desarrollo de éstos de forma generativa, obteniendo un editor basado en GEF y EMF a partir de una serie de modelos que lo describen. Gracias a este Framework no es necesario tener un profundo conocimiento de GEF ni tener que implementar el editor mediante clases Java. Sin embargo, aparecen otros inconvenientes como la baja configurabilidad del editor generado. El editor y sus componentes tienen un aspecto que no suele coincidir con el deseado. Aunque existen muchas opciones en GMF para modificar el aspecto de los componentes del editor, en la mayor parte de las situaciones o bien las opciones parecen no estar implementadas, o no están documentadas o simplemente no existen opciones para realizar el cambio buscado. Como consecuencia, GMF es un buen punto de partida para definir un editor para un DSL pero requiere de un esfuerzo importante modificando el editor inicial para lograr un producto final con un acabado y prestaciones adecuadas.

El desarrollo de un editor con GMF [GMF_Tutorial] implica la creación de cuatro modelos. A continuación se enumeran los modelos necesarios. El orden en que deben ser creados no tiene que coincidir con el que se expone aunque éste parece el más lógico dadas las relaciones que hay entre ellos.

1. **Modelo del dominio** (*domain model*). Este modelo (el metamodelo del lenguaje) define la sintaxis abstracta del DSL.
2. **Modelo de definición gráfica** (*graphical definition model*). Este modelo define la sintaxis concreta: elementos gráficos usados como formas geométricas correspondientes a los nodos y enlaces.
3. **Modelo de herramientas** (*tooling model*). Este modelo define una paleta de herramientas para crear cada uno de los elementos gráficos que pueden aparecer en el DSL.

4. **Modelo de correspondencias** (*mapping model*). Crea asociaciones entre los elementos de los tres modelos anteriores. Una asociación típica es la que asigna a un elemento de la paleta su correspondiente representación en la definición gráfica y su correspondiente clase (denominada EClass en ecore/EMF) en el modelo del dominio. Al seleccionar y ejecutar el elemento de la paleta se visualizaría automáticamente su representación gráfica y se crearía una instancia de la clase del modelo del dominio.

Estos modelos se pueden crear desde cero pero, salvo en el caso del modelo del dominio, una parte considerable de ellos puede ser creada con la ayuda de asistentes. El uso de estos asistentes reduce el tiempo invertido en la creación pero es propenso a la introducción de errores, por lo que los modelos deben ser validados y supervisados antes de continuar. Es importante no arrastrar errores a las siguientes fases del proceso ya que, en el peor de los casos, puede obligar a repetir la mayor parte del trabajo posterior.

Una vez están definidos los cuatro modelos anteriores se crea un quinto modelo de forma automática que es el usado para la generación del código del editor (*generator model*). A partir de este modelo generador se genera el código del editor compuesto por clases Java. Las modificaciones sobre el editor (para que tenga el aspecto deseado y para añadir nuevas funcionalidades) deben introducirse en estas clases aunque también se pueden realizar algunos cambios en el modelo generador. En la Figura 4-5 se puede ver el proceso completo de desarrollo con GMF.

Figura 4-5. Proceso de desarrollo con GMF [GMF Tutorial]

El último paso sería la ejecución del código una vez esté listo mediante el *framework* GMF Runtime. El editor se ejecuta como un nuevo plug-in de Eclipse donde se pueden crear y modificar programas con el DSL, grabándolos en dos archivos que

contienen, respectivamente, un modelo acorde al metamodelo del DSL y la información relativa a la representación gráfica de cada uno de sus elementos.

4.2.2. Sintaxis abstracta

Construir un modelo de la sintaxis abstracta es un primer paso esencial en el desarrollo de un lenguaje. La sintaxis abstracta define los elementos del lenguaje y las reglas que establecen cómo pueden ser combinados. En este apartado se establecen los conceptos del lenguaje a partir de los cuales se deriva la sintaxis abstracta y se explica el metamodelo que la define.

Las capacidades fundamentales que necesita un nodo de una WSN para ejecutar una aplicación típica son **sensorización, actuación, temporización, comunicaciones y almacenamiento**. A partir de éstas se han obtenido los conceptos que componen el lenguaje propuesto, añadiendo nuevas abstracciones para modelar agrupaciones de nodos y tratar las comunicaciones desde el punto de vista del conjunto de nodos. Estos conceptos se pueden clasificar según tres categorías. Estas categorías son:

- 1) Definición de nodos.
- 2) Comunicaciones.
- 3) Comportamiento de los nodos.

El DSL cuenta con dos conceptos para la **definición de nodos**, que permiten especificar los recursos con los que cuenta un nodo:

- ♦ **NodeDefinition**. Las definiciones de nodo indican los recursos que van a ser usados por un conjunto de nodos. Estos recursos se refieren a sensores, actuadores y demás componentes de un nodo que se conectan a las entradas y salidas de su microcontrolador. Tras crear una definición ésta debe asociarse a un grupo de nodos para que puedan acceder a los recursos que la definición contiene.

Además, una definición de nodo proporciona una especificación de los recursos que se exigen a una plataforma hardware para poder ejecutar un programa creado con el DSL. Esta definición no vincula el DSL con ninguna plataforma sino que simplemente especifica el hardware mínimo del que debe disponer una plataforma.

- ♦ **ResourceDefinition**. Una definición de nodo está formada por las definiciones de cada uno de los recursos con que cuenta el nodo. Los elementos de este tipo son los encargados de definir cada uno de los recursos disponibles. Los tipos de recursos disponibles son los siguientes:

- **Sensor.** Se encargan de la adquisición de datos. Se contempla en un principio el uso de sensores de luz, temperatura y humedad, aunque la cantidad de sensores soportados puede ser extendida fácilmente.
- **Port.** Los puertos son los encargados de las comunicaciones con dispositivos físicamente conectados a los nodos como actuadores y ordenadores personales. En este último caso el nodo que posee el puerto funcionaría como pasarela al exterior de la red y permitiría extraer datos de ella. Para la conexión se haría uso de los puertos y buses del microcontrolador: UART (principalmente usada para la comunicación con un PC), I2C (para control de dispositivos) y las entradas y salidas digitales.
- **Storage.** Proporciona almacenamiento temporal o permanente a los datos que manejan los nodos. Asimismo, también permite la definición y el uso de estructuras de datos más complejas como colas FIFO y tablas *hash*. Estas últimas son tablas donde cada uno de sus elementos relaciona una pareja de valores, resultan útiles para almacenar datos recibidos de los nodos vecinos, asociando a cada nodo una lectura de datos. Los dos tipos de datos anteriores, además de permitir la introducción y extracción de valores, permiten realizar operaciones sobre los datos que contienen como inicialización o la obtención de valores máximos, mínimos o medios. En el caso de las tablas *hash* tras realizar las operaciones se puede acceder a cualquiera de los dos datos que relacionan (conocidos como llave y valor).
- **ConfigParam.** Los parámetros de configuración controlan el funcionamiento de los nodos con parámetros de los nodos como el ciclo de trabajo de la radio, la potencia de transmisión y el estado de la batería (sólo lectura).

Para cada uno de estos tipos de *ResourceDefinition* además se pueden usar componentes previamente existentes en las plataformas a las que se realiza la transformación de código. En este caso, las transformaciones generan un esqueleto de código que hay que completar con el código que implementa el *ResourceDefinition*.

Los elementos *ResourceDefinition* no son directamente usables en la especificación del comportamiento de los nodos. Por lo tanto, para poder acceder a ellos se, creará un nuevo elemento de tipo *Resource* (explicado más adelante) en la descripción del comportamiento el cual hará referencia al *ResourceDefinition* en cuestión. *ResourceDefinition* actuará como la definición (declaración) del recurso y *Resource* como la instancia.

El uso de las **comunicaciones** en el DSL se basa por un lado en la especificación de las características de las comunicaciones y, por otro, su uso en los diagramas que describen el funcionamiento de las aplicaciones. Para la especificación de las características se dispone de los siguientes elementos en el DSL:

♦ **Network.** Las redes definen las propiedades de las comunicaciones inalámbricas entre nodos. Cada red que se cree tendrá unas propiedades distintas. Al enviar mensajes se podría elegir una de las redes creadas para poder transmitir usando las propiedades de la red seleccionada. Estos elementos controlan los siguientes parámetros:

- **Mecanismo de propagación de mensajes.** Es el principal parámetro ya que desempeña un papel muy importante en el ámbito de las WSN. De este parámetro dependen directamente el consumo de energía y la alcanzabilidad de los nodos en la red. Se permiten establecer comunicaciones *broadcast* (de uno a muchos), punto a punto (de uno a uno), por inundación y *multihop*. Los dos primeros son los mecanismos más sencillos, pero los que menor alcanzabilidad ofrecen, sólo pudiendo recibir mensajes los nodos dentro del área de visión directa del emisor. Un mecanismo de inundación da mayores garantías de recepción ya que todos los nodos que reciben el mensaje lo van retransmitiendo, pero aumenta considerablemente el consumo de energía. La última opción del DSL emplea un esquema multisalto en el que al establecerse la red se crean rutas entre cada nodo origen y un único nodo destino que funcionaría de sumidero o pasarela con el exterior.
- **Encriptación de las comunicaciones.** Se puede especificar si las comunicaciones precisan de encriptación o si no es necesario.
- **Presencia de nodos móviles.** Se debe indicar esta opción si se espera que alguno de los nodos de la red (*Network*) cambie su ubicación física.
- **Agregación de datos.** Se pueden agregar los datos que llegan de otros nodos en un solo mensaje. De esta forma el consumo de energía es menor que si se emplea un mensaje distinto para cada nodo.

Las elecciones realizadas para cada red definida se usarán para la elección de componentes que formarán el subsistema radio de la implementación final.

♦ **WirelessLink.** Los enlaces inalámbricos definen qué nodos actuarán como origen de las comunicaciones y qué nodos actuarán como destino. Los *WirelessLink* actuarán como una tubería que conecta ambos extremos y lo que se manda por un extremo será recibido en el otro. Sólo se especifican los extremos, si es necesario usar nodos intermedios para alcanzar el destino las rutas serán fijadas en tiempo de ejecución (si las comunicaciones usan una red *multihop*). Las propiedades de las comunicaciones vendrán dadas por un *Network*, por lo tanto, para cada *WirelessLink*, se debe especificar qué *Network* se empleará.

En el caso de que un nodo tenga necesidad de enviar más de un tipo de mensajes (cada tipo conteniendo datos distintos) a un mismo destino deberá

crear un enlace inalámbrico para cada tipo. Esto es necesario para asegurar que el destino sea capaz de interpretar correctamente los datos.

El intercambio de datos se realiza mediante otro concepto denominado *CommUnit*:

- ♦ **CommUnit.** Se trata de un elemento que puede ser configurado para enviar o recibir mensajes vía radio. En ambos casos se deberán asociar al elemento *CommUnit* una serie de datos que formarán el mensaje intercambiado. En el caso de que una misma *CommUnit* mande más de un dato simultáneamente se deben numerar los datos que se pasan a ella. En la *CommUnit* que ejecuta el receptor de los mensajes los datos deben seguir la misma numeración para asegurar que se identifica correctamente cada dato y no se confunden entre ellos.

Para especificar el destino de los mensajes se debe asociar la tarea *CommUnit* con un *WirelessLink*. Esto automáticamente también establece las propiedades de las comunicaciones puesto que los elementos *WirelessLink* se encuentran asociados a una red (*Network*).

El **comportamiento** de los nodos se basa en la existencia de uno o varios flujos de ejecución compuestos por unidades funcionales que ejecutan tareas y que acceden a recursos. Estas tareas se encuentran interconectadas entre sí por enlaces direccionales de forma que cuando una acaba de ejecutarse comienza la ejecución de la siguiente. Como entradas y salidas hacen uso de instancias de recursos que están relacionadas con la definición de los recursos antes expuesta. Los elementos que describen el comportamiento son los siguientes:

- ♦ **FunctionalUnit (FU).** Las unidades funcionales son las encargadas de llevar a cabo las operaciones de los nodos. Normalmente se asocian con uno o varios recursos que actúan como sus entradas o salidas. Temporización y comunicaciones son un tipo especial de unidades funcionales con propiedades especiales. Otras unidades funcionales se utilizan para la adquisición de datos, comunicación directa con otros dispositivos físicos, ejecución de cálculos simples o inicio de un flujo de actividades. Al igual que ocurría con los elementos *ResourceDefinition* existe la posibilidad de crear FU definidas por el usuario, dejando la introducción del código que ejecutan para el último paso del proceso de desarrollo MDE.

Las aplicaciones que se desarrollan para WSN son normalmente sencillas y las tareas ejecutadas en elementos *FunctionalUnit* normalmente requieren un sólo dato como parámetro. En este caso basta con unir el *Resource* asociado al dato a la *FunctionalUnit*. Sin embargo si se precisa usar una *FunctionalUnit* que tome varios parámetros de entrada o varios parámetros de salida habrá que numerar cada una de las entradas y salidas. Con esto se conseguirá que ésta sea capaz de identificar cada uno de los parámetros. Actualmente las *CommUnit* son las únicas unidades funcionales donde se debe tener esta consideración.

- ◆ **OrthogonalFunctionality.** Para simplificar las transformaciones desde el DSL, se ha añadido el concepto *OrthogonalFunctionality* que agrupa unidades funcionales que no guarden relación entre sí. Navegando a través de los enlaces entre unidades funcionales de un mismo *OrthogonalFunctionality* se debe poder acceder al resto de FU que lo forman y nunca a las FU que pertenecen a otros elementos de tipo *OrthogonalFunctionality*.
- ◆ **Resource.** La ejecución de algunas FU implica el acceso a recursos. Como se comentó anteriormente, los recursos se definen en elementos de tipo *ResourceDefinition* asociados a un elemento *NodeDefinition*. Para favorecer la reutilización de estos elementos dentro de una misma aplicación así como su representación en el diagrama, se optó por separar estos *ResourceDefinition* del comportamiento de los nodos. Se pensó para ello en un elemento *Resource* que se comporte como una instancia de uno de los *ResourceDefinition*. Para acceder a un *ResourceDefinition* entonces lo que se debe hacer es crear un nuevo *Resource* y asociarle el correspondiente *ResourceDefinition*.
- ◆ **Timer.** Se encargan de realizar la temporización de las tareas, ejecutando las mismas tras acabar la tarea de temporización. Es posible utilizar dos tipos distintos de temporizadores. Los primeros se programan para saltar una sola vez mientras que el segundo tipo se ejecuta periódicamente, volviendo a reiniciarse la cuenta cada vez que el temporizador vence.

Estos elementos son una parte del comportamiento de los nodos. El conjunto de todo el comportamiento viene definido en una nueva abstracción:

- ◆ **NodeGroup.** Son agrupaciones de nodos que tienen el mismo comportamiento y por tanto ejecutan la misma imagen de código. Este comportamiento está formado por elementos de tipo *OrthogonalFunctionality* que a su vez contienen unidades funcionales, recursos y enlaces entre ellos. El número de nodos que forman parte de un *NodeGroup* debe estar comprendido entre un valor mínimo y un valor máximo que especifica el usuario del DSL. En el rango de nodos definido se entiende que la aplicación que se diseña funcionaría correctamente.

El concepto *NodeGroup*, además de la función de agrupamiento, encierra también la descripción del comportamiento de los nodos y se asocia con un elemento *NodeDefinition*, el cual indica qué requerimientos de hardware son necesarios para ejecutar el comportamiento de los nodos.

Finalmente existe otra abstracción que se usa para agrupamiento:

- ◆ **Region.** Agrupa los nodos según criterios arbitrarios que quedan a elección del desarrollador, por ejemplo pueden indicar agrupaciones físicas de nodos cercanos o agrupaciones lógicas según determinadas funcionalidades. El fin de este concepto es mejorar la comprensibilidad de los diagramas realizados.

Resumiendo, se pueden desarrollar aplicaciones en tres fases las cuales se pueden realizar o bien en paralelo de forma incremental o bien una a continuación de la otra. Estas fases son:

- 1) Definición de los nodos (requerimientos hardware mínimos de los nodos).
- 2) Creación de las redes de comunicaciones.
- 3) Especificación del comportamiento de los nodos.

La primera de estas fases es la creación de un elemento *NodeDefinition* con todos los *ResourceDefinition* que se prevea que serán usados por un conjunto de nodos similares. Como se dijo, cuando se realicen las transformaciones del DSL y se genere una imagen de código, ésta hará uso de estos recursos y por tanto se necesitará una plataforma hardware que cuente con ellos.

En la segunda fase se creará un elemento *Network* por cada red de comunicaciones que se vaya a emplear. Esta red se configurará con las características antes especificadas para este elemento. La creación de elementos *WirelessLink* debe esperar hasta que se definan elementos *NodeGroup* que puedan interconectar.

En la tercera fase el comportamiento de los nodos se expresa en los *NodeGroup*, pero previamente a la creación de elementos de este tipo se necesita crear elementos *Region* que los contengan. A cada *NodeGroup* creado se le deberá asignar una definición de nodo creada en la primera fase.

Posteriormente se debe asignar a cada *NodeGroup* uno o más *OrthogonalFunctionality*, donde cada uno de ellos contiene una serie de *FunctionalUnit*. En cada *OrthogonalFunctionality* debe haber una FU especial que indicará el comienzo de la secuencia de operaciones de esa *OrthogonalFunctionality*. Esta secuencia estará formada por distintas FU enlazadas entre sí, indicando la dirección de estos enlaces el orden de ejecución temporal (tras ejecutarse la FU origen se ejecuta la FU destino). Las FU que necesiten acceder a algún recurso lo harán mediante la conexión de elementos de tipo *Resource* (instancias de recursos) con referencias a los elementos *ResourceDefinition* definidos en la definición de nodo.

Los *NodeGroup* además de contener el comportamiento de los nodos también se utilizan para establecer el sentido de las comunicaciones: nodos origen de las comunicaciones y nodos destino de ellas. Para esto, una vez creados los *NodeGroup* correspondientes se deben unir mediante un *WirelessLink*.

Como se ha comentado, no tiene porqué comenzar con fase tras acabar la anterior. Ejecutarlas en paralelo puede ser útil cuando en un principio no se sabe qué recursos (*ResourceDefinition*) van a ser necesarios y de esta forma pueden ser definidos a medida que sean necesarios en la especificación del comportamiento. Lo mismo ocurre con el uso de redes de comunicaciones (*Network*). En cualquier caso, la única

restricción existente es que tanto los elementos de tipo *ResourceDefinition* como los de tipo *Network* deben existir antes de ser referenciados por *Resource* y *WirelessLink* respectivamente, lo cual es bastante obvio. De la misma manera los elementos *WirelessLink* deben existir previamente a sus referencias desde los elementos *CommUnit*.

A partir de estos conceptos y de la descripción del DSL se debe crear un modelo de la sintaxis abstracta. La más técnica comúnmente empleada para la definición de este modelo en lenguajes gráficos es el uso de metamodelos. Un metamodelo define conceptos y relaciones entre ellos. A partir de cada aplicación descrita con el DSL se generará un nuevo modelo que será instancia de este metamodelo y que será el que se pase a las herramientas de transformación. En la Figura 4-6 se muestra el metamodelo desarrollado para el DSL mediante EMF.

Figura 4-6. Metamodelo del DSL

Consultando los conceptos que definen el DSL se puede entender fácilmente la mayor parte del metamodelo. Sin embargo algunas clases y atributos necesitan alguna aclaración adicional.

En general, se puede ver que hay una serie de atributos que se repiten en la mayoría de las clases. Estos son el nombre, el tipo y la descripción. El atributo nombre se representa como un atributo *name* o por alguna denominación acabada en “ID”. El tipo de cada objeto (determinadas clases tienen clasificaciones asociadas, por ejemplo un objeto *TimerUnit* puede ser o bien un temporizador continuo o de una única temporización) se representa con un nombre derivado de *type* y la descripción por *description*. Esta descripción incluye información textual sobre el elemento al que pertenece. Comúnmente esta información la proporciona el autor del diagrama para clarificar el propósito del elemento, aunque en algún caso muy concreto puede tener una finalidad adicional.

Inspeccionando el metamodelo se puede observar cómo se representan los enlaces entre elementos. Para unir elementos de tipo *FunctionalUnit* entre sí se utiliza *UnitLink*, mientras que los enlaces que conectan *FunctionalUnit* y *Resource* se realizan mediante elementos *ResourceLink*, los cuales poseen un atributo denominado *paramNum* para numerar los *Resource* usados en las *FunctionalUnit* (como se comentó, sólo en casos puntuales es necesario su uso).

Algunas clases además tienen atributos que hacen uso de nuevos tipos de datos (enumerados) definidos en el metamodelo. El nombre de estos atributos acaba en “Type” y se usan para describir que tipo de *FunctionalUnit*, *Timer* o *ResourceDefinition* se empleará. La clase *FunctionalUnit* es la que más posibles variantes contiene. Según el valor de su atributo *type* se pueden tener los siguientes tipos de unidades funcionales:

- START. Señala que comienza el flujo de ejecución en una *OrthogonalFunctionality*.
- CALCULATE_EXPRESSION. Contiene una expresión matemática sencilla con operadores aritméticos (se soportan +, -, *, /, %), de manejo de bits (&, |, ^, <<, >>, ~) y valores constantes. Para referirse a variables de entrada y salida habrá que usar su identificador (atributo *resource_ID* de *ResourceDefinition*). Las expresiones que se introducen son del tipo “*id1* + *id2* = *id3*”. Se pueden introducir varias expresiones separadas por “;”.
- READ. Extrae datos de un recurso (variable, puerto, parámetro de configuración o sensor) para que sean usados en la unidad funcional conectada a su salida (esta FU está unida a la unidad funcional READ mediante un *UnitLink* de la cual es el destino). El recurso de donde se lee el dato se debe conectar como entrada de la unidad funcional (mediante un *ResourceLink*).

En el caso de que el recurso del que se lee sea una cola FIFO (para lo cual su definición de recurso debe de ser de tipo *Storage* y contener un número de elementos mayor que uno), el dato se extrae de dicha cola.

Si por el contrario el dato se ha definido como una tabla *hash* (su definición de recurso es de tipo *Storage* y además su propiedad *isHashTable* está marcada como *true*) entonces se lee el valor asociado a una determinada llave. La llave debe estar almacenada en un recurso de almacenamiento que se conecta a la unidad funcional como entrada (mediante un *ResourceLink* con su *paramNum* igual a 101).

- WRITE. Almacena datos en un recurso (variable, puerto o parámetro de configuración). El dato que se almacena proviene de la unidad funcional conectada a la entrada (origen de un *UnitLink* que tenga como destino a la unidad funcional WRITE) y es almacenado en un *Resource* conectado como salida a la unidad funcional (mediante un *ResourceLink*). Este *Resource* de salida debe contener una referencia a la definición de recurso del nodo que se desea utilizar.

En el caso de que la definición de recurso se haya definido como cola FIFO (lo cual se hace indicando un número de elementos mayor a 1) el dato se introduce en la cola FIFO. Si por el contrario la definición de recurso se ha definido como tabla *hash* (marcando como *true* la propiedad *isHashTable* de la definición de recurso e indicando el tamaño de la tabla) el dato se almacena en la tabla *hash* como un valor asociado a una llave. La llave se debe especificar mediante la lectura de un recurso, el cual se debe conectar como entrada a la unidad funcional (mediante un *ResourceLink* con su *paramNum* igual a 101 (*key*)).

- SEND_MSG_RADIO. Es un tipo especial de *CommUnit* que manda un mensaje por la radio. Los datos del mensaje se obtienen de los recursos de entrada conectados a la unidad funcional y se envía por el enlace inalámbrico asociado a la *CommUnit*.
- RECEIVE_MSG_RADIO. Es un tipo especial de *CommUnit* que recibe un mensaje por la radio se envía proveniente del enlace inalámbrico asociado a la *CommUnit*. Los datos del mensaje se guardan en los recursos de salida conectados a la unidad funcional. Si se quiere leer la potencia recibida se debe numerar el *ResourceLink* que conecta al recurso donde se almacenará la potencia como 100 (*Pow*).
- CONDITIONAL_UNIT. Contiene expresiones condicionales sencillas con operadores relacionales (>, <, ==, >=, <=, !=), operadores aritméticos (+, -, *, /, %), y valores constantes. Para referirse a variables de entrada habrá que usar su identificador (atributo *resource_ID* de *Resource*) para formar expresiones del tipo "*id1* > *id2*". Tras evaluarse la condición de forma positiva se ejecuta la *FunctionalUnit* conectada a la salida de la *CONDITIONAL_UNIT* mediante un

UnitLink numerado como 1 (*true*). En caso de no cumplirse la condición se ejecutará la *FunctionalUnit* conectada mediante un *UnitLink* numerado como 0 (*false*).

- SYNCHRONIZATION. Sincroniza a varias unidades funcionales que se ejecutan antes que ella. Esta unidad funcional queda bloqueada hasta que todas ellas acaban su ejecución. Aún no se encuentra implementada.
- INTERCEPT_MSG. En redes *multihop* un nodo puede usar esta unidad funcional para capturar mensajes no destinados a él. Se incluye esta unidad funcional para que en un futuro se puedan implementar algoritmos de comunicaciones más avanzados mediante el DSL.
- USER_DEFINED. Esta unidad funcional se usa para poder emplear implementaciones previamente existentes en la aplicación. Se pueden crear unidades funcionales genéricas, con tantas entradas y salidas como se desee (definiéndolas en el diagrama), pero sin ningún código asociado. Tras ejecutar las transformaciones se debe introducir el código.
- AVAILABLE_DATA. Se trata de una unidad funcional para crear aplicaciones avanzadas en las que la escritura sobre una variable activa un nuevo flujo de ejecución. Una utilidad de esta unidad es que se pueden programar por separado la adquisición de datos y el envío de datos sin necesidad de que ambas partes sean controladas por temporizadores de la misma frecuencia (temporizadores de distinta frecuencia pueden provocar que se transmitan datos que aún no se han leído o que se sobrescriban datos que aún no se han mandado). Esta unidad funcional aún no está implementada.
- RADIO_TO_UART. Retransmite todos los mensajes recibidos vía radio a la UART. Esto es útil para comunicar una WSN con un PC. Se debe conectar una unidad funcional de este tipo a la entrada de la *CommUnit* que se encargue de la recepción de mensajes.
- UART_TO_RADIO. Retransmite los mensajes recibidos desde la UART a la radio. A su salida se debe conectar una *CommUnit* que se encargue del envío de mensajes.
- RESET. Resetea un recurso cargando el valor 0 en él. En el caso de recursos como colas FIFO o tablas *hash* los vacía.
- SET1. Activa un recurso cargando 1 en todos sus bits.
- MAX. Devuelve el valor máximo de un conjunto. Cuando la unidad funcional tiene conectada (como entrada) una cola FIFO devuelve el valor máximo almacenado en ella. En caso de tratarse de una tabla *hash* se pasa el valor máximo a la unidad funcional conectada como salida que esté numerada

(mediante el atributo *paramNum* del *UnitLink*) como 1 (por defecto) y pasa el valor de la llave asociada a la unidad funcional numerada como 101 (*key*).

- MIN. Devuelve el valor mínimo de un conjunto. Se usa igual que las unidades funcionales de tipo MAX.
- MEAN. Devuelve el valor medio de un conjunto de datos (cola FIFO o tabla *hash*) que se conecta como recurso de entrada.
- WRITE_MEAN. Introduce un valor en una tabla hash (conectada como recurso de entrada) de forma que, para la llave en cuestión, se almacena el valor medio de todos recibidos desde la última vez que se reseteó la tabla. La llave se introduce de la misma forma que se hace con las unidades funcionales WRITE.

Cuando posteriormente se describa una aplicación con el DSL los valores de los atributos del metamodelo serán establecidos o bien gráficamente sobre el diagrama o bien en la pestaña de propiedades de la parte inferior dependiendo del atributo en cuestión. Dado que algunos de estos atributos cuentan con un gran número de variantes, en una primera fase, no se implementarán transformaciones en algunas ellas, sino que se dará soporte al desarrollo de las aplicaciones que más típicamente se emplean en redes de sensores. El hecho de que en la versión actual del DSL se añadan todas ellas se debe a que su adición al metamodelo en fases posteriores podría hacer necesaria una redefinición del editor del DSL y, aparte del trabajo que esto implicaría, podría invalidar los diagramas realizados con él hasta ese momento.

4.2.3. Sintaxis concreta

Se ha elegido una sintaxis concreta gráfica para la representación de los elementos del DSL. A cada uno de los elementos de la sintaxis abstracta definidos en el metamodelo como clases (no abstractas) se ha hecho corresponder una representación gráfica. Los elementos presentes en la sintaxis concreta se pueden clasificar como nodos, enlaces y propiedades. La mayor parte de las clases definidas en el metamodelo se corresponde con nodos, algunos de los cuales pueden actuar como contenedores de otros nodos. Otras clases como *WirelessLink*, *UnitLink* y *ResourceLink* se corresponden con enlaces gráficos representados mediante distintos tipos de flecha. En el caso particular de *ResourceLink* hay una correspondencia con dos representaciones gráficas usadas para representar si el recurso asociado funciona como entrada o como salida a una unidad funcional. El último tipo de elementos de la sintaxis concreta son las propiedades, y éstas se relacionan con los atributos de las clases. Hay dos formas de representar estas propiedades. Una primera forma es darles una representación gráfica que acompañe a la del nodo a la que se asocian. Dado que hacer esto para todas las propiedades sobrecargaría los diagramas realizados con el DSL, algunas de ellas pueden ser editadas en el área de propiedades que se encuentra en la parte inferior del editor del DSL. Esta constituye la segunda forma de representar las propiedades.

Las representaciones de los nodos del DSL son las que se muestran en la Tabla 4-2. La primera columna se usa para indicar las relaciones de contención que hay entre los elementos del DSL.

Contención	Representación	Elemento representado
Editor. Contiene un área de edición de diagramas, paleta de componentes, vista de propiedades y navegador de archivos.
	Region. Agrupa nodos según criterios establecidos por el usuario del DSL.
	NodeGroup. Agrupa nodos (número mínimo y máximo especificados) según comportamiento. En su interior se encuentra el comportamiento.
	OrthogonalFunctionality. Contiene secuencias de ejecución.
	FunctionalUnit. Unidad funcional.
	TimerUnit. Temporizador. Muestra el valor de temporización en milisegundos y el tipo (temporización continua vs única temporización).
	Resource. Accede a un recurso definido mediante un <i>ResourceDefinition</i> (se selecciona en la vista de propiedades)
	NodeDefinition. Definición de los recursos de un nodo.
	ConfigParam. Parámetro de configuración del nodo.
	Port. Puerto de entrada/salida.
	Sensor. Adquisición de datos.
	Storage. Almacenamiento de variables.
	Network. Red de comunicaciones. En la vista de propiedades se editan sus parámetros.

Tabla 4-2. Representación gráfica de los nodos en el DSL para WSN

En cuanto a los enlaces, todos son representados por flechas. *UnitLink* es usado para unir elementos de tipo *FunctionalUnit*, *CommUnit* y *TimerUnit* entre sí mediante un estilo de flecha continuo. *ResourceLink* también hace uso de este estilo de flecha para interconectar elementos *FunctionalUnit* y *CommUnit* con recursos. Se pueden seleccionar dos tipos de flecha según si los recursos son de entrada o de salida. Además, en la vista de propiedades se puede asignar una numeración a los recursos (como se comentó anteriormente esto no suele ser necesario). En el caso de *WirelessLink*, para establecer un enlace de comunicaciones entre dos elementos *NodeGroup*, se usan flechas discontinuas de mayor grosor sobre las cuales se indica el nombre del enlace.

4.3. Conclusiones y aportación a la Tesis

A lo largo de este capítulo se han descrito las principales características de los lenguajes específicos de dominio y se ha definido uno para la descripción de aplicaciones para redes de sensores inalámbricas. Este DSL permite, en primer lugar, simplificar la construcción de aplicaciones mediante el uso de elementos con una representación gráfica intuitiva. En segundo lugar, el tiempo empleado en el aprendizaje del lenguaje es considerablemente menor que el empleado en aprender a programar las redes de sensores mediante lenguajes de programación textuales de menor nivel de abstracción.

La descripción de aplicaciones mediante el DSL definido en este capítulo es el primer paso para la generación de aplicaciones siguiendo el enfoque de Desarrollo de Software Dirigido por Modelos que se ha empleado en esta Tesis. A partir de estas descripciones se obtienen los modelos PIM de mayor nivel de abstracción que serán pasados a las transformaciones con el fin de obtener código ejecutable.

Capítulo 5

Desarrollo de una herramienta MDE para la construcción de aplicaciones WSN

Este capítulo detalla la propuesta que se ha elaborado para llevar a cabo el proceso de transformación desde las descripciones realizadas con el DSL a código ejecutable y que ha sido implementada por la herramienta MoteGen. Esta propuesta se basa en MDA y en el uso de modelos en tres niveles de abstracción. A lo largo del capítulo se detalla cómo se han creado los distintos niveles, exponiendo sus correspondientes metamodelos, y cómo se han realizado las transformaciones entre ellos, comentando además las estrategias y decisiones que han sido tomadas en la elaboración y aplicación de éstas.

5.1. Introducción

De acuerdo con lo visto en los capítulos anteriores, resulta de utilidad aplicar un enfoque de Desarrollo de Software Dirigido por Modelos en el desarrollo de aplicaciones con redes de sensores inalámbricas. En este capítulo se describe cómo se ha llevado a cabo esto, haciendo uso del DSL definido en el capítulo 4 para la descripción de aplicaciones y generando código mediante transformación de modelos. Para este propósito se ha empleado la arquitectura de modelos de MDA y sus conceptos, adaptándolos en algunos de los casos como se verá más adelante.

Los objetivos perseguidos con la aplicación de este enfoque de desarrollo son:

- Elevar el nivel de abstracción en la especificación de aplicaciones WSN. De esta forma no sólo se simplifica el desarrollo de estas aplicaciones y el tiempo destinado a su aprendizaje, si no que se amplía el conjunto de usuarios potenciales del lenguaje a aquellos con menos conocimientos de programación.
- Crear un entorno de desarrollo que sirva como base para la posterior adición de herramientas auxiliares no centradas directamente en el desarrollo de código sino en la mejora de la calidad del software generado. En este sentido, la ejecución de actividades de Verificación y Validación (V&V), simulación y depuración de código resultan de interés.
- Facilitar la integración de los artefactos generados con los obtenidos por otras herramientas similares que se aplican en dominios como la robótica, la visión artificial o la domótica. Para esto se usará un metamodelo en uno de los niveles inferiores de MDA compartido por las distintas herramientas desarrolladas para cada dominio y que posibilita trabajos futuros para integrarlos.

Como MDA propone, la definición de una aplicación consiste en su especificación en un lenguaje de alto nivel de abstracción, su transformación a otros modelos de menor abstracción y, finalmente, la obtención de código a partir de ellos. Siguiendo esta filosofía se han empleado tres niveles donde encuadrar los distintos modelos según su abstracción, cada uno con su correspondiente metamodelo. En el nivel superior se encuentran modelos PIM de alto nivel de abstracción (se pueden llegar a considerar como CIM, lo cual es discutible ya que no queda clara la definición de modelo CIM según el estándar MDA). El DSL desarrollado (explicado en el capítulo 4) permite la construcción de modelos que se sitúan en este nivel de abstracción. En un nivel de abstracción más bajo se dispone de un PIM orientado a componentes, V3-CM, que describe el funcionamiento de las aplicaciones usando máquinas de estados y diagramas de actividad y que sirve como enlace con aplicaciones de otros sistemas reactivos. Por último, se dispone de un nivel que se corresponde con el lenguaje nesC, usado en la plataforma de ejecución para redes de sensores TinyOS. El desarrollo de una aplicación software comienza por tanto con la especificación mediante el DSL y su

transformación posterior a V3-CM, al PSM de nesC y, finalmente, a código ejecutable. En la Figura 5-1 se muestra la propuesta.

Figura 5-1. Propuesta de desarrollo WSN

Para el desarrollo de la herramienta ha sido necesaria la construcción, en primer lugar, del metamodelo correspondiente al PIM para WSN el cual define la sintaxis abstracta del DSL para redes de sensores. Posteriormente se especificó la sintaxis concreta del lenguaje mediante el *plug-in* GMF de Eclipse. De forma paralela se ha desarrollado un nuevo metamodelo para el lenguaje nesC que se encuentra en el nivel PSM. Para este último nivel no ha sido necesaria la creación de una sintaxis concreta pues no se necesita disponer de un lenguaje (gráfico) para la especificación de aplicaciones a este nivel, que serán transformadas desde los niveles superiores.

Se podría haber hecho uso únicamente de estos dos niveles para la construcción de la herramienta, prescindiendo del nivel intermedio entre ellos. Sin embargo se optó por la inclusión de un lenguaje de modelado de propósito general que se encontrara conceptualmente entre ambos. De esta forma, aunque sean necesarias dos transformaciones intermedias en lugar de una, cada una de ellas es más sencilla. Para este nuevo nivel, aunque se contaba con alternativas como UML [UML 07] o SysML [SysML 06], éstas eran demasiado complejas. El lenguaje V3-CM elegido, desarrollado en el grupo de investigación DSIE, y su metamodelo resultan más sencillos y manejan un número de conceptos menor, facilitando la implementación de las transformaciones. Además, este lenguaje está siendo usado dentro del grupo en enfoques de desarrollo de software MDE para domótica [Jiménez 09] y robótica [Alonso 08], por lo que se abre la posibilidad de, en un futuro, integrar estas distintas tecnologías. En este sentido, se puede disponer de una serie de componentes de las diferentes tecnologías que pueden llegar a ser conectados entre sí. En la parte izquierda de la Figura 5-2 se puede apreciar cómo se realiza la transformación para redes de sensores haciendo uso de V3-CM como nivel intermedio, mientras que la reutilización de V3-CM para distintos dominios de aplicaciones se observa en la parte derecha.

Figura 5-2. Uso de V3-CM como nivel intermedio

Para desarrollar la herramienta *MoteGen* se ha empleado el entorno de desarrollo Eclipse y diversas tecnologías basadas en el *framework* de modelado EMF descrito en capítulos anteriores. En los siguientes apartados se describen cada uno de estos niveles

con más detalle así como la forma en la que se han afrontado cada una de las transformaciones entre ellos y las tecnologías que se han empleado.

En el Anexo II se muestra otra aproximación alternativa para el desarrollo de aplicaciones para redes de sensores. Esta aproximación, a diferencia de la empleada en este trabajo de Tesis, se basa en la transformación directa desde las descripciones realizadas con el DSL a código. Por lo tanto se prescinde de las transformaciones entre modelos, buscando simplicidad en la implementación de una herramienta (para la generación de código) frente a otras características deseables.

5.2. Modelos empleados

5.2.1. Nivel PIM – WSN

El nivel PIM es el nivel con mayor abstracción de entre todos los empleados. Las especificaciones realizadas con el DSL desarrollado dan lugar a modelos que se sitúan en este nivel y que forman el punto de partida del proceso de desarrollo de software. Este nivel cuenta con un metamodelo (Figura 5-3) que define la sintaxis abstracta del lenguaje de modelado con el que se describen las aplicaciones. Este metamodelo y el lenguaje asociado se explicaron en el capítulo 4. En él, una aplicación tiene un elemento *Root* de tipo *WSNApplication* que contiene elementos de tipo:

- *Network*. Especifican las propiedades de las comunicaciones (los elementos relativos a las comunicaciones se muestran en la Figura 5-3 en color morado y azul marino).
- *NodeDefinition*. Agrupan recursos hardware (color marrón) como sensores, parámetros de configuración, puertos de entrada/salida y almacenamiento en memoria RAM.
- *WirelessLink*. Establecen un enlace de comunicaciones entre nodos de forma que cuando un nodo que actúe como origen de un *WirelessLink* mande un mensaje se recibirá en el nodo (o nodos) que actúen como destino (color morado).
- *NodeGroup*. Se usan para describir el comportamiento de un grupo de nodos (color azul). Cada comportamiento se asigna a un grupo de nodos donde todos los nodos tienen un comportamiento similar.

Los elementos *NodeGroup* son los más complejos. En ellos, el comportamiento se describe en función de unidades funcionales (*FunctionalUnit*) que se interconectan entre sí mediante enlaces direccionales que determinan su orden de ejecución (y en el cual los datos fluyen). Estas unidades funcionales toman un recurso de entrada o salida sobre el que actuar y sobre el que realizar una tarea (por ejemplo, lectura o escritura) que viene indicada en su atributo *Type*. Además, se dispone de dos clases especiales de unidades funcionales que se usan para temporización (*TimerUnit*) y para las comunicaciones vía radio (*CommUnit*). Estos últimos elementos permiten la selección

de una red de comunicaciones (*Network*) y de los nodos receptores de los mensajes enviados (mediante los elementos *WirelessLink*).

Figura 5-3. Metamodelo PIM-WSN

5.2.2. Nivel PIM – V3-CM

V3-CM es un lenguaje de modelado de arquitecturas basado en componentes que permite la especificación tanto de la estructura de componentes de una aplicación como del comportamiento de éstos. Este lenguaje es una simplificación de UML 2.0, que resulta excesivamente complejo. De UML 2.0 se han tomado una serie de elementos concretos que han sido ligeramente modificados. Gracias a la cercanía entre ambos lenguajes de modelado, una transformación de V3-CM a UML 2.0 es relativamente directa.

V3-CM define tres vistas para describir un sistema. La primera de ellas, la vista estructural, describe la arquitectura del sistema en función de componentes e interconexiones entre ellos. El comportamiento de estos componentes se define en las otras dos vistas: la vista de coordinación que permite el modelado de máquinas de estados y la vista de flujo de datos que describe los diagramas de actividades que se ejecutan para cada estado de la vista anterior.

En la Figura 5-4 se puede ver el metamodelo de V3-CM empleado. Para la vista estructural, mostrada en color azul, se emplean componentes complejos, que contienen e interconectan a otros componentes; componentes simples que tienen una máquina de estados para definir su comportamiento y componentes COTS (*Comercial Off-The-Shelf*) pre-existentes. Los componentes complejos en V3-CM no disponen de una máquina de estados sino que su comportamiento se obtiene a partir del comportamiento de sus componentes internos. La interacción entre componentes se realiza mediante el uso de puertos, por los cuales cada componente expone su funcionalidad al resto. Estos puertos pueden proveer interfaces de forma que otros componentes puedan ejecutar los servicios que provee el componente o bien requerir interfaces (y usar los servicios de otro componente). Finalmente, para que dos componentes se puedan comunicar, deben tener un enlace entre dos de sus puertos que sean compatibles (lo serán cuando los interfaces requeridos en uno de ellos sean provistos en el otro y viceversa).

La vista de coordinación (color verde) hace uso de máquinas de estados simples, que disponen de estados, transiciones entre ellos y actividades asociadas a los estados. Los estados y las transiciones tienen a su vez diagramas de actividad asociados (vista de flujo de datos en color marrón) formados por la interconexión de distintas actividades. Estas actividades se interconectan entre ellas mediante flujos de control y sus parámetros se conectan con parámetros de otras actividades mediante flujos de datos. Entre los tipos de actividades que emplean se encuentran las actividades definidas por el usuario (que definen un código específico para realizar una tarea), las llamadas a servicio que invocan servicios en otros componentes, las condicionales, y las actividades constantes, que se usan para cargar un valor constante en un parámetro.

Figura 5-4. Metamodelo V3-CM

5.2.3. Nivel PSM – nesC

Aunque una transformación directa de los modelos V3-CM a código es posible, resulta excesivamente compleja ya que no hay una relación directa entre los conceptos V3-CM y código nesC (descrito en el capítulo 2). Por este motivo, en el nivel más bajo de abstracción se introducen los modelos específicos de la plataforma nesC (PSM nesC). Este nivel cuenta con su metamodelo (mostrado en la Figura 5-5), que presenta algunas similitudes con el metamodelo de V3-CM, ya que contiene la misma información que éste más la información relativa a la plataforma nesC. Al igual que en V3-CM, se dispone de componentes simples y complejos y los servicios de éstos se exponen a través de interfaces. En lugar de puertos se dispone de instancias de interfaz que son muy similares a los puertos. La diferencia entre ambos es que una instancia de interfaz usa o provee una única interfaz mientras un puerto puede hacer uso de varias. De forma similar a lo que ocurría en las conexiones entre puertos V3-CM, las conexiones entre componentes nesC se realizan entre instancias de interfaces compatibles (un componente debe tener un instancia que provea una interfaz mientras que otro debe tener una instancia que la use).

Una diferencia importante entre V3-CM y el PSM de nesC se encuentra en la forma en la que ofrecen los servicios. V3-CM considera que las interfaces son unidireccionales, siendo el componente que requiere el servicio el único que puede realizar invocaciones. Además los servicios se ejecutan de forma síncrona, de forma que tras realizarse una invocación a un servicio éste se considera ejecutado y se puede disponer de los datos que devuelve. En cambio nesC considera interfaces bidireccionales, formadas por comandos y eventos, donde los comandos fluyen desde el componente que usa la instancia de la interfaz hacia el que la provee y los eventos en sentido contrario. La ejecución de los servicios puede ser tanto síncrona como asíncrona. En caso de hacerse de forma síncrona se invoca un comando que no devuelve el control hasta la finalización del servicio. En el caso de servicios asíncronos la invocación se realiza mediante un comando que finaliza su ejecución antes de que el servicio comience a ejecutarse. Una vez el servicio se haya completado, el componente que provee el servicio lo notifica al que lo solicitó mediante un evento que además contiene los datos de salida del servicio (si existen).

La diferencia más notable entre V3-CM y nesC se tiene a la hora de expresar el comportamiento. Mientras que V3-CM hace uso de máquinas de estados y diagramas de actividades, el metamodelo nesC se basa en la ejecución de comandos, eventos y tareas que expresan su comportamiento mediante la interconexión de bloques de código que ejecutan comportamientos básicos (llamadas a comandos de otros componentes, manipulación de datos, etc.).

Al igual que ocurre en V3-CM, estos bloques básicos, que se corresponden con sus actividades, también disponen de enlaces de control y datos con otros bloques. Los enlaces además tienen un tipo que determinará cómo se transforman a código. En el caso de los enlaces de control, un tipo *Flow* indica que una actividad se ejecuta a

continuación de la otra; un tipo *Call* indica una llamada a una tarea o una función *nesC*; y los tipos *IfTrue* y *IfFalse* se reservan para el caso de condicionales.

Figura 5-5. Metamodelo nesC

En el caso de los enlaces de datos entre los parámetros de dos bloques de código distintos, su importancia radica en que determinan cómo los parámetros de los bloques se transforman a variables. Cada parámetro (punto de intercambio de datos de un bloque de código del PSM-nesC) se corresponde necesariamente con una variable nesC, pero, para no recargar en exceso el código nesC generado, se da la opción de que distintos parámetros se transformen a una única variable. Esto se hace con enlaces de datos cuyo tipo (*linkType*) es *SameVar*, donde la variable generada para el parámetro destino del enlace es la misma que se generó para el parámetro origen del enlace. Por el contrario, un enlace de tipo *Copy* indica que se usará una variable distinta para cada parámetro, y adicionalmente se creará una sentencia de asignación (*var2 = var1;*) en nesC que copiará el valor de la variable origen en la variable destino. Para comparar las dos formas de asignar las variables según qué tipo de enlace de datos se use se muestran la Figura 5-6 y la Figura 5-7. En la primera de ellas se usan enlaces de datos de tipo *SameVar* y se puede ver cómo los parámetros del modelo conectados por un enlace de datos se transforman a una misma variable nesC (se usa el mismo color tanto para la variable como para los parámetros) que tiene el mismo nombre que el parámetro origen del enlace. Por ejemplo, se ve cómo tanto el parámetro *msg* del bloque *InitMsg* como el parámetro *msg2* del bloque *Fill* se transforman a una única variable llamada *msg*. La segunda figura indica cómo se generaría el código si los enlaces fueran todos del tipo *Copy*, generando una variable distinta por cada parámetro (por tanto se tienen el doble de variables). Se puede apreciar el ahorro de líneas de código al usar enlaces *SameVar* (aunque no siempre es posible usarlos, por ejemplo al asignar parámetros que se encuentran en comandos o eventos distintos).

5.3. Transformaciones entre modelos

Se han empleado los niveles de abstracción definidos por el estándar MDA y, para la realización de las transformaciones entre ellos, se ha consultado el mismo. Sin embargo, MDA no da unas guías claras para realizar estas transformaciones, careciendo en ocasiones del suficiente nivel de detalle para poder tener una interpretación no ambigua de los conceptos que propone. Para algunos de estos conceptos se ha recurrido a interpretaciones propias de estos conceptos y en algunos casos a interpretaciones ligeramente distintas a las propuestas por MDA.

Para la implementación de las transformaciones se ha recurrido a la búsqueda de patrones en el modelo de origen para transformarlos en otros patrones predefinidos en un nuevo modelo destino. Estos patrones pueden involucrar desde un único elemento del modelo origen de un tipo concreto, hasta varios elementos del modelo con unas determinadas relaciones. En general, esto es suficiente para la transformación de la mayor parte de los elementos de un modelo y así se han realizado las transformaciones desde el nivel PIM de mayor abstracción al PIM V3-CM (que respecto a éste puede ser considerado como un PSM orientado a componentes).

Figura 5-6. Código para los enlaces de datos de tipo *SameVar*

```
TOS_MsgPtr msg;  
TOS_MsgPtr msg2;  
int output;  
int input;  
int paramNum;  
....  
output = RAM;  
input = output;  
paramNum = 1;  
msg2=msg;  
msg2[paramNum-1] = input;
```

Figura 5-7. Código para los enlaces de datos de tipo *Copy*

Al realizar las transformaciones al PSM de nesC hay que tener en cuenta ciertos detalles adicionales. En primer lugar, algunos elementos del modelo nesC representarán componentes preexistentes (disponibles con TinyOS o desarrollados específicamente para la herramienta *MoteGen*). Es necesario que estos elementos del modelo representen fielmente a estos componentes, ya que a partir de los elementos modelados deben generarse componentes idénticos a ellos (o bien deben usarse los componentes ya existentes). Por esto se ha creado un **modelo de la plataforma TinyOS/nesc** que contiene todos estos componentes predefinidos. De acuerdo con la especificación de MDA, un modelo de la plataforma provee un conjunto de conceptos

técnicos que representan los diferentes tipos de partes que componen una plataforma y los servicios que ésta ofrece. Además un modelo de la plataforma debe indicar la forma en que una aplicación hace uso de la plataforma que abstrae.

MDA no da ninguna información adicional con respecto a los modelos de la plataforma, con la excepción de mencionar la importancia de contar con modelos detallados de la plataforma, expresados por ejemplo en UML, y guardados en un almacén MOF. Por lo tanto, no se aclara la forma en que este modelo de la plataforma participa en el desarrollo de software. En este trabajo el propósito de este modelo de la plataforma es el de ser usado como entrada a las transformaciones, de forma que los modelos nesC resultantes hagan uso de estos elementos predefinidos en ellos incorporándolos a los modelos de salida (en lugar de crear nuevos componentes con sus mismas características). Esto evita que las transformaciones tengan que construir estos elementos del modelo desde cero y asegura que sean totalmente compatibles con los componentes que representan (tienen el mismo nombre e interfaces provistas y usadas). Además, como se verá a continuación, estos modelos pueden ser usados para guiar las transformaciones, concretamente en la generación y uso de los componentes de la plataforma.

El hecho de decidir qué elemento del modelo V3-CM se debe transformar en qué componente del modelo de la plataforma constituye otro reto. Por ejemplo, un componente V3-CM para comunicaciones radio deberá ser transformado en el componente *GenericComm* del modelo de la plataforma TinyOS/nesc. Para realizar la transformación se podría hacer uso de los nombres de los componentes V3-CM, ya que el metamodelo V3-CM no da ninguna otra información de los componentes que los pueda diferenciar. Sin embargo esto obligaría a establecer los nombres de los componentes V3-CM en función de los nombres de los componentes de la plataforma nesc, por lo que los modelos dejarían de ser independientes de la plataforma. La solución que MDA propone es el uso de **marcas**. Estas marcas, son asignadas por un arquitecto a los elementos del modelo PIM y cada una de ellas representa a otro elemento del PSM, de forma que ese elemento origen se transforme en el elemento PSM que indica la marca. Ya que las marcas no son independientes de la plataforma no se consideran parte del PIM. Su uso implica la existencia de un nuevo modelo adicional, un modelo PIM marcado que es el que se pasaría a la transformación (Figura 5-8).

Para la realización de este trabajo de Tesis se ha hecho uso de una visión particular del uso de las marcas de MDA para la transformación entre los modelos V3-CM y nesc que posibilita el desarrollo de reglas de transformación más sencillas y genéricas. Por el contrario, el uso de las marcas tal y como propone el estándar MDA, que además no da detalles sobre su uso, limita la automatización del proceso ya que implica la adición de estas marcas manualmente. Otro inconveniente es el que estas marcas no podrían ser reutilizadas para otras plataformas de ejecución ya que representan conceptos de una plataforma concreta.

Figura 5-8. Uso de marcas en las transformaciones MDA [MDA 03]

A diferencia de lo que MDA propone, lo que se ha hecho ha sido marcar tanto el modelo V3-CM como el modelo de la plataforma nesC. Las marcas se han aplicado sobre componentes, interfaces, servicios y parámetros, pudiendo establecerse relaciones entre ellas. Para transformar un elemento V3-CM en su correspondiente elemento (o elementos) nesC se deben buscar patrones similares entre las marcas de los dos modelos. Por ejemplo, un componente encargado de las comunicaciones radio en V3-CM con una determinada marca se transforma al componente del modelo de plataforma que tenga una marca similar (véase parte superior de la Figura 5-9). Este ejemplo es bastante sencillo pero se pueden realizar transformaciones más complejas haciendo uso de las marcas en las se asigne un servicio V3-CM a más de un servicio nesC, o que un mismo interfaz V3-CM, dependiendo de cuál sea el componente que lo posee (y sus marcas), se transforme a diferentes interfaces nesC. Este último caso se puede ver en la parte inferior de la Figura 5-9 donde se muestran las relaciones entre marcas que hacen que la interfaz V3S-CM *GetI* de los sensores se transforme en una interfaz nesC del tipo *ADC*, la cual se usa para la lectura de conversores analógico-digitales. Sin embargo esta misma interfaz V3-CM en el caso de la lectura de un parámetro de configuración se transforma a la interfaz nesC *GetInt*.

Al utilizar las marcas de esta forma el modelo origen queda marcado con conceptos genéricos válidos para cualquier plataforma (*Network*, *Control*, etc.) y las equivalencias de los elementos origen marcados y los elementos del modelo de la plataforma se establecen mediante marcas en ambos modelos. De esta forma las reglas de transformación se pueden programar de forma genérica, con una única regla para transformar todos los elementos de un mismo tipo (componentes, interfaces, etc.) y mediante marcas en el modelo de la plataforma se especifica cómo se va a transformar cada elemento del origen de acuerdo a su rol (se entiende que el rol se asigna mediante marcas en el modelo V3-CM).

MDA propone que las marcas sean introducidas manualmente por un arquitecto software. Sin embargo, la información necesaria para establecer estas marcas en los modelos V3-CM se encuentra en los modelos creados en el DSL, por lo que estas marcas pueden ser introducidas automáticamente en la transformación a V3-CM. Tras ésta se dispone de un modelo marcado, por lo que si se quisiera obtener un modelo V3-CM no marcado habría que programar una transformación adicional muy sencilla que borrara las marcas del modelo.

Figura 5-9. Uso de las marcas en las transformaciones de la herramienta *MoteGen*

Como lenguaje para la implementación de las transformaciones se ha elegido AGG (*the Attributed Graph Grammar system*), un lenguaje visual que ofrece una aproximación algebraica a las transformaciones utilizando gramática de grafos [Taentzer 05]. Se trata de un lenguaje de transformación de grafos, por lo tanto declarativo. Sus principales ventajas son su simplicidad y su corto periodo de aprendizaje. Además existe una versión disponible para el entorno Eclipse que funciona sobre el framework de modelado EMF conocida como EMT (*EMF Model Transformation framework*).

El lenguaje AGG se basa en la definición de reglas de transformación compuestas por tres bloques. El primero de estos bloques, el LHS (*Left Hand Side*) representa las precondiciones que se deben cumplir en un modelo para que se ejecute la regla. El

resultado de la aplicación de la regla se describe en el RHS (*Right Hand Side*), en el que se crean nuevos objetos en los modelos y se modifican los existentes. El último bloque lo forman las condiciones de aplicación negativa (NAC) que buscan patrones en los modelos de forma que si se encuentran elementos que los cumplen las reglas no se ejecutarán sobre éstos. En la Figura 5-10 se ve un ejemplo de cómo se define una regla con AGG. Esta regla hace uso de tres modelos conformes a tres metamodelos distintos. Para facilitar su comprensión se ha usado la parte izquierda de los bloques LHS y RHS para el modelo de entrada (*V3-CM*), la parte derecha para el modelo de salida (*nesC*) y la parte central para un modelo auxiliar que relaciona los dos anteriores. En este caso la regla se usa para transformar los parámetros de salida de las actividades *V3-CM* en variables de un evento *nesC*. Las precondiciones se pueden ver en el LHS. Destaca que para que se pueda ejecutar la regla previamente debe haberse ejecutado otra regla, identificada por el elemento *ACTNES*, que crea un evento a partir de una actividad. Los elementos del LHS deben ser asociados a elementos del RHS o de lo contrario serán borrados del modelo. Estas asociaciones indican que ambos elementos de las dos partes de la regla hacen referencia a un mismo objeto de los modelos y se representan usando el mismo color o numeración para los dos elementos. Además, en el RHS hay elementos que no existen en el LHS, lo que indica que serán creados tras ejecutar la regla. Uno de estos elementos, del tipo *APAR2VAR*, indica que se ha ejecutado la regla actual, relacionando el parámetro de actividad del modelo origen con la variable del modelo destino. Este elemento puede ser utilizado en el campo NAC para evitar que la regla se vuelva a ejecutar de nuevo sobre el mismo parámetro.

Como se ha visto, el número de modelos de entrada y de salida a las transformaciones no se limita a uno para la entrada y otro para la salida sino que puede ser arbitrario. En este caso se ha tomado, además de los modelos *PIM* y *PSM* y el modelo de la plataforma (cuando es necesario), un modelo adicional para guardar las relaciones que existen entre los nuevos elementos que se han generado en el modelo destino y los elementos del modelo origen desde los que se transforman. En el ejemplo mostrado en la Figura 5-10 los elementos *ACTNES* y *APAR2VAR* pertenecen a este modelo, que cuenta con su propio metamodelo. Gracias a esto se dota al proceso de desarrollo de **trazabilidad** y se hace posible, para cualquier elemento perteneciente a los modelos generados, averiguar cuál de los elementos de los modelos de mayor nivel de abstracción ha dado lugar a él.

Aunque el lenguaje AGG es muy intuitivo y fácil de usar presenta algunos problemas. El principal ha sido la dificultad para depurar las transformaciones. Otros problemas han sido la imposibilidad para componer reglas de transformación a partir de otras ya existentes así como otros problemas relacionados con el entorno (imposibilidad de copiar y pegar objetos y problemas de rendimiento al crear las transformaciones) y con ciertos fallos en la implementación del motor de transformación.

Figura 5-10. Regla AGG

5.3.1. Transformación PIM WSN – V3-CM

A partir de un modelo WSN se generan una serie de componentes V3-CM con una arquitectura similar. Por cada grupo de nodos (*NodeGroup*) del modelo WSN se obtiene una arquitectura de componentes V3-CM como la que se muestra en la Figura 5-11, que representa el comportamiento de un grupo de nodos y puede ser usada para obtener código para ellos. Estas arquitecturas tienen un componente de más alto nivel (no están contenidos en ningún otro componente) que es el que posteriormente dará lugar a cada una de las imágenes de código (sub-aplicaciones) que se cargarán en los nodes. Al ser componentes complejos, este comportamiento viene descrito por sus componentes internos y no por máquinas de estados. Internamente cada uno de ellos está constituido por un componente encargado de arrancar la ejecución de un nodo (o grupo de nodos) y otro componente que encierra el comportamiento de ese nodo, ambos conectados por una interfaz de control. El motivo de aislar la lógica de la aplicación en un componente es posibilitar su reutilización en otras aplicaciones si se necesitaran de nodos con este comportamiento. A su vez este componente está formado por una serie de componentes COTS que se encargan del manejo de los recursos del nodo (comunicaciones, temporizadores, sensores, etc.) y un único componente que actúa como coordinador realizando el procesamiento y las llamadas a los servicios del resto de componentes.

En el coordinador se crea una máquina de estados donde, en el estado *StartX*, se introducen las actividades que se ejecutarán en el grupo de nodos en cuestión. Dentro de este estado se crea un diagrama de actividades donde las actividades son transformadas a partir de las unidades funcionales del modelo origen, correspondiéndose cada una de estas unidades con una o varias actividades según el tipo de unidad funcional. Las actividades que se emplean en el modelo de salida son de los tipos que *V3-CM* contempla: llamadas a servicios, condicionales, constantes y actividades definidas por el usuario.

Figura 5-11. Arquitectura V3-CM generada para cada grupo de nodos

Las actividades definidas por el usuario (elemento *UserDefinedActivity* del metamodelo V3-CM) se emplean para indicar un determinado tipo de procesamiento sobre los datos. El tipo de procesamiento se especifica a través del campo *source* de estas actividades. Cuando, en la siguiente transformación de modelos, estos elementos se conviertan en sus correspondientes elementos PSM de nesC, el valor de este campo debe mantenerse inalterado pues permitirá que en la transformación a texto se seleccione el código nesC correspondiente a la actividad. Estas actividades, además, tienen parámetros y su correcta interpretación constituye el principal reto al generar código para las actividades definidas por el usuario. Para simplificar esta tarea se han asignado los nombres de los parámetros de las actividades de forma que permitan aclarar cuál es el rol que tienen en la actividad. Se ha tomado como convención asignar como nombre de los parámetros una cadena de caracteres formada por la concatenación del rol de la variable más el símbolo '\$' y un número que asegura que ningún otro parámetro tenga el mismo nombre. Un ejemplo de nombre de variable podría ser *"input\$3"*.

A continuación se muestran los distintos tipos de actividades definidas por el usuario que se pueden generar tras la transformación a V3-CM así como sus parámetros asociados. Puesto que toda esta información seguirá inalterada tras la siguiente transformación al PSM-nesC, la transformación de este PSM a texto debe soportar la generación de código para todos estos tipos de actividades.

- *Store*. Almacena un valor (parámetro con el rol *input*) en una variable. El nombre de la actividad representa a la variable en la que se almacena el valor y sigue la misma convención para establecer su nombre que los parámetros.
- *Retrieve*. Extrae un valor de una variable. El nombre de la actividad representa a la variable que almacena el valor y el valor obtenido de ésta se almacena en un parámetro con el rol *output*.

- *InitMsg*. Crea un mensaje para trabajar con él. El mensaje se puede acceder a través de un parámetro con el rol *msg*.
- *Fill*. Introduce un valor en una determinada posición de un mensaje. El mensaje se identifica por un parámetro con el rol *msg*, la posición del mensaje en la que se introduce el valor por un parámetro con el rol *paramNum* y el valor introducido por un parámetro *input*.
- *Extract*. Extrae un valor de una determinada posición de un mensaje. El mensaje se identifica por un parámetro con el rol *msg*, la posición del mensaje en la que se introduce el valor por un parámetro con el rol *paramNum* y el valor introducido por un parámetro *extractData*.
- *GetId*. Obtiene el identificador de red de un nodo. El valor obtenido se devuelve en un parámetro con el rol *NodeID*.
- *Calculate*. Calcula una expresión aritmética. La expresión se almacena en el nombre de la actividad.
- *ListRetrieve*. Extrae un valor de una variable de tipo cola FIFO. El nombre de la actividad representa a la cola que almacena el valor y el valor obtenido de ésta se almacena en un parámetro con el rol *output*.
- *ListStore*. Almacena un valor (parámetro con el rol *input*) en una cola FIFO. El nombre de la actividad representa a la cola en la que se almacena el valor y sigue la misma convención para establecer su nombre que los parámetros.
- *ListMean / HashMean*. Calcula el valor medio de todos los valores de una cola FIFO o tabla *hash*. El valor se almacena en un parámetro con el rol *output*.
- *ListReset / HashReset*. Resetea una cola FIFO o tabla *hash*. Tras esto la variable en cuestión queda vacía.
- *ListMaxRetrieve / ListMinRetrieve*. Devuelve el valor máximo o mínimo de una cola FIFO en un parámetro con rol *output*.
- *HashMaxRetrieve / HashMinRetrieve*. Devuelve el valor máximo o mínimo de una tabla *hash* en su parámetro con rol *output* y la llave asociada a éste en un parámetro de rol *_Key_output*.
- *HashRetrieve*. Lee el valor asociado a una llave (parámetro con rol *_index*) y lo devuelve en un parámetro con rol *output*.
- *HashStore / HashMeanStore*. Introduce un valor (parámetro con rol *input*) asociado a una llave (parámetro con rol *_index*) en una tabla *hash*. *HashMeanStore* almacena la media de todos los valores introducidos desde su último reseteo para la llave en cuestión.

En determinados casos las reglas asociadas pueden llegar a generar gran cantidad de actividades y por lo tanto ser muy grandes y difíciles de manejar, por lo que en algunas reglas se han empleado mecanismos para reducir el tamaño de las reglas (sin embargo no se emplearon en todas y aún se pueden encontrar reglas de transformación excesivamente sobrecargadas). Para disminuir el tamaño de la reglas y facilitar su construcción se ha realizado la composición de reglas mediante el uso de elementos auxiliares que asociados a los elementos del metamodelo de entrada hacen que se pueda ejecutar varias reglas de transformación sobre ellos e interconectar los elementos generados por cada una de la forma que se especifique. El uso de estos elementos ha mejorado también, aunque muy ligeramente, la facilidad de depuración de las transformaciones, ya que de ellos se puede obtener alguna información adicional sobre el estado de la transformación de los elementos.

Las reglas de transformación empleadas para esta primera transformación se pueden observar en el Anexo III. De forma simplificada, las reglas se pueden agrupar como se muestra a continuación:

Transformación del DSL a V3-CM simplificada
Creación de Interfaces V3-CM: Control, Temporizadores, lectura y escritura de enteros y envío y recepción de mensajes.
Cada <i>NodeGroup</i> genera un componente complejo, formado por un COTS de V3-CM llamado START y un coordinador, cada uno con sus respectivos puertos. El coordinador además dispone de una máquina de estados (en otras reglas se conectan estos dos).
Por cada recurso del modelo WSN (Sensores, puertos hardware y parámetros de configuración) se crea un COTS y sus marcas (variarán según el recurso). Los recursos RAM son una excepción y para ellos no se ejecuta la regla.
Por cada <i>Timer</i> se crea un COTS de tipo <i>Timer</i> .
Por cada asociación entre un elemento <i>Network</i> y un <i>NodeGroup</i> se crea un COTS de tipo <i>Network</i> .
Por cada unidad funcional se crean actividades, una o más dependiendo del tipo de unidad funcional. Las actividades que se asocian pueden ser las definidas por el usuario, actividades constantes que permiten usar valores constantes, condicionales y llamadas a las funciones de recursos (sensores, parámetros de configuración y puertos hardware), temporizadores y de la radio. Se indica también el orden en que las actividades y sus parámetros deben conectarse.
Se conectan las actividades asociadas a una misma unidad funcional. Se crean elementos de tipo <i>ControlFlow</i> para los flujos de control y <i>ObjectFlow</i> para los de datos.
Se transforman los <i>UnitLinks</i> que unen unidades funcionales en elementos <i>ControlLink</i> que unen a (a) la última actividad transformada a partir de la unidad funcional que actúa como <i>source</i> del enlace; con (b) la primera actividad de la unidad funcional que actúa como <i>target</i> . De la

misma manera se crean elementos *DataFlow* para conectar los datos de entrada y salida de las unidades funcionales.

5.3.2. Transformación V3-CM a nesC

La transformación de V3-CM a modelos nesC, aunque sigue siendo compleja, no lo es tanto como la primera transformación entre modelos, en parte por la experiencia adquirida en desarrollar la primera transformación. Pero el principal factor que ha simplificado las transformaciones ha sido el hecho de que se puede realizar una transformación directa entre muchos de los conceptos de ambos modelos. Por ejemplo todas las actividades V3-CM definidas por el usuario se corresponden con bloques de código definidos por el usuario en nesC, los cuales toman los mismos datos que sus actividades de origen. En general la transformación de actividades ha sido uno a uno para todos los tipos de actividades V3-CM. Sin embargo, la complejidad al transformar actividades ha venido marcada por la creación de conceptos que no existen en el metamodelo V3-CM, como la implementación de comandos y eventos en los componentes y la ubicación de los distintos bloques básicos de código en éstos.

El principal reto en la implementación de esta segunda transformación ha sido el uso de marcas para establecer relaciones entre los elementos V3-CM y los elementos del modelo de la plataforma nesC. Estas marcas también han ayudado a establecer no sólo cómo cada actividad V3-CM genera una actividad nesC, sino también cómo éstas generan o no implementaciones de eventos. Las relaciones entre modelos que se han establecido haciendo uso de marcas son las siguientes:

- Entre un componente V3-CM y un componente nesC del modelo de la plataforma: Para que la relación se pueda establecer, ambos deben tener marcas con el mismo atributo *data*. Además las marcas deben tener asociados elementos de tipo *tag* similares (las marcas contienen el tipo de componente y poseen tags que indican el subtipo, por ejemplo, *sensor* para la marca y *light_sensor* para el *tag*).
- Entre un puerto con una interfaz V3-CM y una instancia de interfaz nesC: Al igual que antes ambos interfaces deben tener una marca con idéntico valor en el atributo *data* para que se establezca la relación.
- Entre un servicio V3-CM y un prototipo (de comando o evento): Un servicio puede relacionarse con un comando, con un evento o con ambos (en el caso de que tenga una ejecución asíncrona asociada). La relación puede establecerse buscando marcas similares. En el caso de que los servicios de ambos modelos pertenezcan a interfaces con un único servicio sin relacionar, una relación entre ellos se establece sin necesidad de usar marcas.
- Entre un parámetro de un servicio V3-CM y una variable de un prototipo nesC: Cuando un servicio y un prototipo tienen un único parámetro (o variable) esta relación se establece automáticamente. Cuando alguno de ellos tiene más de un

parámetro se debe hacer uso de marcas para establecer estas relaciones. Para esto se debe asignar una marca sobre la variable del modelo de la plataforma de tipo *Parameter*.

- Entre un servicio V3-CM y una constante asignada a una variable de un prototipo nesC: La variable del prototipo debe tener una marca de tipo *ConstParam* cuyo valor en el atributo *data* coincida con el atributo *data* de la marca asociada al servicio.

Las transformaciones completas se encuentran en el Anexo IV. De forma simplificada, el proceso de transformación puede describirse de la siguiente manera:

Transformación de PIM V3-CM a PSM-nesC simplificada
Se establecen relaciones entre los componentes e interfaces de ambos modelos (mediante marcas).
Para cada componente e interfaz del modelo de la plataforma nesC para los que se haya creado una relación se crea un componente o interfaz en el modelo de la aplicación mediante una copia del correspondiente elemento del modelo de la plataforma.
Se crea un componente complejo nesC por cada componente complejo V3-CM. Con los componentes simples se lleva a cabo una transformación similar.
Por cada componente V3-CM enlazado dentro de un componente complejo se crea un componente renombrado que se asigna al componente complejo nesC correspondiente.
En función de las relaciones entre puertos e instancias de interfaz se crean las instancias de interfaz para los componentes nesC obtenidos a partir de COTS V3-CM.
A partir de las instancias de interfaz transformadas para los COTS se crean las instancias de interfaz de los componentes conectados a ellas.
Por cada enlace entre dos puertos se crea una conexión entre dos instancias de interfaz.
Se establecen los valores de los parámetros de las instancias de interfaz parametrizadas de forma que tomen un valor único. Un caso particular es el de las instancias de interfaz parametrizadas del componente radio cuyo valor no debe ser único sino compartido con el componente radio que usan los nodos que se encuentran en el otro extremo de la comunicación.
A partir de las llamadas a un servicio relacionado con un evento se crea una implementación del evento. En el caso de llamadas a un servicio relacionadas con un comando se crea una llamada a dicho comando.
Por cada actividad definida por el usuario se crea un bloque de código definido por el usuario.
Cada bloque de código generado a partir de una actividad debe asignarse a un comando, evento o tarea (lo contendrá). Para ello se

asignan al mismo comando o evento al que se encuentre asignada la actividad V3-CM inmediatamente anterior (la actividad que sea origen de un enlace de control a la actividad en cuestión).
<p>Se crean las tareas según dos criterios para evitar situaciones potencialmente peligrosas, bien por correr el riesgo de generar erróneamente el código o por crear condiciones de carrera potenciales:</p> <ul style="list-style-type: none"> - Si a una actividad llegan dos enlaces de control se creará una nueva tarea y el bloque de código relacionado con la actividad se asigna a ella. - Si una llamada a un comando se realiza desde un evento asíncrono se crea una nueva tarea para realizar la llamada.
Se transforman los enlaces de control a sus equivalentes en el modelo nesC. Los enlaces se asignan al evento o comando al que esté asignado su bloque de código origen.
Se establecen los tipos de los enlaces de control según su bloque de código origen y su bloque destino. Lo mismo se hace para los enlaces de datos.

5.3.3. Ejemplo de transformaciones M2M

La Figura 5-12 muestra un ejemplo de cómo se realizan las transformaciones entre los distintos modelos. En la aplicación WSN descrita, un grupo de nodos (*Monitor*) lee periódicamente del sensor de luz y envía la lectura por radio a otro nodo (*Sink*) que la reenvía a un PC para su análisis. Para ello hace uso de dos elementos *NodeGroup*: *Monitor* y *Sink*. Aplicando las reglas de transformación explicadas en los apartados anteriores, por cada uno de éstos se generan tres componentes V3-CM: un componente simple que contiene el comportamiento de la aplicación y dos componentes complejos que conectan con los recursos (como se mostró en la figura Figura 5-11). El componente simple dispone de una máquina de estados donde, en el estado *StartX*, se crea el diagrama de actividades que se muestra en la parte media de la Figura 5-12. Cada una de las actividades del diagrama se corresponde con las unidades funcionales WSN según muestran las flechas. Para la transformación a nesC las actividades se numeran en la figura y, en el modelo nesC de la parte inferior de la figura, se puede ver el resultado de su transformación. También se puede apreciar que las actividades V3-CM, concretamente las llamadas a servicios, no sólo se transforman a bloques de código del PSM-nesC sino que además generan implementaciones de eventos. La decisión de en qué eventos deben transformarse se realiza mediante marcas (presentes en el modelo V3-CM y en el modelo de la plataforma TinyOS/nesC), de forma que según cuál sea el recurso sobre el que se hace la llamada se generará una implementación para un evento u otro.

Figura 5-12. Ejemplo de transformaciones M2M en la herramienta MoteGen

5.4. Transformación a código

Una vez se dispone del modelo nesC se genera código a partir de éste mediante una transformación a texto. Para su implementación se ha recurrido al lenguaje de transformación MOFScript [Oldevik 05]. Éste se ha desarrollado dentro del proyecto GMT (*Generative Modeling Technologies*), cuya misión es la definición de prototipos en el área MDE para Eclipse. Dentro de este proyecto, el subproyecto *MOFScript* es el encargado del desarrollo de herramientas y *frameworks* para soportar las transformaciones de modelo a texto.

El lenguaje *MOFScript* se usa para generar texto a partir de modelos basados en MOF. Se trata de un lenguaje imperativo basado en la ejecución secuencial. De entre sus ventajas destaca su usabilidad, ya que se trata de un lenguaje sencillo compuesto por un número reducido de construcciones. Además, es similar a otros lenguajes de programación y de scripts existentes y resulta bastante intuitivo. Cuenta con otras características interesantes como el manejo de archivos, expresiones y cadenas así como el uso de variables y colecciones. Para la implementación de la herramienta que se describe en este capítulo, este último tipo de datos ha resultado de gran utilidad en la implementación de las transformaciones.

A modo de ejemplo, en la Figura 5-13, se muestra un fragmento de código *MOFScript*. Este fragmento es una simplificación del empleado para la generación de código para las tareas nesC. En este caso la ejecución de la regla *doTask* para un elemento de tipo *ComplexBlock* genera el código asociado a una tarea (*task*) nesC. En primer lugar se declara una tarea nombrada de acuerdo con su elemento del metamodelo y sin parámetros. Después se crean las variables locales y por último se transforman cada uno de los enlaces de control que parten de la tarea (y como consecuencia también los bloques de código con los que enlazan).

```
mm.ComplexBlock::doTask() {

    println("\ttask void "+ self.name + "() {")
    tabs=2

    //local variables
    self.printLocalVars()

    // code for internal CodeBlocks
    cb.isSource->forEach(cl:mm.ControlLink){
        cl.do()
    }

    tabs=1
    println("\t}\n")
}
```

Figura 5-13. Fragmento del lenguaje MOFScript

Para que la transformación que se implementa con *MOFScript* pueda generar correctamente código a partir de los modelos obtenidos en las transformaciones M2M es necesario que sea capaz de transformar cada uno de los bloques definidos por el usuario creados tras la transformación a V3-CM (en V3-CM se corresponden con actividades definidas por el usuario). En el apartado 5.3.1 se dio una lista de cuáles son estos bloques y cómo se denominan sus parámetros.

A continuación se describen, de forma simplificada, los pasos que se ejecutan para obtener código mediante la transformación M2T:

REGLA		
1	Por cada componente complejo (configuración) de alto nivel que represente el comportamiento de un grupo de nodos se crea un directorio con un archivo <i>makefile</i> . Dentro de este directorio se crearán todos los componentes que sean utilizados por ese comportamiento. Éstos posteriormente serán compilados para obtener una imagen de código ejecutable que sea cargada en los nodos.	
2	Cada componente complejo (de alto nivel o no) se transforma a código.	
	A	Si hace uso de valores constantes (actualmente sólo los tipos AM se consideran) expresados como cadenas de caracteres se les asigna un valor entero.
	B	Se genera el bloque de especificación del componente (<i>configuration</i>).
	1	Se genera su lista de instancias de interfaz, primero las provistas y después las usadas.
	C	Se genera el bloque de implementación del componente. Para componentes complejos (o configuraciones) contiene enlaces entre instancias de interfaz de sus componentes internos.
	1	Se genera la lista de componentes internos, renombrándolos si es necesario.
	2	Por cada elemento <i>Wiring</i> se genera un enlace entre dos instancias de interfaz.
3	Cada componente simple se transforma a código.	
	A	Se genera el bloque de especificación del componente (<i>module</i>).
	1	Se genera su lista de instancias de interfaz, primero las provistas y después las usadas.
	B	Se genera el bloque de implementación del componente. Para componentes simples (o módulos) contiene el código de comandos, eventos, tareas y funciones.
	1	Se genera una lista de las variables de módulo (accesibles desde cualquier punto del componente). Para la realización de esta lista se recorren todos los enlaces de control y bloques de código del módulo.
	a	El uso de determinados tipos de bloques de código definidos por el usuario implica la creación de variables de módulo: <ul style="list-style-type: none"> - Se necesita una variable de tipo mensaje (<i>TOSMsg</i>) en el caso usar bloques <i>InitMsg</i>. - Si se accede a recursos RAM se necesita otra variable de

		módulo de tipo entero.
	b	Se crean variables de módulo para intercambiar datos entre distintos comandos, eventos y tareas.
C		Se transforman a código cada una de las implementaciones de función.
	1	Se declara la implementación de función según su tipo (comando, evento o tarea).
	2	Se declaran las variables locales. Para ello se recorren los bloques de código interno para determinar las que serán empleadas.
	3	Se procesan los enlaces de control que tomen como punto de partida a la implementación de función en cuestión.
	a	Si el enlace de control es de tipo <i>Call</i> se genera una llamada al bloque de código destino del enlace.
	b	<p>Si el enlace es de tipo <i>Flow</i> se genera el código asociado al bloque destino del enlace:</p> <ul style="list-style-type: none"> - Si el bloque es de tipo <i>ExternalCall</i> se hará una llamada a un comando externo a través de una instancia de interfaz usada del componente. - Si el bloque es de tipo <i>UserDefined</i> se generará el código específico del bloque. Se debe realizar una asignación entre los parámetros del bloque y variables existentes. En el caso de bloques condicionales se genera el código en caso de cumplirse la condición y de no cumplirse. - Si el bloque de código contiene información acerca del valor de salida del evento o comando que lo contiene ésta se procesa. <p>Se procesan todos los enlaces de control que salgan del bloque actual.</p>
	4	Se devuelve el valor de salida.

Un pequeño ejemplo de transformación de algunas actividades del PSM nesC en código se puede ver en la Figura 5-6 mientras que en el siguiente capítulo se mostrará el código generado a partir de una aplicación completa.

5.5. Conclusiones y aportación a la Tesis

En el presente capítulo se ha descrito cómo a partir de los modelos descritos con el DSL del capítulo anterior se genera código haciendo de uso de los niveles de abstracción que define MDA. Para cada uno de estos niveles se han descrito sus metamodelos (creados mediante el *framework* EMF) y para cada una de las transformaciones se han dado detalles sobre su implementación (con el lenguaje AGG). Como nivel de mayor abstracción se han usado los modelos generados con el DSL para WSN. En un nivel intermedio se ha empleado el lenguaje de modelado basado en componentes V3-CM y para el nivel de abstracción menor se han empleado modelos PSM del lenguaje nesC de TinyOS. Para estos últimos se ha definido una

transformación a texto (mediante *MOFScript*) que permite finalmente la obtención de código. Tanto los niveles superior (asociado al DSL) como el inferior (PSM-nesC) han sido desarrollados para esta herramienta, mientras que V3-CM existía con anterioridad. Como se ha comentado, V3-CM está siendo usado en proyectos de desarrollo de software dirigido por modelos para robótica y domótica por lo que se posibilita una integración en el futuro entre los artefactos desarrollados en cada estos dominios con los obtenidos a partir de la herramienta que se describe para redes de sensores.

Tras la implementación de los niveles (de modelos y metamodelos) y sus transformaciones, se ha podido comprobar que las transformaciones entre modelos son la parte más complicada de desarrollar. Para ello se ha necesitado emplear un gran número de reglas de transformación complejas que han sido muy difíciles de depurar debido a la carencia de medios disponibles. Para simplificar las transformaciones entre modelos se han tomado diversas consideraciones: desde la composición de reglas (en este caso realizada como reglas que pueden activar la ejecución de otro conjunto de reglas e interconectar los resultados de cada una de ellas) hasta el uso de marcas que guíen las transformaciones a elementos ya existentes en el modelo de la plataforma. De la misma forma se han tenido en cuenta consideraciones en las transformaciones que han permitido que se genere código de forma bastante eficiente, obteniendo un código legible y de un tamaño comparable al que se obtendría al programar directamente en nesC.

Capítulo 6

Uso de la herramienta MoteGen

En este capítulo se describe la herramienta desarrollada en el marco de esta Tesis. Esta descripción se abordará desde el punto de vista de su utilización por los usuarios para desarrollar aplicaciones software con ella, pues los detalles de su implementación ya se han tratado en los dos capítulos anteriores. Por lo tanto, el capítulo se centrará principalmente en describir las funcionalidades que ofrece y dar un ejemplo de su uso. A este respecto se muestra un caso de estudio correspondiente a una aplicación de localización de personas.

A continuación se describe cómo la herramienta *MoteGen* puede ser utilizada por sus usuarios. En primer lugar se mencionan los requisitos con los que se debe contar, tanto a nivel de hardware como de software, para poder ejecutarla y, posteriormente, se describe cómo usarla. En el siguiente punto se presenta una metodología para la construcción de aplicaciones con la herramienta y, finalmente, se muestra una aplicación práctica modelada con ella y el código que se genera.

6.1. Requisitos de la herramienta

A nivel software los requisitos vienen impuestos por las tecnologías empleadas por la herramienta. Estas tecnologías son diversos *frameworks* de modelado y lenguajes de transformación de modelos que deben estar instaladas para poder ejecutar la herramienta. A nivel hardware los requisitos que se exigen para ejecutar la herramienta se refieren principalmente a la cantidad de memoria RAM disponible. Aunque la aplicación puede funcionar con una cantidad menor, se debería contar con al menos un 1 GB de RAM (en el caso del sistema operativo Windows XP en el que ha sido probada).

Para la instalación de la herramienta es necesario instalar todas las tecnologías anteriores de las que hace uso así como crear un espacio de trabajo en Eclipse en el que se importen cada uno de los proyectos que forman la herramienta. En el Anexo I se dispone de una lista de estos proyectos y de localizaciones a partir de las cuales las tecnologías usadas se pueden obtener mediante la instalación automática de Eclipse.

6.2. Uso de la herramienta

Para ejecutar la herramienta se debe abrir un nuevo espacio de trabajo *runtime* en Eclipse. Esto se hace pulsando sobre el botón de la barra de menú de Eclipse. Tras hacer esto se abre una nueva ventana de Eclipse que puede ser usada para editar aplicaciones WSN. En la Figura 6-1 se puede observar una captura de pantalla de la edición de una aplicación. En ella hay una serie de áreas diferenciadas:

- El área de edición gráfica permite añadir elementos al diagrama de la aplicación y modificarlos.
- La paleta de herramientas contiene la lista de elementos del DSL que pueden ser empleados en un diagrama.
- En la barra de menús se ha incluido el elemento WSN-Gen donde se irán incorporando las operaciones que se pueden ejecutar sobre el modelo. En la situación actual sólo se permite generar código (el botón de la barra de menú también se usa para generar código)

- El área de propiedades permite modificar atributos asociados a los elementos del modelo.
- El navegador permite seleccionar archivos almacenados en disco.

Figura 6-1. Áreas de la herramienta MoteGen

Las operaciones que se pueden hacer con la herramienta son las siguientes:

- **Creación de un nuevo modelo de aplicación**

Tras ejecutar esta acción se generan dos archivos correspondientes al modelo de la aplicación y al diagrama. El archivo del modelo, que tiene extensión *wsan_mm* y es conforme al metamodelo WSAN, es el que se pasará a las transformaciones, mientras que el archivo de diagrama contiene información sobre la representación gráfica de los elementos del modelo.

Se debe seleccionar *File -> New -> Other -> Examples -> WSAN_MM Diagram* y, en la ventana de diálogo que aparece, asignar un nombre tanto al fichero del modelo como al de la representación (no se debe pulsar el botón *finish* de la ventana hasta no haber asignado ambos nombres). De momento para poder generar código se debe elegir *"input.wsan_mm"* como nombre para el modelo.

- Edición de modelos y diagramas

La edición de modelos se realiza en el área de edición haciendo uso de los elementos de la paleta. Las propiedades de cada uno de los objetos representados, que pueden representarse o no en el área de edición, son editadas en el área de propiedades.

- Grabación del modelo

Tanto el modelo como el diagrama se graban al pulsar el botón *Save* o al elegir la opción del menú *File -> Save*.

- Validación del modelo

El modelo debe ser validado con respecto a su metamodelo para asegurar que no hay errores (típicamente propiedades obligatorias no asignadas).

Para validar un modelo se debe abrir el archivo del modelo asociado a la aplicación. Una vez abierto se selecciona el objeto *Root* de la aplicación denominado *WSN Application* y en su menú contextual se pulsa sobre *Validate* (véase Figura 6-2).

Figura 6-2. Validación de una aplicación

- Depuración del modelo

Si se han borrado elementos del diagrama pero siguen existiendo en el archivo del modelo deben ser borrados de este último. Esta situación puede darse si se borra con la opción *"Delete from Diagram"* o con la tecla *SUPR*.

- **Generación de código**

Para la generación de código se pulsa sobre el botón o bien en la entrada “*Compile to nesC*” del menú *WSN-Gen*. La carpeta donde se crean los archivos con el código se indica en la consola de Eclipse.

6.3. Consideraciones de uso

A la hora de crear cada uno de los elementos de estos bloques hay que tener en cuenta una serie de consideraciones. La primera de ellas consiste en evitar determinados nombres para las definiciones de recursos que pueden ocasionar problemas. En concreto se deben vigilar los **nombres asignados a los recursos de almacenamiento** ya que serán transformados a variables de módulo en el código nesC y pueden coincidir con alguno de los nombres usados para las variables locales en funciones, eventos y tareas. Por lo tanto, además de usar identificadores de variable C válidos se deben evitar nombres como *g_message* o los formados por la concatenación de *input*, *output*, *paramNum*, *msg*, *extractData*, *NodeID* y un número ya que éstos pueden ser empleados como variables locales (en breve esto se cambiará para que sólo queden restringidos los nombres que empiecen por “_”). Además, hay que evitar repetir nombres en los distintos elementos del diagrama ya que el *framework* de modelado EMF puede confundirlos. Una excepción son los recursos y las definiciones de recursos a las que hacen referencia. Asignar el mismo nombre a estos dos tipos de elementos da lugar a programas más fácilmente comprensibles y no presenta problemas para la generación de código.

Una segunda consideración se debe tener al **borrar elementos** de la especificación de la aplicación. Los elementos se pueden borrar o bien del diagrama o bien del modelo y del diagrama. La primera opción no es recomendable ya que existirían elementos en el modelo que se transformarían a pesar de no aparecer en el diagrama. Por lo tanto hay que asegurarse de que no existan elementos en el modelo no representados:

- Los elementos se deben borrar con la opción “*Delete from model*” que aparece en el menú contextual del elemento.
- Si la eliminación de un elemento implica la desaparición de otros elementos se deben eliminar éstos primero. Por ejemplo, si se va a borrar un recurso antes debe borrarse el *ResourceLink* que conecta al recurso.

Otra consideración al describir las aplicaciones es que, por el momento, sólo se puede realizar una única llamada a sensores y UART por cada componente. Si por ejemplo más de una unidad funcional necesita una lectura de sensor, se deberá hacer una única llamada que almacene el dato y cada una de estas unidades funcionales deberá leer este dato almacenado.

6.4. Metodología de desarrollo

A continuación se propone una metodología sencilla para el desarrollo de aplicaciones con la herramienta *MoteGen*. Esta metodología se basa en la ejecución secuencial de cuatro fases: especificación de requisitos, determinación de elementos de primer nivel, construcción de la aplicación y generación y compilación, aunque es posible ejecutar las fases segunda y tercera simultáneamente.

- Especificación de requisitos

Es necesario conocer de antemano los requisitos de la aplicación tanto funcionales (aquellos que se describen en el bloque de comportamiento del DSL) como no funcionales. Éstos últimos son también importantes porque van a influir en cómo se determinan los elementos del diagrama de la aplicación. Ciertos requisitos tienen mayor importancia y debe quedar claro si la aplicación necesitará hacer uso de ellos antes de comenzar el diseño. Este es el caso de las características del sistema como eficiencia energética, seguridad, movilidad de los nodos o baja latencia en la recepción de datos.

La eficiencia energética es el aspecto que mayor importancia tiene en el diseño de la aplicación pues afecta a un gran número de sus elementos. Puesto que una mayor eficiencia energética va en detrimento de otros parámetros de calidad, debe cuantificarse su importancia. Ésta será mayor cuanto mayor sea el tiempo de vida esperado de la red de sensores y cuanto más activos se encuentren sus nodos. Por el contrario cuanto más accesibles físicamente sean los nodos y menor sea su número, también menor será la importancia de este requisito.

- Determinación de los principales elementos del sistema

Tras un análisis de los requisitos se establecen los elementos necesarios de más alto nivel necesarios en el diagrama: grupos de nodos (*NodeGroup*), definiciones de nodo (*NodeDefinition*), redes de comunicaciones (*Network*) y enlaces inalámbricos (*WirelessLink*).

En primer lugar se deben determinar los distintos tipos de nodo que pueden formar parte de la red de sensores, clasificándolos según roles. A continuación se debe pasar a analizar el intercambio de datos entre éstos. Para esto debe realizar una lista de los mensajes que intercambiarán en la que se indique qué datos contendrá cada uno y qué propiedades de calidad de servicio (*QoS*) necesita cada uno de ellos al ser enviado. Además, cada uno de estos datos debe ir numerado dentro del mensaje para poder identificarlo frente al resto de datos del mensaje. Posteriormente, toda esta información será usada para crear grupos de nodos y establecer comunicaciones entre nodos.

En un siguiente paso se determinan los grupos de nodos (*NodeGroup*). En principio, por cada rol debería necesitarse un grupo de nodos, pero en el caso de que los nodos de un determinado rol deban ser el destino de comunicaciones punto a

punto (con un único destino) no tiene porqué ser así. Si se hiciera de esta forma, sin tomar ninguna medida más, todos los nodos con el rol en cuestión recibirían el mensaje y no sólo el destinatario. Hay dos alternativas para solucionar esto. La primera consiste en, por cada rol, tener un *NodeGroup* para cada uno de sus nodos. La segunda implica que la dirección de destino de los mensajes se incluya en estos como un dato más (al igual que por ejemplo se hace con las lecturas de sensores que se envían por radio) y los nodos al recibir los mensajes comparen este dato con su identificador de nodo.

Una vez creados los grupos de nodos el diseñador les debe asignar un número mínimo de motes que deberían ejecutar este comportamiento y un número máximo de forma que para cualquier número de motes comprendidos entre ambos (incluidos) la aplicación se comporte correctamente.

A continuación se obtienen las redes de comunicaciones necesarias (*Network*) y se asignan a los grupos de nodos que corresponda. Se debe disponer de un elemento *Network* por cada conjunto distinto de parámetros de *QoS*. Tras esto, a cada grupo de nodos se le debe asignar todos los elementos *Network* en cuyas redes participe, entendiendo por participar el ser origen o destino de mensajes de esa red o bien el encaminar mensajes de la red en cuestión. El mecanismo de encaminamiento asignado a cada una de estas redes dependerá principalmente de los requisitos de consumo energético de la red de sensores y de la visibilidad entre nodos. En el caso de nodos con visibilidad directa entre ellos se puede optar por comunicaciones punto a punto (y sin retransmisión) o *broadcast*, mientras que si los nodos están más alejados las opciones son *multihop* e inundación. Éste último presenta el mayor consumo de energía por lo que sólo debería ser empleado en redes que hagan poco uso de las comunicaciones y estén formadas por un número pequeño de nodos o bien para pruebas. En cualquier caso, antes de elegir unos determinados mecanismos de encaminamiento y propiedades de comunicación, se debe tener en cuenta que no todos éstos están actualmente implementados por la herramienta aunque sí lo estén en el lenguaje.

Una vez conocidos los grupos de nodos se pasa a establecer los enlaces inalámbricos (*WirelessLink*) entre ellos. Para esto, por cada comunicación de datos que se realice entre dos grupos de nodos se establece un *WirelessLink* y a éste se le asigna una de las redes *Network* existentes de acuerdo a los requisitos de la comunicación.

Por último, se deben identificar el conjunto de recursos que van a ser usados por los distintos grupos de nodos. A partir de ellos se crearán las definiciones de nodo (*NodeDefinition*). Si los recursos usados por dos grupos de nodos son idénticos, o presentan diferencias muy pequeñas, ambos grupos de nodos pueden usar el mismo *NodeDefinition*. En caso contrario aunque es posible fusionar ambos *NodeDefinition* en uno sólo, resulta más apropiado crear dos definiciones. De esta manera resulta fácil determinar si una plataforma hardware puede ejecutar un comportamiento especificado en un *NodeGroup* simplemente comparando sus recursos con los de la definición de nodo.

- Construcción de la aplicación

En este punto se plasman los anteriores elementos en un diagrama si no se ha hecho simultáneamente a su determinación. Tras esto se define el comportamiento de cada uno de los grupos de nodos según los requisitos funcionales de la aplicación y teniendo en cuenta las consideraciones que se dieron en el apartado 6.3. Además, en el caso de realizarse comunicaciones entre nodos que impliquen el intercambio de más de un dato, se debe usar la numeración de cada uno de los datos que se asignó en la paso previo de la metodología (se numeran los elementos *ResourceLink* que unen recursos y unidades funcionales).

- Generación de código y compilación

Con la descripción de la aplicación finalizada se pasa a su validación. Una vez realizada se procede a la generación de código. La herramienta indicará la localización de los directorios y archivos generados desde donde deberán ser copiados a una carpeta fácilmente accesible en la consola de manejo de TinyOS. Tras la copia de archivos se abre una consola de TinyOS y se procede a la compilación. Puesto que el código para cada grupo de nodos se encuentra en una carpeta distinta habrá que repetir el proceso de compilación para todas ellas. Tras esto se obtiene una serie de imágenes de código ejecutables en los motes que se cargarán a tantos nodos como sea necesario, siempre que se cumpla con las restricciones de número máximo y mínimo de motes por grupo de nodos.

6.5. Ejemplo práctico

La herramienta *MoteGen* permite modelar una gran cantidad de aplicaciones. Resulta muy apropiada para monitorización de datos, con una serie de nodos recogiendo y enviando lecturas de sensores hacia otro dispositivo conectado al PC. Para esto, además, ofrece una serie de funciones que permiten acumular diversas medidas antes de ser enviadas, reaccionar ante valores de éstas que cumplan ciertas condiciones o realizar procesados sencillos como obtener los valores medios, máximos o mínimos de un conjunto de lecturas. Para procesados más complejos se pueden introducir sentencias en lenguaje C en el modelo que operen sobre ellas.

Sin embargo, a pesar de esta inclinación hacia la monitorización de datos, la aplicación que se ha elegido como ejemplo no pertenece a este tipo. En su lugar se ha elegido una aplicación de seguimiento de objetos móviles (*tracking*) por considerarse más avanzada y mostrar mejor la potencia de la herramienta. En este caso la aplicación modelada se encarga de la localización espacial de personas. La aplicación permite conocer en cada momento dónde se encuentran una serie de nodos móviles. Para ello se divide el espacio en zonas en las cuales se ubica un nodo fijo (uno por zona). El nodo móvil debe estimar la proximidad a cada uno de los motes fijos para poder decidir en qué zona se encuentra. Una vez se conozca la zona en la que se encuentra el nodo ésta se representará en los leds, de forma que la persona que lleva el mote pueda conocer su

localización. Además se podría incluir un elemento que mandara por radio la localización de los nodos (no incluido en este ejemplo) de forma que se pudiera guardar un registro de las ubicaciones donde han estado personas, objetos o animales. Para este ejemplo se ha usado la salida por leds ya que resulta más útil para comprobar el correcto funcionamiento de la aplicación.

En un caso real esta aplicación podría tener aplicaciones en visitas turísticas auto-guiadas, o en la realización de operaciones en entornos donde la visibilidad puede ser un problema, por ejemplo en incendios con abundante humo. Si, además, la posición de los nodos se transmite a un PC, la aplicación puede ser útil en el caso de control de personal, pacientes, etc. Otra funcionalidad interesante se podría añadir fácilmente a la aplicación modelada, como por ejemplo que tanto los nodos fijos como los móviles puedan recoger lecturas del entorno (a través de los sensores) para obtener información adicional.

A grandes rasgos, el funcionamiento de la aplicación consiste en tener una serie de nodos emplazados en posiciones fijas, conocidos como nodos *beacon*, que emiten mensajes que identifican al nodo. Los nodos móviles reciben estos mensajes procedentes de distintos nodos y calculan cuál de los nodos *beacon* es el más cercano en función de la potencia con la que se reciben (mayor potencia implica menor distancia). El identificador de este nodo, incluido en los mensajes que manda, corresponderá a la zona en la que se encuentra el nodo móvil.

En la Figura 6-3 se muestra la descripción realizada con el DSL de la herramienta. En ella se ha hecho uso de un recurso de almacenamiento de tipo *hash* llamado *RxPows*. Este tipo de datos consiste en una tabla *hash* que proporciona asociaciones entre llaves y sus correspondientes valores. En el caso de las redes de sensores, estas estructuras pueden ser útiles para asociar a los nodos de una red de sensores las lecturas que han tomado. Concretamente, pueden ser usadas para relacionar a cada uno de los nodos *beacon* (llave) con la potencia recibida desde ellos. Sobre este conjunto de datos más adelante se pueden ejecutar funciones para obtener sus valores medios, máximos o mínimos o las llaves asociadas a ellos. El tamaño que se ha usado para esta estructura es siete, puesto que los nodos sólo disponen de tres leds y por lo tanto pueden representar números en binario entre 0 y 7 (el siete se reserva para indicar que no se recibe señal desde ningún nodo). Por este mismo motivo el número máximo de nodos *beacon* que se usan en la aplicación es también de siete, cada uno con una dirección de nodo distinta correspondiente a estos valores.

El valor de potencia recibido no es constante, presenta fluctuaciones al desplazar ligeramente el nodo e incluso manteniéndolo fijo. Por lo tanto se ha optado por tomar varios valores de potencia para cada nodo y calcular su media. En la aplicación se ha optado por realizar la media de forma simultánea a la introducción de cada valor en la tabla *hash*, para lo cual se ha usado una unidad funcional de tipo *WRITE_MEAN*.

En la descripción de la aplicación se observa cómo se crean dos *NodeGroup*, uno para los nodos *beacon* y otro para los móviles así como definiciones de nodo para cada

uno de ellos. El comportamiento de cada uno de estos grupos de nodo se puede ver como la ejecución de una secuencia de operaciones sobre recursos descritos en cada una de las unidades funcionales del grupo.

Figura 6-3. Aplicación para localización de personas descrita con la herramienta *MoteGen*

Los nodos *beacon* leen cada segundo su dirección de nodo y se escribe en el recurso de almacenamiento *NodeId*. Este valor se manda por radio a través del enlace inalámbrico *WL1* (en la figura se encuentra representado por un flecha con línea discontinua entre ambos *NodeGroup*) que conecta a los nodos *beacon* con los nodos móviles. Posteriormente se carga la dirección del nodo y se muestra en los leds.

En los nodos móviles cada vez que se recibe un mensaje de los nodos *beacon* se almacenan la dirección del nodo *beacon* y la potencia con que se recibió el mensaje. A continuación se carga el valor de potencia recibida y se almacena en la tabla *hash* asociándola a la dirección del nodo *beacon*. En paralelo, cada cinco segundos se procede a determinar en qué zona se encuentra el nodo móvil. Para lo cual en primer lugar se busca la llave asociada al valor máximo de potencia recibida. Esta llave, que coincide

con la dirección de nodo de uno de los *beacons*, se muestra en los leds. Finalmente se borran todos los valores de potencias y direcciones de nodo de la tabla *hash* para volver a comenzar el proceso.

La aplicación se ha ejecutado en una red heterogénea formada por motes de tipo *Telosb* y de tipo *MICAz*, ambos compatibles entre sí ya que usan el estándar de radio *ZigBee*. Se han probado distintas configuraciones. En algunas de ellas los nodos se han emplazado en diversos puntos de una misma habitación y en otras se ha colocado cada nodo en una habitación distinta.

Los resultados obtenidos tras la ejecución de las pruebas han sido satisfactorios y se ha podido localizar a los motes con un margen de error que depende de la densidad de motes fijos colocados. En el caso de realizar pruebas dentro de una única habitación se ha comprobado cómo ésta se divide en una serie de zonas, cada una de las cuales pertenece a uno de los nodos *beacon*. La forma en la que se realiza esta división depende de donde se emplacen los nodos y del entorno que los rodea, en el que se producirán reflexiones de la señal. Así por ejemplo, la colocación de nodos en estanterías de armarios, junto a paredes o insertados en el puerto USB de un PC modifica su diagrama de radiación de los nodos y altera ligeramente la distribución de zonas de la habitación. En el caso de pruebas en distintas habitaciones se ha podido determinar con seguridad la ubicación de cada uno de los motes ya que la señal de motes en otras habitaciones llega más atenuada por lo muros.

6.5.1. Código generado

Al ejecutar la generación de código se crean dos carpetas, una para cada tipo de nodo, con cuatro archivos cada una. Uno de estos cuatro archivos es el controlador, que contiene la lógica de la aplicación. Un segundo archivo, en adelante referido como configuración de segundo nivel, conecta al controlador con los recursos que emplea. Estos recursos son manejados mediante componentes del sistema operativo o bien mediante componentes desarrollados específicamente para la herramienta. El tercer archivo, la configuración de primer nivel, contiene un componente de configuración que es el que será compilado. Este componente conecta la aplicación con el componente llamado *main* del sistema operativo TinyOS encargado de arrancar la aplicación. Por último, el cuarto archivo es un *makefile* usado en la compilación del código de cada grupo de nodos. En las figuras Figura 6-4 a Figura 6-11 se muestra el contenido de estos archivos (en orden inverso al que han sido comentados).

```
COMPONENT=AppMobileNG
PFLAGS +=-I%/lib/Eclipse
include ../Makerules
```

Figura 6-4 Archivo makefile para los nodos móviles

```

configuration AppMobileNG{
}

implementation
{
    components Main as StartAppMobileNG_, ConfMobileNG as ConfMobileNG_;

    StartAppMobileNG_.StdControl -> ConfMobileNG_.ControlConf1BasicControlInterface;
}

```

Figura 6-5. Configuración de primer nivel de los nodos móviles

```

#define AM_Net1 1

configuration ConfMobileNG{
    provides          interface StdControl as ControlConf1BasicControlInterface;
}

implementation
{
    components CoordinatorConfMobileNG as CoordinatorConfMobileNG_, SLedsC as
WPortCompLED_2_, TimerC as TimerComp1_, GenericComm as NetComp_1_;

    CoordinatorConfMobileNG_.NetReq1_RecvI -> NetComp_1_.ReceiveMsg[AM_Net1];
    CoordinatorConfMobileNG_.TimerReq1_Timer_req ->
TimerComp1_.Timer[unique("TimerC")];
    ControlConf1BasicControlInterface      = NetComp_1_.Control;
    ControlConf1BasicControlInterface      = TimerComp1_.StdControl;
    ControlConf1BasicControlInterface      = WPortCompLED_2_.StdControl;
    CoordinatorConfMobileNG_.WPortReq2_IntWrite -> WPortCompLED_2_.SetLeds;
    ControlConf1BasicControlInterface      =
CoordinatorConfMobileNG_.ControlCoor2BasicControlInterface;
}

```

Figura 6-6. Configuración de segundo nivel de los nodos móviles

```

module CoordinatorConfMobileNG
{
    uses {
        interface ReceiveMsg as NetReq1_RecvI;
        interface Timer as TimerReq1_Timer_req;
        interface SetInt as WPortReq2_IntWrite;
    }
    provides          interface StdControl as ControlCoor2BasicControlInterface;
}

```

```

}

implementation
{
    int Pow;
    int RxPows[7];
    uint16_t WriteValue16;
    int _count_RxPows[7];
    int _hlast_RxPows = 0;
    int BeaconID;
    int _hkeys_RxPows[7];

    task void TsCmWriteCall_setLeds_Leds16();

    command result_t ControlCoor2BasicControlInterface.init() {

        return SUCCESS;
    }

    command result_t ControlCoor2BasicControlInterface.start() {

        call TimerReq1_Timer_req.start(TIMER_REPEAT, 5000);
        return SUCCESS;
    }

    command result_t ControlCoor2BasicControlInterface.stop() {

        return SUCCESS;
    }

    event result_t TimerReq1_Timer_req.fired() {
        uint16_t _Key_output9;
        int _aux;
        int _selIndex;

        //Select maximum from hash table
        _aux = 0;
        _selIndex = -1;
        while(_aux < _hlast_RxPows) {
            if (_selIndex == -1) _selIndex=0;
            if (RxPows[_aux] > RxPows[_selIndex]) _selIndex = _aux;
            _aux++;
        }
        if (_selIndex != -1){
            _Key_output9 = _hkeys_RxPows[_selIndex];
        } else {
            _Key_output9 = -1;
        }
    }
}

```

```

        atomic {
            WriteValue16=_Key_output9;
        }
        post TsCmWriteCall_setLeds_Leds16();
        return SUCCESS;
    }

    event TOS_MsgPtr NetReq1_RecvI.receive(TOS_MsgPtr m) {
        uint16_t extractData2;
        uint16_t output8;
        uint16_t output6;
        int _aux;

        atomic {
            Pow = (int8_t)m->strength - 45;
        }
        extractData2=((uint16_t *)m->data)[1-1];           //Extract message param 1
        atomic {
            BeaconID = extractData2;
        }
        atomic {
            output8 = Pow;
        }
        atomic {
            output6 = BeaconID;
        }

        //Write in RxPows and calculate mean for key in output6
        _aux = 0;
        while (_aux < _hlast_RxPows)
            if (_hkeys_RxPows[_aux] == output6) break;
            else _aux++;
        if (_aux == _hlast_RxPows) {
            _hlast_RxPows++;
            _hkeys_RxPows[_aux] = output6;
        }
        if (_hlast_RxPows >= 7) _hlast_RxPows = 7-1;
        _count_RxPows[_aux]++;
        RxPows[_aux] = RxPows[_aux] * ((1.0*_count_RxPows[_aux] -
1)/_count_RxPows[_aux]) + (1.0*output8)/_count_RxPows[_aux];

        return m;
    }

    task void TsCmWriteCall_setLeds_Leds16() {
        int _aux;

        call WPortReq2_IntWrite.set(WriteValue16);
        //Reset RxPows hash table
    }

```



```

        _aux=0;
        while (_aux < _hlast_RxPows) _count_RxPows[_aux++]=0;
        _hlast_RxPows = 0;
    }
}

```

Figura 6-7. Coordinador de los nodos móviles

```

COMPONENT=AppMobileNG
PFLAGS +=-I%/T/lib/Eclipse
include .././Makerules

```

Figura 6-8. makefile para los nodos beacon

```

configuration AppBeaconNG{
}

implementation
{
    components Main as StartAppBeaconNG_, ConfBeaconNG as ConfBeaconNG_;

    StartAppBeaconNG_.StdControl -> ConfBeaconNG_.ControlConf3BasicControlInterface;
}

```

Figura 6-9. Configuración de primer nivel para los nodos beacon

```

#define AM_Net1 1

configuration ConfBeaconNG{
    provides          interface StdControl as ControlConf3BasicControlInterface;
}

implementation
{
    components CoordinatorConfBeaconNG as CoordinatorConfBeaconNG_, SLedsC as
WPortCompLED_1_, TimerC as TimerComp3_, GenericComm as NetComp_2_;

    ControlConf3BasicControlInterface = NetComp_2_.Control;
    ControlConf3BasicControlInterface = TimerComp3_.StdControl;
    CoordinatorConfBeaconNG_.NetReq2_SendI -> NetComp_2_.SendMsg[AM_Net1];
    CoordinatorConfBeaconNG_.WPortReq1_IntWrite -> WPortCompLED_1_.SetLeds;
    ControlConf3BasicControlInterface = WPortCompLED_1_.StdControl;
}

```

```

ControlConf3BasicControlInterface    =
CoordinatorConfBeaconNG_.ControlCoor4BasicControlInterface;
CoordinatorConfBeaconNG_.TimerReq3_Timer_req    ->
TimerComp3_.Timer[unique("TimerC")];
}

```

Figura 6-10. Configuración de segundo nivel para los nodos beacon

```

module CoordinatorConfBeaconNG
{
    uses {
        interface SetInt as WPortReq1_IntWrite;
        interface SendMsg as NetReq2_SendI;
        interface Timer as TimerReq3_Timer_req;
    }
    provides          interface StdControl as ControlCoor4BasicControlInterface;
}

implementation
{
    TOS_Msg g_message;
    TOS_MsgPtr msg10;
    uint16_t WriteValue17;
    int NodeID;

    task void TsCmWriteCall_setLEds2_Leds17();

    task void TsCmSendCall10();

    command result_t ControlCoor4BasicControlInterface.init() {

        return SUCCESS;
    }

    command result_t ControlCoor4BasicControlInterface.start() {

        call TimerReq3_Timer_req.start(TIMER_REPEAT, 1000);
        return SUCCESS;
    }

    command result_t ControlCoor4BasicControlInterface.stop() {

        return SUCCESS;
    }
}

```

```

event result_t NetReq2_SendI.sendDone(TOS_MsgPtr msgSDone, result_t successSD) {
    uint16_t output18;

    atomic {
        output18 = NodeID;
    }
    atomic {
        WriteValue17=output18;
    }
    post TsCmWriteCall_setLEds2_Leds17();
    return SUCCESS;
}

event result_t TimerReq3_Timer_req.fired() {
    uint16_t NodeID15;
    TOS_MsgPtr msg12;
    uint16_t output13;

    NodeID15 = TOS_LOCAL_ADDRESS;
    atomic {
        NodeID = NodeID15;
    }
    msg12=&g_message;
    atomic {
        output13 = NodeID;
    }
    atomic {
        ((uint16_t *)msg12->data)[1-1]=output13;           //Fill message param 1
    }
    atomic {
        msg10=msg12;
    }
    post TsCmSendCall10();
    return SUCCESS;
}

task void TsCmWriteCall_setLEds2_Leds17() {

    call WPortReq1_IntWrite.set(WriteValue17);
}

task void TsCmSendCall10() {

    call NetReq2_SendI.send(TOS_BCAST_ADDR, 8, msg10);
}
}

```

Figura 6-11. Controlador de los nodos beacon

El código generado es bastante eficiente en términos de extensión y de rendimiento comparado con el código nesC introducido directamente por un programador. Aún así, existen diversos puntos en los que su optimización es susceptible de mejorar, principalmente debido a la forma en la que se realiza la transformación de los bloques encargados del acceso a los recursos de almacenamiento del nodo. La modificación de esta transformación no es excesivamente compleja y permitiría ahorrar algunas líneas de código (pero muy pocas), sin embargo en cuanto a rendimiento de los motes y tamaño de la imagen de código compilada se estima que tendría un efecto inapreciable.

6.6. Conclusiones y aportación a la Tesis

En este capítulo se ha mostrado la herramienta desarrollada y su manejo, dando una serie de consideraciones para el desarrollo de aplicaciones y una metodología que permitan definir de forma sencilla aplicaciones conformes a sus requisitos.

Además, se ha validado la herramienta desarrollando una aplicación práctica de localización de personas. Esta aplicación se describe mediante un diagrama sencillo con un esfuerzo mucho menor que el empleado con una aproximación tradicional de desarrollo de software. Previamente al desarrollo de esta aplicación con la herramienta se había desarrollado una aplicación similar, programada directamente en nesC, donde el tiempo necesario para construirla fue bastante mayor (esta aplicación presentó problemas, por lo que se empleó un tiempo más de ciento cincuenta veces mayor en su desarrollo). El aprendizaje del lenguaje nesC se realizó en aproximadamente en ciento cincuenta horas siguiendo el tutorial de nesC. Para el desarrollo la aplicación nesC y la solución de los numerosos problemas que se presentaron se emplearon alrededor de sesenta horas. Por el contrario, el aprendizaje de la herramienta se puede realizar en pocas horas y la descripción de esta aplicación se hizo en veinte minutos.

Capítulo 7

Conclusiones y Líneas Futuras

En este capítulo se presentan las conclusiones obtenidas tras la realización de este trabajo de Tesis, así como los resultados que, hasta el momento, se han obtenido de ella. A continuación se enumeran las principales líneas de investigación que se abren a raíz de su realización de este trabajo y que permitirán su mejora y ampliación.

7.1. Conclusiones

El trabajo de Tesis expuesto ha abordado el desarrollo de aplicaciones software para redes de sensores inalámbricas. Para ello, se ha elaborado una propuesta basada en el estándar MDA y una herramienta que la lleva a la práctica, persiguiendo con éstas la reducción de la complejidad del proceso de desarrollo de software para estas aplicaciones.

Las redes de sensores son una tecnología en auge que puede ser aplicada en un gran número de ámbitos, normalmente al servicio de personal con pocos conocimientos de programación. Como consecuencia, la complejidad que implica el desarrollo de software para esta tecnología impide que éstos puedan desarrollar sus propias aplicaciones. Para resolver este problema, en el marco de esta Tesis, se ha desarrollado un lenguaje específico de dominio (DSL) que permite no sólo especificar una aplicación en función de conceptos gráficos fácilmente comprensibles sino que además evita el trato con los aspectos de la programación de bajo nivel que más dificultan la programación de aplicaciones.

Este DSL se ha integrado dentro de una enfoque de desarrollo MDE con el fin de obtener código ejecutable a partir de las descripciones realizadas con él, así como posibilitar que en un futuro el lenguaje pueda no ser utilizado únicamente para la generación de código sino también para la ejecución de otras tareas, como por ejemplo la simulación de aplicaciones sobre estas descripciones.

La generación de código a partir del DSL es un proceso complejo y resulta complicado poder obtener la totalidad del código de una aplicación. Además, típicamente no se alcanza una automatización total del proceso, siendo necesaria la introducción de información adicional en fases intermedias del proceso. A este respecto uno de los mayores logros de esta Tesis ha sido el poder lograr una generación completamente automatizada de la totalidad del código de las aplicaciones. El principal reto para llegar a esto ha sido la implementación de las transformaciones entre los modelos a distinto nivel de abstracción, por la complejidad de éstas y las dificultades encontradas para su depuración.

La integración del DSL y el conjunto de las transformaciones, tanto las ejecutadas entre modelos como las transformaciones a texto, dan lugar a una herramienta que cumple con las expectativas generadas en torno a la simplificación del proceso de desarrollo de software para redes de sensores. Una valoración preliminar de la mejora que introduce su uso, realizada mediante la comparación de los tiempos empleados por diversos programadores en la realización de aplicaciones frente al empleado a la descripción con el DSL, estima que el tiempo de desarrollo haciendo uso de la herramienta se reduce entre cinco y cien veces (incluso más en algunas situaciones) en función de la complejidad de la aplicación desarrollada y de los conocimientos del programador. Asimismo, se estima que el tiempo necesario para aprender a usar la

herramienta es alrededor de veinte veces menor que el empleado en aprender el lenguaje nesC y el uso de los componentes de TinyOS.

7.2. Aportaciones

Tras la finalización de esta Tesis se considera alcanzado el objetivo marcado inicialmente de generar código automáticamente para redes de sensores. En particular se obtiene código en el lenguaje nesC que hace uso de los componentes provistos por el sistema operativo TinyOS en su versión 1.1. El código funciona correctamente sobre la plataforma hardware *TelosB* aunque algunas de sus aplicaciones han podido ser ejecutadas sobre otras plataformas como *MICA2* o *MICAz*. A continuación se detallan los resultados más relevantes logrados durante el desarrollo de la Tesis:

- Se han estudiado las redes de sensores, confirmando TinyOS como la mejor alternativa como plataforma de implementación. De este estudio también se han extraído los conceptos más relevantes de una red de sensores mediante los cuales se puede especificar el comportamiento de una aplicación.
- Se han estudiado el estándar MDA y las diferentes plataformas de desarrollo que permiten llevarlo a la práctica. De entre éstas se ha elegido el entorno de desarrollo Eclipse y su *framework* de modelado EMF por la gran cantidad herramientas y lenguajes que soportan y por tanto la interoperabilidad que ofrecen.
- Se ha construido un lenguaje específico de dominio (DSL) para redes de sensores basado en la representación gráfica de los principales conceptos de las mismas. Estos conceptos, además de tener una interpretación intuitiva, no están vinculados con ninguna plataforma de ejecución, por lo que las descripciones realizadas con el DSL pueden ser potencialmente usadas para generar código para un gran número de plataformas. Todas estas características hacen que el uso del lenguaje reduzca significativamente el tiempo de desarrollo de aplicaciones y evite la necesidad de aprender a usar el lenguaje de programación y los recursos de una plataforma de ejecución. El desarrollo de este lenguaje ha requerido la creación, en primer lugar, de un metamodelo en el que plasmar la sintaxis abstracta del lenguaje y que recoja los conceptos del lenguaje y la forma en que éstos se relacionan entre ellos. En segundo lugar se han especificado las representaciones visuales de cada uno de sus elementos (sintaxis concreta) mediante el *framework* de modelado gráfico GMF. Este último también ha permitido la construcción de un editor gráfico para la construcción de modelos. GMF ha sido muy útil para la definición del lenguaje, sin embargo conseguir una representación gráfica de los elementos del lenguaje que coincida con el aspecto deseado no ha sido una tarea fácil.
- El DSL permite la construcción de modelos que sirven como punto de entrada a un proceso de generación de código MDA basado en las siguientes perspectivas

(modelos con distintos nivel de abstracción): a) modelos PIM contruidos con el DSL para redes de sensores, b) modelos PIM de menor nivel de abstracción correspondientes al *framework* de modelado V3-CM y c) modelos PSM correspondientes a la plataforma de ejecución nesC. A excepción del nivel V3-CM ya existente, se han tenido que crear los otros dos niveles, mediante la elaboración de un nuevo metamodelo para cada uno de ellos.

- Se han programado transformaciones entre modelos con el lenguaje de grafos AGG: desde el DSL a V3-CM, y desde éste último al PSM-nesC. Para ello, se han establecido mecanismos para realizar estas transformaciones basados en la identificación de patrones en los modelos para los casos más básicos. En casos más complejos, en los que un elemento del modelo origen debe ser transformado a un componente nesC preexistente (seleccionado de un conjunto de componentes en función de un rol asignado al elemento origen), se han usado marcas.

Para el uso de marcas se ha consultado el estándar MDA. Éste las usa para asignar roles a elementos del modelo origen, pero por cada rol debe programarse una transformación. Además, las marcas, dependientes de la plataforma, sólo son válidas para transformar a un determinado PSM. Se ha adoptado un nuevo sistema de marcas para la transformación de V3-CM al nivel PSM-nesC que se basa en el uso de un modelo de la plataforma nesC, que contiene todos los componentes nesC preexistentes (y sus interfaces, instancias de interfaces, etc.), el cual se pasa como entrada a la transformación. Las marcas se han asignado tanto a este modelo como a los modelos V3-CM, de forma que las transformaciones entre modelos se han hecho mediante la identificación de patrones comunes de marcas en ellos. Esto ha permitido la simplificación de las transformaciones, más genéricas ya que por cada rol no debe programarse una transformación. En su lugar, una única transformación puede ser utilizada para transformar elementos marcados con distintos roles, y el elemento que se creará en el modelo destino vendrá determinado por las marcas del modelo de la plataforma. Gracias a esto se puede modificar la forma en la que un rol se transforma sin tener que modificar las transformaciones. Las marcas, además, han sido asignadas de forma automática y no manual como MDA propone, por lo que se mantiene automatizado todo el proceso.

- Las transformaciones desde los modelos PSM-nesC a código se realizan mediante el lenguaje MOFScript. Se basa en la transformación de distintos tipos de bloques de código y sus interconexiones al lenguaje nesC. Los tipos de bloque que deben ser transformados fueron definidos en la transformación a V3-CM y han sido soportados por esta transformación final.
- Se ha construido una herramienta denominada *MoteGen* que soporta la descripción de aplicaciones con el DSL y el posterior proceso de desarrollo de código.

- Se ha elaborado una metodología que permite la construcción de aplicaciones de forma guiada mediante la herramienta *MoteGen*. Esta metodología será probada más ampliamente para poder recoger información que permita mejorarla.
- El proceso de generación de aplicaciones, así como la herramienta que lo implementa han sido validados mediante la construcción de aplicaciones con ella. En esta memoria de Tesis se ha documentado el caso de una aplicación de localización de personas.

7.3. *Divulgación de resultados*

A continuación se enumeran las publicaciones relacionadas con esta Tesis, para la que aún se esperan más en un futuro próximo.

Revistas internacionales

- C. Vicente, 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, Vol. 16, Issue 3-4, pp. 393 – 412, ISSN 0218-8430, December, 2007 (listada en JCR, FI = 0'951)
- F. Losilla, P. Sánchez, B. Álvarez, A. Iborra, *WISE: A Tool for WSAN-Based Applications Using Domain Specific Languages and Code Generation*, IEEE Transactions on Wireless Communications (En proceso de revisión)

Revistas nacionales

- M. Jiménez, J.A. Vera, F. Losilla, P.J. Meseguer, *Redes de Sensores y Actuadores (WSAN) en domótica*, Revista Telecoforum, vol. 5, pp. 59-62. ISSN 1698-2924, 2007
- F. Losilla, P. Sánchez, B. Álvarez, *MDE para el desarrollo de redes de sensores inalámbricas siguiendo un enfoque de líneas de producto*, Revista Telecoforum, vol. 4, pp. 103 -105, ISSN 1698-2924, 2006

Congresos internacionales

- F. Losilla, C. Vicente-Chicote, B. Álvarez, A. Iborra, P. Sánchez, *Wireless Sensor Network Application Development: An Architecture-Centric MDE Approach*, First European Conference on Software Architecture (ECSA 2007), pp. 179-194, LCNS 4758, ISSN 0302-9743, Aranjuez 2007
- F. Losilla, P. Sánchez, C. Vicente, B. Álvarez, A. Iborra, *A WSAN Solution for Irrigation Control from a Model Driven Perspective*, First IFIP International Conference on Wireless Sensor and Actor Networks (WSAN'07), pp. 35 – 46, ISSN 1571-5736, Albacete 2007.

- M. Jiménez, J.A. Vera, F. Losilla, P. Sánchez, A. Iborra, *Experiencia en la integración de Redes de Sensores y Actuadores (WSAN) en domótica*, En: Actas del XIV Seminario Anual de Automática, Electrónica Industrial e Instrumentación (SAAEI'07). ISBN 978-968-9182-52-8. Puebla, México, 2007.
- D. Alonso, C. Vicente-Chicote, P. Sánchez, B. Álvarez, F. Losilla, *Automatic Ada Code Generation Using a Model-Driven Engineering Approach*, 12th International Conference on Reliable Software Technologies (Ada-Europe 2007), pp. 168-179, LNCS 4498, ISSN 0302-9743, 2007

Congresos y jornadas nacionales

- F. Losilla, C. Vicente-Chicote, P. Sánchez, B. Álvarez, *Una metodología para el desarrollo de aplicaciones WSAN siguiendo un enfoque dirigido por modelos*, VI Jornadas de Ingeniería Telemática (JITEL 2007), pp. 313-320, ISBN 978-84-690-6670-6, Málaga, 2007
- F. Losilla, P. Sánchez, B. Álvarez, *Introducción a las Redes de Sensores. Perspectivas para la Ingeniería del Software*, IV Jornadas de Trabajo DYNAMICA, Archena 2005.

Demostraciones

- Demostración de la herramienta *MoteGen* en las XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2009), San Sebastián, septiembre de 2009 (en proceso de revisión)

Informes Técnicos

- F. Losilla, J. Suardiaz, J.A. López, P. Sánchez, B. Álvarez, A. Iborra, P. Alcover, *Monitorización y Control de Riego en cultivos arbóreos*, Informe técnico proyecto MEDWSA (DT-MEDWSA-05), 15 páginas, 2007

Reuniones Científicas

- I Jornadas de Trabajo META: Models, Environments, Transformations and Applications. Ciudad Real, 2007.
- II Jornadas de Trabajo META: Models, Environments, Transformations and Applications. Cartagena, 2008.
- IV Jornadas de Trabajo DYNAMICA (*DYNamic and Aspect-Oriented Modeling for IntegratedComponent-based Architectures*). Archena, 2005

Marco de desarrollo

El trabajo de Tesis que se ha descrito en esta memoria ha sido realizado en el marco de los proyectos MEDWSA (*Marco conceptual y tecnológico para el desarrollo de software de sistemas reactivos*) de la Comisión Interministerial de Ciencia y Tecnología (CICYT TIN2006-15175-C05-02), perteneciente al proyecto coordinado META (*Models,*

Environments, Transformations and Applications) y CON-PARTE-1 (CONtribución a los nuevos PARadigmas y TECnologías de red para las comunicaciones del mañana, TEC2007-67966-C03-01) perteneciente al proyecto coordinado CON_PARTE.

7.4. Trabajos futuros

A consecuencia de la extensión de este trabajo de Tesis no todos los aspectos involucrados en el desarrollo de la herramienta han sido tratados con igual atención, centrándose en los que resultan fundamentales. Así, se ha dado prioridad a la descripción de aplicaciones con el DSL y la generación de código a partir de ellas para poder contar con un enfoque de generación de código completamente automatizado. El resto de aspectos que quedan abiertos presentan una menor importancia para el funcionamiento de la herramienta y no requieren cambios importantes en ésta, por lo que su introducción o ampliación no resulta compleja. A continuación se comentan algunos de estos aspectos.

En un primer lugar, aparecen cuestiones relacionados con la funcionalidad que ofrece el lenguaje. En este sentido, aunque se ha implementado casi la totalidad de los conceptos que propone, aún quedan unos pocos no soportados. Éste es el caso del almacenamiento permanente de datos y de las comunicaciones con características avanzadas. En ambos casos el principal reto para su introducción es la búsqueda de componentes TinyOS que realicen estas tareas o, en caso de no encontrarlos, la programación de nuevos componentes. En el caso del almacenamiento permanente, las transformaciones entre modelos, que son la parte más crítica de la herramienta, no tienen por qué ser modificadas pues ya están preparadas. En el caso de querer emplear comunicaciones radio más avanzadas (por ejemplo comunicaciones *multihop* o cifradas) los cambios en las transformaciones serían mínimos. Simplemente se requeriría marcar los componentes V3-CM encargados de las comunicaciones según las propiedades especificadas en la descripción de la aplicación. Los componentes nesC a los que se deben transformar deberían ser introducidos en el modelo de la plataforma TinyOS con las mismas marcas.

En segundo lugar, es necesario aumentar las restricciones que comprueban que la descripción de una aplicación está bien formada. En este sentido se deben añadir más reglas OCL a la semántica estática de la aplicación que impidan que determinadas relaciones entre los elementos del modelo WSN se den. Esto puede evitar que se modelen aplicaciones con errores o que se produzcan situaciones no soportadas por el proceso de generación de código.

Los cambios anteriores constituyen las principales líneas de actuación que deben seguirse antes de poder distribuir la herramienta para que posibles usuarios que no tengan gran dominio de ella puedan desarrollar aplicaciones WSN de alta calidad. Pero existe otra modificación que podría ser interesante. Ésta consistiría en generar código para la versión 2.0 del sistema operativo TinyOS, popularizada recientemente. Una ventaja importante de su incorporación es la existencia de un mayor catálogo de

componentes que pueden ser usados por las aplicaciones generadas, además de ofrecer una mayor portabilidad de su código entre plataformas hardware. El uso de esta nueva versión, en principio, sólo implicaría modificaciones en la transformación a texto y la construcción de un nuevo modelo de la plataforma TinyOS 2.0.

Por otro lado, existen una serie de ampliaciones que se pueden realizar sobre la herramienta, como son las que a continuación se comentan:

- Generación automática de código a partir de sus requisitos. En lugar de usar el DSL para redes de sensores como punto de partida del proceso de generación de código se pueden tener modelos en un nivel de abstracción superior como punto de partida. Estos modelos se corresponderían con un nivel CIM del estándar MDA y serían transformados a modelos PIM situados en el nivel correspondiente al DSL. Gracias a esta aproximación se puede especificar el comportamiento de una red de sensores de forma en global (macro-programación), en lugar de especificar el comportamiento individualmente para cada nodo (o grupo de nodos). En este sentido se están haciendo avances en otros dominios dentro del grupo DSIE de los que se espera se puedan aprovechar algunos de sus resultados.
- Ejecución de tareas de V&V (Verificación y Validación) sobre las descripciones realizadas con el DSL. Es posible verificar el cumplimiento de algunos de los requisitos de la aplicación (funcionales y no funcionales), como consumos de potencia y otras propiedades que deben cumplirse. De la misma forma, se puede verificar que no se producen ciertas situaciones no deseables durante el funcionamiento de una aplicación (estas situaciones varían según cuál sea la aplicación modelada). La simulación de aplicaciones en el dominio del problema también resulta interesante. En lugar de usar los simuladores en el dominio de la solución (sobre el código generado) se pueden realizar simulaciones sobre las descripciones gráficas realizadas con el DSL. Las simulaciones a este nivel ofrecen una información más útil y fácil de interpretar.
- Integración de los componentes generados por la herramienta con los generados por las herramientas MDE de otros dominios reactivos con los que trabaja el DSIE. Dado que todos los dominios comparten un nivel intermedio con modelos V3-CM, la integración se debe realizar a este nivel. Como resultado se puede conseguir que aplicaciones ejecutadas en las plataformas hardware de los distintos dominios interaccionen, o bien, que aplicaciones sean transformadas a código para dominios distintos al del DSL con el que fueron creadas. Por ejemplo, sería posible ejecutar las aplicaciones WSN sobre robots siempre que dispongan de soporte tanto hardware como de software para el manejo de cada uno de los elementos que se usen en la descripción de la aplicación (sensores, comunicaciones, etc.). Así, el robot podría ejecutar en paralelo tanto código proveniente de un DSL para robótica como del DSL de redes de sensores, e incluso podrían llegar a interactuar.

Bibliografía

- [Akyildiz 02] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. *Wireless sensor networks: a survey*. Computer Networks, 38, pp. 393--422, 2002
- [Alonso 08] D. Alonso, C. Vicente-Chicote, O. Barais, *V3Studio: A Component-Based Architecture Modeling Language*. Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2008), pp. 346-355, 2008
- [Amyot 06] D. Amyot, H. Farah, J. Roy, *Evaluation of Development Tools for Domain-Specific Modeling Languages*, In Proceedings of SAM'2006. pp.183-197, 2006
- [Atkinson 02] C. Atkinson, T. Kühne, *Model-Driven Development: A Metamodeling Foundation*, IEEE Software, 20 (5), pp. 36-41, 2002
- [Bakshi 05] A. Bakshi, V.K. Prasanna, J. Reich, and D. Larner, *The Abstract Task Graph: a methodology for architecture-independent programming of networked sensor systems*, In Proceedings of the 2005 Workshop on End-To-End, Sense-and-Respond Systems, Applications and Services (Seattle, Washington, June 05 - 05, 2005). International Conference On Mobile Systems, Applications And Services. USENIX Association, Berkeley, CA, pp. 19-24, 2005
- [Berkeley] *University of California, Berkeley*, <http://berkeley.edu>
- [Bézivin 01] J. Bézivin, O. Gerbé., *Towards a Precise Definition of the OMG/MDA Framework*, ASE'01, November 2001
- [Bézivin 06] J. Bezivin, *A gentle introduction to a new way of considering the construction and maintenance of information systems*, disponible en <http://www.sciences.univ-nantes.fr/lina/atl/ModelWare/>, última visita marzo 2009
- [Bhatti 05] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, *MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms*, ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks, vol. 10, no. 4, August 2005, guest co-editors P. Ramanathan, R. Govindan and K. Sivalingam, pp. 563-579
- [Bohm 66] C. Bohm, G. Jacopini, *Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules*, Communications of the ACM 9 (5), May 1966
- [Cheong 05] E. Cheong, E.A. Lee, Y. Zhao, *Viptos: A Graphical Development and Simulation Environment for TinyOS-based Wireless Sensor Networks*, In Proceedings of the

3th ACM Conference on Embedded Networked Sensory Systems, ACM, pp. 302-302., November 2005

[Christensen 03] N. H. Christensen, *Domain-specific languages in software development and the relation to partial evaluation*, PhD thesis, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark, 2003

[Crossbow] *Crossbow*, <http://www.xbow.com>, última visita: 04/2009

[Czarnecki 05] K. Czarnecki, *Overview of Generative Software Development*, In J.-P. Banâtre et al. (Eds.): *Unconventional Programming Paradigms (UPP) 2004*, Mont Saint-Michel, France, LNCS 3566, pp. 313–328, 2005

[David 05] Jea David, Srivastava Mani B, *Channels Characteristics for On-Body Mica2Dot Wireless Sensor Networks*, NESL Technical Report TR-UCLA-NESL-200502-03, February 2005

[Demir 06] A. Demir, *Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development*, In *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third international Workshop on Model-Based Methodologies For Pervasive and Embedded Software* (March 30 - 30, 2006). MBD-MOMPES. IEEE Computer Society, Washington, DC, 2006, pp. 75-83

[Deursen 00] A. Deursen, P. Klint and J. Visser, *Domain-specific languages: an annotated bibliography*, ACM SIGPLAN Notices, Vol. 35, No. 6, pp. 26 - 36, 2000

[DSLTools] *Microsoft DSL Tools*, <http://msdn.microsoft.com/vstudio/DSLTools/>

[DSM Forum] *Domain Specific Modeling Forum*, disponible en: <http://www.dsmforum.org/>, última visita marzo 2009

[Dunkels 04] A. Dunkels, B. Grönvall, and T. Voigt., *Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors*, In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, Tampa, Florida, USA, November 2004

[Eclipse] *Eclipse*, www.eclipse.org, última visita marzo 2009

[Emerson 05] M. Emerson, *GME-MOF: A MDA metamodeling Environment for GME*, Master's Thesis: Vanderbilt University, May, 2005

[EMF] *EMF (Eclipse Modeling Framework)*, disponible en <http://www.eclipse.org/emf/>, ultima visita enero 2009

[Favre 04] J. Favre, *Foundations of Meta-Pyramids: Languages vs Metamodels - Episode II: Story of Thotus the Baboon1*, in *Language Engineering for Model-Driven Software Development*, 2004

[Gay 03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, *The nesC Language: A Holistic Approach to Networked Embedded Systems*, Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003

[GEF] *GEF (Graphical Editing Framework)*, disponible en <http://www.eclipse.org/gef/>, última visita enero 2009

[GME 04] Institute for Software Integrated Systems, *The Generic Modeling Environment (GME)*, 2004, disponible en: <http://www.isis.vanderbilt.edu/Projects/gme/>, última visita febrero 2009

[GMF Tutorial] *GMF Tutorial*, disponible en: http://wiki.eclipse.org/GMF_Tutorial, última visita en enero 2009

[GMF] *GMF (Graphical Modeling Framework)*, en <http://www.eclipse.org/gmf/>, última visita en enero 2009

[Greenfield 04] J. Greenfield, K. Short, S. Kent, J. Crupi, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools* (1st ed.), Wiley, 2004

[Guizzardi 02] G. Guizzardi, L. Ferreira Pires and M. J. Sinderen, *On the Role of Domain Ontologies in the design of Domain-Specific Visual Modeling Languages*, In *2nd Workshop on Domain-Specific Visual Languages*, 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2002). Seattle, Washington, USA, 2002.

[Haan 09a] J. Haan, *Roles in Model Driven Engineering*, disponible en: <http://www.theenterprisearchitect.eu/archive/2009/02/04/roles-in-model-driven-engineering>

[Haan 09b] J. Haan, *Model Driven Engineering tools compared on user activities*, disponible en: <http://www.theenterprisearchitect.eu/archive/2009/02/18/>

[Han 05] C. Han, R. K. Rengaswamy, R. Shea, E. K. and M. Srivastava, *SOS: A dynamic operating system for sensor networks*, Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys), 2005

[Hill 00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. *System Architecture Directions for Networked Sensors*. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press., pages 93--104, Cambridge, MA, 2000

[Huang 03] G. Huang, *Casting the Wireless Sensor Net*, July/August 2003 MIT Technology Review, pp. 51-56

[Iborra 09] A. Iborra, D. Alonso, F. Ortiz, J.A. Pastor, P. Sánchez, B. Álvarez, *Experiences using Software Engineering in the design of Service Robots*, *Robotics and Automation*

Magazine, Special Issue on Software Engineering for Robotics, Vol. 16, No. 1, pp. 24-33, 2009

[IEEE] *IEEE*, <http://www.ieee.org>

[Intel] *Intel Corporation*, <http://www.intel.com>

[Jiménez 09] M. Jiménez, F. Rosique, P. Sanchez, B. Álvarez, A. Iborra, *Habitation: a Domain Specific Language for Home Automation*, IEEE Software, Special Issue on Domain Specific Languages & Modeling, 2009

[Jones 96] C. Jones, *Programming Languages Table, Release 8.2, March 1996*, Software Productivity Research, disponible en : <http://www.mindspring.com/~dway/smalltalk/docs/0langtbl.pdf>, Ultima visita enero 2009

[Juang 02] P. Juang, H. Oki, Y. Wang, et al. *Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet*. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), Oct. 2002

[Kang 90] K. Kang, S.Cohen, J.Hess, W.Novak and S.Peterson, *Feature-Oriented Domain Analysis (FODA)*, Software Engineering Institute, CMU/SEI-90-TR-21, 1990

[Kent 02] S. Kent, *Model Driven Engineering*, Proceedings of IFM2002, LNCS 2335, Springer, pp. 286-298, 2002

[Kleppe 03] A. Kleppe, W. Bast, and J. Warmer. *Mda Explained: the Model Driven Architecture: The Model Driven Architecture: Practice and Promise*, Addison-Wesley Professional, 2003

[Kolovos 06] D. S. Kolovos, R. F. Paige, T. Kelly, and F. A. C. Polack, *Requirements for domain-specific languages*, in Proceedings of the First ECOOP Workshop on Domain-Specific Program Development, Nantes, France, 2006

[Ledeczi 01] A. Ledeczi et al., *The Generic Modeling Environment*, In Proceedings of WISP'2001, Budapest, Hungary, May 2001

[Ledeczi 05] A. Ledeczi, G. Balogh, Z. Molnar, P. Volgyesi, M. Maroti, *Model Integrated Computing in the Large*, in IEEE Aerospace Conference 2005, pp. 1 -8, march 2005

[Lennox 04] J. Lennox et al., *CPL: A Language for User Control of Internet Telephony Services*, Internet Engineering Task Force, IPTEL WG, April. 2004

[Lorincz 04] K. Lorincz, D. Malan, T.. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, S. Moulton, and M. Welsh, *Sensor Networks for*

Emergency Response: Challenges and Opportunities. In *IEEE Pervasive Computing*, Special Issue on Pervasive Computing for First Response, Oct-Dec 2004

[Madden 05] S. Madden, et al., *TinyDB: An Acquisitional Query Processing System for Sensor Networks*, *ACM Transactions on Database Systems*, vol. 30, pp. 122-173, No.1, March 2005

[Mainwaring 02] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, J. Anderson, *Wireless Sensor Networks for Habitat Monitoring*, *WSNA'02*, Atlanta, Georgia, September 2002

[Marrón 05] P.J. Marrón, O. Saukh, M. Krüger, and C. Große. *Sensor Network Issues in the Sustainable Bridges Project*, European Projects Session of the 2nd European Workshop on Wireless Sensor Networks, 2005

[MDA 03] Object Management Group (OMG), *MDA Guide v1.0.1*, disponible en: <http://www.omg.org/docs/omg/03-06-01.pdf>

[Mellor 02] S. J. Mellor, M. J. Balcer, *Executable UML: a Foundation for Model-Driven Architecture*, Addison-Wesley, 2002

[Mens 04] T. Mens, K. Czarnecki, and P. Van Gorp, *A Taxonomy of Model Transformations*, In *Dagstuhl Seminar Proceedings 04101*, 2004

[Mernik 05] M. Mernik, J. Heering, A. M. Sloane, *When and how to develop domain-specific languages*, *ACM Computer Surveys Vol 37*, No. 4, pp. 316-344, 2005

[MetaEdit] MetaCase, *MetaEdit +*, disponible en: <http://www.metacase.com/>, última visita enero 2009

[MOF 06] Object Management Group (OMG), *Meta Object Facility (MOF) Core Specification*, disponible en: <http://www.omg.org/docs/formal/06-01-01.pdf>

[Mohagheghi 07] P. Mohagheghi, V. Dehlen, *An Overview of Quality Frameworks in Model-Driven Engineering and Observations on Transformation Quality*, In L. Kuzniarz, J. L. Sourrouille, & M. Staron (Ed.), *Proceedings of the 2nd Workshop on Quality in Modeling, MoDELS 2007*, (pp. 3-17). Nashville, TN, USA

[MSDN] *Software Factories*, en MSDN, disponible en: <http://msdn.microsoft.com/>

[Muñoz 04] J. Muñoz, J. Pelechano: *MDA a Debate*, Actas del Primer Taller sobre Desarrollo Dirigido por Modelos, MDA y Aplicaciones (DSDM'04), 2004

[Newton 07] R. Newton, G. Morrisett, M. Welsh, *The regiment macroprogramming system*, In *Proceeding of 6th the International Conference on Information Processing in Sensor Network*, Cambridge, Massachusetts, USA, pp. 489-498, 2007

[Oldevik 05] J. Oldevik, T. Neple, R. Gronmo, J. Aagedal, and A. Berre, *Toward Standardised Model to Text Transformations*, Proceedings of Model Driven Architecture. Foundations and Applications: First European Conference, ECMDA-FA, Springer Berlin / Heidelberg, pp. 39 – 253, 2005

[Ortiz 07] F.J. Ortiz, J.A. Pastor, D. Alonso, B. Álvarez, P. Sánchez , *Experiences using a component-oriented architectural framework for robots and its improvement with a MDE approach*, Lecture Notes in Computer Science, vol. 4758, pp. 335-338, 2007

[Palmer 97] J.D. Palmer, *Traceability*, Software Requirements Engineering, 1997

[Pister 00] K. Pister, *Vision of sensor networks in 2010*, disponible en: <http://robotics.eecs.berkeley.edu/~pister/SmartDust/in2010>

[Ranganath 03] Ranganath, V. P.; Childs, A.; Greenwald, J.; Dwyer, M. B.; Hatcliff, J.; Singh, G. Cadena: enabling CCM-based application development in Eclipse. In Proceedings of the OOPSLA Workshop on Eclipse Technology Exchange, Anaheim, California, October 2003, pp. 27-27

[Riesco 02] M. Riesco, *Especificación de Sistemas Reactivos Distribuidos utilizando Estelle síncrono*, disponible en: <http://www.di.uniovi.es/~albizu/tesis/Memoria.pdf>, última visita: marzo 2009

[Römer 04] K. Römer, F. Mattern, *The design space of wireless sensor networks*, in IEEE Wireless Communications, December 2004, pp. 54-61

[Saad 08] M. Al Saad, E. Fehr, N. Kamenzky, J. Schiller, *ScatterFactory2: An Architecture Centric Framework for New Generation ScatterWeb*, New Technologies, Mobility and Security, 2008. NTMS '08

[Santos 07] P. Santos-Neto, R. Resende, & C. Pádua, *Requirements for Information Systems Model-Based Testing*, in Proceedings of the 2007 ACM symposium on Applied computing, Seoul, Korea, 2007, pp. 1409-1415

[Schmidt 06] D. C. Schmidt, *Guest Editor's Introduction: Model Driven Engineering*, Computer, vol. 39, no. 2, pp. 25-31, 2002

[Selic 03] B. Selic, *The Pragmatics of Model-Driven Development*, in IEEE Transactions on Software Engineering, vol. 20, issue 5, IEEE Computer Society, pp. 19-25, 2003

[Sendall 03] S. Sendall, & W. Kozaczynski, *Model Transformation - the Heart and Soul of Model-Driven Software Development*, IEEE Software (vol. 20, no. 5), pp. 42-45, 2003

[Solheim 06] I. Solheim, & T. Neple, *Model Quality in the Context of Model-Driven Development*. In L. F. Pires, & S. Hammoudi (Ed.), Proceedings of the 2nd International Workshop on Model-Driven Enterprise Information Systems, MDEIS 2006 (pp. 27-35). Paphos, Cyprus: INSTICC Press 2006

- [SysML 06] Object Management Group (OMG), *Systems Modeling Language (OMG SysMLTM) Specification*. 2006
- [Taentzer 05] Taentzer, G., *AGG: The Attributed Graph Grammar System* (2005), <http://tfs.cs.tu-berlin.de/agg>
- [Tolvanen 05] Tolvanen, J.-P., Kelly, S., *Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences*, Proceedings of the 9th International Software Product Line Conference, H. Obbink and K. Pohl (Eds.) Springer-Verlag, LNCS 3714, pp. 198 – 209, 2005
- [Tolvanen 06] Tolvanen J.P., *Domain-Specific Modeling: How to Start Definig Your Own Language*, 2006, disponible en <http://www.devx.com/enterprise/Article/30550>, última visita enero 2009
- [UML 07] Object Management Group (OMG), *UML v2.1.2*, disponible en: <http://www.omg.org/spec/UML/2.1.2/>
- [Volgyesi 02] P. Volgyesi, A. Ledeczi. *Component-Based Development of Networked Embedded Applications*. 28th Euromicro Conference, Sep 2002
- [Warneke 01] B. Warneke, M. Last, B. Liebowitz and K. Pister, *Smart Dust: Communicating with a Cubic-Millimeter Computer*, IEEE Computer, 34, pp. 44–51, 2001
- [Werner-Allen 05] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. *Monitoring volcanic eruptions with a wireless sensor network*. In EWSN'05, Istanbul, Turkey, January 2005
- [Weiss 95] D.. Weiss, *Software Synthesis: The FAST Process*, in Proceedings of the International Conference on Computing in High Energy Physics, 1995
- [Wile 01] D. Wile, *Supporting the DSL Spectrum*, Journal of Computing and Information Technology (CIT), Volume 9, Issue 4, pp. 263–287, 2001
- [XMI 05] Object Management Group (OMG), *MOF 2.0/XMI Mapping Specification, v2.1*, disponible en: <http://www.omg.org/cgi-bin/apps/doc?formal/05-09-01.pdf>
- [ZigBee] ZigBee Alliance, *ZigBee*, disponible en www.zigbee.org

Anexo I. Instalación de la herramienta MoteGen

Para instalar la aplicación es necesario disponer Eclipse instalado junto con una serie de paquetes de modelado: **EMF v2.4.0**, **GEF v3.4.0**, **GMF v2.1.0**. Tanto Eclipse como los paquetes anteriores se pueden instalar automáticamente descargando la distribución *Eclipse Modeling Tools* desde <http://www.eclipse.org/downloads/>

Una vez instalados se debe a proceder a instalar los paquetes no incluidos en esta distribución para lo cual es recomendable hacer uso de la actualización automática de Eclipse (*Help -> Software Updates -> Find and Install*). En este caso es necesario especificar una *url (update site)* en la que buscar las actualizaciones. Los paquetes que se deben actualizar son:

EMF Model Transformation (transformaciones M2M mediante AGG)

Update site: <http://user.cs.tu-berlin.de/~emftrans/update>

MOFScript

Prerrequisito: ANTLR plug-in

- Update site: <http://antreclipse.sourceforge.net/updates/>

Update site: <http://download.eclipse.org/modeling/gmt/mofscript/update/>

Tras tener todos los paquetes instalados se procede a instalar la herramienta *MoteGen*. Para ello se debe disponer de una carpeta con todos los archivos de la herramienta. Dentro de ella, en la carpeta *Projects*, se encuentran todos los proyectos que componen la herramienta: *CIM2PIM*, *CIM2PIMrules*, *GMFWSAN*, *GMFWSAN.diagram*, *GMFWSAN.edit*, *GMFWSAN.editor*, *nesC*, *nesC.edit*, *nesC.editor*, *PIM2PSM*, *PIM2PSMrules*, *V3Studio*, *V3Studio.edit*, *V3Studio.editor*.

Éstos se deben importar a un *Workspace* de Eclipse mediante:

File->import->Existing Ptojects into Workspace

En la ventana de diálogo que aparece se debe buscar la carpeta de la que cuelguen todos estos proyectos, seleccionarlos todos automáticamente mediante el botón *Select All*, marcar la casilla *Copy projects into workspace* y pulsar sobre el botón *Finish*.

Finalmente se debe copiar la librería de archivos *nesC* de la herramienta (que serán usados por los componentes generados automáticamente). Esta librería se encuentra dentro de las carpetas de instalación de la herramienta, concretamente en la carpeta llamada *Eclipse*. Ésta, junto con todo su contenido, debe ser copiada en el

directorio del sistema operativo nesC que contiene las librerías de componentes, típicamente:

C:\Archivos de programa\UCB\cygwin\opt\tinyos-1.x\tos\lib

Anexo II. Generación de código para redes de sensores mediante MetaEdit+

Introducción

De forma paralela al trabajo que se ha expuesto en esta Tesis se ha llevado a cabo otra aproximación al desarrollo de software para redes de sensores distinta a la adoptada en esta Tesis. Esta nueva aproximación se enmarca dentro de un Proyecto Final de Carrera realizado por D. José Antonio Moñino Martínez y dirigido por D. Fernando Losilla López. En ella se ha desarrollado una herramienta para la especificación y generación de código para aplicaciones WSN (*Wireless Sensor Networks*) mediante el uso del entorno comercial de desarrollo MetaEdit+ [MetaEdit] de la compañía *MetaCase*. Como resultado se obtiene código en el lenguaje nesC de TinyOS que ha sido probado sobre motes *TelosB*.

La aproximación seguida con esta herramienta es más simple que la que se ha presentado a lo largo de esta Tesis. Se basa en el uso de un DSL para redes de sensores para la descripción de aplicaciones y la transformación de cada uno de sus elementos a código mediante la ejecución de un script. Esta aproximación ha sido válida para el desarrollo de aplicaciones simples para la monitorización de datos pues se ha podido encontrar correspondencias directas de cada uno de los elementos básicos del DSL (sensores, temporizadores, etc.) con componentes nesC (a los que se transforman). En este caso las transformaciones son considerablemente más sencillas que las transformaciones haciendo uso de modelos intermedios. Sin embargo cuando para un elemento del DSL no se puede encontrar una correspondencia con un componente nesC, su transformación se complica. Como consecuencia el DSL sólo permite el uso de ciertos conceptos de las redes de sensores fácilmente transformables con esta aproximación. Esto restringe el conjunto de aplicaciones que pueden ser modelados con esta herramienta. Sin embargo, las aplicaciones susceptibles de ser descritas con ella, se pueden desarrollar de forma muy sencilla.

Hubiera sido posible seguir una aproximación similar haciendo uso del entorno Eclipse en lugar de *MetaEdit+*. Tras la experiencia obtenida en el desarrollo de ambas herramientas se ha llegado a la conclusión que emplear Eclipse sería más complejo, principalmente por la facilidad para la construcción que *MetaEdit+* ofrece frente a la tecnología GMF de Eclipse. Sin embargo, el lenguaje de transformación *MOFScript* disponible en Eclipse es más potente e intuitivo que el lenguaje de scripts de *MetaEdit+*.

DSL para redes de sensores

El DSL implementado con *MetaEdit+* dispone de una serie de categorías en las que se encuadran sus conceptos como se muestra a continuación.

Sensor	Elemento hardware que mide el valor de ciertas magnitudes físicas (temperatura, humedad, etc.).
Control	Elemento que al ser activado inicia la ejecución de un proceso, por ejemplo un botón.
Temporizador	Elementos que periódicamente activa la ejecución de un proceso
Actuador	Elemento receptor de eventos para los cuales realiza un procesamiento (envío a un PC, encendido de un led, etc.)
Radio	Elemento para la comunicación entre motes

Una aplicación para redes de sensores se modela mediante la interconexión de elementos del DSL correspondientes a los anteriores tipos. Las interconexiones realizadas entre elementos se representan mediante flechas que representan el orden en que la aplicación va ejecutar sus distintas tareas. La ejecución comienza cuando se activa un elemento de Control, que provoca la ejecución secuencial de todos los elementos conectados a él. En la siguiente figura se puede ver un ejemplo. En ella cuando se pulsa el botón de usuario del mote se enciende un led. En paralelo, al encenderse el mote (se activa el elemento START), se arranca un temporizador que periódicamente (cada 250 ms) solicita una lectura de la humedad relativa. Cuando el valor obtenido es mayor del 60% se apaga el led anterior.

Ilustración 1. Aplicación descrita con el DSL de MetaEdit+

A continuación describen cada uno de los elementos del DSL.

Categoría	Elemento	Icono	Propiedades
Control	Start: Representa el inicio de la ejecución en una aplicación		Ninguna.
Temporizador	Timer One Shot: Temporizada que se ejecuta una sola vez		<p>Milliseconds: Valor de temporización. Type: Timer Repeat vs Timer One Shot</p>
	Timer Repeat: Temporizador que se ejecuta repetidamente
Sensor	Mide la luminosidad total incluyendo infrarrojos (TSR)		<p>Type: Tipo de sensor usado Threshold: Tipo de umbral para las lecturas del sensor: [>,<==,!=,>=,<=, 'Value']. El término <i>Value</i> indica que no hay detección de umbral. El segundo campo indica el valor del umbral (expresado en °C para la temperatura, luxes para la luminosidad, tanto por ciento para la humedad relativa, etc.).</p>
	Mide la luminosidad fotosintéticamente activa (PAR)
	Mide temperatura exterior
	Mide humedad
	Mide la temperatura interna del microcontrolador del mote
Radio	Rfm: Acceso al canal radio. Dependiendo de si es el primer elemento de un flujo de ejecución o el		<p>Handler ID : Identificador de una</p>

	último actuará como emisor o receptor respectivamente		comunicación de datos. Un emisor y su receptor deben usar el mismo <i>Handler ID</i> . Store: Numero de lecturas enviadas por mensaje (mientras no se llega a ese número éstas se van acumulando).
Control	Button: Entrada digital de los motes (p. ej. <i>TelosB</i>). Generará un evento cada vez que ésta se active.		Port: Número de la entrada digital en el microcontrolador (el valor por defecto 27 se corresponde con el botón de usuario de los motes <i>TelosB</i>).
Actuador	Computer: Transmisión de datos a un PC. Transmite los datos que llegan del elemento que le precede		Handler ID: Identificador del mensaje que se pasa al PC. Se usa para procesar el mensaje en el PC de forma conveniente. Store: Cantidad de datos que se transmiten al PC (en un mensaje) Channel: Se usa para hacer el mensaje compatible con la herramienta <i>Oscilloscope</i> (disponible en la instalación de TinyOS) la cual representa los datos recibidos en gráficas 2D.
Actuador	Led: Representa los leds instalados en un mote (en particular un mote <i>TelosB</i>). Se usa para activarlos y desactivarlos.		Colour: Color del led (azul, amarillo o rojo). Action: Acción que se ejecutará sobre el led. Las opciones son 'On', 'Off' y 'Toggle' (encender, apagar y conmutar).

Generación de código

La generación de código se lleva a cabo mediante la ejecución de una serie de scripts de texto. Como se comentó, estos scripts generan un nuevo componente por cada elemento del DSL. Los componentes generados tienen una estructura predefinida, que se completa con la información existente en las propiedades de los elementos. Una vez transformados los componentes se procede a conectarlos entre sí, lo que se hace en la misma forma que sus elementos de origen en la descripción con el DSL. Por lo tanto, las relaciones entre elementos del DSL se transforman a enlaces entre instancias de interfaz de los componentes. Cada componente tendrá una instancia de interfaz de entrada por la que recibirá peticiones de servicio y datos de entrada y una instancia de interfaz de salida por la que solicitará servicios de otros componentes y les enviará datos de salida. El problema para realizar las conexiones entre las instancias de interfaz (asociadas a un interfaz) de dos componentes es la heterogeneidad de los interfaces de los distintos componentes nesC. Haciendo uso de los componentes TinyOS estándar no se puede realizar esta interconexión ya que cada uno usa interfaces distintas. Como consecuencia se ha optado por encapsular todos los componentes del sistema operativo en otros componentes que provean y usen un mismo tipo de interfaz. Esta interfaz se ha denominado *IntOutput* y permite el intercambio de datos de tipo entero. En caso de componentes que no requieran datos de entrada el paso de un entero a través de esta interfaz activa al componente. En la siguiente figura se puede ver la arquitectura de componentes (con sus interfaces e instancias de interfaz). Hace uso de un componente *main*, provisto por el sistema operativo TinyOS, que se encargará de inicializar al resto de componentes. Los componentes *Start*, *RadioReceive*, *Button* son el inicio de un flujo de ejecución, activando por medio de las instancias de interfaz *IntOutput* a los componentes *Timer* o *Sensor*, los cuales mandan sus datos a los componentes *RadioSend*, *Led* o *UART*.

Anexo III. Transformación M2M a PIM V3-CM

A continuación se muestran las reglas empleadas en la transformación desde los modelos asociados a las descripciones del DSL a modelos V3-CM. Por cada una de ellas se muestra su nombre y su capa. La capa de una regla determina el orden en que será ejecutada, de forma que primero se ejecutan todas las reglas de la capa 0, a continuación las de la capa 1 y así sucesivamente.

CreateInterfaceControl, capa 2

CreateTopLevelApp, capa 2

CreateDataTypes, capa 2 *CreateInterfaceSensor, capa 2* *CreateInterfaceSetInteger, capa 2*

CreateInterfaceGetInteger, capa 2 *NodeGroupToComplexComponent, capa 2* *TimerToCOTS, capa 2*

CreateStateMachine, capa 2 *AuxFunctionalUnitString*, capa 0 *CreateAux*, capa 0

ReadTask, capa 2

ReadActivation, capa 2

WriteTask, capa 2

WriteActivation, capa 2 *Load0Task*, capa 2 *BLinkTaskAoriginFU*, capa 2

BLinkTaskAoriginUL, capa 2 **BLinkTaskCControlFlow, capa 2** **BLinkTaskDDataFlow, capa 2**

BlockTaskNoIn, capa 2 **BlockTaskNoOut, capa 2** **BlockTaskIn, capa 2** **BlockTaskOut, capa 2** **BlockTaskDone, capa 2**

AuxTimerString, capa 2 *TimerOneShotTask*, capa 2 *TimerOneShotActivation*, capa 2

SendMessageActivation, capa 2 *SendMessageTask, capa 2*

SetMessageParam, capa 2 *MulLinkFirst*, capa 2 *MulLinkConnect*, capa 2

SetMessageParamUnique, capa 2 *BlockTaskFirst, capa 2* *BlockTaskLast, capa 2* *MulLinkLast, capa 2*

ReceiveMessageTask, capa 2

ExtractTask, capa 2

GetMessageParam, capa 2 *GetMessageParamUnique, capa 2*

RadioToUARTTask, capa 2 **UARTToRadioActivation, capa 2**

UARTToRadioTask, capa 2

ReadVarActivation, capa 2

ReadVarTask, capa 2

WriteVarActivation, capa 2

WriteVarTask, capa 2

UnitLinkToEdge, capa 2

ExtractVarTask, capa 2

FillVarTask, capa 2

StartTask, capa 2

StartTaskActivation, capa 2 **ConditionalTaskActivation, capa 2** **ConditionalTask, capa 2**

ConditionalTrue, capa 2 *ConditionalFalse, capa 2* *CalculateExpressionActivation, capa 2*

CalculateExpressionTask, capa 2 **DeleteInitMessage, capa 2** **DeleteInitMsgParam, capa 2**

ExtractVarStrengthTask, capa 2 **GetIdActivation, capa 2** **GetIdTask, capa 2**

MulLinkConnectLastData, capa 2 *SetMessageParamNone*, capa 2

SetActivation, marca 2 *ListMarkA*, capa 1 *ListMarkB*, capa 1

HashMarkA, capa 1 *HashMarkB*, capa 1 *ResetComplexObjectMark*, capa 1

MeanComplexObjectMark, capa 1 *MaxComplexObjectMark*, capa 1 *MinComplexObjectMark*, capa 1 *ListProcess*, capa 4 *DelListifHash*, capa 3

HashProcess, capa 4 *MeanRenameAct*, capa 3 *WriteMeanArrayRenameAct*, capa 3 *MaxRetrieveRenameAct*, capa 3

HashSelectKey, capa 3 *ResetRenameAct*, capa 3

Anexo IV. Transformación M2M a PSM-nesC

A continuación se muestran las reglas empleadas en la transformación desde los modelos V3-CM a modelos PSM-nesC. Por cada una de ellas se muestra su nombre y su capa. La capa de una regla determina el orden en que será ejecutada, de forma que primero se ejecutan todas las reglas de la capa 0, a continuación las de la capa 1 y así sucesivamente.

COTSMarksRelation, capa 0

CreateAuxxMM, capa 0

DeleteInvalidMarkAssoc, capa 0

InterfaceMarkRelationProvides, capa 0

TrackStdControlInterface, capa 0

CopyComplexComponent, capa 2 *CopyInterfaces, capa 2* *TrackPMComponents, capa 2*

TrackPMInterfaces, capa 2 *TrackPMInterfaceInstances, capa 2* *ComplexComponentToConfiguration, cap 3*

ComponentToAlias, capa 3 *PortToInterfaceInstanceUses, capa 3*

SimpleComponentToModule, capa 3 *PortControlToProvidedInterface*, capa 3

PortToInterfaceInstanceProvidesFromDelegation, capa 3 *PortLinkAssemblyToWiring, capa 3*

PortControlToUsedInterface, capa 3 *SetParametrizedInterfaces*, capa 4

SetSendInterfaceParam, capa 5 *ServiceMarkRelationBConstantParams, capa 6* *ServiceMarkRelationCParameters, capa 6*

ServiceCallToNesca_Event, capa 6 *ActivityNesComponentRelation*, capa 6

ServiceMarkRelationDAddProto, capa 6 *ServiceCallToNesCB*, capa 6

UserDefinedToNesCA, capa 6 *ActivityToNesCTask, capa 6* *ActivityToNesCLast, capa 6*

ControlFlowToNesC, capa 6 *ServiceMarkRelationERelateRest, capa 6*

BasicBlockAssigningOwnerA, capa 8 *ConditionalToNescActivityA*, capa 6 *ConditionalToNescIfTrue*, capa 12

ConditionalToNescCallIfFalse, capa 12 *BasicBlockAssingOwnerB*, capa 8 *ConstantActivityToNescNop*, capa 6

AssignControlLinksA, capa 8 *AssignControlLinksB*, capa 8 *RedirectControlLinks*, capa 9

ActivityParamToVariableA_Events, capa 9 *ActivityParamToVariableB_Commands*, capa 10 *ActivityParamToVariableC_BasicBlocks*, capa 10

ObjectFlowToDataFlow, capa 10 *DataLinkAssignOwnerA*, capa 10 *DataLinkAssingOwnerB*, capa 10

RenameCalculateExpression, capa 10 **CreateConstantsParamPrototypeCalls**, capa 10 **SetDataLinkSameVariableTypeA**, capa 11

ConnectParamPrototypeCalls, capa 10 *SetDataLinkSameVariableTypeB*, capa 11 *SetControlLinkCallType*, capa 10

SetConstantParams, capa 10 *SetControlLinkFlowTypeA*, capa 11 *SetControlLinkFlowTypeB*, capa 11

SetApplications, capa 12 *SetDataLinkCopyVariableType*, capa 11 *CreateDataLinkProto*, capa 12

SetWiringAssemblyKind, capa 3 *SetWiringDelegationKind*, capa 3

Anexo V. Transformación M2T a código

```

/**
 * transformation NesC
 *
 */

texttransformation NesCM2T (in mm:"NesC") {
    var typeEqs:Hashtable

    var tCBParams:Hashtable // Relates variables with input types to CodeBlocks (input, output, pnum, etc.)
    var tVarNames:Hashtable // Relates variables with their actual names in .nc files (without $)
    var globalVarDecl:Hashtable // Global variables declaration
    var globalVarCB:Hashtable // Relates CodeBlock's types with the global variable they produce
    var tabs:Integer // tabulation control
    var outputDir:String // Root output directory
    var ReturnVal:String // return value of the complex block in generation
    var numParamsMsg:Integer
    var app:mm.Application // Auxiliar variable
    var enums:Hashtable // Constant declaration
    var AMid:Integer // AM constants count
    var uniqueComps:Hashtable // generated components

    mm.Application::main(){
        stdout.println ("Generating nesC files...")

        outputDir="c:\\Temp\\"
        app=self

        //initilizations
        tabs=0
        AMid=0

        // Fill variable related tables
        self.associateDataTypes()
        self.initGlobalVarsList()

        // generate code
        self.createNodeImages()

        stdout.println ("Done")
    }

    // Assign data types equivalencies (from models to nesC language)
    mm.Application::associateDataTypes(){
        typeEqs.put("integer","uint16_t")
        typeEqs.put("int","uint16_t")
        typeEqs.put("message","TOS_MsgPtr")
        typeEqs.put("Message","TOS_MsgPtr")
        typeEqs.put("TOS_MsgPtr","TOS_MsgPtr")
    }

    // Get name without $
    mm.InterfaceInstance::getActualRename():String{
        if (self.renamedAs.indexOf("$")>0)
            result=self.renamedAs.subStringBefore("$")
        else
            result=self.renamedAs
    }

    mm.Application::getActualName(s:String):String{
        if (s.indexOf("$")>0)
            result=s.subStringBefore("$")
        else
            result=s
    }

```

```
}

// Create simple component
mm.SimpleComponent::create(path:String){
  var sComp:mm.SimpleComponent
  sComp=Self

  tCBParams.clear()
  tVarNames.clear()
  globalVarDecl.clear()

  sComp.createVarToTypeTable(tCBParams, tVarNames)

  file sal(path+sComp.name+".nc")

  sal.println("module "+sComp.name+"\n{")
  sComp.UsesAndProvides()
  sal.print("}")
  n(2)

  var cont:integer
  sal.println("implementation\n{")

  // global variables
  sComp.declareGlobalVars(globalVarDecl, tVarNames)
  globalVarDecl->forEach(vardec){ println("\t"+vardec+";") }
  n(2);

  // task prototypes
  sComp.objectsOfType(mm.TaskImplementation)->forEach(cb:mm.TaskImplementation){
    var str:String
    if (cb.name.indexOf("$")>0) str=cb.name.subStringBefore("$")
    else str=cb.name
    println("\ttask void "+str+"();\n")
  }

  // commands code
  sComp.objectsOfType(mm.CommandImplementation)-
>forEach(cb:mm.CommandImplementation){
    cb.doComplexBlockExt(cb.implements, cb.instanceref)
  }

  // events code
  sComp.objectsOfType(mm.EventImplementation)->forEach(cb:mm.EventImplementation){
    cb.doComplexBlockExt(cb.implements, cb.instanceref)
  }

  // tasks code
  sComp.objectsOfType(mm.TaskImplementation)->forEach(cb:mm.TaskImplementation){
    cb.doTask()
  }
  println("}\n");
}

//}

// task code
mm.ComplexBlock::doTask(){
  var cb:mm.ComplexBlock=Self
  var taskName:String
  tabs=1
  // get actual name
  if (cb.name.indexOf("$")>=0)
    taskName=cb.name.subStringBefore("$")
  else taskName=cb.name
  println("\ttask void "+taskName+"() {}")
  tabs=2

  //local variables
  self.printLocalVars()
```

```

        // code for internal CodeBlocks
        cb.isSource->forEach(cl:mm.ControlLink){
            cl.do()
        }

        tabs=1
        println("\t}\n")
    }

    // print with current tabulation
    mm.CodeBlock::prTab(str:String){
        var i:Integer
        i=tabs
        i->forEach(n){tab(1)}
        print(str)
    }
    mm.UserDefBlock::prTab(str:String){
        var i:Integer
        i=tabs
        i->forEach(n){tab(1)}
        print(str)
    }
    mm.ExternalCall::prTab(str:String){
        var i:Integer
        i=tabs
        i->forEach(n){tab(1)}
        print(str)
    }
    mm.ControlLink::prTab(str:String){
        var i:Integer
        i=tabs
        i->forEach(n){tab(1)}
        print(str)
    }
}

// code for a command or event
mm.ComplexBlock::doComplexBlockExt(proto:mm.Prototype, iInst:mm.InterfaceInstance){
    var cb:mm.ComplexBlock=self

    // the code is obtained from codeblocks or directly from code attribute?
    if (proto==null) println("\t"+cb.code)
    if (proto!=null) {
        if (proto.returnType.equals("result_t")) ReturnVal="SUCCESS"
        else ReturnVal=null

        // command/event declaration
        print("\t")
        if (proto.isAsynchronous==true) print("async ")
        if (cb.oclIsTypeOf(mm.CommandImplementation))
            print("command ")
        else if (cb.oclIsTypeOf(mm.EventImplementation))
            print("event ")
        print(proto.returnType+" ")

        var interfaceName: String=cb.instanceref.getActualRename()
        print(interfaceName+".")
        print(app.getActualName(proto.name))
        if (cb.instanceref.parameters.size()>0){ // parameter of the interface

            iInst.parameters->forEach(lv:mm.Variable)
            print("[ "+lv.type+" "+lv.name+" ]")
        }
        print("(")
        if (cb.implements.arguments.size()>0){ // with parameters
            var aux:Integer=cb.implements.arguments.size()
            proto.arguments->forEach(lv:mm.Variable){
                print(lv.type+" "+lv.name)
            }
        }
    }
}

```

```

        if (aux>1) print(" ", " ")
        aux=aux-1
    } // end foreach
} // end of parameters
print("") {\n"}

    // local variables
    self.printLocalVars()

    tabs = 2

    // code for task calling and CodeBlocks
    var clList:List
    cb.isSource->forEach(cl:mm.ControlLink){

        cl.do()
    }

    // print return value obtained in CodeBlocks creation
    if (!(RetVal=null))
        println("\t\treturn "+RetVal+";")
    println("\t\t\n")

}

// print local variables
mm.ComplexBlock::printLocalVars(){
    var blockVars:Hashtable
    self.internalBlocks->forEach(bb:mm.BasicBlock){
        if (bb.oclIsTypeOf(mm.UserDefBlock) && (bb.code.endsWith("Mean"))){
            blockVars.put("\t\tint _mean, _count;", "\t\tint _mean, _count;")
        }
        if (bb.oclIsTypeOf(mm.UserDefBlock) && (bb.code.startsWith("Hash"))){
            blockVars.put("\t\tint _aux;", "\t\tint _aux;")
        }
        if ((not bb.code.indexOf("Max")=-1) || (not bb.code.indexOf("Min")=-1)){
            blockVars.put("\t\tint _selIndex;", "\t\tint _selIndex;")
        }
        bb.blockParams->forEach(bp:mm.Variable){
            var dlList:List
            dlList=bp.isTarget->select(dl:mm.DataLink |

                if (dlList.isEmpty()){
                    var !:List=globalVarDecl.keys()->select(k:String |

                        if (l.isEmpty()){ //if it is not global

                            if (not bp.constantValue)
                                println("\t\t"+typeEqs.get(bp.type)+"")

                        }
                    } // end dlList.isEmpty
                } // end foreach variable
            } // end foreach basicblock
            blockVars->forEach(v:String){ println(v)}
            println("")
        }

    // Print task call or generate code for the conected CodeBlock
    mm.ControlLink::do(){
        // task call
        if (self.linkType.equals("Call") && self.target.oclIsTypeOf(mm.TaskImplementation)){
            self.prTab("post "+app.getActualName(self.target.name) + "();\n")
        }
        // execute next CB (and the following ones)
        if (self.linkType.equals("Flow")){
            self.target.do()
            if (not self.target.isSource.isEmpty()){
                self.target.isSource->forEach(cl:mm.ControlLink){
                    cl.do()
                }
            }
        }
    }
}

```



```

    }
  }
}

}

// Code for a UserDefinedBlock (generic)
mm.UserDefBlock::do(){
  var params:Hashtable

  // assign CB variables to input types for CBs
  self.blockParams->forEach(p:mm.Variable){
    params.put(tCBParams.get(p.name),tVarNames.get(p.name))
  }

  // select type
  if (self.code.equals("NOP")){}
  if (self.code.equals("Retrieve")){self.doRetrieve(params)}
  if (self.code.equals("Store")){self.doStore(params)}
  if (self.code.equals("ListRetrieve")){self.doListRetrieve(params)}
  if (self.code.equals("ListStore")){self.doListStore(params)}
  if (self.code.equals("HashReset")){self.doHashReset(params)}
  if (self.code.equals("HashStore")){self.doHashStore(params)}
  if (self.code.equals("HashMaxRetrieve")){self.doHashMaxRetrieve(params)}
  if (self.code.equals("StoreStrength")){self.doStoreStrength(params)}
  if (self.code.equals("CalculateExp")){self.doCalculateExp(params)}
  if (self.code.equals("GetID")){self.doGetID(params)}
  if (self.code.equals("Conditional")){self.doConditional(params)}
  if (self.code.equals("Fill")){self.doFillMsg(params)}
  if (self.code.equals("Extract")){self.doExtractMsg(params)}
  if (self.code.equals("InitMsg")){self.doInitMsg(params)}

  // copy variables for use in CBs not belonging to the current one
  var dList:List
  self.blockParams->forEach(bp:mm.Variable){
    dList=bp.isSource->select(dl:mm.DataLink | dl.linkType.equals("Copy"))
    dList->forEach(dl:mm.DataLink){
      self.prTab("atomic {\n") //avoid data races and warnings
      self.prTab("\t"+tVarNames.get(dl.target.name)+"="+tVarNames.get(bp.name)+";\n")
      self.prTab("}\n")
    }
  } // end foreach variable
}

// Store CB
mm.UserDefBlock::doStore(params:Hashtable){
  self.prTab("atomic {\n")
  tabs=tabs+1
  self.prTab(self.name.substringBefore("$")+ " = ")
  print(params.get("input")+";\n")
  tabs=tabs-1
  self.prTab("}\n")
}

// StoreStrength CB
mm.UserDefBlock::doStoreStrength(params:Hashtable){
  var msg:String
  msg=params.get("msg")
  self.prTab("atomic {\n")
  tabs=tabs+1
  self.prTab(self.name.substringBefore("$")+ " = ")
  print("(int8_t)+msg+"->strength - 45;\n")
  tabs=tabs-1
  self.prTab("}\n")
  returnVal=msg;
}

```

```
}

// Retrieve CB
mm.UserDefBlock::doRetrieve(params:Hashtable){
  self.prTab("atomic {\n")
  tabs=tabs+1
  self.prTab(params.get("output")+" = ")
  print(self.name.substringBefore("$")+";\n")
  tabs=tabs-1
  self.prTab("}\n")
}

// List Store CB
mm.UserDefBlock::doListStore(params:Hashtable){
  self.prTab("//Insert intolist(queue)\n")
  self.prTab("atomic {\n")
  tabs=tabs+1
  self.prTab(self.name.substringBefore("[")+["_first_ "+ self.name.substringBefore("[")+ "++"]+" = ")
  print(params.get("input")+";\n")
  self.prTab("if (_first_ "+ self.name.substringBefore("[")+ " >
"+self.name.substringAfter("[").substringBefore("]")+" ) _first_ "+self.name.substringBefore("[")+ " = 0;\n")
  tabs=tabs-1
  self.prTab("}\n\n")
}

// List Retrieve CB
mm.UserDefBlock::doListRetrieve(params:Hashtable){
  self.prTab("//Extract from list(queue)\n")
  self.prTab("atomic {\n")
  tabs=tabs+1
  self.prTab(params.get("output")+" = ")
  print(self.name.substringBefore("[")+["_last_ "+ self.name.substringBefore("[")+ "++];\n")
  self.prTab("if (_last_ "+ self.name.substringBefore("[")+ " >
"+self.name.substringAfter("[").substringBefore("]")+" ) _last_ "+self.name.substringBefore("[")+ " = 0;\n")
  tabs=tabs-1
  self.prTab("}\n\n")
}

// List Retrieve CB
mm.UserDefBlock::doListMean(params:Hashtable){
  var prim:String
  var ult:String
  var ram:String

  prim="_first_ "+ self.name.substringBefore("[")
  ult="_last_ "+ self.name.substringBefore("[")
  ram=self.name.substringBefore("[")

  self.prTab("//Calculate mean value\n")
  self.prTab("_mean=0;\n")
  self.prTab("_count=0;\n")
  self.prTab("while (" +prim+" != "+ult+" ) {\n")
  tabs = tabs+1
  // self.prTab("atomic {\n")
  // tabs=tabs+1
  self.prTab("_mean+= "+ram+"["+prim+"++];\n")
  // tabs=tabs-1
  // self.prTab("}\n")
  self.prTab("_count++;;\n")
  self.prTab("if (" +prim+" >= "+self.name.substringAfter("[").substringBefore("]")+" ) "+prim+" = 0;\n")
  tabs=tabs-1
  self.prTab("}\n")
  self.prTab("if (_count > 0) "+params.get("output")+"=((float)_mean)/_count;\n\n")
}

//Hash reset
mm.UserDefBlock::doHashReset(params:Hashtable){
  var ult:String
```

```

var ram:String
var cont:String

ult="_hlast_" + self.name.substringBefore("[")
ram=self.name.substringBefore("[")
cont="_count_" + self.name.substringBefore("[")

self.prTab("//Reset "+ram+" hash table\n")
self.prTab("_aux=0;\n")
//while (_aux<ult) cont[_aux]=0;
self.prTab("while (_aux < "+ult+" ) "+cont+"[_aux++]=0;\n")
self.prTab(ult+" = 0;\n\n")
}

//Hash write mean store - CHANGE!
mm.UserDefBlock::doHashStore(params:Hashtable){
  var ult:String
  var ram:String
  var cont:String
  var index:String
  var tam:String
  var input:String

  ult="_hlast_" + self.name.substringBefore("[")
  ram=self.name.substringBefore("[")
  cont="_count_" + self.name.substringBefore("[")
  index=params.get("_index")
  tam=self.name.subStringAfter("[").subStringBefore("]")
  input=params.get("input")

  self.prTab("//Write in "+ram+" and calculate mean for key in "+ index+ "\n")
  self.prTab("_aux = 0;\n")
  self.prTab("while (_aux < "+ult+" )\n")
  self.prTab("\tif (_hkeys_ "+ram+"[_aux] == "+index+" ) break;\n")
  self.prTab("\telse _aux++;;\n")
  self.prTab("\tif (_aux == "+ult+" ) {\n")
  self.prTab("\t\t"+ult+"++;;\n")
  self.prTab("\t_hkeys_ "+ram+"[_aux] = "+index+";;\n")
  self.prTab("}")
  self.prTab("\tif (" +ult+" >= "+tam+" ) "+ult+" = "+tam+"-1;\n")
  self.prTab(cont+"[_aux]++;;\n")
  // ram[_aux]=(ram[_aux] * ((1.0*cont[_aux] - 1)/(cont[_aux])) + (1.0*input)/(cont[_aux])
  self.prTab(ram+"[_aux] = "+ram+"[_aux] * ((1.0*"+cont+"[_aux] - 1)/"+cont+"[_aux] +
(1.0*"+input+")/"+cont+"[_aux];\n\n")
}

//Hash hash maximum retrieve
mm.UserDefBlock::doHashMaxRetrieve(params:Hashtable){
  var ult:String
  var ram:String
  var tam:String
  var key:String
  var val:String

  ult="_hlast_" + self.name.substringBefore("[")
  ram=self.name.substringBefore("[")
  tam=self.name.subStringAfter("[").subStringBefore("]")
  key=params.get("_Key_output")
  val=params.get("output")

  self.prTab("//Select maximum from hash table\n")
  self.prTab("_aux = 0;\n")
  self.prTab("_selIndex = -1;\n")
  self.prTab("while(_aux < "+ult+" ) {\n")
  self.prTab("\tif (_selIndex == -1) _selIndex=0;\n")
  self.prTab("\tif (" +ram+"[_aux] > "+ram+"[_selIndex]) _selIndex = _aux;\n")
  self.prTab("\t_aux++;;\n")
  self.prTab("}")
  self.prTab("\tif (_selIndex != -1){\n")

```

```

        if (not key=null)
            self.prTab("\t"+key+" = _hkeys_"+ram+"[_selIndex];\n\n")
            //"+key+" = -1;\n")
        if (not val=null)
            self.prTab(val+" = "+ram+"[_selIndex];\n\n")
self.prTab("} else {\n")
    if (not key=null)
        self.prTab("\t"+key+" = -1;\n")
    if (not val=null)
        self.prTab(val+" = 0;\n\n")
self.prTab("}\n\n")

//_aux=0;
//_selIndex=-1;
//while(_aux < ult) {
//    if (ram[_aux]>ram[_selIndex]) _selIndex = aux;
//    _aux++;
//}
//if (_selIndex != -1)
//    key=_hkeys_ram[_selIndex];
//else key = -1;

}

// Calculate Expression CB
mm.UserDefBlock::doCalculateExp(params:Hashtable){

    self.prTab(self.data)
    if (not self.name.endsWith(",")) print(",")
    print("\n")
}

// GetId CB
mm.UserDefBlock::doGetId(params:Hashtable){
    self.prTab(params.get("NodeID")+ " = ")
    print("TOS_LOCAL_ADDRESS;\n")
}

// Conditional CB
mm.UserDefBlock::doConditional(params:Hashtable){
    var clList:List
    var clList2:List
    // condition
    self.prTab("if (" +self.data+" ) {\n")
    tabs = tabs + 1
    // True
    clList = self.isSource->select(cl:mm.ControlLink | cl.linkType.equals("ifTrue"))
    clList->forEach(cl:mm.ControlLink){
        cl.target.do()
        cl.target.isSource->forEach(cl2:mm.ControlLink){
            cl2.do()
        }
    }
    tabs = tabs - 1
    self.prTab("}\n")
    // false
    clList = self.isSource->select(cl:mm.ControlLink | cl.linkType.equals("ifFalse"))
    if (not clList.isEmpty()){
        self.prTab("else {\n")
        tabs = tabs + 1
        clList->forEach(cl:mm.ControlLink){
            cl.target.do()
            cl.target.isSource->forEach(cl2:mm.ControlLink){
                cl2.do()
            }
        }
        tabs = tabs - 1
        self.prTab("}\n")
    }
}

```

```

    }

}

// External call CB
mm.ExternalCall::do(){
    var numArgs:Integer
    var argSet:List
    var dlinks:List
    var aux:Integer=0
    var vproto:mm.Variable
    var v:mm.Variable

    // command call
    if (self.protoCalled.oclIsTypeOf(mm.CommandPrototype)){
        self.prTab("")

        // copy integer return value to variable
        if (self.protoCalled.returnType.equals("uint16_t")){
            self.blockParams->forEach(bp:mm.Variable){
                var dlinks:List= bp.isSource->select(dl:mm.DataLink | dl.linkType.equals("ProtoCall"))
                if (dlinks.size()==0){
                    print(tVarNames.get(bp.name)+"=")
                }
            }
        }

        // call
        print("call " + self.interfaceCalled.getActualRename() + "." + app.getActualName(self.protoCalled.name) +
"(")

        numArgs=self.protoCalled.arguments.size()
        // copy variables assigned to each argument, ordered
        numArgs->forEach(i){
            aux=i
            argSet=self.protoCalled.arguments->select(arg:mm.Variable | arg.order = aux)
            vproto=argSet.first()
            self.blockParams->forEach(bp:mm.Variable){
                dlinks=bp.isSource->select(dl:mm.DataLink | dl.target=vproto)
                if (not dlinks.isEmpty()){
                    print(tVarNames.get(dlinks.first().source.name))
                    if (aux<numArgs) print(", ")
                }
            }
        }
        println(");")

        // copy variables for use in CBs not belonging to the current one
        var dllist:List
        self.blockParams->forEach(bp:mm.Variable){
            dllist=bp.isSource->select(dl:mm.DataLink | dl.linkType.equals("Copy"))
            dllist->forEach(dl:mm.DataLink){
                self.prTab("atomic {\n")
            }
        }
        self.prTab("\t"+tVarNames.get(dl.target.name)+"="+tVarNames.get(bp.name)+";\n")
        self.prTab("}\n")
    }
}

else {
    stdout.println("In "+self.name+"::do(). Event call not considered")
}

}

// FillMsg CB
mm.UserDefBlock::doFillMsg(params:Hashtable){
    var pnum:String
    var msg:String

```

```

        var input:String

        pnum=params.get("paramNum")
        msg=params.get("msg")
        input=params.get("input")
        self.prTab("atomic {\n")
        self.prTab("\t((uint16_t *)"+msg+"->data)["+pnum+"-1]="+input+";\t\t\t\t//Fill message param
"+pnum+"\n")
        self.prTab("}\n")
    }

    // ExtractMsg CB
    mm.UserDefBlock::doExtractMsg(params:Hashtable){
        var pnum:String
        var msg:String
        var extractData:String

        pnum=params.get("paramNum")
        msg=params.get("msg")
        extractData=params.get("extractData")
        //self.prTab(extractData+"=((uint16_t *)"+msg+")["+pnum+"-1];\t\t\t\t//Extract message param
"+pnum+"\n")
        self.prTab(extractData+"=((uint16_t *)"+msg+"->data)["+pnum+"-1];\t\t\t\t//Extract message param
"+pnum+"\n")
        ReturnVal=msg
    }

}

// InitMsg CB
mm.UserDefBlock::doInitMsg(params:Hashtable){
    var msg:String

    msg=params.get("msg")
    self.prTab(msg+"=&g_message;\n")
    numParamsMsg=0
}

// Relate CB types with the global variables they require
mm.Application::initGlobalVarsList(){
    globalVarCB.put("InitMsg","TOS_Msg g_message")
    globalVarCB.get("InitMsg")
}

// Iterate through CB to find neccessary global variables
mm.SimpleComponent::declareGlobalVars(global:Hashtable, tVar:Hashtable){
    var cList:List
    self.objectsOfType(mm.ComplexBlock)->forEach(compBlock:mm.ComplexBlock){
        compBlock.objectsOfType(mm.BasicBlock)->forEach(basicB:mm.BasicBlock){
            if (basicB.code.startsWith("Store") || basicB.code.startsWith("Retrieve")){
                global.put(basicB.name.substringBefore("$"), "int " +
basicB.name.substringBefore("$"))
            }
            if (basicB.code.startsWith("List")){
                global.put(basicB.name.substringBefore("$"), "int " +
basicB.name.substringBefore("$"))
                global.put("_first_"+basicB.name.substringBefore("$"), "int " + "_first_" +
basicB.name.substringBefore("[")+ " = 0")
                global.put("_last_"+basicB.name.substringBefore("$"), "int " + "_last_" +
basicB.name.substringBefore("[")+ " = 0")
            }
            if (basicB.code.startsWith("Hash")){
                global.put(basicB.name.substringBefore("$"), "int " +
basicB.name.substringBefore("$"))
                global.put("_hkeys_"+basicB.name.substringBefore("$"), "int _hkeys_" +
basicB.name.substringBefore("$"))
                global.put("_hlast_"+basicB.name.substringBefore("$"), "int " + "_hlast_" +
basicB.name.substringBefore("[")+ " = 0")
            }
        }
    }
}

```

```

        global.put("_count_"+basicB.name.substringBefore("$"), "int _count_" +
basicB.name.substringBefore("$"))
    }
    if ((basicB.code!=null) && globalVarCB.get(basicB.code)!=null)
        global.put(basicB.code, globalVarCB.get(basicB.code))
    basicB.objectsOfType(mm.Variable)->forEach(v:mm.Variable){
        clList = v.isTarget->select(cl:mm.DataLink | cl.linkType.equals("Copy"))
        if (not clList.isEmpty()){
            global.put(v.name, typeEqs.get(v.type) + " " + tVar.get(v.name))
        }
    } // foreach variable
} // foreach basicblock
} // foreach complexcomp

}

// Fill tables for types and names of variables
mm.SimpleComponent::createVarToTypeTable(tTypes:Hashtable, tNames:Hashtable){
    self.objectsOfType(mm.ComplexBlock)->forEach(compBlock:mm.ComplexBlock){
        compBlock.hashVariables(tTypes, tNames)
        compBlock.objectsOfType(mm.BasicBlock)->forEach(basicB:mm.BasicBlock){
            basicB.hashVariables(tTypes, tNames)
        }
    }
} //falta variables de comandos y eventos?
}

// Prepare variable name for tables
mm.CodeBlock::hashVariables(ht:Hashtable, hn:Hashtable){
    var str:String

    // types are the part before $
    self.objectsOfType(mm.Variable)->forEach(v:mm.Variable){
        if (v.name.indexOf("$")>=0)
            ht.put(v.name, v.name.substringBefore("$"))
        else
            ht.put(v.name, v.name)

        // names are the type + its cb number
        str=v.getVarName()
        if (str.indexOf("$")>=0)
            str = str.substringBefore("$") + str.substringAfter("$")
        hn.put(v.name, str)
    }
}

// navigates datalinks to find origin variable
mm.Variable::getVarName():String{
    var name:String
    var pointer:mm.Variable
    var clList:List
    var clList2:List

    pointer=self

    while(pointer!=null)
    {
        if (pointer.constantValue=true)
            name=pointer.name.substringBefore("$") // constant -> do not navigate, not connected
        else name=pointer.name
        clList=pointer.isTarget->select(cl | cl.linkType.equals("SameVar"))
        pointer=clList.first().source
    }
    result = name
}

```

```

// Create a configuration file
/*****
mm.Configuration::create(path:String){
    var cont:integer=0
    var listaUniqueComp: list
    var duplicate:Boolean=false
    var listaComp: list
    var nonSystemC:List

    var d:mm.Configuration
    d=self
    file sal(path+d.name+".nc")

    // print constants (alfanumerical)
    d.wirings->forEach(w:mm.Wiring){
        if ((w.paramSource!=null) && (w.paramSource.startsWith("AM"))){
            if (enums.get(w.paramSource)=null){
                AMid=AMid+1
                enums.put(w.paramSource, AMid)
            }
            println("#define " + w.paramSource + " " + enums.get(w.paramSource))
        } // no probado

        if ((w.paramTarget!=null) && (w.paramTarget.startsWith("AM"))){
            if (enums.get(w.paramTarget)=null){
                AMid=AMid+1
                enums.put(w.paramTarget, AMid)
            }
            println("#define " + w.paramTarget + " " + enums.get(w.paramTarget))
        }
    }

    println("")

    sal.println("configuration "+d.name+"{")
    d.UsesAndProvides()
    sal.print("}")
    n(2)
    cont=0

    sal.println("implementation{n{")
    //component and alias enumeration
    sal.print("\tcomponents ")
    listaComp.clear()
    listaUniqueComp.clear()
    d.wirings->forEach(w:mm.Wiring){
        listaComp.add(w.source.owner)
        listaComp.add(w.target.owner)
    } // end wiring
    var i:Integer
    i=0
    d.renamedComponent->forEach(a:mm.Alias){
        if (not a.baseComponent.name.equals(a.renamedAs)){
            if (i>0) sal.print(", ")
            sal.print(a.baseComponent.name+" as "+ a.renamedAs)
        }
        if (a.baseComponent.name.equals(a.renamedAs))
            sal.print(a.baseComponent.name)
        i=i+1
    }
    sal.println("; \n")

    d.CreateWirings()

} // end module Configuration

```



```

mm.Configuration::CreateWirings()
{
    var nombreInterface: String
    var nombreComponente: String
    // var auxString: String
    self.wirings->forEach(w:mm.Wiring){

        // para el source
        nombreComponente=w.source.owner.name // de momento le doy el nombre original
        // ahora veo si el componente está renombrado con un Alias
        self.renamedComponent->forEach(a:mm.Alias)
            if(w.source.owner.name.equals(a.baseComponent.name)) nombreComponente=
a.renamedAs //componente renombrado con Alias
        // ahora veamos si la interfaz está renombrada
        // pero antes hagamos que el nombre del componente quede vacío si es el propio configuration
        nombreComponente=nombreComponente+" "
        if(w.source.owner.name.equals(self.name)) nombreComponente=""
        if(!w.source.interface.name.equals(w.source.getActualRename()))
            print("\t"+nombreComponente+w.source.getActualRename()) //interfaz renombrada
        if(w.source.interface.name.equals(w.source.getActualRename()))
            print("\t"+nombreComponente+w.source.interface.name) //interfaz no renombrada
        //pongamos la parametrización del interfaz si existe
        // en wiring paramSource y paramTargert
        if(w.paramSource!=null) print("[ "+w.paramSource+"]")

        if (w.wiringKind.equals("DELEGATION"))
            print("\t =\t ")
        else print("\t ->\t ")

        // para el target
        nombreComponente=w.target.owner.name // de momento le doy el nombre original
        // ahora veo si el componente está renombrado con un Alias
        self.renamedComponent->forEach(a:mm.Alias)
            if(w.target.owner.name.equals(a.baseComponent.name))
nombreComponente=a.renamedAs //componente renombrado con Alias
        // veamos si la interfaz está renombrada
        // pero antes hagamos que el nombre del componente quede vacío si es el propio configuration
        //if(w.target.owner.name.equals(self.name)) nombreComponente=""

        if(!w.target.interface.name.equals(w.target.getActualRename()))
            print(nombreComponente+" "+w.target.getActualRename()) //interfaz renombrada
        if(w.target.interface.name.equals(w.target.getActualRename()))
            print(nombreComponente+" "+w.target.interface.name) //interfaz no renombrada
        if(w.paramTargert!=null) print("[ "+w.paramTargert+"]")
        print(",")
        //....
        n(1)
        //....
    }
    println("\n}")
} // end module configurations
/*****

mm.Component::UsesAndProvides() {
    if(self.uses.size()==1) print("\tuses")
    if (self.uses.size()>1) {n(1) println("\tuses {}")}
    self.uses->forEach(b:mm.InterfaceInstance) {
        if (!b.getActualRename().equals(b.interface.name)) print("\t\tinterface "+b.interface.name+" as
"+b.getActualRename())
        if (b.getActualRename().equals(b.interface.name)) print("\t\tinterface "+b.interface.name)
        if (b.parameters.size()>0)
            b.parameters->forEach(t:mm.Variable){print("[ "+t.type+" "+t.name+"]")}
    }
    println(",")
}

```

```
    } // end uses
    if (self.uses.size()>1) println("\t}")

    // provides
    if(self.provides.size()==1) print("\tprovides")
    if (self.provides.size()>1) {n(1) println("\tprovides {}")}
    self.provides->forEach(b:mm.InterfaceInstance){
        if (!b.getActualRename().equals(b.interface.name)) print("\t\tinterface " + b.interface.name + " as
"+b.getActualRename())
        if (b.getActualRename().equals(b.interface.name)) print("\t\tinterface
"+b.interface.name)
        if (b.parameters.size()>0)
            b.parameters->forEach(t:mm.Variable){print("[ "+t.type+"
"+t.name+"]")}
        println(";")
    } //end provides
    if (self.provides.size()>1) println("\t}")
}

// create makefiles and its components
mm.Application::createNodeImages(){

    self.configurations->forEach( conf:mm.Configuration){
        uniqueComps.clear()
        file sal(outputDir+ conf.name+"\\Makefile")
        sal.println("COMPONENT="+conf.name+"\nPFLAGS +=-I%T/lib/Eclipse\\ninclude
../../Makerules")
        conf.create(outputDir+ conf.name+"\\")
        conf.createInternalComps(outputDir+ conf.name+"\\")
    }
}

mm.Configuration::createInternalComps(path:String){
    self.renamedComponent->forEach(a:mm.Alias){
        if (a.baseComponent.systemComp=false){
            if (uniqueComps.get(self.name+a.baseComponent.name)=null){
                uniqueComps.put(self.name+a.baseComponent.name,"Done")
                a.baseComponent.create(path)
                if (a.baseComponent.oclIsTypeOf(mm.Configuration))
a.baseComponent.createInternalComps(path)
            }
        } //end if systemcomp
    } // end foreach
}

} // file
```