ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Aplicación del Modelado Específico de Dominio a las Redes de Sensores Inalámbricos

AUTOR: José Antonio Moñino Martínez
DIRECTOR: Fernando Losilla López

Diciembre / 2007

Autor	José Antonio Moñino Martínez
E-mail del Autor	josemonino@gmail.com
Director(es)	Fernando Losilla López
E-mail del Director	fernando.losilla@upct.es
Título del PFC	Aplicación de modelado específico de dominio a las redes de sensores inalámbricos
Descriptores	DSM, MetaEdit+, TinyOS, WSN

Resúmen

En este proyecto final de carrera se detalla la obtención de una herramienta para el desarrollo de aplicaciones de forma automática que facilite la tarea de construcción de software para las redes de sensores inalámbricas.

Se consigue elevar el nivel de abstracción del lenguaje ahorrando tiempo al programador en el desarrollo de las aplicaciones. Así se evita la programación completa de aplicaciones mediante lenguajes tradicionales, y en su lugar se usan modelos que describen la aplicación en función de conceptos y terminología propios de las redes de sensores.

La herramienta usada es Metaedit+ en su versión 4.5 y la plataforma destino es TinyOS - Telos rev. B.

Titulación	Ingeniero de Telecomunicación
Intensificación	Sistemas y Redes de Telecomunicación
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Diciembre- 2007

A mi director y amigo Fernando Losilla por su paciencia y dedicación.

A mis familiares que tanto les debo.

A Jose Ramón por su valiosa aportación.

A todos mis amigos y compañeros de carrera.

Para ti Esther .

Índice de contenidos

INTRODUCCIÓN **13**

I. PRESENTACIÓN 13
II. PRINCIPIOS BÁSICOS DEL DSDM 15
III. ARQUITECTURA DIRIGIDA POR MODELOS (MDA) 18
IV. MODELADO ESPECÍFICO DEL DOMINIO 21
i. Decisión *23*
ii. Análisis *24*
iii. Diseño *24*
iv. Implementación *24*
v. Despliegue *25*
vi. Ventajas e inconvenientes *26*
V. REDES DE SENSORES INALÁMBRICAS 27

REDES DE SENSORES INALÁMBRICAS **29**

I. INTRODUCCIÓN 29
II. HISTORIA 31
III. CARACTERÍSTICAS DE LAS WSN 32
i. Arquitecturas de las WSN *35*
ii. Protocolos de las WSN *36*
iii. Zigbee, estándar WSN *39*
iv. Problemas de las WSN *42*
IV. APLICACIONES DE LAS WSN 44
i. Aplicaciones militares *45*
ii. Aplicaciones medioambientales *46*
iii. Aplicaciones sanitarias *47*
iv. Aplicaciones del hogar: domótica *48*
v. Otras aplicaciones comerciales *49*
V. NODOS SENSORES 51
VI. EJEMPLOS DE MOTES: MICAS Y TELOS 57
i. Micas *57*
ii. Telos *62*
iii. Resumen comparativo *64*
VII. TINYOS 66
i. NesC *67*
ii. Herramientas de TinyOS *70*
VIII. FUTURO DE LAS WSN 73

METAEDIT+ **77**

I. INTRODUCCIÓN A LA HERRAMIENTA 77
II. HERRAMIENTAS DE METAMODELADO 80
i. Herramienta de propiedades (Property Tool) *80*
ii. Herramienta de objetos (Object Tool) *83*
iii. Herramienta de relaciones (Relationship Tool) *84*
iv. Herramienta de roles (Role Tool) *85*
v. Herramienta de puertos (Port Tool) *86*
vi. Herramienta de grafos (Graph Tool) *87*
III. EDITORES DE ASPECTO VISUAL 93
i. Editor de diálogos (Dialog Editor) *93*
ii. Editor de símbolos (Symbol Editor) *94*
IV. EDITORES DE LOS MODELOS 98
i. Editor de diagramas (Diagram Editor) *98*

ii. Editor de matrices (Matrix Editor) *99*
iii. Editor de tablas (Table Editor) *101*
V. INFORMES Y GENERACIÓN DE CÓDIGO 102
i. Informes independientes del DSL (predefinidos) *102*
ii. Informes dependientes del lenguaje: el navegador de informes *103*
iii. El lenguaje para escribir informes *105*
VI. EL REPOSITORIO 110
VII. POSIBILIDADES AVANZADAS DE METAEDIT+ 112
i. Importación y exportación en XML *112*
ii. API de acceso *113*

MANUAL DE USUARIO 115

I. INTRODUCCIÓN 115
II. GRÁFICO 116
III. OBJETOS 119
i. Start *119*
ii. Timer *119*
iii. Sensor *120*
iv. Rfm *121*
v. Button *122*
vi. PC *123*
vii. Leds *124*
IV. RELACIONES 126

IMPLEMENTACIÓN DEL DSM 129

I. INTRODUCCIÓN 129
II. METAMODELO 130
III. GENERACIÓN DE CÓDIGO 135

EJEMPLO DE APLICACIÓN PRÁCTICA 141

I. ESCENARIO 141
II. SOLUCIÓN 142
i. Modelo 1 ..... *142*
ii. Modelo 2 ..... *143*
iii. Modelo 3 ..... *144*
III. PUESTA EN MARCHA 146

CONCLUSIONES Y LÍNEAS DE FUTURO 147

I. CONCLUSIONES 147
II. LÍNEAS DE FUTURO 148

BIBLIOGRAFÍA Y REFERENCIAS 149

Índice de figuras

FIGURA 1: UN NUEVO NIVEL DE ABSTRACCIÓN EN LOS LENGUAJES 15
FIGURA 2: UN METAMODELO SIMPLE "ESQUEMA RELACIONAL" 16
FIGURA 3: ELEMENTOS BÁSICOS DE MDA 18
FIGURA 4: EJEMPLO DSM 21
FIGURA 5: RED DE SENSORES INALÁMBRICOS 29
FIGURA 6: TAMAÑO DE LOS NODOS SMART DUST 31
FIGURA 7: ELEMENTOS DE UNA WSN 32
FIGURA 8: ARQUITECTURA WSN CENTRALIZADA 35
FIGURA 9: ARQUITECTURA WSN DISTRIBUIDA 36
FIGURA 10: NIVELES FÍSICO, RED Y APLICACIÓN 39
FIGURA 11: COMPARATIVA ESTÁNDARES INALÁMBRICOS 40
FIGURA 12: ESPECTRO DE ESTÁNDARES 41
FIGURA 13: APLICACIONES ZIGBEE 42
FIGURA 14: APLICACIONES DE LAS WSN 44
FIGURA 15: TRACKING DE ANIMALES 45
FIGURA 16: RECONOCIMIENTO DEL ENEMIGO 46
FIGURA 17: DESPLIEGUE DE SENSORES PARA RECONOCIMIENTOS 46
FIGURA 18: DETECCIÓN DE INCENDIOS 47
FIGURA 19: APLICACIÓN DOMÓTICA 48
FIGURA 20: NODOS SENSORES 51
FIGURA 21: COMPARACIÓN PLATAFORMAS PARA NODOS 53
FIGURA 22: ESTADOS DE UN NODO SENSOR 55
FIGURA 23: DISTRIBUCIÓN DEL CONSUMO DE ENERGÍA 56
FIGURA 24: ESTRUCTURA DE RED Y MOTES 57
FIGURA 25: MICAZ 58
FIGURA 26: DIAGRAMA DE BLOQUES MICAZ 58
FIGURA 27: CARACTERÍSTICAS TÉCNICAS DE LOS MICA2 59
FIGURA 28: MICA2 60
FIGURA 29: DIAGRAMA DE BLOQUES MICA2 60
FIGURA 30: MICA2DOT 61
FIGURA 31: DIAGRAMA DE BLOQUES MICA2DOT 61
FIGURA 32: SENSORES MTS300 62
FIGURA 33: TIPOS DE SENSORES PARA MICAS 62
FIGURA 34: TELOSB 63
FIGURA 35: DIAGRAMA DE BLOQUES DEL TELOSB 63
FIGURA 36: EVOLUCIÓN DE LOS MOTES 64
FIGURA 37: COMPARATIVA DE TIEMPOS Y CONSUMOS 65
FIGURA 38: ESTRUCTURA DE UN COMPONENTE 68
FIGURA 39: FICHEROS DE UNA APLICACIÓN 69
FIGURA 40: TINYVIZ 70
FIGURA 41: SURGE VIEW 71
FIGURA 42: TINYDB 72
FIGURA 43: TWISTER 74
FIGURA 44: TÚNEL WSN 75
FIGURA 45: METAMETAMODELO DEL LENGUAJE GOPPRR 78
FIGURA 46: HERRAMIENTA DE PROPIEDADES 80
FIGURA 47: HERRAMIENTA DE OBJETOS 83
FIGURA 48: HERRAMIENTA DE RELACIONES 85
FIGURA 49: HERRAMIENTA DE ROLES 86
FIGURA 50: HERRAMIENTA DE PUERTOS 87
FIGURA 51: HERRAMIENTA DE GRAFOS 88
FIGURA 52: DEFINIDOR DE TIPOS DEL GRAFO 89
FIGURA 53: DEFINIDOR DE LOS ENLACES DEL GRAFO 89

FIGURA 54: DEFINIDOR DE RESTRICCIONES 90
FIGURA 55: JERARQUÍA DE LENGUAJES EN UML 92
FIGURA 56: EDITOR DE DIÁLOGOS 93
FIGURA 57: EDITOR DE SÍMBOLOS APLICADO AL OBJETO TIMER 95
FIGURA 58: BARRA DE EDICIÓN DE SÍMBOLOS 95
FIGURA 59: EDITOR DE SÍMBOLOS PARA EL ROL "PROVIDES" 97
FIGURA 60: EDITOR DE DIAGRAMA PARA UN CASO DE MODELO DE MOTE 99
FIGURA 61: BARRA DE HERRAMIENTAS DEL EDITOR DE DIAGRAMAS 99
FIGURA 62: VISTA DEL EDITOR DE MATRICES PARA UN CASO DE VENTAS E INVENTARIO 100
FIGURA 63: BARRA DE HERRAMIENTAS DEL EDITOR DE MATRICES 101
FIGURA 64: EDITOR DE TABLAS REFLEJANDO LAS METAS DE UN SISTEMA DE INVENTARIO 101
FIGURA 65: NAVEGADOR DE INFORMES 104
FIGURA 66: TABLA DE PALABRAS RESERVADAS 106
FIGURA 67: EJEMPLO DE CICLO EN UNA MÁQUINA DE ESTADOS 108
FIGURA 68: COMMIT / ABANDON 111
FIGURA 69: EJEMPLO DE LA ESTRUCTURA DEL XML 112
FIGURA 70: HERRAMIENTA DE LA API 114
FIGURA 71: VISTA PRINCIPAL METAEDIT+ 116
FIGURA 72: OPCIÓN CREATE GRAPH 116
FIGURA 73: DIÁLOGO CREATE GRAPH 117
FIGURA 74: DIÁLOGO DE PROPIEDADES DE GRÁFICO 117
FIGURA 75: VENTANA DE MODELADO 118
FIGURA 76: OBJETO START 119
FIGURA 77: DIÁLOGO TIMER 119
FIGURA 78: TIMER ONE SHOT 120
FIGURA 79: TIMER REPEAT 120
FIGURA 80: DIÁLOGO SENSOR 120
FIGURA 81: LUMINOSITY TSR 121
FIGURA 82: LUMINOSITY TSR THRESOLD 121
FIGURA 83: LUMINOSITY PAR 121
FIGURA 84: LUMINOSITY PAR THRESOLD 121
FIGURA 85: TEMPERATURE 121
FIGURA 86: TEMPERATURE THRESOLD 121
FIGURA 87: HUMIDITY 121
FIGURA 88: HUMIDITY THRESOLD 121
FIGURA 89: INTERNAL TEMP. 121
FIGURA 90: INTERNAL TEMP. THRESOLD 121
FIGURA 91: DIÁLOGO RFM 122
FIGURA 92: OBJETO RFM 122
FIGURA 93: DIÁLOGO BUTTON 122
FIGURA 94: OBJETO BUTTON 123
FIGURA 95: DIÁLOGO PC 123
FIGURA 96: OBJETO PC 124
FIGURA 97: DIÁLOGO LEDS 124
FIGURA 98: LEDS YELLOW ON 125
FIGURA 99: LEDS RED ON 125
FIGURA 100: LEDS VALUE 125
FIGURA 101: LEDS GREEN TOGGLE 125
FIGURA 102: TABLA DE RELACIONES 127
FIGURA 103: DEPENDENCIAS DE SÍMBOLOS 130
FIGURA 104: CREACIÓN DE INTERFACE 131
FIGURA 105: SÍMBOLO PROVIDES 132
FIGURA 106: CREACIÓN DE MOTE 133
FIGURA 107: GRAPH TYPES 133
FIGURA 108: GRAPH BINDINGS ROLE USES 134
FIGURA 109: GRAPH BINDINGS ROLE PROVIDES 134
FIGURA 110: RFM EN AMBOS ROLES 135
FIGURA 111: ESCENARIO EJEMPLO 141
FIGURA 112: MODELO 1 142
FIGURA 113: MODELO 2 144

FIGURA 114: MODELO 3 145
FIGURA 115: CENTRO DE CONTROL 146

Capítulo 1
Introducción

I. Presentación

Las redes de sensores inalámbricas constituyen una nueva disciplina de estudio de gran interés en los últimos años. Esta nueva tecnología permite un mejor conocimiento acerca de los fenómenos que se estudian, ya que permite el despliegue de un elevado número de nodos, no sujetos a las restricciones impuestas por el cableado. Las nuevas necesidades específicas de estas redes inalámbricas han hecho que las plataformas usadas para el desarrollo de software en otros dispositivos inalámbricos no sean adecuadas para estas redes. Nuevos sistemas operativos, cada uno con su propio conjunto de herramientas, han aparecido para llenar este hueco. De entre todos estos destaca TinyOS por la gran aceptación que ha tenido entre la comunidad investigadora. Sin embargo, a pesar de la aportación de los sistemas operativos, las aplicaciones finales para redes de sensores deben enfrentarse al cumplimiento de una serie de requisitos, muchas veces enfrentados entre sí y no tienen en cuenta la experiencia previa ni los artefactos generados en el diseño de aplicaciones de características similares.

Este proyecto fin de carrera pretende desarrollar un metamodelo para la construcción de software sobre redes de sensores de forma automática. El desarrollador únicamente necesita expresar la aplicación deseada mediante un diagrama con elementos del entorno de las redes de sensores (temporizadores, sensores, antenas de transmisión o recepción...) y el metamodelo aquí desarrollado generará el código equivalente automáticamente sin necesidad de más parámetros que los que se pidan durante la elavoración del diagrama. En otras palabras lo que se pretende es elevar el nivel de abstracción del lenguaje mediante el paradigma del DSM (Modelado específico de dominio).

Con el uso del DSM aplicado a las mencionadas redes de sensores inalámbricas se obtiene un aumento significativo de la productividad en la tarea de programar ya que aportan beneficios como:

- **Automatización**. Si bien la aparición de sistemas operativos con lenguajes de programación de alto nivel para las WSN ya había supuesto un aumento de la productividad en este campo, la aplicación del DSM a estos lenguajes puede permitir un nuevo aumento de productividad ya que un elemento en el modelo supondrá varias instrucciones en el lenguaje.
- **Correspondencia Directa**. Existe una correspondencia directa entre el modelo conceptual de la solución y el modelo DSM. Se facilita en gran medida la expresión de la solución al problema reduciendo el salto semántico entre el modelo conceptual que representa la solución para cierta tarea y el código o diagrama que la expresa en un formato ejecutable.

- **Mantenimiento minimizado.** Debido a la correspondencia directa resulta más fácil modificar modelo cuando se producen cambios en los requisitos funcionales.
- **Reducción del tiempo de prueba**. Del modelo se generará el código que ejecute la solución tal y como se ha expresado. Deja de ser necesario por tanto el control de excepciones y de errores en la escritura del código permitiendo al programador concentrarse únicamente en la solución a su problema.

Para la tarea se usa la herramienta privada y ya consolidada como una de las que mejores resultados aportan a este tipo de proyectos: Metaedit+. Concretamente se ha usado la versión 4.5, la más reciente hasta la fecha. Como plataforma destino para el código generado se ha usado el sistema operativo más extendido entre las redes de sensores: TinyOS. Además, el código generado atiende a los componentes de los nodos sensores TelosB, que por sus características son los que mejor comportamiento tienen para el ahorro de baterías, algo muy importante para este tipo de tecnología.

A continuación se explica la estructura de la memoria para dar fluidez a su lectura y aportar una mejor compresión de la misma:

- En el presente capítulo se introduce a los diferentes paradigmas de la ingeniería software entre los que destaca en este proyecto el DSM como se ha comentado. También se introduce brevemente al lector en las redes de sensores inalámbricas, de evidente importancia en este proyecto.
- En el capítulo 2 se explican de manera más detallada las redes de sensores inalámbricas y el sistema operativo sobre el que se basa el proyecto (TinyOS).
- En el capítulo 3 se explica a modo de manual de usuario la herramienta usada para crear el metamodelo (Metaedit+) dejando claras las posibilidades que ofrece.
- El capítulo 4 pretende enfocar el metamodelo creado como herramienta para el desarrollo de software sin adentrar en términos de desarrollo del mismo, únicamente expresa la forma de trabajar con él para crear aplicaciones de redes de sensores.
- En el capítulo 5 se explica el camino seguido para la implementación del metamodelo.
- En el capítulo 6 se puede encontrar un ejemplo de aplicación práctica en el que se desarrolla da solución a un problema en un escenario ficticio. Se explican todos los pasos a seguir desde el punto de vista del programador adentrando únicamente en términos prácticos.
- En el capítulo 7 se encuentran las conclusiones a las que se ha llegado con el presente proyecto y las posibles ampliaciones que podrían hacerse para completar esta herramienta.

II. Principios básicos del DSDM

La principal idea compartida por todos los paradigmas englobados dentro del Desarrollo de Software Dirigido por Modelos (DSDM) es la conveniencia de que los programadores empleen lenguajes de más alto nivel de abstracción que los lenguajes de programación, esto es, lenguajes que manejen conceptos más cercanos al dominio de la aplicación. Estos lenguajes que proporcionan mayor nivel de abstracción se denominan lenguajes de modelado o lenguajes específicos del dominio (DSL) que es donde se centra precisamente el propósito del proyecto.

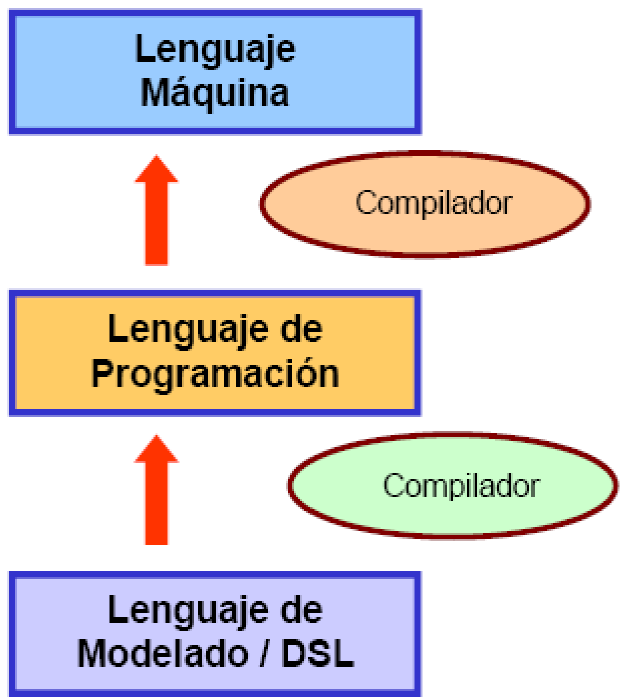


Figura 1: Un nuevo nivel de abstracción en los lenguajes

Como muestra la Figura 1, el uso de un DSL requiere, lógicamente, la existencia de un compilador que transforme el código expresado con un DSL en código en un lenguaje de programación (o en otro DSL, como paso previo a la generación de código). En los últimos años ha surgido un área muy activa relacionada con la creación de herramientas que permitan definir DSL y que incorporen compiladores DSL-DSL (denominados herramientas de transformación modelo-modelo en el ámbito de MDA) o DSL-código (transformación modelo-código).

La creación de un DSL no es sencilla aunque se dispongan de las herramientas apropiadas. La teoría subyacente a la creación de lenguajes de modelado o DSL se denomina metamodelado y los diferentes aspectos que abarca son: lenguajes de metamodelado, sintaxis abstracta, sintaxis concreta, semántica y transformaciones.

Un lenguaje de metamodelo es un lenguaje para describir otros lenguajes. Los lenguajes de metamodelo más extendidos son MOF de OMG y Ecore para Eclipse; otro ejemplo representativo es XMF de Xactium. Una definición de un lenguaje a través de un lenguaje de metamodelo se denomina metamodelo.

Un metamodelo define la sintaxis abstracta de un lenguaje que establece los conceptos y relaciones entre ellos, e incluye las reglas que determinan qué es un modelo bien formado. La Figura 2 muestra un ejemplo de metamodelo que define un esquema relacional, como una agregación de tablas.

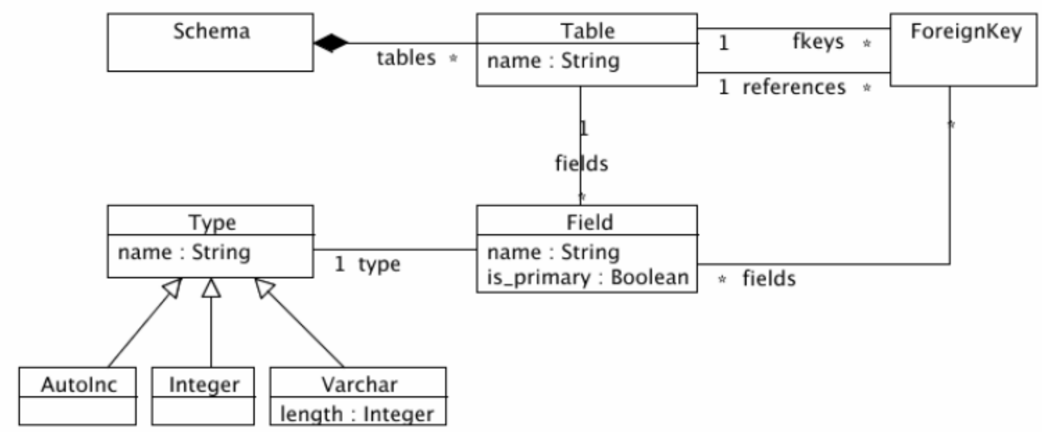


Figura 2: Un metamodelo simple "esquema relacional"

El metamodelo deber ir acompañado de una definición de la sintaxis concreta o notación para expresar los modelos que conforman a la sintaxis abstracta. Supuesta una notación textual. La semántica se refiere al significado de los conceptos y de las relaciones en el lenguaje, lo cual es necesario para comprender el lenguaje. Por ejemplo, para el metamodelo anterior formaría parte de la semántica el hecho de que en una tabla corresponde al concepto matemático de relación. Hay cuatro diferentes enfoques para describir la semántica de un metamodelo:

- **Traducción**: Se traducen los conceptos en conceptos de otro lenguaje que tenga una semántica precisa.
- **Operacional**: Modelado del comportamiento operacional de los conceptos.
- **Extensional**: Se extiende la semántica de los conceptos de otro lenguaje.
- **Denotacional**: Asocia objetos matemáticos a cada objeto del lenguaje.

Dado un modelo o una especificación en cierto DSL es necesaria una transformación a código que puede ser directa o indirecta a través de uno o más pasos en los que hay una generación de otro modelo o especificación expresada en otro lenguaje de modelado o DSL intermedio, así se habla de transformaciones modelo-modelo M2M o modelo-código M2T. Estas últimas se resuelven a través de lenguajes de plantillas tales como *Velocity* o *FreeMaker*. A las transformaciones modelo-modelo se han dedicado grandes esfuerzos de investigación y desarrollo en los últimos cuatro años y han surgido diferentes lenguajes de transformación como ATL, Tefkat y MTF. Los distintos lenguajes propuestos se pueden clasificar en tres enfoques: declarativos, imperativos e híbridos. Existe consenso al considerar más apropiados los lenguajes híbridos ya que el estilo declarativo es muy restrictivo para expresar transformaciones complejas.

El interés actual por los lenguajes de transformación y las herramientas de transformación (o compiladores de modelos) recuerda a la situación vivida a principios de los sesenta con la aparición de los primeros lenguajes de programación y toda la teoría relacionada con los compiladores e intérpretes. Como dijimos anteriormente, son muchos los que consideran que el DSDM plantea un salto en el nivel de abstracción del mismo orden de magnitud que con los lenguajes de programación.

Como es lógico, actualmente se están dedicando importantes esfuerzos al desarrollo de herramientas que permitan la definición de lenguajes de modelado o DSL, estas herramientas incorporan mecanismos para: i) crear metamodelos, ii) asociar una notación cualquiera a los elementos de un metamodelo, iii) expresar transformaciones para la generación de modelos o código a partir de un modelo que conforme a cierto metamodelo, y iv) realizar consultas sobre los metamodelos creados. El componente de estas herramientas que permite crear un metamodelo y asociarle una sintaxis concreta se denomina editor de metamodelos.

III. Arquitectura Dirigida por Modelos (MDA)

La principal idea sobre la que se articula MDA (Model Driven Architecture) es la separación de la especificación de la funcionalidad de una aplicación de su implementación sobre una plataforma concreta, con el objetivo de que el desarrollador sólo se preocupe de la lógica del negocio y herramientas específicas generen todo el código relacionado con las plataformas de implementación.

En este sentido, MDA propone que el desarrollador cree una descripción de la lógica del problema desde una perspectiva independiente de la computación (CIM, Computation Independent Model) que es donde se mejor se podrían encajar las reglas de modelado o metamodelo del presente proyecto. Desde aquí se crean los modelos independientes de la plataforma (PIM, Platform Independent Model) que en este proyecto se obvian ya que sigue el paradigma del DSM en lugar de MDA y por tanto del modelo anterior se pasa directamente a código (DSL-Código). A partir de los modelos PIM se generan modelos específicos de una plataforma concreta (PSM, Platform Specific Model) y finalmente el código. Como vemos en la Figura 3, cada modelo puede ser expresado en un lenguaje diferente (representado como L1, L2 y L3 en la figura), y las trasformaciones CIM-PIM, PIM-PSM y PSM-código requieren de herramientas de transformación que reciben como entrada, además del modelo origen, una definición de transformación que establece el mapping entre el lenguaje del modelo origen y del lenguaje destino. Esta definición de transformación es expresada mediante un lenguaje de transformación.

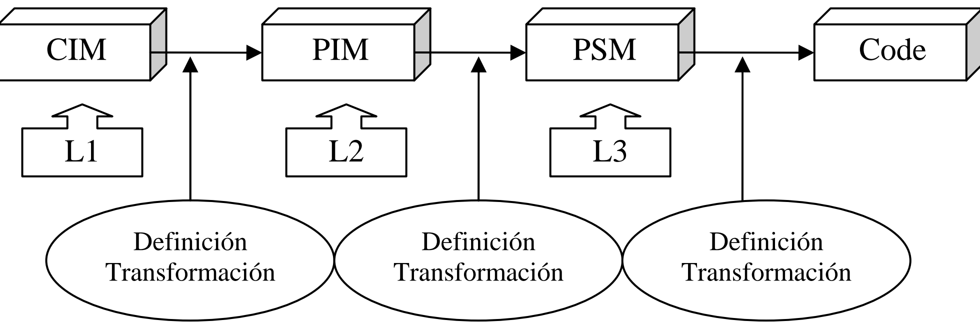


Figura 3: Elementos básicos de MDA

El desarrollador debe especificar la lógica de negocio mediante modelos CIM independientes de la computación, en su más alto nivel y lo más cercano al dominio del problema. Aquí es donde se realiza la primera transformación al modelo PIM que describe la funcionalidad del sistema de forma independiente a las plataformas de implementación específicas. Ambos modelos son expresados en algún lenguaje de modelado y a partir de esos modelos se podrá generar automáticamente modelos específicos de las plataformas y el código de la aplicación.

En un principio los perfiles UML fueron la alternativa más utilizada como lenguaje de modelado pero ahora han ganado en aceptación nuevos lenguajes definidos a partir de lenguajes de metamodelado como MOF.

Junto a la automatización y al empleo de lenguajes de modelado, el uso de estándares es el tercer pilar en el que se sustenta MDA. OMG ha definido un conjunto de estándares para soportar su propuesta como son: MOF, UML, OCL, XMI y QVT.

UML es un lenguaje de modelado visual que ha tenido gran aceptación en la comunidad del software y que miles de desarrolladores utilizan para crear modelos que le ayudan a razonar sobre el sistema que quieren crear y a documentar sus decisiones de diseño. Con la aparición de herramientas MDA, como OptimalJ, ArcStyler y AndroMDA, se ha podido hacer realidad uno de los objetivos iniciales en el diseño de UML: la generación de código a partir de modelos, pero no sólo el código esqueleto sino también código que incorpora funcionalidad concreta. En el terreno de las herramientas MDA para aplicaciones de tiempo real y sistemas embebidos existen soluciones MDA (UML Ejecutable) que genera el 100% del código de la aplicación y en el caso de herramientas MDA para sistemas empresariales hay una generación entre un 70% y un 80% del código de la aplicación.

MDA ha revitalizado el modelado de software. Hasta ahora han sido pocas las empresas que han practicado modelado intensivo en sus proyectos (Grady Booch, uno de los creadores de UML, estimaba en 2002 que sólo un 5% de los desarrolladores utilizaban UML en sus proyectos y la mayoría para documentar), pero una vez que existan herramientas MDA maduras es lógico pensar que las empresas incorporarán el modelado a sus procesos de desarrollo puesto que los modelos son los artefactos a partir de los cuales se crea toda o parte de la aplicación.

Un **perfil UML** se define como una extensión de un subconjunto de UML orientada a un dominio. Un perfil se obtiene a partir de una especialización de un subconjunto de UML utilizando los conceptos que incorpora el mecanismo de extensión de UML: *estereotipos*, *restricciones* y *valores etiquetados*. Como resultado se obtiene una variante de UML para un propósito específico; existen perfiles adoptados por OMG para CORBA, Java, EJB o C++. Los perfiles UML pueden utilizarse como lenguajes de modelado para crear modelos PIM y PSM y esta es la solución empleada por la mayoría de herramientas, como es el caso de OptimalJ, ArcStyler y AndroMDA.

OCL (*Object Constraint Languaje)* es un lenguaje de especificación que permite definir modelos más precisos y completos, mediante expresiones que pueden establecer el cuerpo de una operación de consulta, un invariante de clase, las pre y postcondiciones de una operación, o especificar las reglas de derivación para atributos o asociaciones.

MOF (*Meta Object Facility*) es el lenguaje de metamodelado propuesto por el OMG. Un lenguaje de modelado se define mediante un metamodelo o descripción de los elementos del lenguaje y de las relaciones existentes entre ellos. MOF es un lenguaje para crear metamodelos (por ejemplo, el metamodelo de UML ha sido definido con MOF), y es, por tanto, un elemento básico de MDA. En realidad, la especificación de MOF, distingue entre el lenguaje EMOF (Essential MOF, el núcleo de MOF) y CMOF (Complete MOF). EMOF es un subconjunto de CMOF que proporciona las primitivas básicas de metamodelado, mientras que CMOF añade mecanismos avanzados de metamodelado, como por ejemplo para extensibilidad y reflexión. En la plataforma Eclipse se ha definido Ecore como lenguaje de metamodelado, que es compatible con EMOF.

Los conceptos básicos que proporciona MOF son clase, asociación, atributo, operación, generalización y el concepto de paquete como elemento organizativo. A la hora de crear un metamodelo, los conceptos del lenguaje se representan como clases,

sus propiedades como atributos y operaciones, las relaciones estructurales entre ellos como asociaciones y agregaciones, y una relación de "un concepto es una especialización de otro" como una generalización.

Una herramienta imprescindible para que MDA tenga éxito es el desarrollo de editores específicos para metamodelos. Hasta ahora como no han existido herramientas de esta naturaleza se ha optado por el uso de perfiles UML para crear lenguajes de modelado, pero la tendencia está cambiando. Entre los generadores de editores para MDA destaca el proyecto GMF (*Graphical Modeling Framework*) dentro de Eclipse, que ofrece un marco basado en el uso de modelos para la definición de editores gráficos. Otro proyecto Eclipse relacionado con MDA es GMT destinado a crear herramientas de transformación.

Lenguaje QVT (Query-View Transformation). OMG ha definido el lenguaje QVT para expresar transformaciones. La propuesta de estándar QVT, que aún no ha sido aceptada, propone tres lenguajes de características diferentes: *Relations*, *Core* y *Operational Mappings*. El lenguaje *Relations* es un lenguaje declarativo, *Operacional Mappings* es imperativo, y ambos pueden implementarse sobre el lenguaje *Core*, que es un lenguaje de transformación con primitivas de bajo nivel. Actualmente la versión definitiva del estándar no ha sido publicada, y no existe ninguna implementación completa de la propuesta actual.

Herramientas. Tres de las herramientas MDA más extendidas son OptimalJ, ArcStyler y AndroMDA. Las dos primeras son comerciales y la segunda es open source. AndroMDA, al igual que ArcStyler, está basada en el concepto de cartucho como artefacto que engloba el lenguaje de modelado y las transformaciones para una determinada plataforma, existiendo en la actualidad cartuchos para la plataforma Java y para Web Services. En cambio, Optimal-J está dirigida a una arquitectura concreta: una arquitectura de tres capas con Struts y EJB.

IV. Modelado Específico del Dominio

Este paradigma es más simple en sus planteamientos que MDA. El desarrollo específico del dominio (DSM, Domain Specific Modeling) plantea elevar el nivel de abstracción por encima de los lenguajes de programación, de modo que sea posible especificar la solución mediante conceptos del dominio, siendo el código final generado automáticamente a partir de esas especificaciones de alto nivel. Los expertos en un dominio crean los lenguajes específicos del dominio (DSL) y sus generadores de código, y los desarrolladores los usan para especificar una solución de alto nivel, de una forma más productiva que escribiendo el código en un lenguaje de programación.

La automatización es factible debido a que tanto el lenguaje y los generadores se ajustan a los requisitos, es decir, se estrecha el dominio o espacio de la solución al que se puede aplicar el lenguaje, a veces reduciéndose al ámbito de los productos de una empresa. La Figura 4 muestra un ejemplo de código expresado con un DSL de Nokia para crear software de teléfonos móviles.

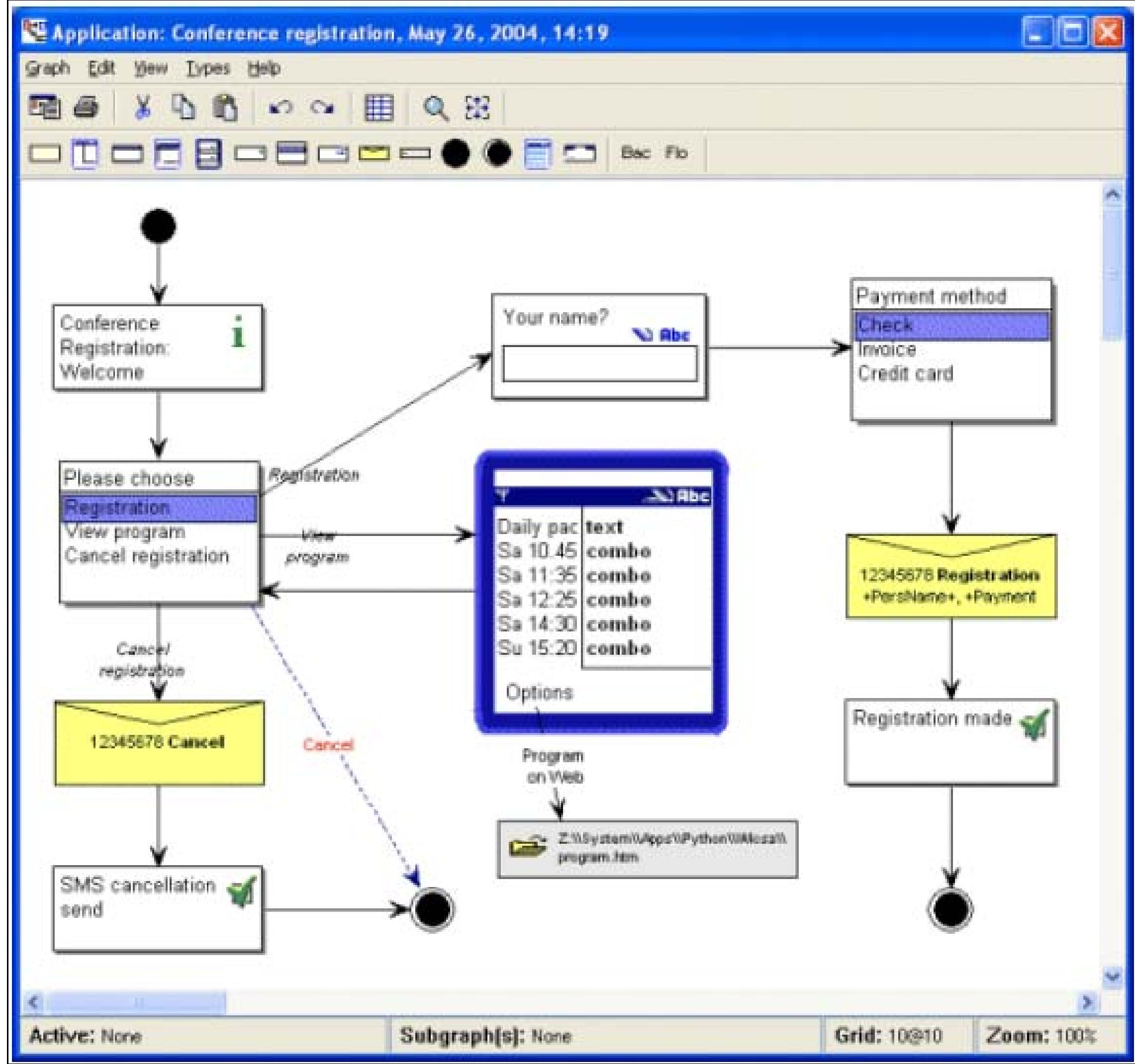


Figura 4: Ejemplo DSM

S. Dmitriev y M. Fowler han acuñado el término *"Programación Orientada a los Lenguajes"* para referirse al desarrollo específico de dominio basado en la creación de DSL, y han descrito los entornos o *language workbench* necesarios para crear DSL. Algunos de los entornos más maduros en el ámbito comercial son Metaedit+ de Metacase (herramienta en la que se basa el presente proyecto), MPS de JetBrains, DSL Tools de Microsoft y XMF/Mosaic de Xactium, y en el ámbito académico GME de Vanderbilt University y el proyecto GMF de Eclipse dentro del contexto de MDA.

Tradicionalmente se ha programado utilizando lenguajes que, en mayor o menor grado, nacieron con el propósito de poder abordar cualquier tipo de problemas, quedando al margen la naturaleza del problema. Aunque esta afirmación resulta algo relativa, pues casi siempre el diseño de cualquier lenguaje de propósito general ha estado orientado al desarrollo en ciertos ámbitos, como puede ser Fortran para cálculo numérico, Cobol para programación de gestión, Pascal para la enseñanza de la programación o PHP y ASP para la programación de aplicaciones Web, etc. Este tipo de lenguajes denominados de propósito general, aunque en muchos casos son considerados lenguajes de alto nivel, no aportan la suficiente abstracción como para atacar directamente el problema sin tratar con aspectos técnicos internos y alejados del dominio del problema.

Esta es la razón de que aunque proporcionan una gran potencia en relación a los lenguajes ensambladores todavía es posible pensar en un paso más que los acerque a los conceptos de los problemas como medio para facilitar la programación y entonces acortar el tiempo de programación y reducir los costes de la producción de software.

A lo largo de esta década se ha revitalizado el interés por los lenguajes específicos del dominio (DSL) con el surgimiento del paradigma del desarrollo dirigido por modelos en especial con MDA, el Desarrollo específico del dominio y las factorías de software. Debemos tener en cuenta que este concepto no es algo nuevo, ya en los años 50 se idearon DSL para programar aplicaciones en máquinas controladas numéricamente. Es más, muchas de las notaciones y lenguajes ampliamente utilizados hoy día pueden considerarse lenguajes específicos del dominio. Algunos de ellos bastante conocidos son: la notación BNF, EXCEL, HTML, LATEX, la aplicación Make, MATLAB, SQL, VHDL, junto a un largo etcétera.

Un DSL se diseña para suplir unas necesidades muy específicas dentro de un dominio de aplicación concreto, contrastando así con los lenguajes de propósito general. El objetivo es la creación de lenguajes especializados para poder, de forma más sencilla, modelar artefactos de software en el ámbito para el que fue prescrito el DSL, y sea capaz de generar código de estos modelos (gráficos o texto). El ámbito de aplicación determinará qué conceptos son aquellos que deberán aparecer en un DSL para permitir el modelado de dichos elementos.

Se trata pues de construir un lenguaje en el que queden plasmados todos los aspectos necesarios para el diseño de software dentro de un dominio de problema. Por ejemplo, debemos tener en cuenta que no resultan aplicables los conceptos en el diseño de una base de datos para una aplicación Web o multimedia.

Un DSL no es estrictamente textual por definición sino que puede ser de tipo gráfico. Es aquí donde juega su papel el DSM pues la utilidad de las herramientas de metamodelado consistirá en dar soporte al diseño de lenguajes gráficos, que permitirán generar el código de forma automática en base al modelo gráfico.

Además, el uso de un metamodelo (DSM) para la creación de una aplicación dentro de un dominio hace más fácil la tarea. Esto es debido a que los distintos componentes ya no son comandos de texto dentro del lenguaje sino gráficos intuitivos que acercan todavía más la herramienta al dominio del problema.

La creación de un DSL no es sencilla y no está siempre justificada. A continuación presentamos un resumen del método para crear DSL, que aunque está centrado en DSL textuales, se pueden extraer ideas válidas para DSM.

En el desarrollo de un DSL pueden distinguirse cinco fases que se resumirán seguidamente y son:

- Decisión
- Análisis
- Diseño
- Implementación
- Despliegue

Para cada una de estas fases se han definido una serie de patrones que facilitan su seguimiento.

i. Decisión

La intención del desarrollo de un DSL es la de mejorar la productividad ahorrando costes en el desarrollo y mantenimiento de software. La fase de decisión no es una tarea fácil pues no siempre está justificado el desarrollo de un DSL. Hay que tener en consideración numerosos factores a la hora de afrontar la decisión de crear un nuevo DSL, y siempre buscar alternativas existentes antes de embarcarse en el desarrollo partiendo de la nada. Evidentemente resultará más económico adoptar un DSL existente que el diseño de uno nuevo.

Para facilitar el proceso de decisión se han elaborado una serie de patrones de decisión:

- **Notación:** Si se dispone de una notación (existente o nueva) específica para el dominio puede resultar un factor decisivo. Surgen dos subpatrones importantes:
    - **Transformar la notación de visual a textual:** para DSL textuales podemos adoptar notaciones visuales disponibles en forma de texto para simplificar la composición de programas o especificaciones complejas.
    - **Proveer de notación amigable a una API existente:** Se trata de convertir una API en un DSL.
- **AVOPT (Análisis, Verificación, Optimización, Paralelización y Transformación):** No resulta factible para aplicaciones escritas con lenguajes de propósito general ya que los patrones del código fuente son demasiado complejos o no están bien definidos. El uso de un DSL apropiado permite estas operaciones. Este patrón se solapa con la mayoría de los demás.
- **Automatización de tareas:** En muchos casos los programadores que utilizan lenguajes de propósito general gastan demasiado tiempo en tareas tediosas y repetitivas que siguen siempre el mismo patrón. En muchos casos estas tareas pueden automatizarse generando código de forma automática mediante un DSL apropiado.
- **Línea de producto:** Los miembros de una línea de productos poseen una estructura común y están construidos en base a un mismo conjunto básico de elementos.
- **Representación de la estructura de datos:** El código dirigido por datos se basa en estructuras de datos inicializadas cuya complejidad las hace difíciles de escribir y mantener. El uso de un DSL quizá permita expresarlas más fácilmente.

- **Estructuras de datos transversales:** Complejas estructuras de datos transversales pueden ser expresadas más fácilmente mediante un DSL apropiado.
- **Sistemas front-end:** Un DSL basado en sistemas front-end se utiliza para manipular la configuración y adaptación del sistema.
- **Interacciones:** Las interacciones por medio de menús o texto suelen especificarse mediante un DSL apropiado, debido a que son tareas complejas y repetitivas.
- **Construcción de GUI:** Las interfaces de usuario habitualmente se diseñan mediante un DSL.

ii. Análisis

El análisis consiste en identificar el dominio y obtener el conocimiento necesario de las fuentes disponibles, como pueden ser documentación técnica relacionada, expertos en el dominio, código lenguajes de propósito general existente, encuestas a los clientes, etc.

De esta fase se trata de obtener una terminología específica del dominio junto con la semántica expresada de manera más o menos abstracta. Para llevar a cabo esta fase se proponen tres patrones:

- **Formal:** Se utilizan técnicas de análisis de dominio específicas.
- **Informal:** El dominio se analiza de manera informal.
- **Extraído del código:** Se obtiene el dominio mediante la inspección "manual" o mediante aplicaciones de código escrito con lenguajes de propósito general.

iii. Diseño

La forma más sencilla de diseñar un DSL consiste en basarse en un lenguaje existente y preferiblemente bien extendido y estandarizado. Esto supone una implementación más sencilla y mayor familiarización para los usuarios. Para el diseño en base a un lenguaje existente se proponen tres patrones:

- **Piggyback[2]:** Un lenguaje existente se utiliza parcialmente obteniendo parte de sus características para el diseño de un DSL.
- **Especialización:** Consiste en restringir un lenguaje existente haciéndolo más simple.
- **Extensión:** Se toma como partida un lenguaje existente y se extienden sus características introduciendo nuevos conceptos.

Para la descripción de un DSL podemos utilizar técnicas formales e informales:

- **Formales:** El DSL se describe utilizando técnicas formales de definición semántica como las gramáticas atribuidas, reglas de escritura o máquinas de estados.
- **Informales:** El DSL se define de de manera informal mediante el lenguaje natural u otros medios.

iv. Implementación

Las técnicas de implementación basados en lenguajes de propósito general no resultan demasiado útiles para DSL, por lo que se utilizan técnicas específicas que consiguen reducir en gran medida el esfuerzo total del desarrollo. Imagínese por un

momento el esfuerzo que supondría implementar un DSL como TinyOS con las técnicas de programación habituales y sin ningún soporte específico. Herramientas como DSL Tools o MetaEdit+ suponen un gran avance en este sentido.

Los patrones propuestos para esta fase en el caso de DSL textuales son:

- **Intérprete:** El DSL se construye en base a un intérprete estándar. Se utiliza especialmente para lenguajes de carácter dinámico o que la velocidad de ejecución no supone un problema. Respecto a la compilación supone mayor simplicidad, mejora sobre el control del entorno de ejecución y facilita su extensión.
- **Compilador:** El DSL se traduce a un lenguaje base y llamadas a librerías. Al contrario que la interpretación tiene un carácter estático. Los compiladores para DSL suelen denominarse **generadores de aplicaciones**.
- **Preprocesadores:** Se traduce a un lenguaje base de manera que el análisis estático queda limitado por lo que haga el procesador del lenguaje. Se definen cuatro subpatrones:
- **Procesamiento de Macros:** Expansión de las definiciones de Macros.
- **Transformación fuente a fuente:** El código del DSL se traduce a un código en lenguaje base.
- **Pipeline:** El DSL es manipulado en etapas sucesivas en las que se va transformando.
- **Procesamiento léxico:** Simplemente se realiza una exploración léxica sin realizar complejos análisis sintácticos.
- **Embebido:** El DSL se construye de forma que quede embebido dentro de un lenguaje de propósito general, definiendo nuevas clases y operadores. Los lenguajes dinámicos son los más utilizados para aplicar esta técnica. En [Sánchez-Cuadrado, 2006] se puede encontrar un ejemplo de lenguaje de transformación de modelo definido como un lenguaje embebido en Ruby.
- **Compilador/intérprete extensible:** Un compilador o intérprete para lenguajes de propósito general se extiende con reglas de optimización específicas del dominio y generadores de código específico. La extensión de un compilador suele ser mucho más costosa que para el caso de los intérpretes.
- **COTS (Commertial Off-The-Selft):** Utilización de herramientas o notaciones existentes aplicadas a un dominio específico.
- **Híbrido:** Las técnicas anteriores pueden combinarse.
- **Entornos de desarrollo:** las más recientes técnicas de implementación de DSL se basa en entornos e desarrollo específicamente orientados para esta tarea, especialmente para los DSL de tipo gráfico, cuya complejidad aumenta en comparación con los de tipo textual. Las dos aplicaciones motivo de nuestro estudio son buenos ejemplos de ello, pues nos abstraen casi en su totalidad de la utilización de lenguajes de propósito general.

v. Despliegue

El despliegue del DSL, una vez finalizadas todas las fases previas, es de vital importancia para garantizar su éxito comercial. Deben proporcionarse ciertas facilidades que permitan a los usuarios finales del DSL implantarlo de manera sencilla en su entorno de desarrollo.

Esta tarea viene muy determinada por la implementación que se ha realizado del DSL, que suele ser un factor limitante a la hora de proporcionar mecanismos de despliegue adecuados para un entorno. Por ejemplo, si para la implementación hemos

utilizado un entorno de desarrollo específico para DSL, dicho entorno debería proporcionar mecanismos para facilitar el despliegue de los DSL creados. En cambio, si nos basamos en la extensión un compilador o intérprete para lenguajes de propósito general el panorama puede ser bien distinto ya que, en principio, no fue diseñado para tales propósitos.

vi. Ventajas e inconvenientes

Ventajas

Los DSL por el simple hecho de estar diseñados específicamente para un dominio de aplicación son, por naturaleza, mucho más expresivos si los comparamos con los lenguajes de propósito general.

También resultan más sencillos para el programador, que conoce perfectamente el dominio para el que fue diseñado el DSL. Podemos decir que los DSL acercan la programación a personal cualificado en el dominio pero con escasos conocimientos de programación.

La verificación de los programas también se ve simplificada al permitir un mayor número de propiedades que podemos comprobar. Al contrario que sucede con los lenguajes de propósito general en DSL podemos definir restricciones a nivel semántico que se encargan de asegurar el cumplimiento de ciertas propiedades críticas.

Al simplificarse la programación se reducen los costos de desarrollo y mantenimiento aumentando la productividad.

Por otra parte, un DSL ofrece pautas y funcionalidades intrínsecas que refuerzan la reutilización. En la implementación de un DSL se captura el conocimiento del dominio, de forma implícita ocultando los patrones comunes del programa o explícitamente exponiendo los parámetros apropiados al programador del DSL. Así cualquier usuario reutiliza necesariamente componentes de la biblioteca y conocimiento del dominio.

Inconvenientes

El desarrollo de un DSL es muy difícil, requiere conocimiento amplio del dominio y de desarrollo de lenguajes. Los expertos no suelen tener ambos conocimientos. Además las técnicas utilizadas son muy variadas y precisan de una mayor atención en todos los factores.

Otro problema fundamental es el aumento en el costo de las tareas de documentación, soporte técnico, mantenimiento y estandarización, en las que se produce un sustancial aumento en esfuerzo y tiempo. Es un factor a tener en cuenta especialmente en proyectos de gran magnitud.

V. Redes de sensores inalámbricas

Recientes avances en las tecnologías de adquisición de datos, microelectrónica y transmisión RF han permitido el auge de las redes de sensores inalámbricas. Durante los últimos años éstas se están comenzando a aplicar a la recolección de datos en diversos entornos. Principalmente son usadas por investigadores especialistas en este campo que persiguen la evolución de esta tecnología. Sin embargo, cada vez es más frecuente encontrarse con aplicaciones reales que sirven para otros fines científicos. Así, es posible encontrar aplicaciones donde las redes de sensores se usan para monitorizar fenómenos naturales o estudiar comportamientos de animales o personas.

En cualquiera de los casos a los que se aplican las redes de sensores, la forma en que estas funcionan es similar. Todo gira entorno a unos nuevos dispositivos, los motes, que constituyen la pieza central de las redes de sensores. Un mote, también conocido como nodo sensor, es un elemento que combina capacidades de recolección, procesado y transmisión de datos en un mismo dispositivo, logrando todo esto con un reducido coste económico, tamaño y consumo de potencia. Al esparcir estos dispositivos en las inmediaciones del fenómeno que se pretende analizar se pueden obtener estudios que superan notablemente en calidad a los realizados con tecnologías anteriores. Estos beneficios se deben a las características que se han citado sobre los nodos sensores. Su bajo coste permite desplegar una red en la que haya una cantidad considerable de nodos, lo cual da una mayor resolución para el estudio del fenómeno de interés. El uso de la tecnología *wireless* para estos nodos evita el uso de cableado y, por tanto, da mayor libertad a la hora de situar los nodos. Además, su funcionamiento autónomo posibilita trabajar sin la presencia humana que, en muchos casos de estudio, alteraría los comportamientos estudiados o pondría en riesgo la integridad de las personas.

Puesto que se trata de una tecnología joven, la investigación realizada hasta ahora se ha dedicado principalmente a sentar las bases para el desarrollo de aplicaciones con redes de sensores. La mayor parte de los esfuerzos han ido dirigidos hacia el desarrollo de hardware y de técnicas que mejoren las prestaciones de estas redes. Así, en estos campos, se han desarrollado numerosas plataformas hardware, sensores y algoritmos. La gran cantidad de nuevos artefactos disponibles para redes de sensores ha hecho que se haya ampliado su funcionalidad, no siendo meramente la de recolectar datos. Como consecuencia, el concepto que se tenía de estas redes está cambiando. En un principio se las consideraba como redes homogéneas ya que todos sus elementos eran de un mismo tipo. Sin embargo ya podemos encontrarnos con redes heterogéneas donde los nodos pueden integrar diversos sensores, e incluso alguno de ellos, los nodos actuadores, tienen capacidad de actuación sobre el entorno.

Todos los anteriores avances están llevando a una complejidad creciente que hace a las aplicaciones más difíciles de desarrollar. Si bien se disponen de sistemas operativos que simplifican la programación de éstas, se carece de reglas y modelos que guíen su proceso de desarrollo. Es en este punto donde la ingeniería del software puede jugar un papel importante. El objetivo que se plantea es la consecución de un modelo arquitectónico que cubra los requerimientos propios de una red de sensores. En este sentido se están realizando *frameworks* para redes de sensores inalámbricas como TinyCubus, que se centra principalmente en la adaptabilidad y reconfiguración de la red. Sin embargo, éste, como la mayoría de *frameworks* disponibles, no ofrece una estructuración clara que diferencie los bloques funcionales en los que se dividen las aplicaciones.

Capítulo 2

Redes de Sensores Inalámbricas

I. Introducción

El MIT identificó en Febrero de 2003 las diez tecnologías emergentes que cambiarán el mundo y las redes de sensores inalámbricas aparecían en primer lugar.

Las Redes de Sensores Inalámbricas (Wireless Sensor Networks, WSN) consisten en un conjunto de nodos de pequeño tamaño, de muy bajo consumo y capaces de una comunicación sin cables, interconectados entre sí a través de una red y a su vez conectados a un sistema central encargado de recopilar la información recogida por cada uno de los sensores.

Este novedoso tipo de redes se han convertido en un campo de estudio que se encuentra en continuo crecimiento, ya que últimamente han surgido una serie de dispositivos que integran un procesador, una pequeña memoria, sensores y comunicación inalámbrica.

Al estar dotados con procesador, estos nodos son capaces de realizar ciertas computaciones locales sobre los datos tomados, lo que permite una serie de ventajas como una reducción de tráfico a través de la red, ya que sería procesada localmente, y consecuentemente una lógica descarga de trabajo del computador central.

Figura 5: Red de sensores inalámbricos

Las redes de sensores con cables no son nuevas y sus funciones incluyen medir niveles de temperatura, líquido, humedad etc. Muchos sensores en fábricas o coches por ejemplo, tienen su propia red que se conecta con un ordenador o una caja de controles a través de un cable y, al detectar una anomalía, envían un aviso a la caja de controles.

La diferencia entre los sensores que todos conocemos y la nueva generación de redes de sensores inalámbricas es que estos últimos son inteligentes, es decir, capaces de poner en marcha una acción según la información que vayan acumulando, y no están limitados geográficamente por un cable fijo.

Los nuevos avances en la fabricación de microchips de radio, nuevas formas de enrutado y nuevos programas informáticos relacionados con redes están logrando eliminar los cables de las redes de sensores, multiplicando así su potencial.

El ámbito de aplicación de este tipo de sistemas, como veremos, es muy amplio: monitorización de entornos naturales, aplicaciones para defensa, aplicaciones médicas en observación de pacientes, etc. El motivo del éxito de este tipo de redes de sensores se debe a sus especiales características físicas.

A los nodos de las redes se les imponen unas restricciones de consumo severas. El motivo de la imposición de estas restricciones es la necesidad de que los nodos sean capaces de operar, por sí mismos, durante periodos largos de tiempo, en lugares donde las fuentes de alimentación son si no inexistentes, de baja potencia. El tamaño es otra restricción que cada vez se hace más necesaria para la mayoría de las aplicaciones, de manera que las tarjetas o nodos que forman las redes de sensores sean cada vez de menor tamaño.

Desde el punto de vista del software, para la realización de las aplicaciones para redes de sensores, la Universidad de Berkeley e Intel han desarrollado una plataforma específica para este tipo de sistemas, que tiene en cuenta las restricciones de los nodos. En particular se ha desarrollado un sistema operativo, llamado TinyOS, cuya característica principal reside en que al ser modular resulta ideal para instalarse en sistemas con restricciones de memoria.

Se desarrolló también un lenguaje de programación, llamado nesC, de sintaxis muy parecida a C, basado en componentes, y a partir del cual se rediseñó una primera versión de TinyOS de modo que actualmente está íntegramente implementado sobre nesC. Tanto nesC como TinyOS están basados en componentes e interfaces bidireccionales. Además, actualmente, Berkeley e Intel han desarrollado diversas aplicaciones a modo de ejemplo, simuladores de ejecución, y varias universidades internacionales están dedicando esfuerzos al desarrollo de aplicaciones usando esta emergente tecnología.

Existen otras empresas que son proveedores de esta tecnología, el mayor de estos es Crossbow Technology, que ha desarrollado redes de sensores a gran escala para su uso comercial.

Las últimas investigaciones apuntan hacia una eventual proliferación de redes de sensores inteligentes, redes que recogerán enormes cantidades de información hasta ahora no registrada que contribuirá de forma favorable al buen funcionamiento de fábricas, al cuidado de cultivos, a tareas domésticas, a la organización del trabajo y a la predicción de desastres naturales como los terremotos. En este sentido, la computación que penetra en todas las facetas de la vida diaria de los seres humanos está a punto de convertirse en realidad.

Si los avances tecnológicos en este campo siguen a la misma velocidad que han hecho en los últimos años, las redes de sensores inalámbricas revolucionarán la capacidad de interacción de los seres humanos con el mundo.

II. Historia

Las redes de sensores nacen, como viene siendo habitual en el ámbito tecnológico, de aplicaciones de carácter militar.

La primera de estas redes fue desarrollada por Estados Unidos durante la guerra fría y se trataba de una red de sensores acústicos desplegados en el fondo del mar cuya misión era desvelar la posición de los silenciosos submarinos soviéticos, el nombre de esta red era SOSUS (Sound Surveillance System). Paralelamente a ésta, también EE.UU. desplegó una red de radares aéreos a modo de sensores que han ido evolucionando hasta dar lugar a los famosos aviones AWACS, que no son más que sensores aéreos. SOSUS ha evolucionado hacia aplicaciones civiles como control sísmico y biológico, sin embargo AWACS sigue teniendo un papel activo en las campañas de guerra.

A partir de 1980, la DARPA comienza un programa focalizado en sensores denominado DSN (Distributed Sensor Networks), gracias a él se crearon sistemas operativos (Accent) y lenguajes de programación (SPLICE) orientados de forma específica a las redes de sensores, esto ha dado lugar a nuevos sistemas militares como CEC (Cooperative Engadgement Capability) consistente en un grupo de radares que comparten toda su información obteniendo finalmente un mapa común con una mayor exactitud.

Estas primeras redes de sensores tan sólo destacaban por sus fines militares, aún no satisfacían algunos requisitos de gran importancia en este tipo de redes tales como la autonomía y el tamaño. Entrados en la década de los 90, una vez más DARPA lanza un nuevo programa enfocado hacia redes de sensores llamado SensIt, su objetivo viene a mejorar aspectos relacionados con la velocidad de adaptación de los sensores en ambientes cambiantes y en cómo hacer que la información que recogen los sensores sea fiable.

Ha sido a finales de los años 90 y principios de nuestro siglo cuando los sensores han empezado a coger una mayor relevancia en el ámbito civil, decreciendo en tamaño e incrementando su autonomía. Compañías como Crossbow han desarrollado nodos sensores del tamaño de una moneda con la tecnología necesaria para cumplir su cometido funcionando con baterías que les hacen tener una autonomía razonable y una independencia inédita.

El futuro ya ha empezado a ser escrito por otra compañía llamada Dust Inc, compuesta por miembros del proyecto Smart Dust ubicado en Berkeley, que ha creado nodos de un tamaño inferior al de un guisante y que, debido a su minúsculo tamaño, podrán ser creadas múltiples nuevas aplicaciones.

Figura 6: Tamaño de los nodos Smart Dust

III. Características de las WSN

Los recientes avances en microelectrónica, wireless y electrónica digital han permitido el desarrollo de nodos sensores de bajo coste, reducido tamaño, bajo consumo y que se comunican de forma inalámbrica.

El desarrollo de estos nodos sensores ha dado la posibilidad de crear redes basadas en cooperación de los nodos, con una notable mejora sobre redes de sensores tradicionales, que se suelen desplegar de dos modos:

- Sensores que se encuentran lejos del fenómeno, grandes y muy complejos para distinguir el objetivo del ruido del entorno.
- Muchos sensores con posición y topología cuidadosamente seleccionada. Transmiten datos de adquisición a nodos centrales que realizan la computación.

Como ya hemos comentado, las WSN se componen de miles de dispositivos pequeños, autónomos, distribuidos geográficamente, llamados nodos sensores. Estos gozan de capacidad de cómputo, almacenamiento y comunicación en una red conectada sin cables, e instalados alrededor de un fenómeno objeto para monitorizarlo.

Una vez se produzcan eventos, toma de medidas o cualquier actividad programada con el fenómeno en cuestión los nodos enviarán información a través de la red, hasta llegar a un sistema central de control que recogerá los datos y los evaluará, ejecutando las acciones pertinentes en comunicación con otros sistemas o en la propia red de sensores.

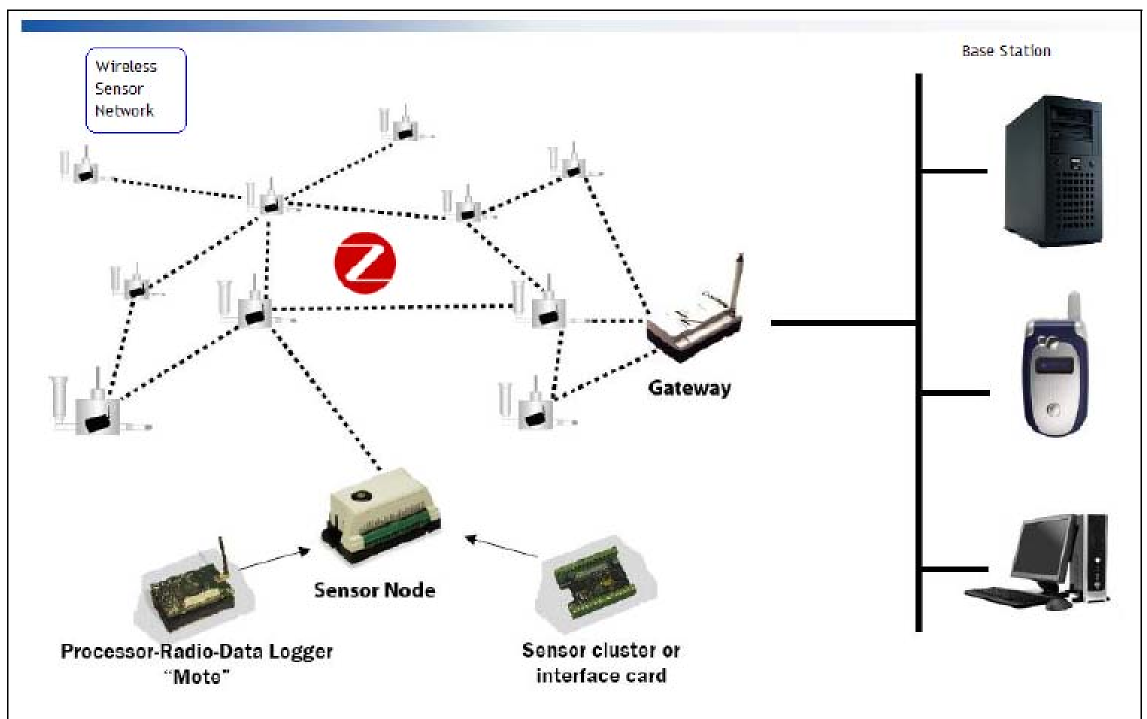


Figura 7: Elementos de una WSN

Tal y como vemos en esta figura podemos establecer, por tanto, una serie de elementos que componen de forma general una WSN:

1. **Sensores.**

De distinta naturaleza y tecnología, toman del medio la información y la convierten en señales eléctricas.

2. **Nodos sensores.**
   Son los procesadores de radio, que toman los datos del sensor a través de sus puertas de datos, y envían la información a la estación base.
3. **Pasarelas o Gateways.**
   Elementos para la interconexión entre la red de sensores y una red TCP/IP.
4. **Estaciones base.**
   Recolector de datos basado en un ordenador común o sistema integrado.
5. **Red inalámbrica.**
   Típicamente basada en el estándar 802.15.4 - ZigBee.

Características de una WSN

Aunque la naturaleza de los nodos que emplean las redes pueda ser muy distinta y la misión a realizar pueda variar dependiendo del tipo de aplicación, se pueden identificar una serie de características comunes a todas ellas y que son las que principalmente las identifican:

- Gran Escala.
  La cantidad de nodos que se despliega en una red puede llegar hasta los miles, pudiendo crecer su número a lo largo de la vida de la red. La red se va a componer de muchos sensores densamente desplegados en el lugar donde se produce el fenómeno y, por lo tanto, muy cerca de él.
- Topología variable.
  La posición en que se coloca cada nodo puede ser arbitraria y normalmente es desconocida por los otros nodos. La localización no tiene porqué estar diseñada ni preestablecida, lo que va a permitir un despliegue aleatorio en terrenos inaccesibles u operaciones de alivio en desastres. Por otro lado, los algoritmos y protocolos de red deberán de poder organizarse automáticamente.
- Recursos limitados.
  Los sensores, a cambio de su bajo consumo de potencia, coste y pequeño tamaño disponen de recursos limitados. Los dispositivos actuales más usados, los mica2, cuentan con procesadores a 4 MHz, 4 Kbytes de RAM, 128 Kbytes de memoria de programa y 512 Kbytes de memoria flash para datos. Su radio permite trasmitir a una tasa de 38.4 KBaudios.
- Cooperación de nodos sensores.
  Realizan operaciones simples antes de transmitir los datos, lo que se denomina un procesamiento parcial o local.
- Comunicación.
  Los nodos sensores usan comunicación por difusión y debido a que están densamente desplegados, los vecinos están muy cerca unos de otros y la comunicación multihop (salto múltiple de uno a otro) consigue un menor consumo de potencia que la comunicación single hop (salto simple). Además, los niveles de transmisión de potencia se mantienen muy bajos y existen menos problemas de propagación en comunicaciones inalámbricas de larga distancia.
- Funcionamiento autónomo.
  Puede que no se acceda físicamente a los nodos por un largo periodo de tiempo. Incluso tal vez esto nunca ocurra. Durante el tiempo en el que el

nodo permanece sin ser atendido puede que sus baterías se agoten o su funcionamiento deje de ser el esperado.

Requisitos para una WSN

Para que una red pueda funcionar de acuerdo con las anteriores características surgen una serie de retos que la aplicación debe resolver. Estos, que se detallan a continuación, son los requisitos no funcionales del sistema:

- Eficiencia energética.
  Es uno de los asuntos más importantes en redes de sensores. Cuanto más se consiga bajar el consumo de un nodo mayor será el tiempo durante el cual pueda operar y, por tanto, mayor tiempo de vida tendrá la red. La aplicación tiene la capacidad de bajar este consumo de potencia restringiendo el uso de la CPU y la radio FM. Esto se consigue desactivándolos cuando no se utilizan y, sobre todo, disminuyendo el número de mensajes que generan y retransmiten los nodos.
- Autoorganización.
  Los nodos desplegados deben formar una topología que permita establecer rutas por las que mandar los datos que han obtenido. Los nodos necesitan conocer su lugar en esta topología, pero resulta inviable asignarlo manualmente a cada uno, debido al gran número de estos. Es fundamental, por tanto, que los nodos sean capaces de formar la topología deseada sin ayuda del exterior de la red. Este proceso no sólo debe ejecutarse cuando la red comienza su funcionamiento, sino que debe permitir que en cada momento la red se adapte a los cambios que pueda haber en ella.
- Escalabilidad.
  Puesto que las aplicaciones van creciendo con el tiempo y el despliegue de la red es progresivo, es necesario que la solución elegida para la red permita su crecimiento. No sólo es necesario que la red funcione correctamente con el número de nodos con que inicialmente se contaba, sino que también debe permitir aumentar ese número sin que las prestaciones de la red caigan drásticamente.
- Tolerancia a fallos.
  Los sensores son dispositivos propensos a fallar. Los fallos pueden deberse a múltiples causas, pueden venir a raíz del estado de su batería, de un error de programación, de condiciones ambientales, del estado de la red, etc. Una de las razones de esta probabilidad de fallo radica precisamente en el bajo coste de los sensores. En cualquier caso, se deben minimizar las consecuencias de ese fallo. Por todos los medios se debe evitar que un fallo en un nodo individual provoque el mal funcionamiento del conjunto de la red.
- Tiempo real.
  Los datos llegan a su destino con cierto retraso. Pero algunos datos deben entregarse dentro de un intervalo de tiempo conocido. Pasado éste dejan de ser válidos, como puede pasar con datos que impliquen una reacción inmediata del sistema, o se pueden originar problemas serios como ocurriría si se ignora una alarma crítica. En caso de que una aplicación tenga estas restricciones debe tomar las medidas que garanticen la llegada a tiempo de los datos.
- Seguridad.
  Las comunicaciones wireless viajan por un medio fácilmente accesible a personas ajenas a la red de sensores. Esto implica un riesgo potencial para

los datos recolectados y para el funcionamiento de la red. Se deben establecer mecanismos que permitan tanto proteger los datos de estos intrusos, como protegerse de los datos que estos puedan inyectar en la red.

Según la aplicación que se diseñe algunos de los anteriores requisitos cobran mayor importancia. Como ejemplo podemos considerar una aplicación que controle el comportamiento de animales salvajes dentro de un parque natural. Para determinar su localización, cada uno de estos animales llevaría sujeto un pequeño sensor. En esta situación la capacidad de autoorganización cobraría gran importancia. Sin embargo, si pensamos en una red con nodos inmóviles, como los usados en la red domótica de una oficina, este mismo atributo sería de menor importancia.

Es necesario encontrar el peso que cada uno de estos requisitos tiene en el diseño de la red, pues normalmente unos requisitos van en detrimento de otros. Por ejemplo, dotar a una red de propiedades de tiempo real podría implicar aumentar la frecuencia con la que se mandan mensajes con datos, lo cual repercutiría en un mayor consumo de potencia y un menor tiempo de vida de los nodos.

Esto lleva a buscar, para cada aplicación, un compromiso entre los requisitos anteriores que permita lograr un funcionamiento de la red adecuado para la misión que debe realizar.

i. Arquitecturas de las WSN

Tomando como elementos principales de la red a los nodos sensores, las pasarelas (gateway) y las estaciones base, podemos distinguir dos tipos principales de arquitecturas.

Arquitectura centralizada

En este tipo de arquitectura los nodos de una red que estudian un fenómeno enviarán sus datos directamente a la pasarela más cercana, que dirige el tráfico de esa red en concreto.

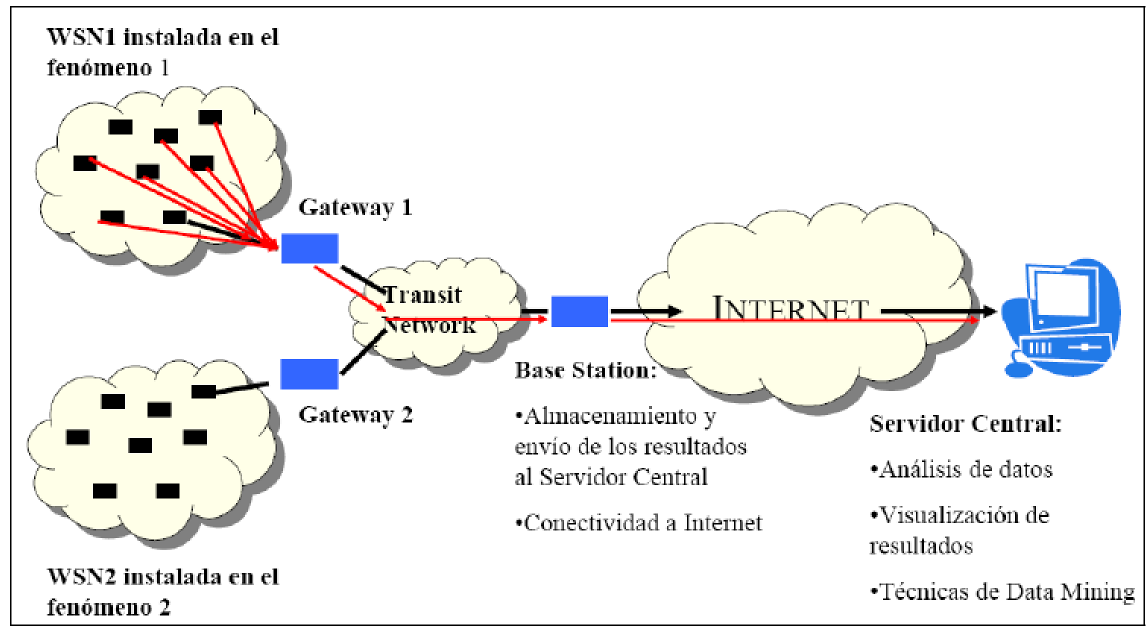


Figura 8: Arquitectura WSN centralizada

Si tenemos en cuenta que el ciclo de vida de un nodo consiste en despertarse, medir, transmitir y dormirse, y cada vez que transmita su mensaje irá a la pasarela, estaremos creando dos grandes problemas para la red:

1. Cuello de botella en las pasarelas.
2. Mayor consumo de energía por las comunicaciones.

Como resultado, el tiempo de vida de la red será relativamente corto.

Arquitectura distribuida

Dada la naturaleza intrínseca de las redes de sensores, normalmente se tiende a este tipo de arquitectura con una computación distribuida. De hecho, como comentábamos en las características principales de una WSN, son redes basadas en cooperación de los nodos.

Los nodos sensores se van a comunicar entre sus nodos vecinos y van a cooperar entre ellos, ejecutando algoritmos distribuidos para obtener una única respuesta global que un nodo (cluster head) se encargará de comunicar a la estación base a través de las pasarelas pertinentes.

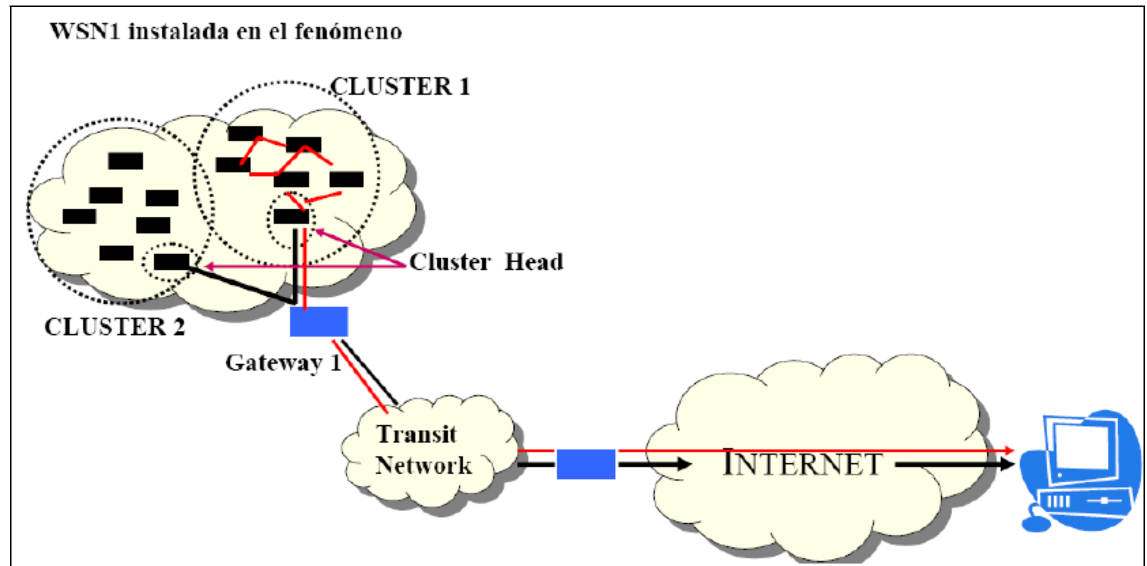


Figura 9: Arquitectura WSN distribuida

De esta forma se evitan los problemas que surgían en la arquitectura centralizada y, además, se mantienen las características y ventajas que comentamos al comienzo de este capítulo.

ii. Protocolos de las WSN

A continuación vamos a ver por qué son diferentes las WSN de las redes tradicionales y por qué existen algoritmos y protocolos que no se ajustan a las redes de sensores, ya que se encuentran diferencias fundamentales en los principales objetivos de ambas redes.

Las WSN utilizan una comunicación inalámbrica y, obviamente, son diferentes de las redes cableadas. También son diferentes de las tradicionales redes inalámbricas como las redes celulares, Bluetooth o las móviles ad hoc (MANETS). En estas redes, el objetivo es optimizar el rendimiento y el retardo. Aunque las MANETS comparten las características de desarrollo ad hoc y configuración automática de los nodos, el consumo de potencia no es una prioridad. Bluetooth también trata los mismos problemas de limitación de potencia pero el grado de bajo consumo de potencia que se

requiere en las WSN es mucho mayor. Además, los nodos sensores son frecuentemente expuestos a extremas condiciones ambientales, haciéndolos propensos a frecuentes fallos en los nodos. Esto conlleva unas restricciones estrictas en las WSN, no como en las otras redes.

Capa física

Aunque la transmisión en las WSN puede ser por infrarrojos, radio o medio óptico, la banda industrial, científica y médica de los 915MHz (ISM) se ha hecho muy popular en las redes de sensores. La deficiencia de usar infrarrojos o medios ópticos para la transmisión es que requieren que los nodos transmisor y receptor mantengan una línea de contacto visible. Sin embargo, algunas especificaciones inalámbricas como Bluetooth, HomeRF, y las redes LAN Wireless especificadas por el IEEE 802.11b operan todas en la frecuencia de los 2.4GHz.

Capa de enlace

La responsabilidad de la capa de enlace es establecer un enlace fiable o una infraestructura de red sobre la que los datos puedan ser encaminados. Existen especificaciones como Bluetooth que utilizan la multiplexación por división en el tiempo (TDMA) con saltos de frecuencia mientras que las redes LAN inalámbricas especificadas por el 802.11b utilizan un método de acceso al medio con detección de portadora y evitando colisiones (CSMA/CA).

La necesidad de un nuevo protocolo de capa MAC para WSN radica en que las dificultades de las WSN son muy distintas de los problemas que tenían las especificaciones existentes. Por ejemplo, el alcance de un piconet, que es una colección de ocho nodos, en una red Bluetooth es de 10 metros, mientras que el alcance requerido es mucho más pequeño en una WSN. En las redes celulares, las estaciones base forman una columna cableada proporcionando una estructura parcial a la red. En las WSN, no hay estaciones base.

Un protocolo de capa MAC para las WSN es el llamado MAC autoorganizado para redes de sensores (SMACS), que configura la capa de enlace. El algoritmo de escucha y registro (EAR) permite a los nodos sensores móviles interconectar nodos estacionarios. SMACS actúa al crear la red detectando los nodos vecinos usando transferencia de mensajes. En SMACS, un canal se define con un par de intervalos de tiempo. La detección de vecinos y la asignación de canales se combinan en una fase, para que cuando los nodos vayan a escuchar a sus vecinos, ya hayan formado una red conectada. No hay jerarquías asumidas en SMACS y por esto se forma una topología llana. El algoritmo EAR tiene el problema del control de la movilidad cuando se introducen nodos móviles en la red.

Capa de red

El encaminamiento en las WSN es bastante similar al de los protocolos ad hoc en las redes MANETS. La diferencia es que en los algoritmos de enrutamiento ad hoc, el consumo de potencia es secundario. En redes Bluetooth, las comunicaciones de un nodo master con siete esclavos definen un piconet. Cuando los piconets están interconectados para formar redes dispersas, las diferentes topologías limitan a los nodos que las forman. Para una WSN con la posibilidad de cientos de nodos, esto no será suficiente. Y no sólo eso, sino que también el coste proyectado de un dispositivo Bluetooth es menos de $4 mientras que el precio estimado de un nodo sensor es de menos de $1. Además, los requisitos de potencia de los nodos sensores son mucho menores que para Bluetooth.

Varios algoritmos se han propuesto para el encaminamiento de WSN. El principal objetivo de los algoritmos es el bajo consumo. Se pueden clasificar como aquellos que determinan:

1. Rutas con los nodos de mayor potencia a lo largo de la ruta.
2. Rutas que consuman la mínima energía.
3. Rutas con el mínimo número de saltos.
4. Rutas en las que la mínima potencia disponible es la máxima entre todos los demás caminos.

Capa de transporte

La necesidad de una capa de transporte radica en que las WSN necesitan ser conectadas a una red más grande, como Internet. Las WSN se conectan a Intenet por medio de pasarelas.

El protocolo de la capa de transporte que conecta el usuario con la pasarela podría ser TCP o UDP, ya existentes. Sin embargo, el protocolo que conecta la pasarela y los nodos sensores tendría que ser diferente ya que no hay un esquema de direccionamiento global en una red de sensores. La limitación de memoria en los nodos sensores conlleva una cuestión importante en tanto que se prefieren protocolos que requieran un bajo almacenamiento de información de estado antes que otros del tipo TCP.

Capa de aplicación

Los nodos sensores tienen muchas aplicaciones distintas. Diseñar una capa de aplicación tiene el mérito de que las WSN pueden ser conectadas a grandes redes como Internet. El direccionamiento de nodos es una cuestión importante aquí ya que, no como en otras redes, los nodos sensores no tienen un identificativo global.

Los protocolos de la capa de aplicación como el Protocolo de Administración de Sensores (SMP) y el Protocolo de Petición de Sensores y Entrega de Datos (SQDDP) están actualmente en investigación. El SQDDP introduce una interfaz de peticiones para emitir peticiones, responderlas y recopilar las respuestas de las peticiones.

Aunque las aplicaciones de las WSN son diversas, las últimas investigaciones indican que se van a tener que realizar más desarrollos en el direccionamiento de varios protocolos en todas las capas. Además, se necesita una plataforma común donde se puedan probar los algoritmos propuestos. Las simulaciones de WSN son difíciles y normalmente están vinculadas a una aplicación en particular.

El área de las WSN es un área en expansión y en los próximos años veremos los resultados de los esfuerzos que se están realizando en estas aplicaciones.

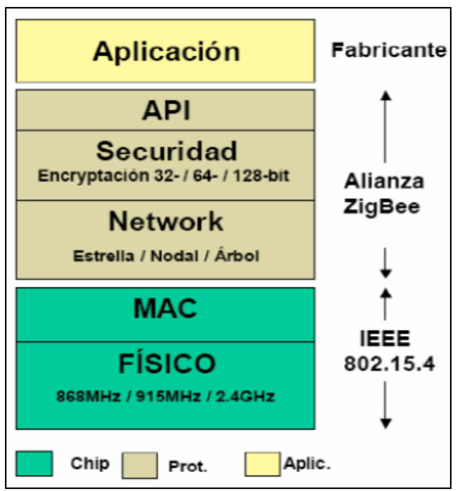


Figura 10: Niveles físico, red y aplicación

iii. Zigbee, estándar WSN

ZigBee es el estándar de la norma IEEE 802.15.4 que define el protocolo y la interconexión de dispositivos con comunicación vía radio para redes de área personal inalámbricas (WPAN) patrocinado por la ZigBee Alliance.

La tecnología está diseñada con el objetivo de ser más simple y barata que otras WPANs tales como Bluetooth, y está apuntando su uso al de aplicaciones de bajas tasas de datos y bajo consumo de potencia.

El estándar usa CSMA/CA como acceso al medio, acceso múltiple con detección de portadora evitando colisiones y soporta topologías en estrella y punto a punto. Opera en las bandas libres de los 2.4 Ghz, 915 MHz y 868 MHz, y, al igual que WiFi, usa DSSS (Direct Sequence Spread Spectrum) como método de transmisión y se focaliza en las capas inferiores de red (Física y MAC).

La transmisión se realiza a 20 kbps para el caso de las frecuencias de 915/868 MHz mientras que aumenta a 250 kbps para las que son de 2.4 GHz. Finalmente, el rango de transmisión está entre los 10 y 75 metros, dependiendo de la potencia de transmisión y del entorno.

Estándar	Wi-Fi 802.11g	Wi-Fi 802.11b	Bluetooth 802.15.1	ZigBee 802.15.4
Aplicación principal	WLAN	WLAN	WPAN (sustituir cable entre dos dispositivos)	Control y monitorización
Memoria necesaria	1MB+	1MB+	250KB+	4KB - 32KB
Vida Batería (días)	0,5 - 5	0,5 - 5	1 – 7	100 - 1000+
Tamaño Red	32 nodos	32 nodos	7	255 / 65.000
Velocidad (Kbps)	54 Mbps	11 Mbps	720 Kbps	20 – 250 Kbps
Cobertura (metros)	100	100	10 (v1.1)	1 – 100
Parámetros más importantes	Velocidad y Flexibilidad	Velocidad y Flexibilidad	Coste y perfiles de aplicación	Fiabilidad, bajo consumo y muy bajo coste

Figura 11: Comparativa estándares inalámbricos

En esta tabla comparativa podemos observar cómo cada estándar tiene unas características distintas en cuanto a velocidad de transmisión, memoria necesaria o vida de las baterías, por lo que se enfocan para unas aplicaciones y parámetros clave muy distintos.

Una red ZigBee puede estar formada por hasta 255 nodos los cuales tienen la mayor parte del tiempo el transmisor ZigBee dormido con objeto de consumir menos que otras tecnologías inalámbricas. El objetivo es que un sensor equipado con un transmisor ZigBee pueda ser alimentado con dos pilas AA durante al menos 6 meses y hasta 2 años.

Como comparativa, la tecnología Bluetooth es capaz de llegar a 1 Mbps en distancias de hasta 10 m operando en la misma banda de 2,4 GHz, sólo puede tener 8 nodos por celda y está diseñado para mantener sesiones de voz de forma continuada, aunque pueden construirse redes que cubran grandes superficies ya que cada ZigBee actúa de repetidor enviando la señal al siguiente, etc.

En la siguiente gráfica se puede ver claramente dónde se ajusta cada estándar, en función de su alcance y su tasa de datos, aunque habría que tener en cuenta factores que diferencian el Bluetooth de Zigbee como es el consumo de potencia que hemos comentado.

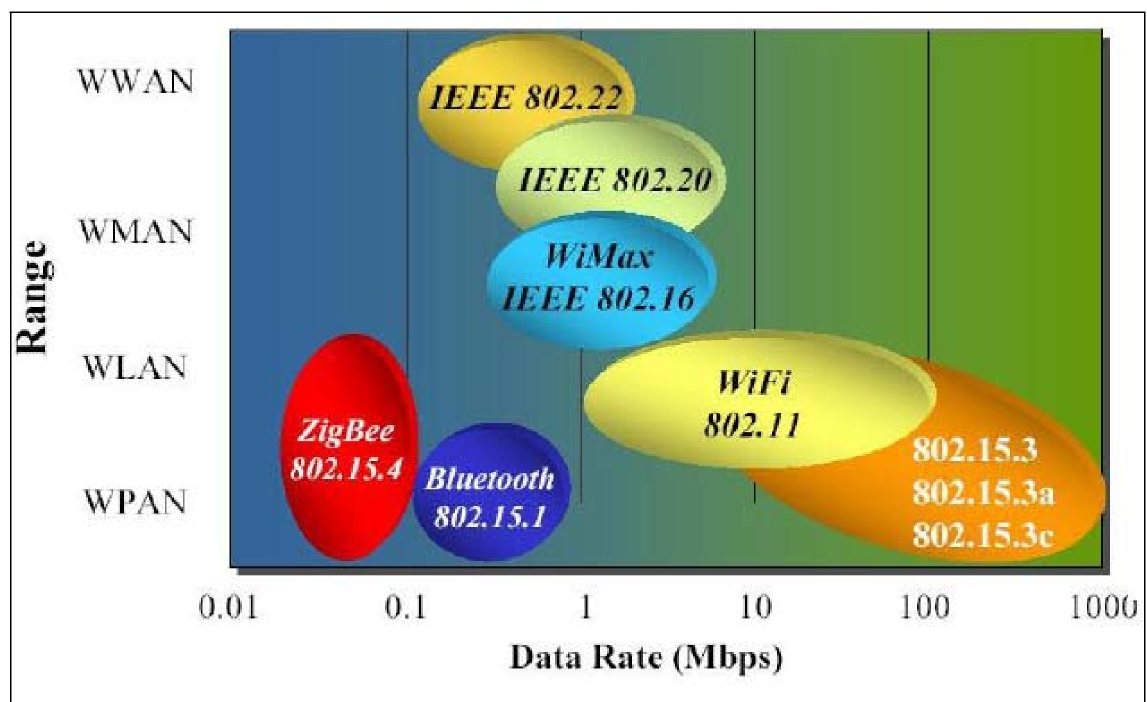


Figura 12: Espectro de estándares

Se espera que los módulos ZigBee sean los transmisores inalámbricos más baratos jamás producidos de forma masiva, con un coste estimado alrededor de los 2 euros. Dispondrán de una antena integrada, control de frecuencia y una pequeña batería. ZigBee Alliance es una alianza, sin ánimo de lucro, de más de 100 empresas, la mayoría de ellas fabricantes de semiconductores, con el objetivo de auspiciar el desarrollo e implantación de una tecnología inalámbrica de bajo coste. La alianza de empresas está trabajando codo con codo con IEEE para asegurar una integración, completa y operativa. Los principales mercados de la ZigBee Alliance son la automatización de viviendas, edificios y la automatización industrial.

Además de ser el estándar aceptado y utilizado por las WSN, ZigBee es un sistema ideal para redes domóticas, específicamente diseñado para reemplazar la proliferación de sensores y actuadores individuales. ZigBee fue creado para cubrir la necesidad del mercado de un sistema a bajo coste, un estándar para redes Wireless de pequeños paquetes de información, bajo consumo, seguro y fiable.

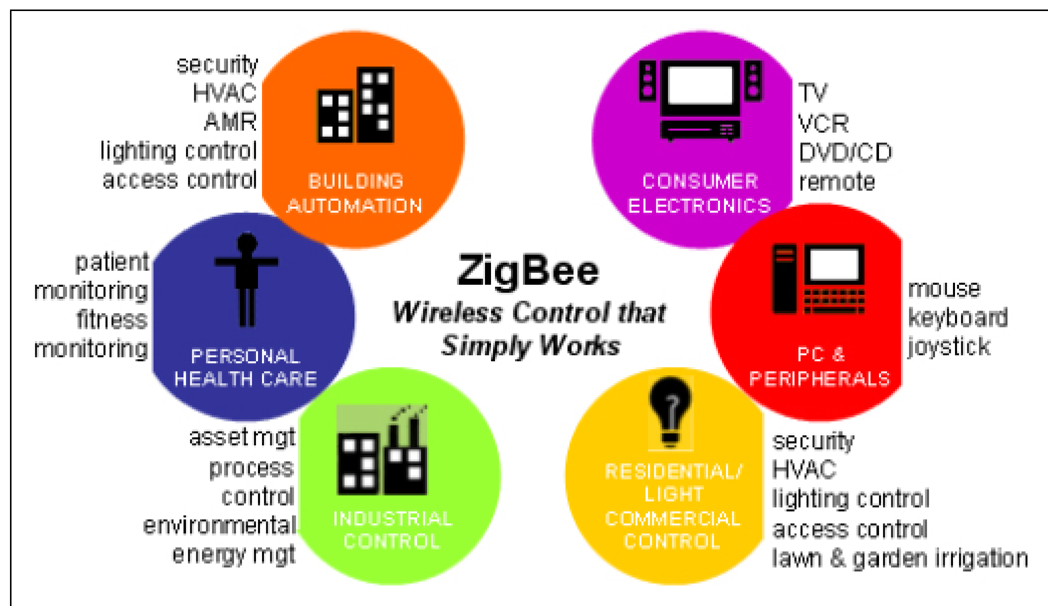


Figura 13: Aplicaciones Zigbee

iv. Problemas de las WSN

Es evidente que las tecnologías de redes de sensores nos ofrecen multitud de aplicaciones y utilidades, como veremos en el apartado siguiente, pero también conllevan un importante esfuerzo de producción eficiente tanto hardware como software.

Las características que tienen este novedoso tipo de redes requieren, como hemos visto, de protocolos y estrategias concretas a la hora de crearlas, tanto a nivel de red como a nivel de componentes individuales, ya sean los sensores con sus limitaciones de potencia o procesador, las pasarelas para enrutar de forma eficiente este tipo de redes o las estaciones base para evaluar y ejecutar respuestas de forma adecuada ante los datos que van recibiendo.

Podemos, por tanto, describir una serie de problemas que están en estudio para conseguir posibles soluciones o mejoras, tanto a nivel hardware como de programación:

1. **Optimización del consumo de energía en los nodos sensores.**
   Uno de los problemas más importantes es el consumo de energía de los nodos. Para lograr que éste sea mínimo y, por tanto, conseguir un máximo tiempo de vida de la red habrá que tener en cuenta:
   - La comunicación de mensajes es el primer consumidor de energía.
   - La CPU puede quedarse en un estado sleep de bajo consumo mientras no tenga que procesar ni enviar nada.
   - Economizar la distancia de las comunicaciones.
   - Técnicas de software: programación eficiente de líneas de código.
2. **Ancho de banda y cobertura de red limitados.**
   Se debe trabajar teniendo en cuenta que estas redes tienen una cobertura limitada, dadas las características de sus componentes, y creando sistemas que se ajusten adecuadamente.
3. **Los recursos de computación son limitados.**
   Los nodos tienen ciertas limitaciones tanto a nivel del procesador como la capacidad de almacenamiento de la memoria. Dependiendo del tipo de sensor,

hay diferencias, aunque gracias a las últimas novedades tecnológicas estas limitaciones se van reduciendo.

4. **Soluciones ad-hoc para redes ad-hoc.**
   Como hemos visto en el capítulo de protocolos, muchas de las soluciones ad-hoc no son válidas para este tipo de redes, debido a las diferencias existentes tanto en tecnología como en objetivos.
5. **La topología de red es muy dinámica.**
   Las WSN tienen como uno de sus objetivos crear redes móviles cuya topología va cambiando continuamente, redes caracterizadas por:
   - Nodos móviles.
   - Nodos con alta probabilidad de fallo.
   - Nodos que entran en el sistema.
   - Cuantos más nodos haya en la red mayor será el rendimiento.

IV. Aplicaciones de las WSN

La WSN puede consistir en muchos tipos diferentes de sensores, como pueden ser sísmicos, magnéticos, térmicos, acústicos, radar, IR, etc. Los distintos tipos de sensores existentes pueden monitorizar una gran variedad de condiciones ambientales, que incluyen:

- Temperatura, humedad, presión.
- Condiciones de luz, movimiento de vehículos, niveles de ruido.
- Composición del suelo.
- Presencia o ausencia de cierto tipo de objetos.
- Niveles de estrés mecánico en objetos (maquinaria, estructuras, etc.).
- Características de velocidad, dirección y tamaño de un objeto.

Los nodos sensores pueden adoptar diversas formas de trabajo, pueden actuar en modo continuo, por detección de eventos, por identificación de eventos, toma de datos localizados o como control local de actuadores.

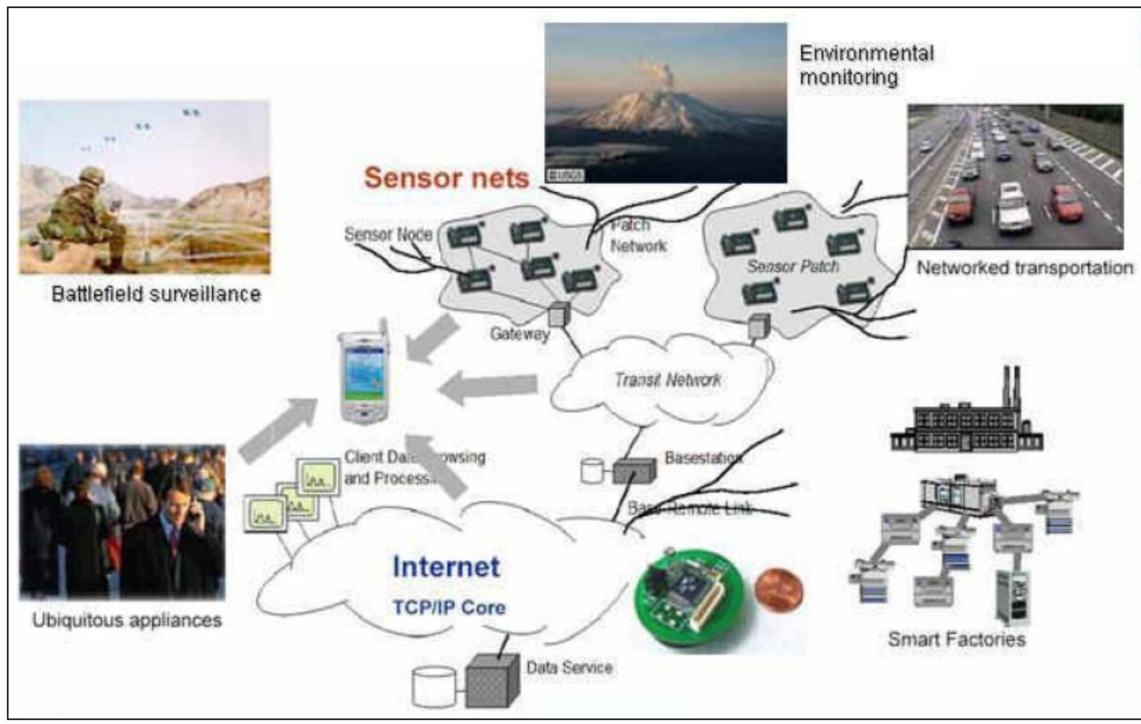


Figura 14: Aplicaciones de las WSN

Si tenemos el tipo de monitorización que van a realizar los sensores, podemos hacer una primera clasificación de aplicaciones, en tres tipos distintos que tendrían las siguientes propiedades:

Monitorización del entorno.

Este tipo de aplicaciones se caracterizan por un gran número de nodos sincronizados que estarán midiendo y transmitiendo periódicamente en entornos puede que inaccesibles, para detectar cambios y tendencias.

La topología es estable y no se requieren datos en tiempo real, sino para análisis futuros. Ejemplos: control de agricultura, microclimas, etc.

Monitorización de seguridad.

Son aplicaciones para detectar anomalías o atranques en entornos monitorizados continuamente por sensores. No están continuamente enviando datos, consumen menos y lo que importa es el estado del nodo y la latencia de la comunicación: se debe informar en tiempo real.

Como ejemplos tenemos el control de edificios inteligentes, detección de incendios, aplicaciones militares, seguridad, etc.

Tracking.

Aplicaciones para controlar objetos que están etiquetados con nodos sensores en una región determinada. La topología aquí va a ser muy dinámica debido al continuo movimiento de los nodos: se descubrirán nuevos nodos y se formarán nuevas topologías.

Figura 15: Tracking de animales

Redes híbridas

Son aquellas en las que los escenarios de aplicación reúnen aspectos de las tres categorías anteriores.

El concepto de microsensores comunicados de forma inalámbrica promete muchas nuevas áreas de aplicación. De momento las vamos a clasificar en: militares, entorno, salud, hogar y otras áreas comerciales. Por supuesto es posible ampliar esta clasificación.

i. Aplicaciones militares

Las WSNs pueden ser parte integral de sistemas militares C4ISRT (command, control, communications, computing, intelligence, surveillance, reconnaissance and targeting) que son los que llevan las órdenes, el control, comunicaciones, procesamiento, inteligencia, vigilancia, reconocimientos y objetivos militares.

El rápido y denso despliegue de las redes de sensores, su autoorganización y tolerancia a fallos las hace una buena solución para aplicaciones militares. Ofrecen una solución de bajo coste y fiable para éstas ya que la pérdida de un nodo no pone en riesgo el éxito de las operaciones.

Ejemplos de aplicación en esta área son:

Monitorización de fuerzas aliadas, equipamiento y munición. Cada equipo, tropa, vehículo o arma crítica lleva integrado un sensor para informar de su estado a líderes o niveles superiores. Se usan nodos recolectores donde se recopila toda la información para luego transmitirla a niveles superiores.

Reconocimiento del terreno y fuerzas enemigas. Se pueden desplegar y obtener información valiosa antes de que el enemigo los intercepte sobre rutas de acceso, posibles caminos e incluso movimientos del enemigo.

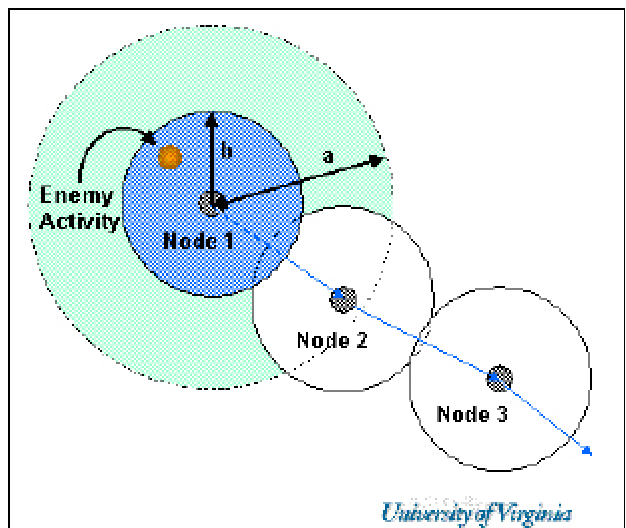


Figura 16: Reconocimiento del enemigo

Adquisición de blancos. Se pueden incorporar los nodos a sistemas de guiado de armas inteligentes.
Valoración de daños, antes o después de los ataques.
Reconocimiento de ataques nucleares, biológicos y químicos, desplegándolos en la región y usándolos como sistema de aviso. También para reconocimiento después de que ocurra sin tener que exponer a equipos de reconocimiento a radiaciones o agentes químicos.

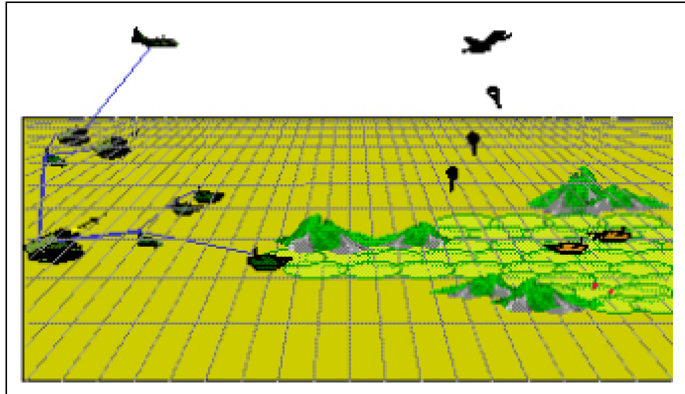


Figura 17: Despliegue de sensores para reconocimientos

ii. Aplicaciones medioambientales

En este campo tenemos aplicaciones como el seguimiento de aves, animales e insectos; monitorización de condiciones ambientales que afectan al ganado y las cosechas; irrigación; macroinstrumentos para la monitorización planetaria de gran escala; detección química o biológica; agricultura de precisión (monitorización de niveles de pesticidas, polución y erosión del terreno); detección de incendios; investigación meteorológica o geofísica; detección de inundaciones; mapeado de la biocomplejidad del entorno; y estudios de la polución.

Podemos destacar cuatro de estas aplicaciones:

Detección de fuego en bosques.

Se podrían desplegar millones de sensores de manera estratégica, aleatoria y densa en el bosque que informaran del origen exacto de un incendio antes de que se haga incontrolable. Los sensores podrían tener métodos de obtención de energía como placas solares, ya que serían abandonados durante meses o años sin mantenimiento.

Además, debido a la densidad de su despliegue serían capaces de cooperar para realizar una recolección de datos cooperativa y evitar obstáculos en las transmisiones, como árboles y rocas que pueden bloquear la línea de visión entre sensores.

Figura 18: Detección de incendios

Mapeado de la biocomplejidad del entorno medioambiental.

Requiere enfoques sofisticados para integrar información en escalas temporal y espacial. Avances tecnológicos en sensorización remota y recolección de datos automatizada han permitido una resolución espacial, espectral y temporal con un coste por unidad de área que se ha reducido en progresión geométrica. Además, la conexión de estos sistemas a Internet permite a usuarios remotos controlar, monitorizar y observar la biocomplejidad del medio ambiente.

Aunque los satélites y sensores aéreos son útiles en la observación a gran escala de la biodiversidad (p.e. complejidad espacial de especies de plantas dominantes), no tienen la precisión suficiente como para observar la biodiversidad a pequeña escala, que es la parte más importante en la biodiversidad de un ecosistema. Por ello, se necesita un despliegue de nodos sensores para observar la biocomplejidad.

Ejemplo: Reserva James en el sur de California con tres redes de monitorización, cada una con 25-100 sensores.

Detección de inundaciones.

Ejemplo: sistema ALERT desplegado en EEUU. Hay sensores de lluvia, nivel de agua y meteorológicos. Proporcionan información a una estación base centralizada. Hay proyectos de investigación que investigan enfoques distribuidos en interacción con los nodos sensores en campo para proporcionar consultas en momentos fijos (snapshot) o en espacios temporales largos (long-running queries).

Agricultura de precisión.

Beneficios obtenidos de monitorizar niveles de pesticidas en agua potable, niveles de erosión del suelo y niveles de polución del aire en tiempo real.

iii. Aplicaciones sanitarias

Proveer interfaces para los discapacitados; monitorización integral de pacientes; diagnósticos; administración de medicamentos en hospitales; monitorización de los movimientos y procesos internos de insectos u otros pequeños animales; tele-monitorización de datos fisiológicos humanos; y seguimiento y monitorización de pacientes en un hospital.

Tele-monitorización de datos fisiológicos humanos.

Los datos recolectados se pueden almacenar durante períodos largos de tiempo, usados para exploración médica. La WSN instalada puede monitorizar y detectar el comportamiento de personas mayores, como por ejemplo una caída. Los sensores, por su reducido tamaño, dan al usuario una mayor libertad de movimiento y permiten a los médicos identificar antes síntomas predefinidos.

Además, permiten una mayor calidad de vida a los usuarios comparados con los centros de tratamiento. En la facultad de medicina de Grenoble - Francia, se ha diseñado una Vivienda Inteligente para la Salud para validar la viabilidad de dichos sistemas.
Seguimiento y monitorización de médicos y pacientes.

Cada paciente lleva un sensor pequeño y ligero. Cada sensor tiene una tarea específica, por ejemplo monitorizar la presión arterial y la frecuencia cardiaca. Los médicos también llevan sensores, lo que permite a otros médicos localizarlos en el hospital.

Administración de medicación en hospitales.

Si colocamos un sensor en la medicación se reduce el riesgo de errores, porque el paciente también llevará un sensor que identifique sus alergias y medicación requerida.

iv. Aplicaciones del hogar: domótica

El objetivo de esta aplicación es programar y configurar sensores para adaptarlos a entornos del hogar, posibilitando aplicaciones que favorezcan los objetivos de la domótica.

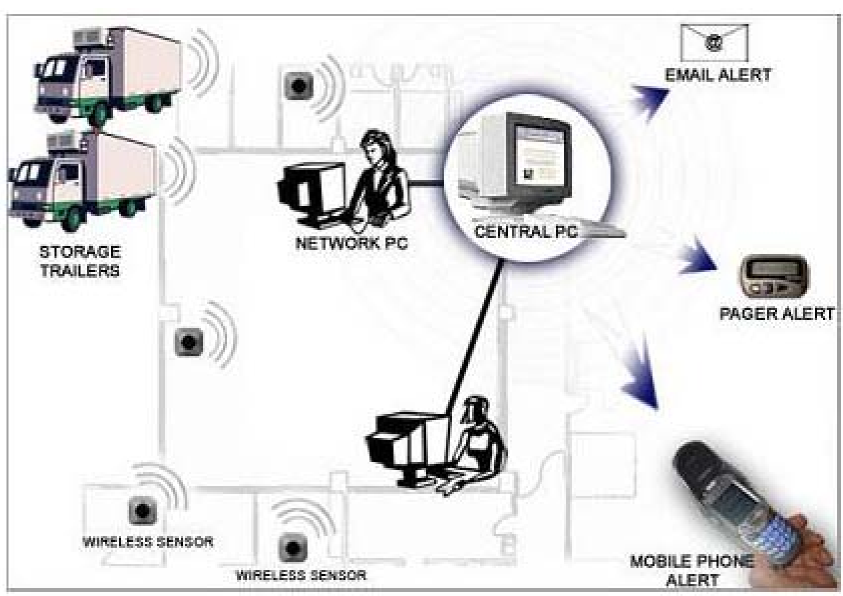


Figura 19: Aplicación domótica

Los nodos sensores pueden ser introducidos en aparatos domésticos como aspiradoras, microondas, hornos, frigoríficos y VCRs. Esto permite que sean manejados remotamente por los usuarios finales mediante una comunicación que se realizaría vía satélite o Internet.

A través de las redes de sensores pueden crear hogares inteligentes donde los nodos se integran en muebles y electrodomésticos. Los nodos dentro de una habitación se comunican entre ellos y con el servidor de la habitación. Estos servidores de habitaciones se comunican también entre ellos dando así conectividad entre distintas habitaciones.

Se crea lo que llamamos un entorno inteligente, cuyo diseño puede tener dos enfoques:

- Desde el punto de vista humano: el entorno se adapta a necesidades del usuario final en términos de capacidades de entrada-salida.
- Desde el punto de vista tecnológico: hay que desarrollar nuevas tecnologías hardware, soluciones de redes y servicios middleware.

Los sensores pueden integrarse en muebles y aparatos. Se comunican entre ellos y con servidores o actuadores de habitación, que a su vez se comunican con otros servidores de habitación. Todos ellos se integran y organizan con los dispositivos integrados existentes para autoorganizarse, autorregularse y autoadaptarse basándose en modelos de control.

v. Otras aplicaciones comerciales

Otras aplicaciones comerciales son la monitorización de la fatiga de materiales; teclados virtuales de edificios; gestión de inventario; monitorización de la calidad de productos; construcción de oficinas inteligentes; control ambiental en edificios de oficinas; control de robots y guiado en entornos de fabricación automática; juguetes interactivos; museos interactivos; control y automatización de procesos; monitorización de desastres; estructuras inteligentes; máquinas de diagnóstico; transporte; instrumentación de fábricas; control local de actuadores; detección y monitorizado de robo de coches; seguimiento y detección de vehículos; e instrumentación de cámaras de procesado de semiconductores, maquinaría giratoria, túneles de viento y cámaras anecoicas. Algunas de estas aplicaciones son:

Control ambiental en edificios de oficinas.

Normalmente la calefacción o el aire acondicionado se controlan desde una central, por lo que la temperatura dentro de la habitación puede variar unos pocos grados debido a que la distribución de aire no está uniformemente distribuida y el control es central. Se puede desplegar una WSN para controlar el flujo de aire y la temperatura en diferentes partes de la habitación. Con esta tecnología se reduciría el consumo ahorrando dos cuatrillones de BTUs (British Termal Unit), que supondría 55 millones de $ en US, reduciendo en 35 millones de toneladas las emisiones de dióxido de carbono.

Museos interactivos.

Para que los niños interactúen con objetos y experimentos, y reaccionen al toque o al habla. También permitiría el aviso y localización de personas dentro del recinto. (Ejemplo: exploratorium en museo de San Francisco).

Gestión de inventarios.

En el almacén, cada artículo lleva pegado un sensor, de modo que se puede conocer en todo momento su localización y número de artículos por categoría. Para dar de alta nuevos artículos se les pega el sensor y se llevan al almacén.

V. Nodos sensores

Las redes de sensores inalámbricas, como hemos visto, se componen de diversos elementos formando el conjunto total de la red en sí, como son las pasarelas, las estaciones base, la tecnología inalámbrica, etc. Pero los principales elementos que crean y sobre los que se sustentan estos novedosos sistemas son los nodos sensores. Todo gira entorno a estos nuevos dispositivos, que constituyen la pieza central de las redes de sensores. El DSM creado en este proyecto se centra únicamente en el ámbito del nodo sensor obviando conceptos como la red o un conjunto de sensores.

Por lo tanto, dedicaremos este apartado a examinar sus principales características y los distintos sensores que están apareciendo en el mercado, así como los que hemos utilizado en nuestro proyecto para generar el código correspondiente.

Un mote, también conocido como nodo sensor (la traducción literal sería mota de polvo), es un dispositivo capaz de observar y tomar medidas de un fenómeno objeto. Es lo que se denomina en inglés como sensing.

El mote combina capacidades de recolección, procesado y transmisión de datos en un mismo dispositivo, logrando todo esto con un reducido coste económico, tamaño y consumo de potencia.

Los sensores que lleva incorporado el nodo pueden ser de diferentes tipos: presión, humedad, temperatura, movimiento, etc. dando lugar a las distintas aplicaciones posibles que comentamos en el apartado anterior.

Figura 20: Nodos sensores

De esta forma, podemos establecer una serie de características generales que se dan para los nodos sensores:

1. Integran sensores para realizar mediciones. éstos pueden ser de luz, temperatura, presión, humedad, etc.
2. Están limitados en diferentes aspectos:

- Energía. Ya que suelen estar alimentados por medio de baterías y el bajo consumo es una de sus prioridades.
- Capacidad de cómputo y memoria. Aún no disponen de grandes capacidades de procesador y de almacenamiento, aunque este campo está desarrollándose y ampliándose cada día.
- Memoria. La capacidad de almacenamiento también es limitada.

3. Hacen un uso intensivo de la CPU para el procesamiento, y de la Radio para enviar y recibir mensajes. De hecho, ésta es una de las bases de esta tecnología.
4. Los sensores son de bajo coste. (1$ en el 2005).
5. Alta probabilidad de fallo, teniendo en cuenta las condiciones a las que se exponen los sensores. Deben tener costes de producción muy bajos y ser desechables.
6. Son autónomos y operan de forma independiente.
7. Se adaptan al entorno.

Por otro lado, podemos establecer una serie de factores que evalúan y catalogan los tipos y características de los nodos sensores. Estos factores serían: Energía, flexibilidad, robustez, seguridad, comunicación, computación, sincronización, tamaño y coste.

Un nodo sensor está formado por cuatro componentes básicos:

1. Unidad sensora. Sensores y conversores analógico-digitales que convierten las señales analógicas en digitales para el microprocesador.
2. Procesador. Normalmente asociado a una pequeña unidad de almacenamiento. Gestiona los procesos que permiten al nodo sensor colaborar con otros para realizar las tareas asignadas.
3. Transceptor. Conecta el nodo a la red, realizando las operaciones de trasmisión y recepción de mensajes.
4. Alimentación. Uno de los componentes más importantes, se obtiene a partir de baterías aunque puede estar ayudado de un generador, como placas solares que obtienen energía del entorno.

Todo esto tiene que caber en un módulo del tamaño de una caja de cerillas y, a veces, en un tamaño de un centímetro cúbico para poder suspenderlo en el aire.

	Wins NG 2.0	iPAQ	Berkeley MICA Mote	Smart Dust
Parts cost	$100s	$100s	$10s	<$1
SIZE (cm^3)	5300	600	40	.002
Weight (g) including battery	5400	350	70	.002
Battery Capactity (Kj)	300	35	15	(Less)
Sensors	Off-board	Microphone & light sensors	Integrated on PCB: acceleration, temperature, light, sound	MEMS sensors to be integrated
Memory	32 MB RAM 32 MB flash	64 MB RAM 32 MB flash	4 KB RAM 128 KB flash	(Less)
CPU	Hitachi SH4	StrongARM or Xscale	Atmega 103L	(Less powerful)
Operating System	Linux	WinCE or Linux	TinyOS	(smaller)
Processing capability	400 MIPS/1.4 GFLOPS	240 MIPS	4 MIPS	(Less)
Radio Range	100 m.	100 m.	30 m.	(Shorter)

Figura 21: Comparación plataformas para nodos

Aunque cada vez hay procesadores más pequeños y rápidos, las unidades de procesamiento y almacenamiento en los nodos son recursos escasos. Por ejemplo, Smart Dust mote lleva un Atmel AVR8535 de 4MHz con memoria flash de instrucciones de 8K y 512 bytes de RAM y 512 bytes de EEPROM. El sistema operativo TinyOS que usan ocupa 3500 bytes de memoria quedando 5400 para código. Otro caso: uAMPS tiene un microprocesador SA-1110 de 59-206 MHz y ejecuta un sistema operativo multihilo.

Los transceptores pueden ser dispositivos ópticos activos o pasivos (como en smart dust motes), o radiofrecuencia. Radiofrecuencia requiere modulación, filtro pasa banda, filtrado, desmodulación y circuito multiplexor, lo que los hace complejos y caros. Además, las pérdidas en la transmisión pueden llegar a ser mayores que la distancia a la cuarta, porque están muy cerca del suelo.

Aún así, se prefieren transceptores RF en la mayoría de proyectos de investigación porque los paquetes transportados son pequeños y las tasa de transferencia pequeñas (generalmente menores a 1Hz), y la reutilización de frecuencias es alta debido a las cortas distancias de comunicación.

Por ello se puede usar electrónica de radio de bajo ciclo de trabajo. Aún así, el diseño de una electrónica de comunicaciones energéticamente eficiente es todavía un reto. Por ejemplo, Bluetooth consume demasiada energía sólo en encendidos y apagados.

Como muchas veces son inaccesibles, la vida del sensor depende de la fuente de energía, que además es escasa por las limitaciones en el tamaño. En una Smart Dust mote se dispone de 1J de capacidad de batería. En WINS (Wireless Integrated Network Sensors) el consumo medio es de 30uA para proporcionar tiempos de vida largos. Se puede ampliar la vida con recolección de energía, en muchos casos con células solares.

En algunas aplicaciones los nodos pueden llevar componentes adicionales: sistema de localización (GPS), generador de energía y movilizador. Muchas tareas requieren conocer la posición porque los sensores se despliegan de manera aleatoria y se dejan desatendidos. Además se necesita en muchos protocolos. A menudo, se supone que los nodos tienen GPS con precisión de menos de 5m. Equipar a todos los nodos con GPS es inviable para WSNs, por ello, un enfoque alternativo es dotar a unos nodos con GPS que ayudan a los demás a determinar su posición.

Optimización del consumo de energía

El consumo de energía de los nodos sensores, como ya hemos comentado, es uno de las principales limitaciones que tienen estos dispositivos. Por ello, se han realizado ciertas estrategias hardware y software para conseguir un ahorro de energía, de modo que el tiempo y la duración del mote en la red con suficiente energía sea el máximo posible.

Un nodo sensor tiene tres estados de funcionamiento:

1. Sleep.

Estado en el que el mote está durmiendo o inactivo. Se pretende que esté la mayor parte del tiempo posible en este estado y que su consumo sea el mínimo.

2. Wakeup.

Estado de cambio, en el que el nodo se despierta y va a pasar a un estado activo. Se produce cuando el sensor recibe algún cambio, estímulo o interrupción programada dentro de sus funciones de detección y análisis. Uno de los objetivos es que el tiempo de wakeup sea mínimo para pasar rápidamente al estado de trabajo.

3. Active.

Es el estado activo del mote, donde está realizando el trabajo de adquisición, procesado y trasmisión de datos. Por supuesto, este tiempo debe ser mínimo para volver cuanto antes al estado sleep, ya que el consumo será el mayor de los tres que se dan en cada fase.

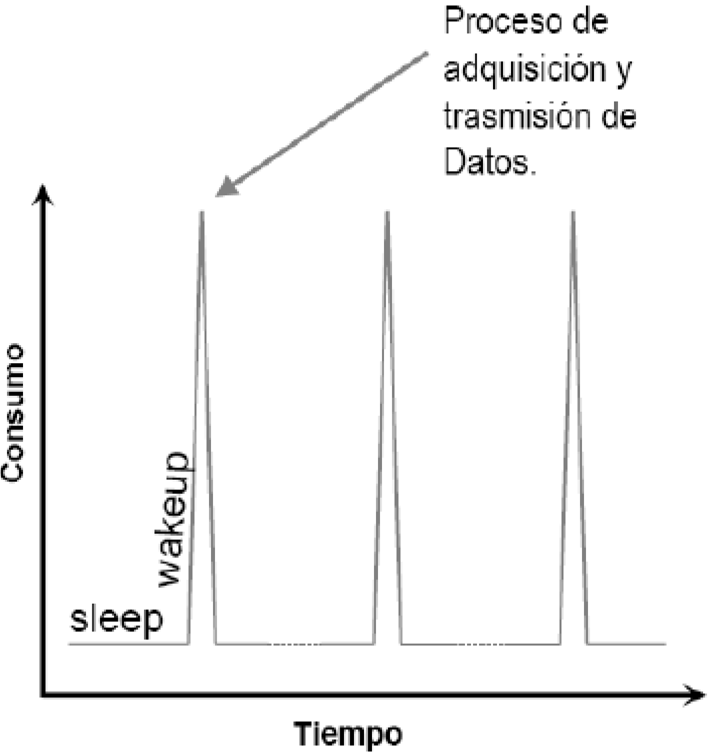


Figura 22: Estados de un nodo sensor

A continuación, podemos ver en una serie de tablas las cantidades de energía que se consumen tanto para la recepción como la transmisión de mensajes. Se puede apreciar cómo además de las operaciones de radio, otras como la modulación o el manejador radio también consumen su parte de energía y utilizan parte el procesador.

Por supuesto, la recepción y trasmisión radio hacen que los niveles de consumo de energía puedan llegar hasta los 4uJ en el caso de la transmisión, por lo que un factor importante para reducir estos consumos sería reducir el tiempo de estado activo del mote, como sabemos.

1) RECEPCIÓN

Componets	Packet reception work breakdown	Percent CPU utilization	Energy (nJ/bit)
AM	0.05%	0.02%	0.33
Packet	1.12%	0.51%	4.58
Radio Handler	26.87%	12.16%	182.38
Radio decode thread	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Toal	100.00%	100.00%	2028.66

2) TRANSMISIÓN

Componets	Packet transmission work breakdown	Percent CPU utilization	Energy (nJ/bit)
AM	0.03%	0.01%	0.18
Packet	3.33%	1.59%	23.89
Radio Handler	35.32%	16.90%	253.55
Radio decode thread	4.53%	2.17%	32.52
RFM	56.80%	27.18%	407.17
Radio Transmission	-	-	1800
Idle	-	52.14%	-
Toal	100.00%	100.00%	4317.89

Figura 23: Distribución del consumo de energía

VI. Ejemplos de motes: Micas y Telos

En este apartado, vamos a describir un tipo de motes en concreto: los Micas y Telos, ya que son los que han sido objeto de investigación en nuestro proyecto. Realizaremos un breve resumen de características y aplicaciones posibles de ambos tipos, viendo ventajas de cada uno y la evolución que han sufrido con los nuevos avances.

En los últimos años las WSN han evolucionado mucho, principalmente por la creación de estos nuevos motes. La Universidad de Berkeley se puede considerar la principal responsable de este avance, ya que fue allí donde se desarrolló el primer mote. Varios años después, junto con Intel Research Laboratory Berkeley, han diseñado nuevos motes, los MICA2 y los MICA2DOT, que son los más usados para investigación en redes de sensores.

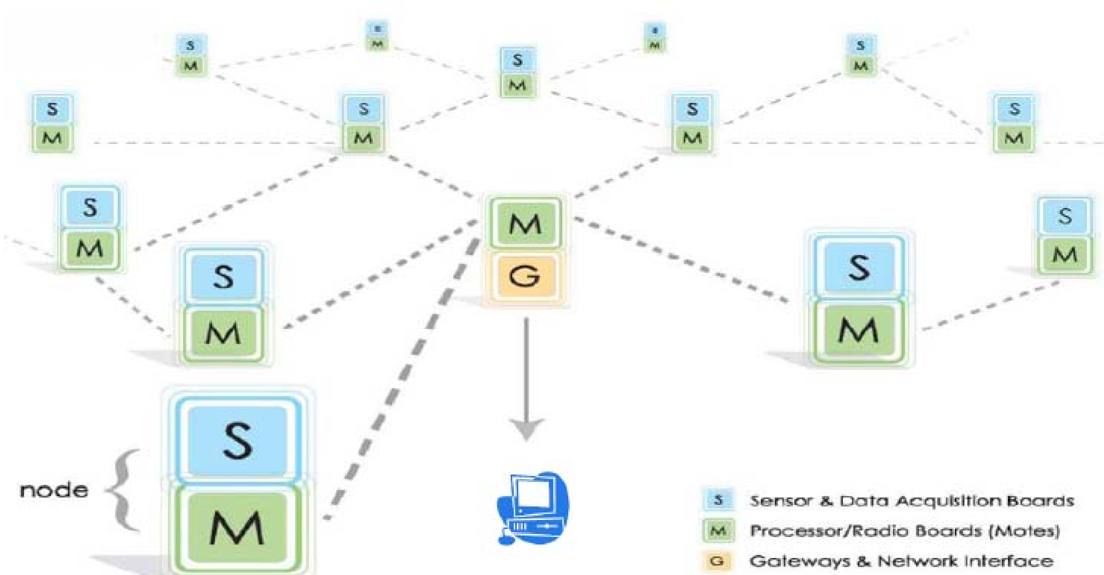


Figura 24: Estructura de red y motes

Dentro de una arquitectura WSN, los nodos sensores se diferencian en dos partes: MPR, placa del procesador y radio y MTS, placa de sensores o también puede que lleve adquisición de datos. En el caso de los Micas son placas que pueden ir por separado y unirlas mediante pines de conexión, mientras que en el caso de los Telos, está todo integrado en el mismo mote, teniendo una serie de sensores concretos, dependiendo del tipo de Telos.

i. Micas

Dentro de la familia de los micas, podemos encontrar varios tipos. Veremos las características de los Micaz, Mica2 y Mica2Dot. En la tabla de características de la siguiente página podemos apreciar cómo ha evolucionado el tipo de procesador o, por otro lado, cómo se ha pasado a trabajar en frecuencias de 2.4GHz con el Micaz

comparado con la banda de los 433MHz o 915MHz de los otros modelos. La tasa de datos es mucho menor en los de menor frecuencia, pero no es un inconveniente para nuestros objetivos.

Micaz

Los Micaz son una de las últimas generaciones de motes que trabaja en la banda de frecuencias de 2400 MHz a 2483.5 MHz. El MPR2400 (Micaz) usa el Chipcon CC2420, bajo la norma protocolo IEEE 802.15.4, un transmisor integrado ZigBee y un microcontrolador Atmega128L.

Figura 25: Micaz

Usa, al igual que el Mica2, 51 pines de conexión I/O y una memoria flash. Además, sus aplicaciones son compatibles. A continuación, vemos el diagrama de bloques.

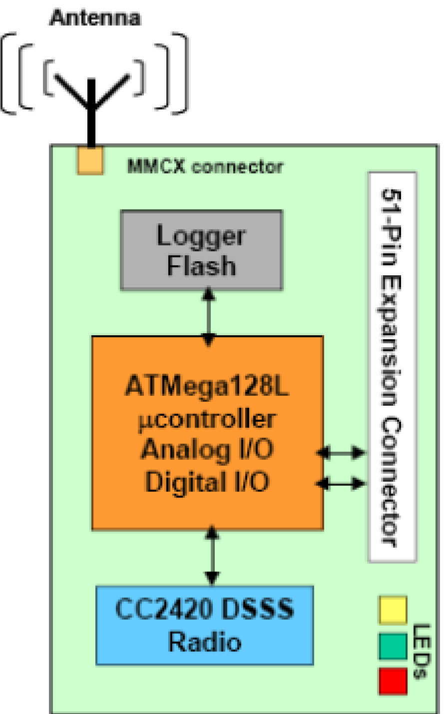


Figura 26: Diagrama de bloques Micaz

En esta tabla se puede ver una comparativa de las características de los Micas:

Mote Hardware Platform		MICAz	MICA2	MICA2DOT	MICA
Models (as of April 2005)		MPR2400	MPR400/410/420	MPR500/510/520	MPR300/310
MCU	**Chip**	ATMega128L			ATMega103L
	Type	7.37 MHz, 8 bit		4 MHz, 8 bit	4 MHz, 8 bit
	Program Memory (kB)	128			
	SRAM (kB)	4			
Sensor Board Interface	**Type**	51 pin		18 pin	51 pin
	10-Bit ADC	7, 0 V to 3 V input		6, 0 V to 3 V input	7, 0 V to 3 V input
	UART	2		1	2
	Other interfaces	DIO, I2C		DIO	DIO, I2C
RF Transceiver (Radio)	Chip	CC2420	CC1000		TR1000
	Radio Frequency (MHz)	2400	315/433/915		433/915
	Max. Data Rate (kbits/sec)	250	38.4		40
	Antenna Connector	MMCX		PCB solder hole	
Flash Data Logger Memory	Chip	AT45DB014B			
	Connection Type	SPI			
	Size (kB)	512			
Default power source	Type	AA, 2×		Coin (CR2354)	AA, 2×
	Typical capacity (mA-hr)	2000		560	2000
	3.3 V booster	N/A			✓

Figura 27: Características técnicas de los Mica2

Mica2

Los motes Mica2 son los módulos de tercera generación de motes que se usan para redes de sensores inalámbricas de baja potencia. Mejoran bastante las características del Mica original:

- Diseñado específicamente para redes de sensores integradas.
- Distintas frecuencias de transmisión con amplio rango.
- Más de un año de batería mediante los modos sleep.
- Soportan reprogramación inalámbrica a distancia.
- Amplia variedad de placas de sensores compatibles: luz, temperatura, presión, aceleración, acústica, detectores magnéticos, etc.
- Las distintas aplicaciones en las que se utilizan estos motes son, principalmente, las WSN, la seguridad y vigilancia, la monitorización ambiental o las redes inalámbricas de gran escala.

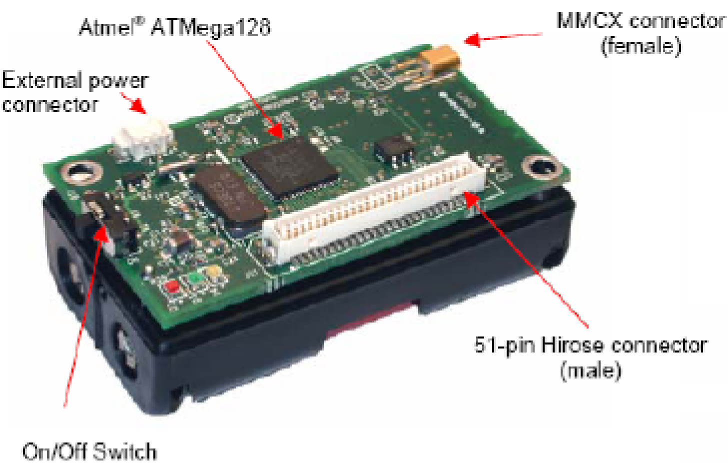


Figura 28: Mica2

Los Mica2 se dividen en tres modelos dependiendo de su frecuencia de uso: MPR400 (915MHz), MPR410 (433MHz), y MPR420 (315MHz). En nuestros laboratorios disponemos de varios de estos motes, concretamente los MPR400, que trabajan a 915MHz.

Estos motes usan el Chipcon CC1000, con radio modulada en FSK. Todos los modelos usan un potente microcontrolador Atmega128L y una radio de frecuencia sintonizable en un rango amplio. Tanto los MPR4x0 como los MPR5x0 son compatibles y pueden comunicarse entre ellos.

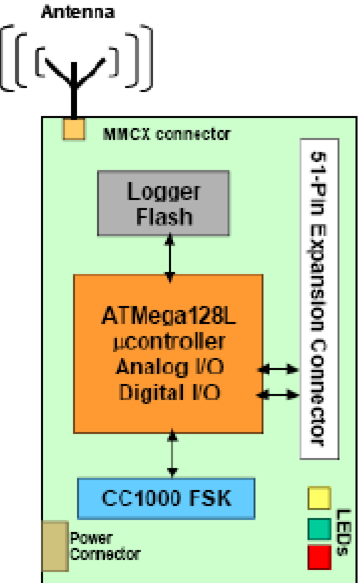


Figura 29: Diagrama de bloques Mica2

Mica2Dot

Los Mica2Dot son un tipo de motes diseñados especialmente para aplicaciones donde el tamaño físico es fundamental. Al igual que los Mica2 hay tres modelos dependiendo de su frecuencia: MPR500 (915MHz), MPR510 (433MHz), y MPR520 (315MHz). El resto de características son similares a las de los Mica2, lo más

importante es la forma física y el reducido tamaño que poseen, tal y como podemos ver en las imágenes.

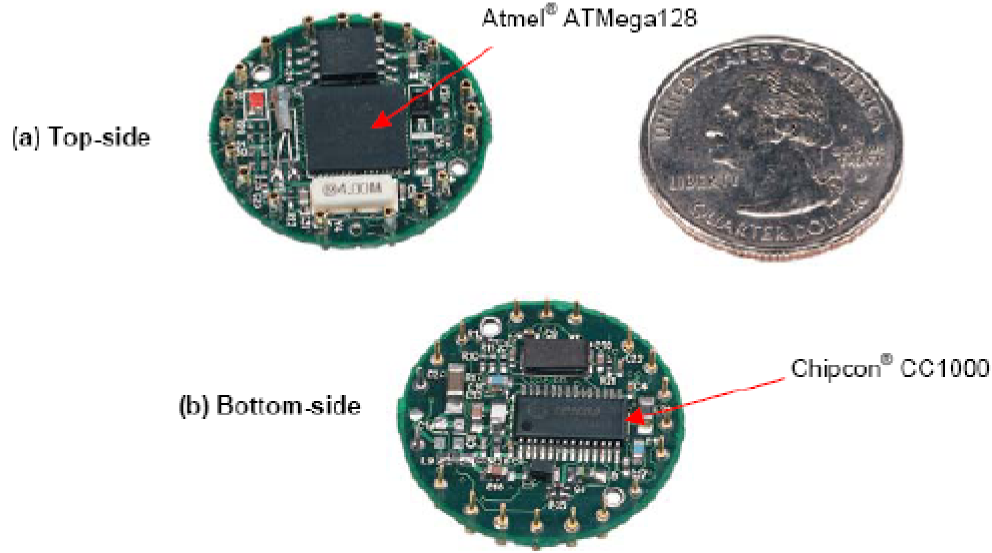


Figura 30: Mica2Dot

Su correspondiente diagrama de bloques lo podemos ver a continuación, apreciando que los pines se han colocado de forma periférica:

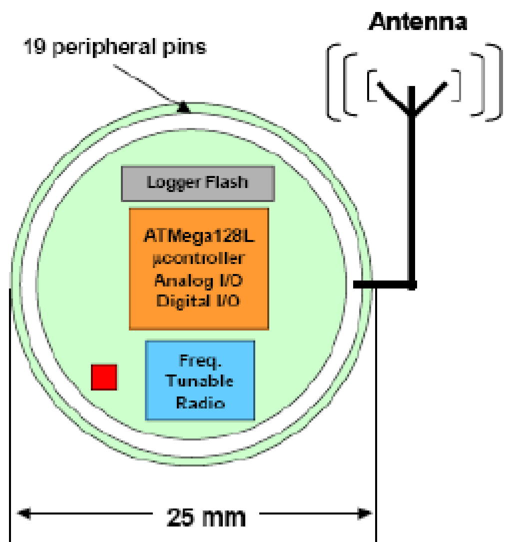


Figura 31: Diagrama de bloques Mica2dot

Sensores para Micas

Las series de placas de sensores MTS o de adquisición de datos MDA han sido diseñadas para la familia de los Micas. Hay una gran variedad de sensores que permiten un amplio margen de modalidades diferentes. La tabla de la siguiente página muestra los sensores disponibles actualmente para cada tipo de mote.

A modo de ejemplo, presentamos el modelo con el que hemos realizado algunas pruebas de aplicaciones junto con los Mica2 de que disponemos en el laboratorio.

La placa sensor es el modelo MTS300CA. Este tipo de placa lleva incorporada sensores de luz y temperatura. Además llevan un micrófono para detectar sonidos, y un resonador piezoeléctrico, lo que se denomina un buzzer, para producir un sonido a una frecuencia determinada.

Figura 32: Sensores MTS300

El modelo MTS310CA incluye también acelerómetro y magnetómetro. Con este tipo de sensores la variedad de aplicaciones posibles aumenta, incluyendo detección de vehículos, detección de seísmos de baja intensidad, movimiento, rangos acústicos, robótica, etc.

Crossbow Part Name	Motes Supported	Sensors and Features
MTS101CA	MICA, MICA2	Light, temperature, prototyping area
MTS300CA	MICA, MICA2	Light, temperature, microphone, and buzzer
MTS310CA	MICA, MICA2	Light, temperature, microphone, buzzer, 2-axis accelerometer, and 2-axis magnetometer
MTS400CA	MICA2	Ambient light, relative humidity, temperature, 2-axis accelerometer, and barometric pressure
MTS420CA	MICA2	Same as MTS400CA plus a GPS module
MTS510CA	MICA2DOT	Light, microphone, and 2-axis accelerometer
MDA300CA	MICA2	Light, relative humidity, general purpose interface for external sensors
MDA500CA	MICA2DOT	Prototyping area

Figura 33: Tipos de sensores para Micas

ii. Telos

Los TelosB (TPR2400) son los motes que más importancia tienen para nosotros, ya que hemos desarrollado la mayor parte del proyecto con ellos, programando las aplicaciones en varios de estos dispositivos.

Este tipo de motes reúne todo lo esencial para estudios de laboratorio en una plataforma simple, incluyendo la capacidad de programación por USB, una antena

integrada con sistema radio IEEE 802.15.4, un procesador de bajo consumo con una memoria extendida y un conjunto de sensores, concretamente en el modelo TPR2420.

Figura 34: TelosB

Las características generales de los TelosB son:

- Transmisión RF de acuerdo con la norma IEEE 802.15.4/ZigBee.
- Banda de frecuencias desde 2.4 a 2.4835 GHz, compatible con ISM.
- Velocidad de transferencia de datos de 250kbps.
- Antena integrada.

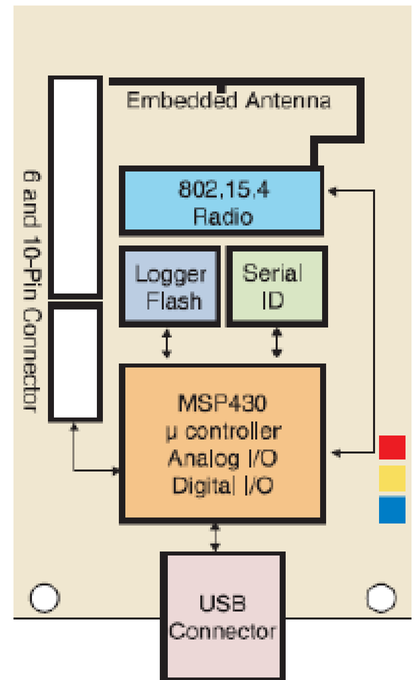


Figura 35: Diagrama de bloques del TelosB

- Micro-controlador MSP430 a 8MHz con 10kB de RAM.
- Bajo consumo.
- Flash externa de 1Mb para almacenamiento de datos.
- Programación y toma de datos vía USB.
- Redes de sensores inalámbricas
- Conjunto de sensores de luz, temperatura y humedad.

- Soporta TinyOS para implementación y comunicación de redes.

Esta plataforma consigue un bajo consumo de potencia permitiendo una larga vida a las baterías además de tener un tiempo mínimo en el estado de wakeup, otro de los objetivos dentro de las estrategias de bajo consumo.

Los TelosB se alimentan de dos baterías AA, aunque si es conectado mediante el puerto USB para programación o comunicación, la alimentación la proporciona el ordenador. También se proporciona la capacidad de añadir dispositivos adicionales. Los dos conectores de expansión de los que dispone pueden ser configurados para controlar sensores analógicos, periféricos digitales y displays LCD.

Con todas estas características, el mote TelosB no sólo proporciona más facilidad para programación, más flexibilidad y más prestaciones, sino que es el que menos consumo de potencia ofrece, muy por debajo de los que había hasta ahora, permitiendo alargar la vida de los nodos considerablemente y siendo ésta su principal baza frente a los otros dispositivos.

iii. Resumen comparativo

Las diferentes plataformas de motes que hemos visto se diferencian en sus características hardware lo que les proporciona distintas cualidades a unas de otras: mayor procesador, frecuencias de transmisión, velocidad de transmisión de datos, consumo de energía, etc.

En la siguiente tabla podemos ver la evolución que han sufrido los distintos prototipos y, lo que es más importante para nosotros, la comparación con las prestaciones que tienen y ofrecen los TelosB.

Mote Type	*WeC*	*René*	*René 2*	*Dot*	*Mica*	*Mica2Dot*	*Mica 2*	*Telos*
Year	1998	1999	2000	2000	2001	2002	2002	2004
Microcontroller								
Type	AT90LS8535		ATmega163		ATmega128			TI MSP430
Program memory (KB)	8		16		128			60
RAM (KB)	0.5		1		4			2
Active Power (mW)	15		15		8		33	3
Sleep Power (μW)	45		45		75		75	6
Wakeup Time (μs)	1000		36		180		180	6
Nonvolatile storage								
Chip	24LC256				AT45DB041B			ST M24M01S
Connection type	I^2C				SPI			I^2C
Size (KB)	32				512			128
Communication								
Radio	TR1000				TR1000	CC1000		CC2420
Data rate (kbps)	10				40	38.4		250
Modulation type	OOK				ASK	FSK		O-QPSK
Receive Power (mW)	9				12	29		38
Transmit Power at 0dBm (mW)	36				36	42		35
Power Consumption								
Minimum Operation (V)	2.7		2.7		2.7			1.8
Total Active Power (mW)	24				27	44	89	41
Programming and Sensor Interface								
Expansion	none	51-pin	51-pin	none	51-pin	19-pin	51-pin	10-pin
Communication	IEEE 1284 (programming) and RS232 (requires additional hardware)							USB
Integrated Sensors	no	no	no	yes	no	no	no	yes

Figura 36: Evolución de los motes

Como ya hemos comentado anteriormente, se puede apreciar que los consumos de potencia del procesador son muy inferiores en comparación con los otros motes, la velocidad de transferencia es bastante superior y facilidades como la conexión USB

para la programación o el llevar la antena integrada, hacen de este prototipo uno de los mejores.

En la figura siguiente, observamos más concretamente los tiempos de cambio de estado, así como la potencia consumida en cada estado, obteniendo siempre los mínimos resultados para los Telos, y consiguiendo un mayor tiempo de vida del nodo.

MICA2	MICAZ	TELOS
– 0.2 ms wakeup	– 0.2 ms wakeup	– 0.006 ms wakeup
– 30 mW sleep	– 30 mW sleep	– 2 mW sleep
– 33 mW active	– 33 mW active	– 3 mW active
– 21 mW radio	– 45 mW radio	– 45 mW radio
– 19 kbps	– 250 kbps	– 250 kbps
– 2.5V min	– 2.5V min	– 1.8V min

Nodos enviando un mensaje de sincronización cada 3 minutos y con dos baterías AA:

453 días	328 días	945 días

Figura 37: Comparativa de tiempos y consumos

Conclusiones

Los Telos son los motes con el menor consumo de potencia hasta la fecha. Incluyen numerosas mejoras que permiten investigar en las WSN mientras que los dispositivos van siendo más fáciles de usar y más baratos en lo que al coste se refiere.

Otras características, como la protección hardware contra escritura y la estabilidad de la señal radio, concretan aún más las actuales investigaciones. Los investigadores pueden experimentar con el nuevo estándar IEEE 802.15.4 y usar trabajos existentes con TinyOS. Una flexibilidad adicional permite que el software pueda configurar o deshabilitar módulos hardware.

Los TelosB son módulos robustos con unas grandes prestaciones de bajo consumo de potencia que no existían en diseños anteriores.

VII. TinyOS

Las redes de sensores inalámbricas están basadas en nodos de pequeño tamaño que tienen ciertas limitaciones tanto de memoria como de procesador. Además, sufren también el importante inconveniente del consumo de potencia, por lo que toda mejora tanto en hardware como software tiene que ir enfocada hacia esos objetivos que incrementen las prestaciones de estos dispositivos.

El sistema operativo que lleva a cabo estas estrategias de software y, por tanto, se ha convertido en la base de la programación de los motes, es el TinyOS. TinyOS (Tiny Micro-Threading Operating System) se trata de un pequeño sistema operativo dirigido por eventos, que ha sido desarrollado por la Universidad de Berkeley y ofrece un marco para el desarrollo de aplicaciones orientado a componentes.

Su lenguaje de programación, nesC, es un extensión de C que permite la definición de componentes y la interconexión de ellos. No tiene gestión de procesos, en su lugar tiene dos hilos de ejecución, uno destinado a la ejecución de eventos y otro a la de tareas, entre los cuales el cambio de contexto es muy rápido.

Es muy ligero y puede ser ejecutado en dispositivos con prestaciones limitadas pues, tanto el espacio que ocupa en memoria flash, como la RAM que consume, son muy pequeños. Por tanto, es ideal para el tipo de dispositivos como los motes.

La creación de nuevos componentes hardware, tanto motes como placas de sensores, es posible. Basta con crear un fichero de cabecera en el que se describa el conexionado del hardware y la creación de componentes software que, a través de llamadas a las librerías C del microcontrolador o a través de macros, realicen la comunicación con el hardware.

La gestión de potencia corre a cargo de TinyOS, que se encarga de desactivar los recursos hardware que no están siendo usados. Permite un modo de bajo consumo de potencia mientras no se están transmitiendo o recibiendo datos así como también establecer periodos en los que, cíclicamente, la radio se apaga. Se crean los estados de actividad o de latencia que comentábamos anteriormente, y se gestionan los tiempos de paso de un estado a otro, tanto de wakeup, como la vuelta al estado sleep.

En cuanto a las comunicaciones, permite tanto comunicación broadcast a todos los nodos como encaminamiento multi-salto. En este segundo caso la aplicación es libre de elegir el algoritmo de encaminamiento que desee pero siguiendo unas normas. Todos los algoritmos de encaminamiento que se desarrollen, deben cumplir con una arquitectura de componentes que define el sistema operativo para que las aplicaciones puedan fácilmente cambiar el algoritmo que usan por otro.

Existen otras características dignas de mencionar sobre TinyOS, como son la reprogramación en red de todo el código de un mote, o el uso de componentes para cifrado de datos (TinySec). Estas características y el hecho de que estuviera disponible bastante antes que otros sistemas operativos, desde principios de 2003, han hecho que sea el sistema operativo más extendido para redes de sensores.

Sin embargo, TinyOS tiene algunos inconvenientes. Puesto que sólo hay un hilo de ejecución para tareas y el planificador de estas es una cola FIFO, no se tienen garantías de tiempo real. Este hilo de ejecución para tareas no ofrece gestión de prioridades, lo único que TinyOS hace al respecto es permitir que los nuevos eventos desalojen a las tareas que puedan ejecutarse.

El entorno de programación que ofrece puede resultar muy complicado para programadores novatos, que deben tratar con cuestiones de programación asíncrona y temporización.

Además hay algunas características de las que este sistema operativo no dispone, como mecanismos fiables de sincronización de datos entre nodos o protección de memoria, si bien el carecer de esta última le hace requerir menos recursos para su ejecución.

Tampoco proporciona conectividad con grandes infraestructuras de servicios como Internet. Aunque esto último es un problema menor, ya que es posible el uso de aplicaciones que sirvan de puente entre las redes de sensores y otras redes como las basadas en TCP/IP. La reprogramación en red sí es posible, pero, como se comentó anteriormente, se debe reprogramar todo el código que se ejecuta en el mote.

TinOS ha sido incorporado a decenas de plataformas y numerosas placas de sensores. Una amplia comunidad lo utiliza en simulaciones para desarrollar y probar varios protocolos y algoritmos. Hasta el momento, hay unas 10000 descargas y 500 grupos de investigación y compañías están usando TinyOS en los motes de Berkeley/Crossbow. Numerosos grupos contribuyen activamente con el código en su sitio web sourceforge y trabajan juntos para establecer servicios de red estándar, formados desde una base de experiencia directa y perfeccionados a través de análisis competitivos en un entorno abierto.

Proyectos TinyOS

Presentamos una serie de ejemplos de proyectos con TinyOS que se están realizando actualmente en la Universidad de Berkeley:

- Calamari: Soluciones de localización para redes de sensores.
- FPS: Un protocolo de red para programación de la potencia de radio en WSN.
- Great Duck Island: "Nuestro objetivo es permitir a los investigadores de cualquier parte del mundo dedicarse a la monitorización no-intrusiva de la vida salvaje y sus habitantes. Los motes sensores están monitorizando el hábitat de la isla y transmitiendo sus lecturas por satélite, lo que permite a los investigadores descargar los datos del entorno en tiempo real a través de Internet."
- Maté: Máquinas virtuales de aplicación específica para redes TOS. Permiten la programación de motes mediante simples scripts.
- PicoRadio: Redes inalámbricas de muy bajo consumo.
- SSIS: Sensores de detección de integridad estructural. Informan de la localización y daños durante y después de un terremoto.
- TinyDB: Sistema de procesamiento de peticiones para la extracción de información de una red de sensores TinyOS.
- XYZ On A Chip: Redes de sensores inalámbricas integradas para el control de entornos interiores en edificios.

i. NesC

NesC es una extensión del lenguaje de programación C diseñado para plasmar los conceptos de estructuración y los modelos de ejecución de TinyOS. Los conceptos básicos que hay tras nesC son:

Separación de construcción y composición: los programas se hacen mediante componentes que son "conectados" para formar los programas completos. Los componentes tienen una concurrencia interna en forma de tareas. Los hilos de control

pueden pasar a un componente a través de sus interfaces. Estos hilos pueden ser de ejecución de tareas o interrupciones hardware.

Las especificaciones del comportamiento de un componente se hacen por medio de interfaces. Las interfaces pueden ser proporcionadas o usadas por los componentes. Las interfaces proporcionadas son para representar la funcionalidad que el componente proporciona al usuario, y las interfaces usadas representan la funcionalidad que el componente necesita para hacer su trabajo.

Las interfaces son bidireccionales: especifican un conjunto de funciones para ser implementadas por el proveedor de la interfaz (los métodos) y un conjunto para ser implementados por el usuario de la interfaz (los eventos). Esto permite que una interfaz simple represente una interacción compleja entre componentes (por ejemplo, registrar algo de interés de un evento, seguido por una llamada cuando el evento ocurre). Esto es importante porque todos los métodos largos en TinyOS (por ejemplo, enviar paquetes) son de no-bloqueo; se indica que se han completado a través de un evento (envío realizado).

Mediante las especificaciones de las interfaces, un componente no puede llamar al método enviar hasta que proporcione una implementación del evento envío realizado.
Los componentes están conectados estáticamente a otros por medio de las interfaces.

Esto incrementa la eficiencia del tiempo de ejecución, fomentando el diseño robusto y permitiendo un mejor análisis estático de los programas. NesC está diseñado bajo la expectación de que el código será generado por compiladores de programas-completos. Esto debería también permitir una mejor generación de código y análisis.

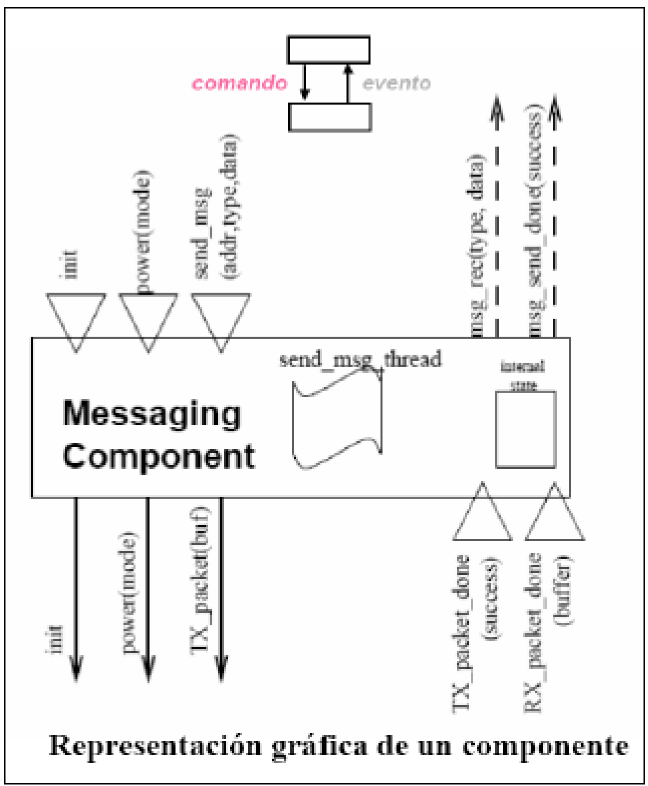


Figura 38: Estructura de un componente

Estructura de un componente

Una aplicación nesC consiste en uno o más componentes unidos para formar un ejecutable. Un componente proporciona y usa interfaces. Estas interfaces son el punto de acceso al componente y son bidireccionales. Una interfaz declara un conjunto de

funciones (métodos) que la proveedora debe implementar y otro conjunto de funciones (eventos) que la usuaria debe implementar. Para llamar a los métodos de una interfaz, se deben implementar los eventos de esa interfaz. Un componente simple puede usar o proporcionar múltiples interfaces y múltiples instancias de la misma interfaz.

Hay dos tipos de componentes en nesC: los módulos y las configuraciones. Los módulos proporcionan el código de la aplicación, implementando una o más interfaces. Las configuraciones se usan para ensamblar los componentes unos con otros, conectando las interfaces que usan unos componentes a las interfaces que les proporcionan otros. Esto es lo que se llama wiring (conexionado). Cada aplicación nesC se describe por una configuración de alto nivel que conecta todos los componentes que contiene.

Los ficheros que van a componer una aplicación serán:

miaplicacion.nc

Será el fichero configuración del componente. En su código contendrá dos apartados:

• Configuración. En general vacía, sólo contendrá algo si se pretende crear un componente no mediante su implementación de código directa (en Module) sino ensamblando otros componentes ya creados.

• Implementación. Aquí es donde se definen las conexiones que hay entre los diferentes componentes que utilizan la aplicación.

miaplicacionM.nc

Será el módulo del componente. En él se definen las interfaces que se proporcionan y que se usan y, por supuesto, contiene la implementación de la aplicación en sí.

Puede haber más ficheros para una misma aplicación, si se incluyen librerías en las que se definen diferentes estructuras necesarias para la aplicación, como en el ejemplo:

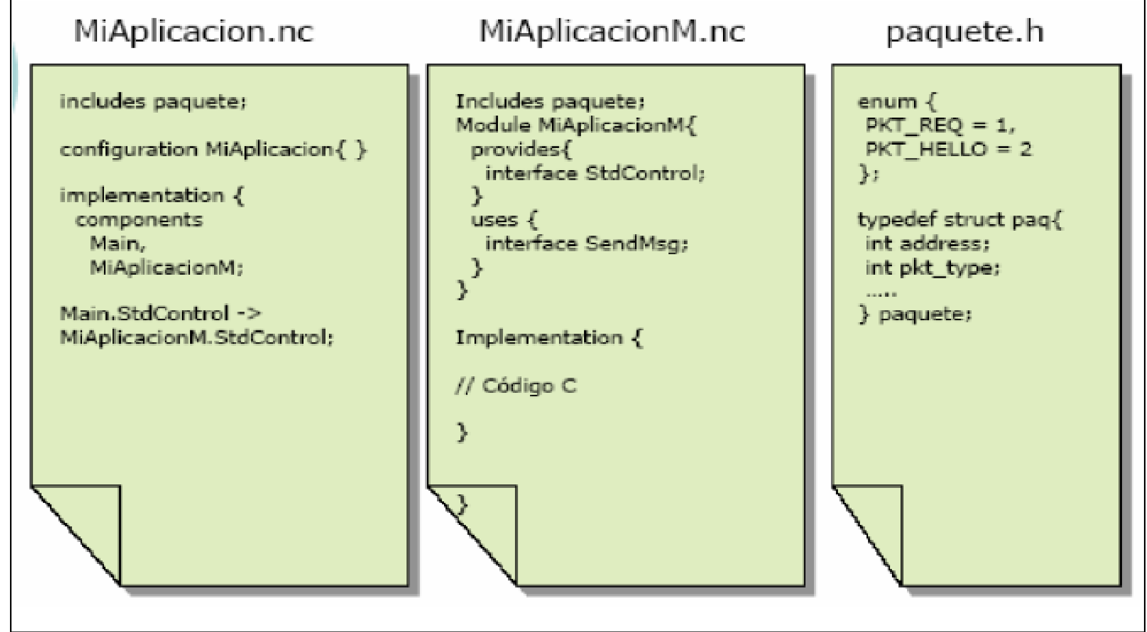


Figura 39: Ficheros de una aplicación

ii. Herramientas de TinyOS

Las herramientas que a continuación se describen ayudan en la tarea del desarrollo de una aplicación pero no forman parte de la aplicación que finalmente se encargará de recoger información en los nodos. Estas herramientas pertenecen al entorno de desarrollo de TinyOS. Otros entornos de desarrollo tienen sus propias herramientas de simulación y visualización de datos.

TOSSIM / TinyViz es un simulador de eventos discretos para TinyOS. TOSSIM (TinyOS SIMulator). Se compila directamente desde componentes TinyOS a la plataforma de destino especificada. Gracias a esto se pueden introducir sentencias en el código generado que informen del estado de la simulación. Por ejemplo, se puede saber cuándo la simulación transmite un paquete, enciende un led, provoca una interrupción del reloj, etc. Permite una simulación bastante completa de una red. Entre sus posibilidades se encuentra la simulación de las transmisiones de datos a nivel de bit fijando una probabilidad de error de bit. También puede tomar lecturas de los sensores, los datos obtenidos de ellos son en principio aleatorios, aunque existe la posibilidad de que sean introducidos por el usuario de TOSSIM.

TinyViz es la interfaz gráfica de TOSSIM. Permite ver la topología de la red aunque la calidad de la imagen que ofrece no es buena. Soporta la adición de plug-ins con nueva funcionalidad. Un ejemplo es el mencionado anteriormente que permite forzar los valores que toman los sensores.

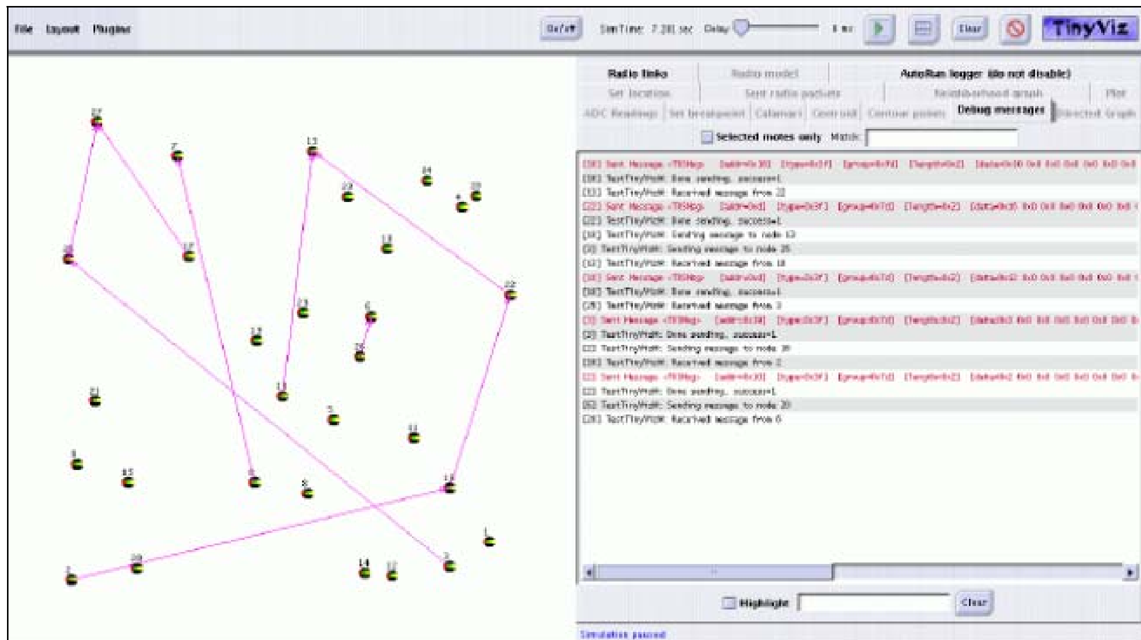


Figura 40: TinyViz

Surge View

Surge View es una aplicación Java que viene incluida con TinyOS. Permite monitorizar una red y analizar el funcionamiento del mallado de la red. Sus características incluyen:

- Descubrimiento y configuración automática de la red.
- Visionado de la topología de red.
- Redes de sensores inalámbricas
- Almacenamiento y visualización de estadísticas de la red como rendimiento, calidad de los enlaces, etc.

- Herramienta gráfica para el visionado de datos.

Figura 41: Surge View

SerialForwarder

Esta aplicación java permite recibir paquetes por el puerto serie del PC y reenviarlos a través de otros puertos del PC. De esta forma otros programas, como por ejemplo Surge View, pueden comunicarse con la red de sensores a través de un nodo conectado al PC que hace de pasarela.

NesDoc

Esta utilidad permite generar documentación automáticamente a partir del código fuente de un programa. Por cada fichero fuente genera un fichero HTML con un gráfico que describe el conexionado de los componentes del fichero a través de sus interfaces y una descripción textual sacada de los comentarios de los ficheros fuente. Por esto, para aprovechar todas las características de NesDoc es necesario seguir una serie de reglas al comentar los ficheros fuente.

GRATIS II / GME

GRATIIS II (Graphical Development Environment for TinyOS) es un entorno que ofrece una representación gráfica de los componentes implicados en una aplicación tinyOS. Ofrece un traductor que permite transformar entre los modelos gráficos que usa y los ficheros de configuración de NesC (en ambos sentidos).

GME (Generic Modeling Environment) es entorno de modelado sobre el que se instala GRATIS como un metamodelo de NesC.

TinyDB

TinyDB es un sistema de procesado de consultas para extraer información de una red de sensores con TinyOS. Convierte la red en una tabla de una base de datos distribuida, donde existe una columna por cada tipo de dato que se pretende leer (temperatura, luz, etc.).

Las consultas se realizan en el lenguaje TinySQL, una extensión del lenguaje de consultas SQL. Las extensiones realizadas a SQL permiten que al pedir una lectura se puedan especificar la frecuencia de muestreo y el periodo de tiempo durante el que se tomarán muestras entre otros parámetros. Incluso es posible que TinyDB ajuste estos parámetros para conseguir un determinado tiempo de vida de los nodos.

Estas consultas, que son realizadas por una aplicación ejecutada sobre un PC, provocan en los nodos la lectura de datos. Cada vez que un dato consultado sea leído, éste se introducirá en un mensaje y se reenviará por la red de vuelta al PC que solicitó el dato. Para ahorrar energía, el número de mensajes retransmitidos se reduce organizando

los nodos en una estructura en árbol en la que cada nodo sólo se comunicará con su nodo padre y sus nodos hijo.

TinyDB está implementado como un framework extensible. A través de esta extensibilidad se pretende que, en un futuro, se añadan nuevos eventos y atributos de los nodos y se pueda invocar comandos para la realización de tareas de control y actuación.

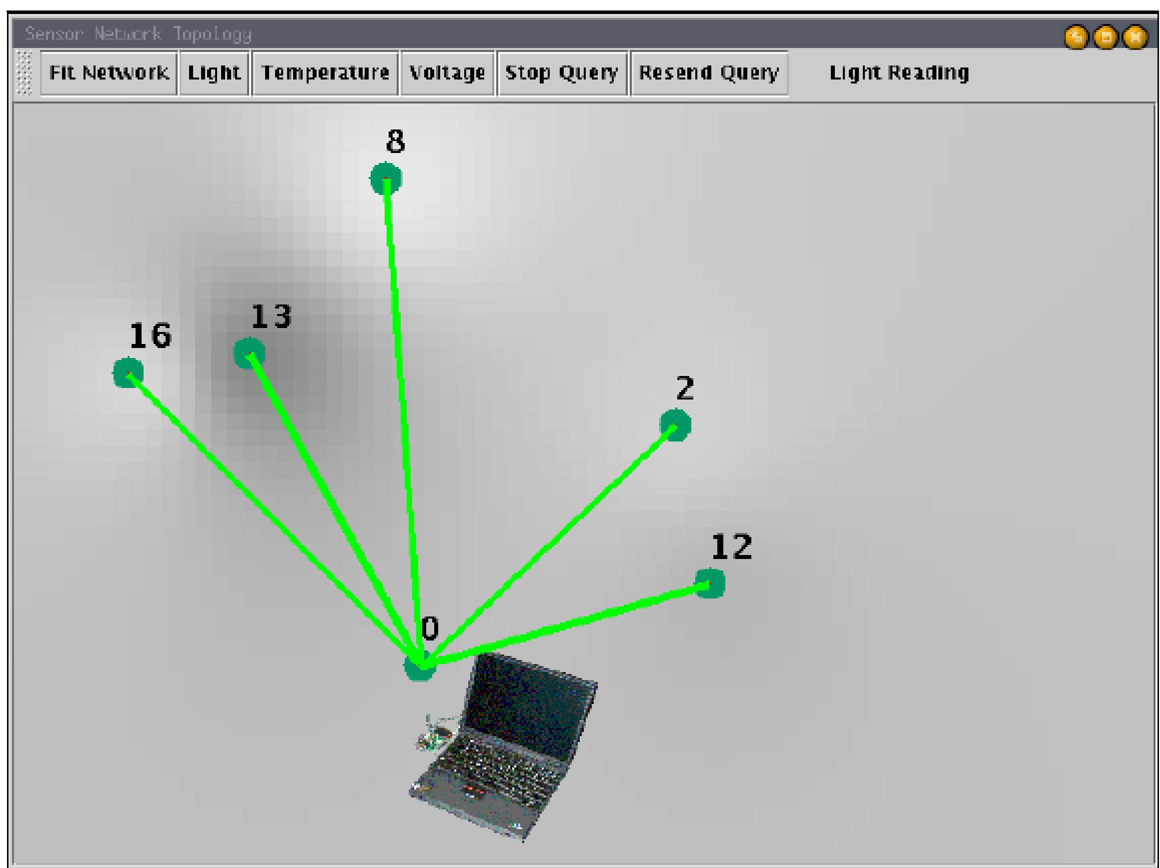


Figura 42: TinyDB

Bombilla / Maté

Maté es la maquina virtual de TinyOS, Bombilla es el conjunto de componentes que se sitúan sobre el sistema operativo para ejecutar los scripts de Maté. El lenguaje en el que se escriben estos scripts tiene un número muy limitado de operaciones pero permite que las aplicaciones se implementen con pocas líneas de código.

El código en Bombilla se divide en cápsulas de 24 bytes. Una de estas cápsulas se encarga de la recepción de mensajes, otra de la transmisión de mensajes y otra de los eventos del reloj que permiten disparar la adquisición de datos. Si es necesario el uso de algoritmos puede haber otras cuatro cápsulas adicionales que los contengan.

Las instrucciones de Maté hacen que la programación ya no sea asíncrona, ahora se sabe qué momento van a llegar datos de los sensores, lo cual facilita la tarea de programar la aplicación. Cuando se pide un dato, el programa queda esperando hasta que el sensor obtiene y devuelve ese dato. La desventaja es clara: mientras se espera ese dato se pueden dar situaciones de bloqueo.

Maté es adecuado para aplicaciones simples que necesitan ser reprogramadas a menudo. Para la reprogramación basta introducir las nuevas cápsulas en la red y los propios nodos las adoptarán si contienen una versión más moderna que la que están ejecutando. Sin embargo, Maté no permite realizar operaciones matemáticas que sean más complejas que una resta, ni tampoco puede ejecutar programas extensos.

VIII. Futuro de las WSN

Las características de flexibilidad, movilidad, alta fidelidad en sensorización, bajo coste y rápido despliegue de las WSN crean muchas nuevas áreas de aplicación interesantes para la sensorización remota. En el futuro, este amplio rango de áreas de aplicación hará de las redes de sensores una parte integral de nuestras vidas.

Sin embargo, la realización de las redes de sensores debe satisfacer las restricciones introducidas por factores como la tolerancia a fallos, escalabilidad, coste, hardware, cambios en la topología, entorno y consumo energético. Puesto que estas restricciones son muy exigentes y específicas de las redes de sensores, se requieren nuevas técnicas para este tipo de redes. En la actualidad hay muchos investigadores involucrados en el desarrollo de tecnologías necesarias para las diferentes capas de la pila de protocolo de las redes de sensores. Además de estos proyectos, se requiere más trabajo en los problemas descritos y más desarrollos para solucionar los temas de investigación abiertos que hemos estado viendo en este capítulo.

Debemos tener en cuenta que estamos tratando con una tecnología bastante reciente en la que hay muchos diseños pero pocos "funcionan", no existe lo que se llama una killer application que cree una nueva forma de mercado (como fue la tecnología móvil) y que el 99% de las redes son cableadas.

Si resumiéramos los factores que están actualmente impidiendo el desarrollo deberíamos resaltar:

- No existen tendencias claras de SO o plataformas hardware.
- Falta de estándares o protocolos comunes.
- Limitación de recursos: energía, capacidad de CPU, memoria.
- David Culler: "The lack of an overall sensor network architecture" (La falta de una arquitectura general para redes de sensores).

Sin embargo, hay mucho trabajo por hacer en todos estos aspectos. Tanto a nivel físico, como de computación: sistemas operativos, algoritmos distribuidos, etc. como de comunicación: protocolos de enrutamiento, mantenimiento de la topología, descubrimiento de vecinos, etc.

Cada vez van saliendo nuevas soluciones que permiten mejorar cada uno de estos apartados. Por ejemplo, una posible solución distribuida sería la creación de Middleware, que establezca una interoperabilidad entre los sistemas operativos y una aplicación, de tal forma que proporcione interfaces de alto nivel para enmascarar la complejidad de las redes y protocolos o que permita a los desarrolladores centrarse en cuestiones específicas de la aplicación.

En un futuro no muy lejano veremos cómo las redes de sensores empezarán a verse en todo tipo de aplicaciones como las que hemos visto en este capítulo y en muchas mas que irán surgiendo. Problemas como las limitaciones de memoria o procesador irán desapareciendo con las nuevas nanotecnologías y MEMs, lo que permitirá bajar mucho más el consumo de potencia, alargar la vida de los nodos y quizá cambiar la perspectiva de estas redes hacia nuevos campos de actuación.

En la película de Hollywood Twister, los meteorólogos persiguen tornados para acoplarles un barril con sensores en su camino que pudieran meterse en el corazón del tornado. Los científicos podían medir así el tornado desde dentro con la información que enviaban los sensores.

El laboratorio nacional de tormentas graves (NSSL) informó de que la película estaba basada en un trabajo del Dr. Al Bedard que desarrolló un instrumento para observar tornados. Era un barril dotado de sensores en su exterior para medir viento, presión y temperatura, y llevaba unas baterías incorporadas. Lo que no tenía muy en cuenta eran los inconvenientes de seguridad como los objetos que podían chocar contra el barril.

Figura 43: Twister

Aunque la película Twister exageraba la realidad en su momento, ahora ya no es imposible pensar en la utilización de redes de sensores para aplicaciones de ese tipo en un futuro. Nodos sensores de bajo peso pueden introducirse en un tornado, tomar datos y transmitirlos a una red donde los usuarios podrían evaluar y analizar los datos. La robustez y el coste de la red de sensores superarían la alta probabilidad de fallo en los nodos para asegurar el éxito del proyecto y, además, conseguir una información relevante desde el corazón del tornado, para el bien de los meteorólogos.

Otra aplicación que podríamos considerar futurista es la que se expone en la siguiente figura.

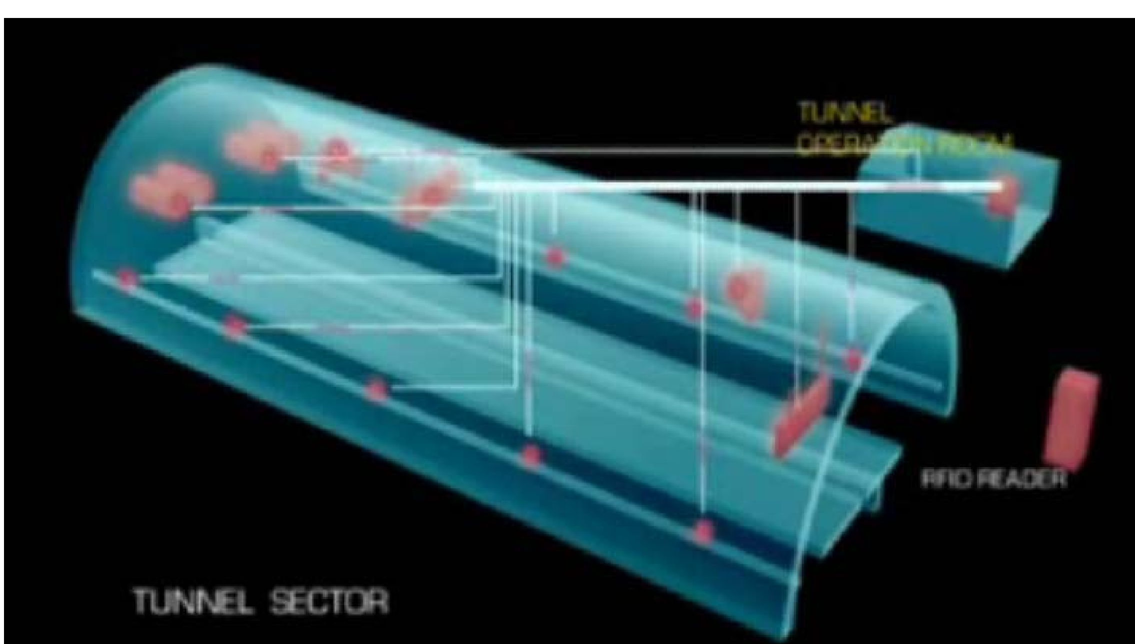


Figura 44: Túnel WSN

En él se muestra un escenario de un túnel con bastante tráfico en el que ocurre un accidente y provoca un incendio en su interior. El túnel está dotado de sensores que transmiten información de todo tipo a las centrales de control de la ciudad: niveles de temperatura, daños, número de coches y sus características, etc.

Cada coche implicado se muestra como una red de sensores propia, así como cada persona lleva sus sensores que indican sus niveles de salud.

La comunicación y el tiempo de respuesta, evidentemente, son inmediatos. El tráfico es desviado hacia otras carreteras y los sistemas de emergencia del túnel hacen su parte hasta que llegan los refuerzos humanos.

Los vehículos de emergencia también disponen de sistemas de control con información actualizada de la situación en el interior del túnel, e introducen robots de detección y rastreo de los distintos niveles de peligro que pudiera haber. Finalmente, los propios bomberos son una PAN (Personal Area Network) móvil, y van provistos de varios sensores que indican sus características internas y externas, así como su posición. Los cascos van provistos de cámara y pantallas que informan de cada uno de los parámetros clave para tomar las mejores decisiones.

En EEUU están ya experimentando en un proyecto para bomberos llamado Gíreles FireFighter System en el que están diseñando este tipo de cascos, así que parece que no estamos ante una realidad tan futurista.

Capítulo 3
MetaEdit+

I. Introducción a la herramienta

Este capítulo se dedicará a la descripción de la herramienta Metaedit+, desarrollada por la empresa Metacase [Metacase], y destinada a la creación de lenguajes específicos del dominio (DSL). Proporciona un entorno gráfico, amigable y fácil de usar que integra diferentes editores, navegadores de datos y otras herramientas que en principio son suficientes para crear cualquier DSL, crear modelos expresados en ese DSL y generar código (en un sentido amplio, podemos hablar de cualquier tipo de texto deducido del modelo). En Metaedit+, el lenguaje de metamodelado se denomina GOPPRR, que proviene de Graph, Object, Property, Port, Relationship y Role, que son los meta-tipos que proporciona para describir los lenguajes de modelado. Nosotros nos referiremos a estos meta-tipos como Grafo, Objeto, Propiedad, Puerto, Relación y Rol.

Aunque se permiten otras formas de representar los modelos, como por ejemplo texto, el modo más usado es el gráfico, que es que permite mayor riqueza y expresividad. En este modo de edición, los modelos son grafos cuyos elementos fundamentales son (siguiendo la terminología de Metaedit+) los objetos (vértices del grafo) y las relaciones (aristas del grafo). A diferencia de otras herramientas, el entorno de trabajo que utiliza no diferencia claramente los conceptos de sintaxis abstracta y sintaxis concreta. De esta forma, cuando se define un nuevo elemento del metamodelo y sus propiedades, se diseña también el aspecto visual del mismo. Además de herramientas para crear la sintaxis abstracta y concreta de los metamodelos, Metaedit+ proporciona un lenguaje propio para expresar cómo se realiza la generación de código, informes de consistencia o cualquier otro tipo de salida a partir del modelo. Con el término informe nos referiremos a aquellos programas escritos en el lenguaje de informes propietario de Metaedit+, y que sirve para generar una salida en base a un modelo. Con todo ello se permite construir un modelo gráfico a partir del cuál se pueden crear instancias de las que se puede generar distintos ficheros de código.

Para hacernos una idea un poco más precisa de qué es Metaedit+, vamos a enumerar y describir brevemente los editores, herramientas y navegadores fundamentales que antes mencionábamos, y que lleva integrados. Antes, sin embargo merece la pena destacar que en Metaedit+, como en cualquier herramienta de esta naturaleza, existe un repositorio destinado a almacenar los elementos de los metamodelos y de los modelos, y que por tanto juega un papel esencial. Más adelante comentaremos este repositorio.

La especificación de la sintaxis abstracta del lenguaje de modelado se realiza con el Metaedit+ Method Workbench (menú Metamodel del entorno). Éste proporciona un conjunto de herramientas potente y a la vez sencillo, de acuerdo con los conceptos del lenguaje de metamodelado GOPPRR. En la Figura 45 se muestra el metametamodelo del lenguaje de metamodelado empleado en Metaedit+.

Dichas herramientas de desarrollo:

- **Herramienta de objetos (***Object Tool***)**: para especificar tipos de objetos que son componentes básicos de los metamodelos.
- **Herramienta de relaciones (***Relationship Tool***)**: para indicar los conectores entre tipos de objetos.
- **Herramienta de roles (***Role Tool***)**: sirve para indicar los tipos de objetos que intervienen en una relación jugando un determinado rol. Un objeto puede jugar roles diferentes en distintas relaciones.
- **Herramienta de puertos (***Port Tool***)**: para especificar semántica adicional respecto a cómo los tipos de roles se conectan con los tipos de objetos.
- **Herramienta de grafos (***Graph Tool***)**: una vez definidos los tipos de objetos, relaciones, roles y puertos, permite establecer las reglas para la conexión de todos estos elementos.
- **Herramienta de propiedades (***Property Tool***)**: permite modificar las propiedades de cualquier tipo de elemento, algo posible con las propias herramientas, y también permite crear nuevos tipos de datos.

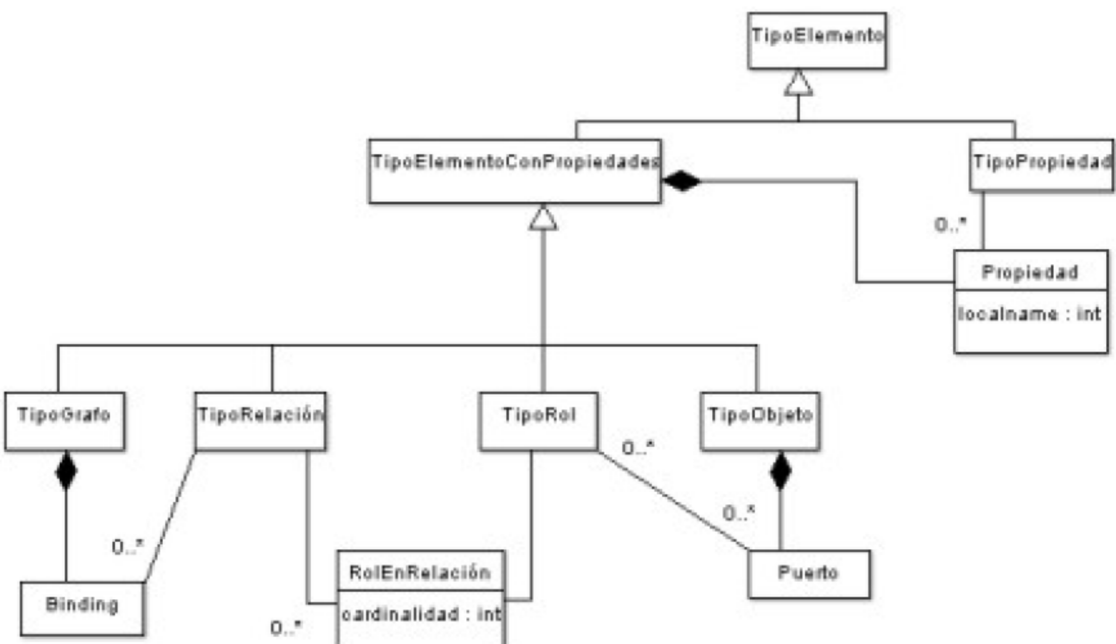


Figura 45: Metametamodelo del lenguaje GOPPRR

El resto de herramientas de desarrollo con las que podemos modificar el aspecto de a información que se muestra al usuario son:

- **Editor de símbolos (***Symbol Editor***)**: permite especificar y editar la presentación gráfica de los elementos definidos como parte del lenguaje (objetos, relaciones, roles y puertos). Por tanto, es la herramienta para establecer la sintaxis concreta.
- **Editor de diálogos (***Dialog Editor***)**: permite alterar la disposición de los elementos (*layout*) de los diálogos utilizados para la edición de las propiedades de los elementos de diseño.
- **Navegador de informes (***Report Browser***)**: permite crear tus propios *reports* que se añaden a los que incorpora por defecto. Los editores de Metaedit+ permiten crear y modificar modelos para cualquiera de los lenguajes de modelado definidos.

Estos editores ofrecen diferentes vistas y representaciones de un mismo modelo. Los tres editores que se proporcionan son:

- **Editor de diagramas (***Diagram Editor***)**: permite realizar representaciones gráficas de los modelos.
- **Editor de matrices (***Matrix Editor***)**: permite visualizar los modelos en forma de matrices.
- **Editor de tablas (***Table Editor***)**: muestra representaciones tabulares de las propiedades de los objetos en los modelos.

Además de las herramientas de desarrollo, hay herramientas para gestionar y consultar los metamodelos y modelos:

- **Navegador de grafos (***Graph Browser***)**: proporciona una vista jerárquica[13] del repositorio, de forma que las instancias del grafo aparecen en estructura de árbol. Permite inspeccionar el contenido del grafo seleccionado, y realizar operaciones sobre su contenido.
- **Navegador de tipos (***Type Browser***)**: se utiliza para visualizar y abrir editores de los elementos de los metamodelos, incluyendo los tipos de grafo, objeto, relación y rol. Se puede mostrar los elementos de esos tipos por proyecto, y los editores abiertos sobre los elementos de diseño. También permite abrir, cerrar o crear proyectos.
- **Navegador de objetos (***Object Browser***)**: nos proporciona una vista jerárquica de los datos del diseño basado en las estructuras de agregación de grafo, objeto y propiedad, esto es, un árbol en que para cada grafo se muestran sus objetos, y sus propiedades asociadas.
- **Gestor de grafos (***Graph Manager***)**: proporciona una vista del repositorio en forma de jerarquía gráfica. El gestor de grafos permite inspeccionar el contenido del grafo seleccionado, y realizar operaciones sobre ese grafo y su contenido. También se puede utilizar para exportar entre repositorios la información de los modelos creados, o para inspeccionar el uso de los elementos individuales, para conocer qué elementos de diseño que referencian al elemento seleccionado.
- **Navegador de metamodelo (***Metamodel Browser***)**: se utiliza para editar y consultar el metamodelo de un lenguaje de modelado de un proyecto abierto, mostrando los tipos que contiene y las relaciones entre ellos. Con esta herramienta, el desarrollador del lenguaje tiene acceso a todos los metamodelos de los lenguajes de los proyectos abiertos.
- **Gestor de tipos (***Type Manager***)**: permite exportar especificaciones de metamodelos a otros repositorios[14], y eliminar especificaciones de metamodelos no deseadas.
- **Herramienta de información (***Info Tool***)**: muestra información acerca de un tipo dado, indicando qué tipos usa y qué otros tipos lo usan.

En los siguientes apartados del capítulo vamos a tratar las herramientas básicas que el framework Metaedit+ nos proporciona para la creación de herramientas de modelado.

II. Herramientas de metamodelado

Son las herramientas básicas para diseñar el metamodelo de acuerdo al lenguaje de metamodelado GOPPRR. Son las herramientas de *propiedades*, *objetos*, *relaciones*, *roles*, *puertos*, y *grafos*. Forman la parte fundamental del *Metaedit+ Method Workbench*. Las herramientas de objeto, relación, rol y puerto son muy similares. La de propiedades también es parecida. La de grafos es la más compleja de ellas y es la clave para crear los grafos. En siguientes apartados las comentaremos.

i. Herramienta de propiedades (Property Tool)

Es la herramienta para diseñar los tipos de las propiedades que especifican información del resto de tipos del grafo (objetos, relaciones, roles, puertos y grafos). Con ella se pueden consultar los tipos de las propiedades definidas, crear otras nuevas o modificar las existentes. Como ejemplos de tipos de propiedades podemos pensar en las variables, direcciones de correo electrónico, teléfonos, fechas, o precios. Las propiedades se añaden desde la herramienta del tipo de elemento a que corresponde, y los nuevos tipos de propiedades se definen con esta herramienta. En la Figura 46 se define un nuevo tipo de propiedad denominado *Plataforma PLC*.

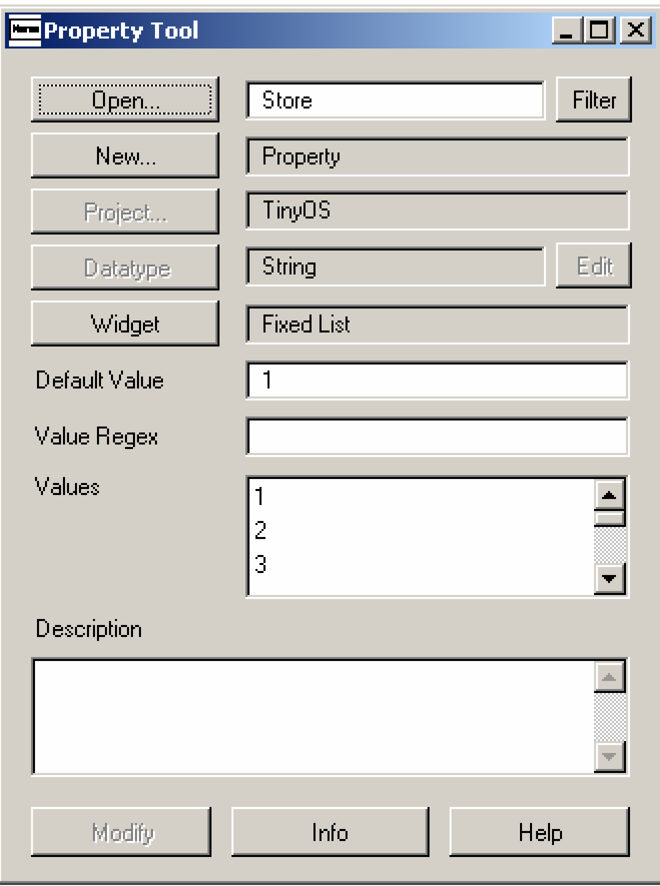


Figura 46: Herramienta de propiedades

La primera de las cajas de texto se utiliza para especificar el nombre de un nuevo tipo de propiedad. A un lado se encuentra el botón *Open* que sirve para abrir un tipo de propiedad existente en caso de desear modificar una de ellas, y al otro lado está el botón *Filter*, que permite restringir la selección del tipo a los tipos de propiedades de un tipo de datos específico o componente gráfico (widget). Si no se especifica dicho filtro, al pulsar el botón *Open* el diálogo que muestra una lista para seleccionar el tipo, mostrará todos los tipos de propiedades.

La segunda de las cajas de texto especifica el supertipo del tipo de la propiedad. Se puede seleccionar pulsando el botón *New* al lado de la caja de texto. Por defecto el supertipo (el tipo raíz) de las propiedades es el tipo *Property*. Esto denota que existe la jerarquía de herencia entre tipos de propiedades. Pero esto no es exclusivo de los tipos de propiedades, puesto que es posible organizar elementos de un lenguaje de metamodelado (tipos de grafos, objetos, relaciones, roles, puertos o propiedades) en jerarquías de herencia. La herencia permite a un nuevo tipo de objeto, relación, rol o puerto disponer (obligatoriamente) de las propiedades de sus ancestros. Hay cierta herencia de comportamiento, como podría ser que un objeto pueda tomar parte en un enlace en que se incluye un ancestro de ese tipo de objeto. De forma similar, una propiedad cuyo tipo de datos es de un determinado tipo (no puede corresponder con un tipo de propiedad), puede contener grafos, objetos, relaciones, puertos o roles cuyos tipos sean descendientes del tipo dado.

En el tercer campo encontramos el nombre del proyecto al que pertenece el tipo de propiedad. El proyecto sirve para englobar bajo un mismo marco los tipos de elementos de los tipos de grafos para un determinado tipo de grafo, de forma que estarán disponibles si el proyecto al que pertenecen está abierto. Con el botón *Project* que le precede se puede seleccionar el proyecto al que se asociará, de entre los proyectos abiertos en ese preciso instante.

El siguiente cuadro expresa el tipo de datos subyacente para el tipo de propiedad dada. El tipo de datos se selecciona de una lista que aparece al pulsar el botón *Datatype*. Si el tipo es un tipo complejo, esto es, un tipo grafo, objeto, relación o rol, con el botón *Edit* contiguo al citado campo se podrá acceder a la herramienta del tipo correspondiente. Vamos a enumerar y describir los tipos de datos soportados:

- **String**: cadena de caracteres ordinarios sin ningún formato. Este tipo habilita el campo de componente gráfico de interfaz de usuario en que se presenta, el campo de valor por defecto, e incluso el de lista de valores dependiendo del tipo de componente indicado. Hablaremos acerca de ellos más adelante.
- **Text**: cadena de caracteres de mayor extensión que *String* utilizado para descripciones y que admite caracteres de formato, como retornos de carro, y formatos básicos como distintos tipos de fuentes, cursiva, u otros. El componente gráfico que se utiliza para visualizarlo es una caja de texto multilínea con barra de scroll vertical. Este tipo de propiedades se pueden modificar con un editor de textos, lo que resulta útil para textos extensos o con formato.
- **Number**: número entero, real o en notación científica. El valor que toma el tipo por defecto es 0.
- **Boolean**: tipo de datos booleano, que toma valores *verdadero* o *falso*. Se visualiza como una casilla de verificación que por defecto está desmarcada, lo que se interpreta con el valor *falso*.
- **Collection**: define una lista de *Strings* u objetos, inicialmente vacía, en la que el usuario puede añadir o borrar elementos. Cuando se selecciona este tipo de datos, aparece un diálogo solicitando el tipo de datos de la colección, que puede

ser: grafo, objeto, relación, puerto, rol o *String*. El componente gráfico que lo visualiza es una lista con una barra scroll vertical, y los elementos se introducen o borran utilizando el menú contextual que aparece al aplicarlo sobre la lista.

- **Graph, object, relationship, role**: los grafos, objetos, relaciones y roles también se pueden usar como valores de una propiedad. Al seleccionar este tipo de datos, la herramienta pregunta por el tipo concreto del elemento del grafo (tipo concreto de grafo, objeto, relación o rol). También se permiten instancias de los subtipos para los valores de estas propiedades, o de los supertipos de todos los tipos de ese metatipo (tipos raíz). El componente gráfico que aparece cuando tenemos propiedades de este tipo es una caja de texto no editable en la que está escrito el nombre del elemento. Con el menú contextual se pueden crear nuevas instancias del tipo subyacente en la propiedad o señalar instancias existentes. Estos tipos de datos son potentes, permitiendo propiedades que son complejas estructuras de datos.

El botón indicado por *Widget* servirá para indicar el componente gráfico que se mostrará para visualizar el tipo de dato seleccionado. Sólo aparece si el tipo de datos seleccionado es *String*. El componente seleccionado aparecerá entonces en la caja de texto situada contiguamente. Los componentes gráficos disponibles son:

- **Input Field**: se trata de una caja de texto en la que se puede escribir una cadena.
- **Fixed List**: una lista desplegable en la que los valores de la lista han sido prefijados y no se pueden editar, simplemente se puede seleccionar uno de los que ya hay.
- **Overridable List**: lista desplegable en la que pueden existir una serie de cadenas prefijadas, pero que permite la introducción de un nuevo valor que no se incluirá en la lista, y no podrá ser seleccionado en un futuro si su valor se cambia a otro de los preestablecidos.
- **Editable List**: lista desplegable en la que pueden existir una serie de valores prefijados, y se permite la introducción de nuevas cadenas que se incluirán en la lista, para ofrecer la posibilidad de seleccionarlos de nuevo posteriormente. Este tipo de componente gráfico permite mediante el botón *Reset Projects* que los valores de las listas editables que hayan sido introducidos por los usuarios (valores no prefijados), se borren.
- **External Element**: se utiliza cuando una cadena representa el nombre de un fichero, un recurso, o comando del sistema operativo. Se visualiza con el mismo componente gráfico que si se tratase de un campo de texto normal, pero el menú que muestra dispone de un elemento denominado *Execute* que envía el contenido de la propiedad como una cadena para que la interprete el sistema operativo.

El penúltimo campo que aparece cuando trabajamos con *Strings*, o con *Number* es el del valor por defecto. El valor que aquí introduzcamos será el que toma la propiedad cuando se crea.

En la ventana de la herramienta aparece un área de texto bajo la etiqueta *Description*, que se utiliza para esbozar una breve descripción de la propiedad. Por último comentar los botones que aparecen en la parte baja de la ventana. *Modify* se utiliza para confirmar la modificación del tipo, *Info* abre la herramienta de información del tipo, y *Help* proporciona ayuda acerca del manejo de la herramienta de objetos.

Existe una propiedad (apréciese que no nos referimos a tipo de propiedad) que se puede definir y que tiene implicaciones a la hora de utilizar los informes predefinidos.

El tipo de esta propiedad puede denominarse de cualquier modo, pero el nombre local de la propiedad tiene que ser *Documentation*. Al ejecutar ciertos informes predefinidos en grafos en que algunos objetos poseen esta propiedad, se tienen en cuenta como campos de documentación y la información que contienen se incluyen en los informes.

ii. Herramienta de objetos (Object Tool)

Es la herramienta con la que definiremos, consultaremos y modificaremos lo que serán los tipos de objetos (los tipos de objetos son los metatipos y sus instancias u objetos son los vértices del grafo que representa el modelo). Ejemplos de tipos de objetos pueden ser: Persona, Tabla, Etapa, Proceso, Entidad financiera u Organización. La Figura 47 muestra la ventana de la herramienta para la definición de un tipo de objeto *Sensor* con las propiedades *Type, Thresold y Operation* y que pertenece al proyecto *TinyOS*.

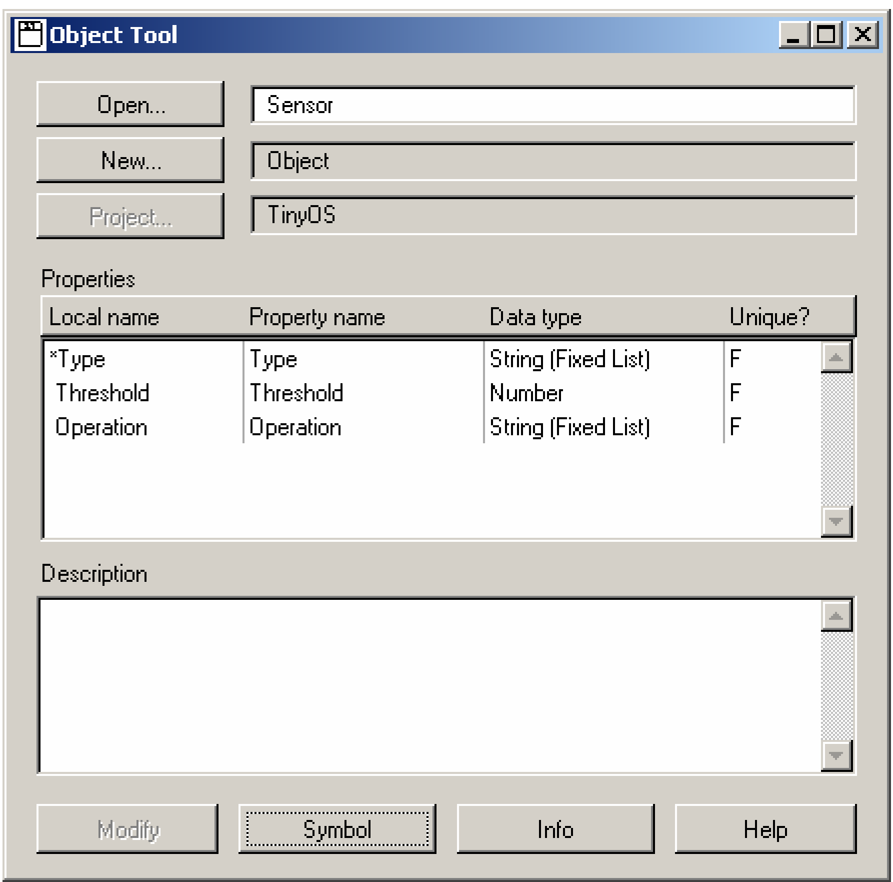


Figura 47: Herramienta de objetos

En la ventana que se muestra hallamos una serie de partes. La primera caja de texto se utiliza para introducir el nombre del tipo de objeto en caso de que se desee crear uno nuevo. El botón *Open* tiene el significado anteriormente expuesto. La segunda caja de texto especifica el nombre del supertipo del tipo de objeto. Por defecto aparecerá seleccionado *Object*, que es el supertipo de todos los tipos de objetos. Para cambiar el supertipo se debe pulsar el botón *New* y seleccionar uno de la lista. Los subtipos heredan obligatoriamente las propiedades de sus padres. El tercer cuadro de texto especifica el proyecto en que se engloba el tipo de objeto.

La tabla que aparece bajo el título *Properties* enumera las propiedades que incorporamos al tipo de datos definido. En la tabla aparecen cuatro columnas. La

primera de ellas es el nombre de la propiedad dentro de ese objeto, la segunda el tipo de propiedad, la tercera indica el tipo de datos subyacente para el tipo de propiedad. La última columna nos permite definir la restricción de unicidad sobre la propiedad en el modelo, de forma que si toma el valor *T* la aplicación se asegura de que el valor introducido para la propiedad sea único, y si toma el valor *F* no lo tendrá en cuenta. De las cuatro columnas, la primera y la cuarta se pueden modificar, mientras que las otras son sólo informativas y se alterarían con la herramienta de propiedades. Una de las propiedades siempre tendrá a la izquierda del nombre un asterisco, lo que refleja que esa propiedad (no hablamos de tipo de propiedad, sino de la instancia del tipo) es el identificador del objeto. Por defecto se establece a la primera propiedad que introducimos, pero se permite cambiarlo. El identificador es útil a la hora de la generación de código. No se trata realmente de un identificador que utilice Metaedit+ para distinguir internamente las instancias de los modelos, pues para eso ya se dispone de un identificador interno (el *OID*).

El área de texto que aparece más abajo se utiliza para proporcionar una breve descripción del tipo de objeto. De los botones que aparecen en la parte baja de la ventana, *Modify* se utiliza para confirmar la modificación del tipo, *Symbol* abre el editor de símbolos del que hablaremos más adelante, *Info* abre la herramienta de información del tipo, y *Help* proporciona ayuda acerca del manejo de la herramienta de objetos.

iii. Herramienta de relaciones (Relationship Tool)

Se utiliza para definir lo que serán los tipos de relaciones (tipos de aristas) entre los tipos de objetos (tipos de vértices) del tipo de grafo (lo que comúnmente se expresa como las asociaciones entre las clases del metamodelo). Algunos ejemplos de tipos de relaciones pueden ser las transiciones, los flujos de datos o de información, o las relaciones familiares. Los objetos y las relaciones conforman los conceptos más fundamentales a la hora de crear los grafos. El manejo de esta herramienta es muy similar al de la herramienta de objetos. La Figura 48 muestra como se definiría la relación *Interface* que pertenece al proyecto *TinyOS*.

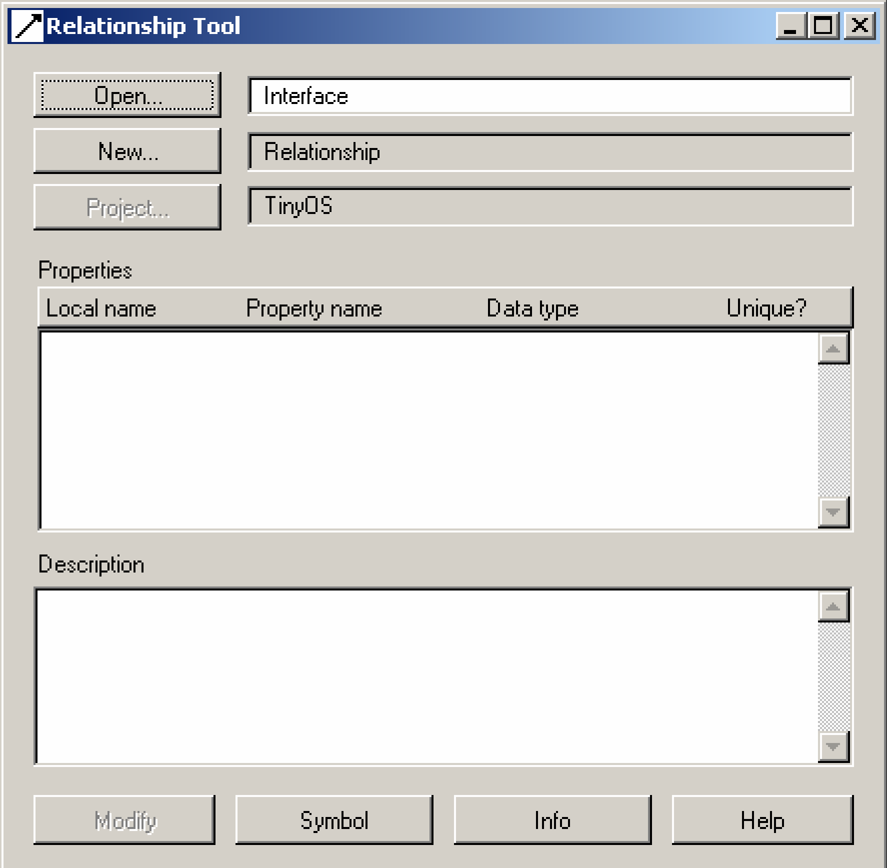


Figura 48: Herramienta de relaciones

iv. Herramienta de roles (Role Tool)

Un *rol* especifica el papel que juegan los objetos de los extremos de la relación a la que se conectan. Por ejemplo, en una *interface* tendríamos los roles *uses* y *provides* para los objetos correspondientes que serían los extremos de la relación, o en una relación de herencia de un diagrama UML las clases jugarían los roles *padre* e *hijo*. Esta herramienta se utiliza para definir los tipos de roles. En este caso, a diferencia del resto de tipos, podemos decir que es equivalente hablar de roles que de tipos de roles. La razón es que no se pueden crear instancias de estos tipos, de forma que podemos decir que esta herramienta define roles. La herramienta es similar a las dos anteriores. En la Figura 49 se muestra cómo definir un rol *Provides* para el proyecto *TinyOS*.

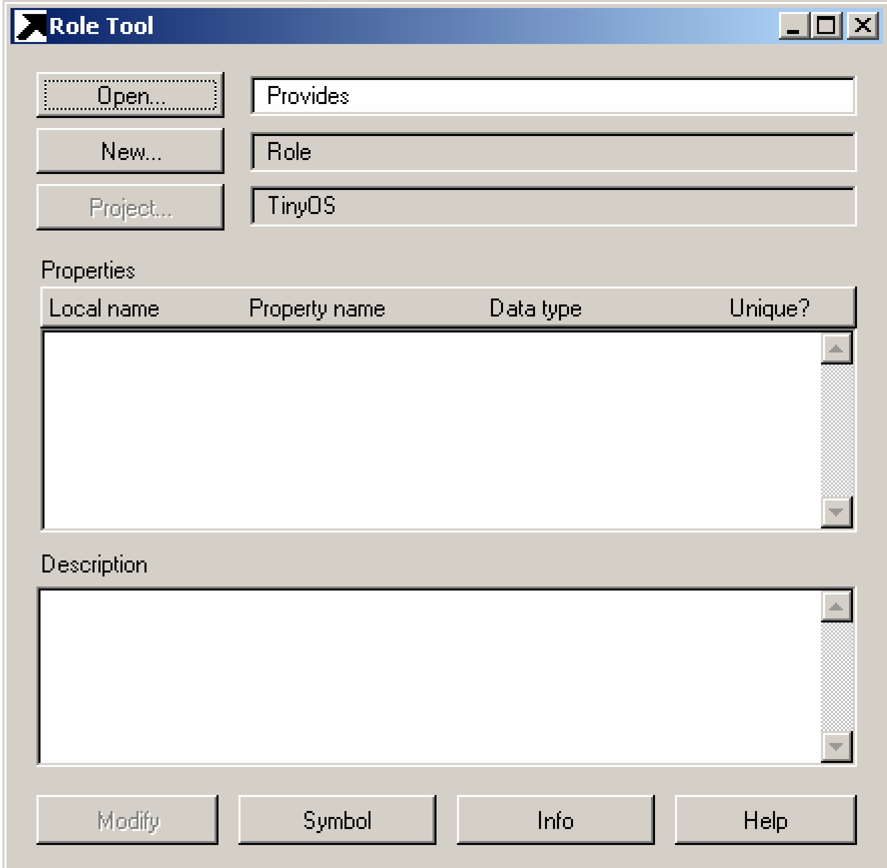


Figura 49: Herramienta de roles

v. Herramienta de puertos (Port Tool)

Con ella crearemos, consultaremos o modificaremos los tipos de puertos de los objetos. A nivel gráfico, un *puerto* es el punto del símbolo de un objeto al que se pueden enganchar las transiciones, más concretamente los roles. Los puertos que se definen en los símbolos deben tener un tipo. Estos tipos son los que se construyen con esta herramienta. Es posible asociar restricciones semánticas a un tipo de puerto. Este aspecto se mencionará posteriormente en la herramienta de grafos. Como ejemplo de aplicación podemos imaginarnos el establecimiento de un requisito para un árbol genealógico de forma que todas las instancias de relaciones familiares representarán en el origen a un padre (rol padre), y deben conectarse al borde inferior del símbolo *Persona*. Del mismo modo en el extremo opuesto de la relación se tendría el rol hijo, que se conectaría al borde superior del citado símbolo. La herramienta es similar a las tres anteriores. La Figura 50 muestra como se definiría un tipo de puerto denominado *Provides*. En este caso el botón de *Symbol* estará deshabilitado siempre, puesto que un puerto no tiene símbolo.

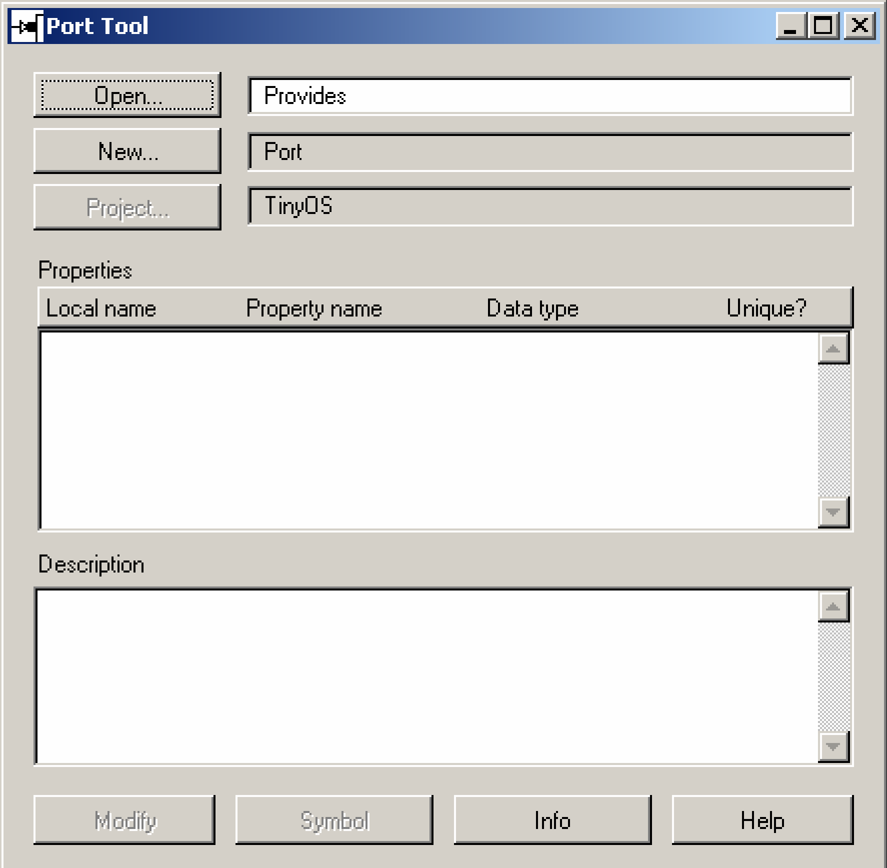


Figura 50: Herramienta de puertos

vi. Herramienta de grafos (Graph Tool)

Proporciona la funcionalidad que permite combinar objetos, relaciones, roles y puertos para crear un lenguaje de modelado o DSL. Como es lógico también permite realizar modificaciones a nivel de metamodelo y desde ella se pueden abrir las herramientas de desarrollo de los tipos de elementos del metamodelo. Es la herramienta más compleja de las que hemos descrito y es la clave para definir los metamodelos, estableciendo los elementos que lo formarán, los enlaces que tendrá y sus restricciones de conectividad entre otras cosas. La Figura 51 muestra el aspecto de la herramienta cuando se define un lenguaje para nesC.

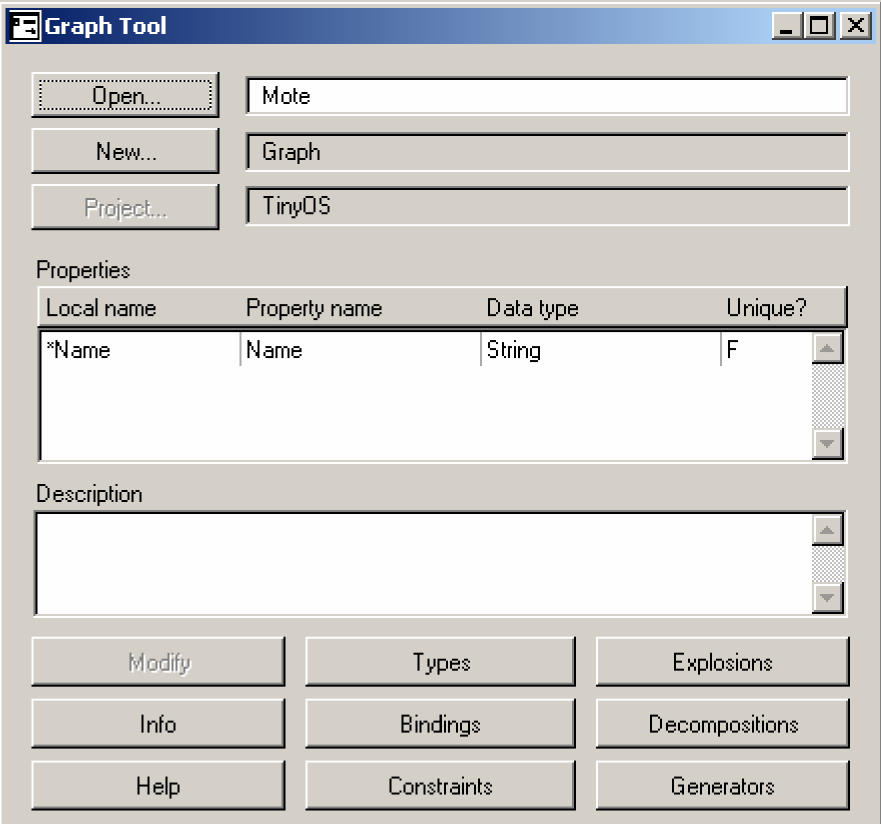


Figura 51: Herramienta de grafos

Se observa que se denomina *Mote* a los diagramas que se realizarán y que poseen la propiedad *Name* (el nombre del programa en nesC.

Los tres primeros campos de texto son similares a los del resto de herramientas y especifican el nombre del tipo de grafo, el supertipo (*Graph* por defecto) y el proyecto a que se asocia. La tabla de propiedades también es semejante a las herramientas anteriores, permitiendo especificar el nombre y tipo de las propiedades del tipo de grafo, la restricción de unicidad, y la declaración de la propiedad que hace las veces de identificador. El área de texto que se sitúa debajo de la tabla servirá para describir el tipo de grafo.

En la parte baja de la ventana aparecen nueve botones. Los botones de la primera columna ya se han comentado en herramientas anteriores, y sirven para modificar el tipo de grafo, abrir la herramienta que muestra información de éste, y la ayuda de la herramienta. Los otros seis botones introducen nuevas funcionalidades.

El botón *Types* constituye la *herramienta de tipos*, que nos sirve para indicar los tipos de objetos, relaciones y roles que se utilizarán en el tipo de grafo. Por cada tipo de objeto y relación que aquí indiquemos, aparecerá un botón en la barra de herramientas del editor de diagramas, más el icono de la relación genérica si le hemos indicado que en el tipo de grafo participa más de un tipo de relación. En la Figura 52 podemos ver el aspecto de esta herramienta para el lenguaje nesC.

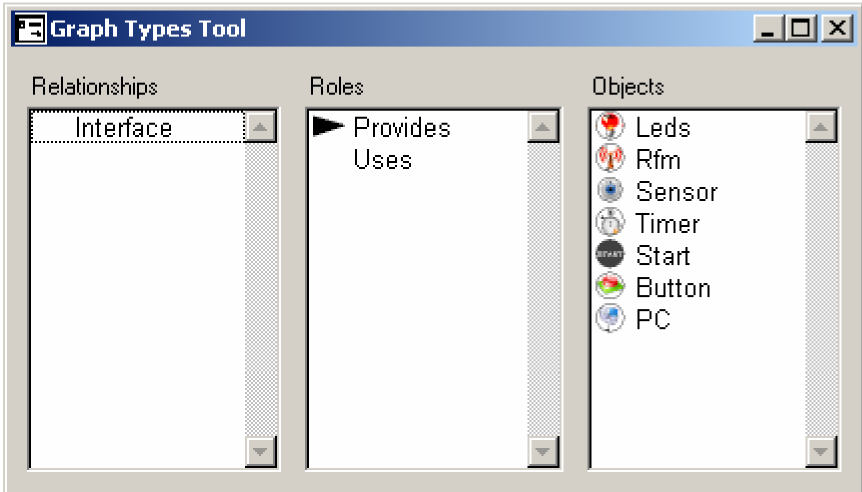


Figura 52: Definidor de tipos del grafo

Al pulsar el botón *Bindings* nos aparecerá la *herramienta de enlaces* en la que indicaremos para cada relación de las que hemos introducido antes, qué roles tiene asociados, y para cada uno de ellos, la cardinalidad, los objetos a los que se pueden conectar, y los puertos de los objetos a los que se puede enganchar (los objetos deben ser aquellos que contienen alguno de los puertos a los que se pueden enganchar). En la Figura 53 podemos ver el aspecto de esta herramienta para la definición de *bindings* en el caso del lenguaje nesC.

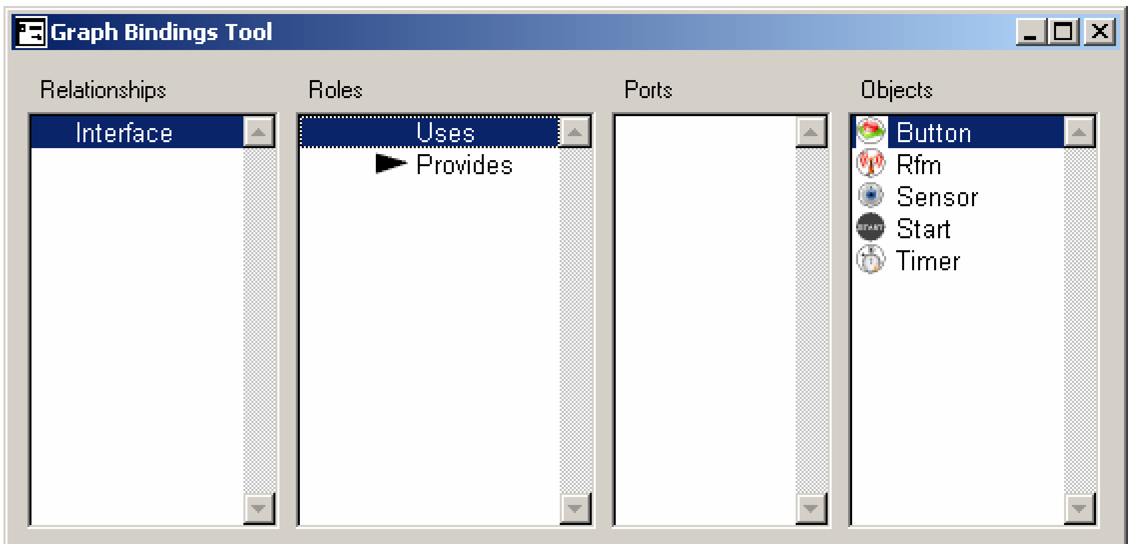


Figura 53: Definidor de los enlaces del grafo

Apréciese que mediante la cardinalidad de los roles permitimos relaciones n-arias, y no sólo binarias. Un detalle importante para la implementación es que el orden (descendente) en que se especifican los roles es el orden en que se enganchan los extremos de la relación. A modo de ejemplo, si definimos una transición que tiene como roles *Origen* y *Destino*, el primer rol de la lista debe ser *Origen*, pues es el que corresponde al extremo de la relación que se engancha al primer objeto, mientras que *Destino* corresponderá al segundo extremo de esta relación binaria. Por esta razón, es posible modificar no sólo la cardinalidad de los roles, sino también el orden. Para introducir un enlace (*binding)*, en primer lugar se introduciría en la primera columna el tipo de relación, y en la segunda columna un rol de esa relación. El siguiente paso sería

especificar en la cuarta columna los objetos con los que se permitirá conectar el rol de esa relación, y por último en la tercera columna se indicarían los puertos de los tipos de objetos que hemos introducido antes a los que se permite conectar el rol.

La herramienta que surge con el botón *Constraints* es la *herramienta de restricciones*. Se utiliza para especificar restricciones del tipo de grafo que no se pueden especificar con la herramienta de enlaces que acabamos de comentar.

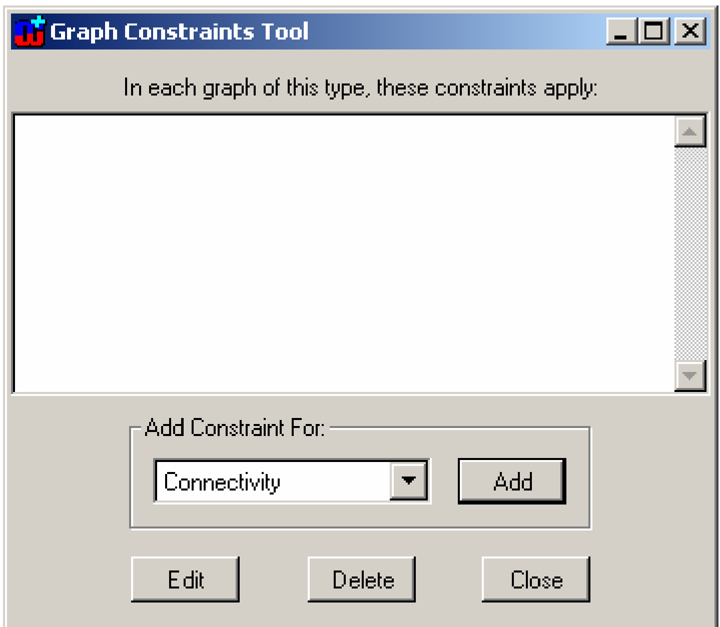


Figura 54: Definidor de restricciones

La ventana de la herramienta muestra las restricciones sobre el tipo de grafo que hay definidas, y permite la modificación (botón *Edit*), el borrado (botón *Delete*) y la creación de otras nuevas. Podemos incorporar restricciones de dos tipos: de conectividad (botón *Connectivity*) y de puertos (botón *Ports*). Las *restricciones de conectividad* se utilizan para limitar que un objeto tenga conectados a lo sumo cierta cantidad de roles o relaciones de cierto tipo. Las *restricciones de puertos* aseguran que todos los puertos de cierto tipo tengan el mismo o diferente valor para cierta propiedad.

Esta herramienta amplía el abanico de restricciones que se pueden definir sobre un modelo. No obstante, hay restricciones semánticas que no pueden cubrir los mecanismos de restricción vistos hasta ahora. Este es el caso de la restricción de evitar que cierto tipo de relaciones formen ciclos o la limitación de que no puedan existir en el grafo más de un determinado número de instancias de cierto tipo. Desde una perspectiva todavía más genérica, observamos que es realmente imposible ofrecer una herramienta de metamodelado que permita establecer cualquier restricción que se nos ocurra mediante un entorno gráfico, pues las posibilidades son infinitas. Es posible en la herramienta ofrecer la posibilidad de restringir ciertas posibilidades que sean frecuentes, como la que hemos citado de evitar los ciclos formados por determinada relación, pero nunca se cubrirán todas. La solución pasa por mecanismos que permitan un control más fino de los elementos del modelo, como por ejemplo, mediante código.

En el apartado de generación de código (apartado 5.5) trataremos las soluciones que nos aporta Metaedit+ para expresar restricciones más complejas, y cómo desatisfactoria es la solución ofertada.

Metaedit+ permite la posibilidad de manejar subgrafos. Existen dos posibilidades para tal fin:

Explosión: permite seleccionar un elemento del modelo y asociarle un nuevo modelo. Un elemento puede tener varias explosiones en un único modelo, y un conjunto de explosiones en cada modelo en que se use. La explosión se utiliza a menudo entre los modelos de diferentes lenguajes. Un ejemplo típico de explosión lo encontramos en los diagramas de UML, en los que a una clase se le puede asociar un diagrama de transición de estados para describir su comportamiento.
Descomposición: trabaja de la misma manera que la explosión, pero el origen de la descomposición solamente puede ser un tipo de objeto. La diferencia con la explosión es que ésta crea un enlace simple entre un elemento y un modelo, mientras que la descomposición también maneja las relaciones asociadas al elemento. A diferencia de la descomposición, la explosión permite conectar un mismo elemento de diseño a varios modelos, y diferentes enlaces para el mismo elemento utilizado en diferentes modelos, mientras que un objeto puede tener únicamente una descomposición, que es la misma sea donde quiera que se usa el objeto. A nivel de instancia, sólo se permite una descomposición de un objeto, y se aplica en todos los modelos que contienen el objeto. A nivel de tipo, cada tipo de objeto en un modelo puede tener varios posibles tipos de modelo en los que se puede descomponer legalmente. Un ejemplo lo podemos encontrar en el tutorial *Watch Example* que incorpora la documentación de ayuda de Metaedit+.

En la herramienta de grafos, el botón *Explosions* abre un diálogo en el que se expresa para cada tipo de relación, rol u objeto que se desee explotar, los tipos de grafos que se le pueden asociar. De manera semejante, mediante la opción *Descompositions* se abrirá un cuadro de diálogo en que se indicará lo mismo que antes, pero con la limitación de que solamente es aplicable a tipos de objetos.

El último de los botones que queda por comentar es el de *Reports* que abre el navegador de informes, del que comentaremos sus posibilidades más adelante, en la sección 6.5. Puede apreciarse que la herramienta de grafos permite crear lo que serán los tipos de grafos (metamodelos), esto es, los lenguajes gráficos. Por tanto podríamos utilizar el mecanismo de herencia que proporciona Metaedit+ para definir lenguajes formados por varios sublenguajes. Este es el caso de UML, en el que una misma notación se incluye diversos tipos diagramas (lenguajes gráficos). Los sublenguajes heredarían del lenguaje que los engloba, y esto se expresaría en la herramienta de grafos seleccionando como supertipo del grafo al lenguaje gráfico padre, en lugar de seleccionar el supertipo de los grafos, *Graph*. En la Figura 55 mostramos un ejemplo de lo comentado para el caso de UML, que se puede encontrar entre los proyectos del repositorio *demo*.

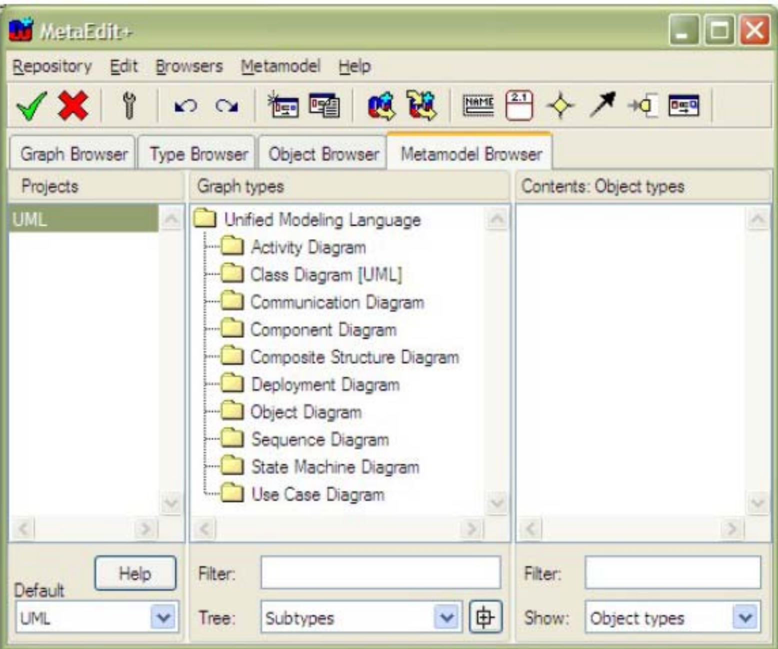


Figura 55: Jerarquía de lenguajes en UML

III. Editores de aspecto visual

Metaedit+ dispone de algunas herramientas que permiten cambiar el aspecto en que los modelos realizados siguiendo un metamodelo se pueden presentar a los usuarios, como los símbolos de los diagramas o los diálogos de propiedades. Aquí también podríamos englobar la generación de informes o código, pero dada su importancia y mayor complejidad, le dedicamos exclusivamente el apartado posterior a éste.

i. Editor de diálogos (Dialog Editor)

Las herramientas de objetos, relaciones, roles y grafos permiten abrir el editor de diálogos para modificar la disposición de los elementos del diálogo (*layout*) utilizados para editar las propiedades de una instancia de ese tipo. Mostramos la ventana de este editor en la Figura 56. En este diálogo se puede modificar la disposición gráfica del mismo, tanto de los botones como de las etiquetas, listas desplegables o cajas de texto de las propiedades de *Diagrama GRAFCET*, que son *Plataforma PLC*, *Nombre*, y *Documentation*.

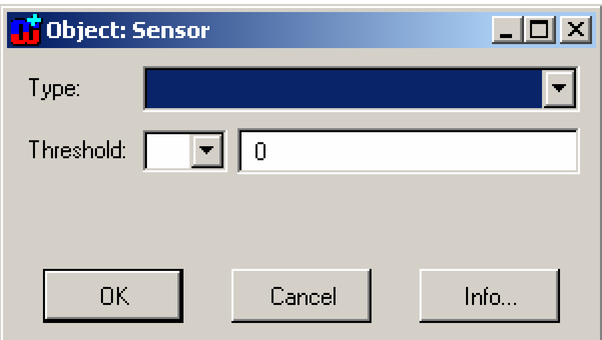


Figura 56: Editor de diálogos

El aspecto de este editor es casi idéntico al que aparece cuando se utiliza dicho diálogo. Una diferencia es que los valores por defecto de cuadros de texto y listas desplegables no aparecen en este diálogo. Cada componente gráfico (*widget*) del diálogo viene determinado por el tipo de datos y el tipo de componente gráfico de su tipo de propiedad. De esta forma, una cadena es caja de texto de una sola línea (véase la propiedad *Threshold* en la Figura 56), un texto se representa con un área de texto, una lista es una lista desplegable (véase la propiedad *Type* en la Figura 56), etc. Las etiquetas que aparecen son los nombres locales de las propiedades en ese tipo de objeto. El editor permite mover, cortar, pegar o eliminar las etiquetas, cajas de texto o listas desplegables y los botones que aparecen en el diálogo, pero no permite modificar el valor de las etiquetas, que aparecen como los nombres locales de las propiedades. Otros aspectos que se pueden personalizar de los diálogos son: se puede incorporar o suprimir una barra de desplazamiento horizontal o vertical, se pueden redimensionar los componentes gráficos del diálogo de forma que se mantengan unas dimensiones relativas a las dimensiones de la ventana, se permite alinear una serie de componentes gráficos de forma relativa entre ellos, especificar la separación (relativa o absoluta) del grupo de componentes al que se aplica la selección, ajustar el ancho y/o alto de los componentes para igualar los seleccionados, o especificar cuáles son las dimensiones

máximas y mínimas a las que se permitirá redimensionar el diálogo cuando se utilice, e incluso indicar el orden en que se seleccionarán los campos del diálogo cuando pulsemos la tecla del tabulador. También se puede alterar la fuente de forma independiente a la plataforma, pero los tipos son limitados y no se puede cambiar el tamaño o color.

ii. Editor de símbolos (Symbol Editor)

Este editor es una herramienta de dibujo especial para crear los símbolos gráficos con los que se representarán los tipos de objetos, roles, o relaciones en los editores de diagramas o de matrices. Por tanto, es la herramienta para especificar la sintaxis concreta de un lenguaje de modelado o DSL. Los símbolos se pueden construir con gran variedad de formas, colores y campos de texto. Los valores de las propiedades también se pueden mostrar en las cajas de texto, pudiendo modificarse la fuente y disposición del texto. Cada forma puede tener una condición ligada a ella, que determina si se muestra dependiendo del valor de una propiedad. Los símbolos de objetos y relaciones pueden definir áreas conectables que determinan cómo y dónde las líneas de los roles se unirán a ellos. Además, el editor de símbolos permite al usuario almacenar y recuperar símbolos de la *librería de símbolos* en el proyecto por defecto actual. La Figura 57 muestra el aspecto del editor de símbolos utilizado para crear el símbolo del tipo de objeto *Timer*.

En la ventana del editor distinguimos varias partes. La primera de ellas es una barra de menús con la que se pueden llevar a cabo todas las opciones disponibles en el editor, como grabar en la librería y exportar como bitmap con el menú *Symbol*; agrupar una serie de formas en una sola o modificar el formato de la forma con el menú *Edit*; alinear las figuras con el menú *Align*; mostrar y alinear a la rejilla con el menú *View*, o consultar la ayuda del editor con *Help*. La segunda parte es una barra de herramientas con las opciones más habituales a la hora de trabajar con los símbolos. El área siguiente es el lienzo sobre el que pintar los símbolos. La tercera barra permite indicar el estilo, color y grosor de la línea que forma el borde de la figura, y el color de relleno de ésta, de forma más rápida que recurriendo a menús. La última parte es la barra de estado que se encuentra en la parte inferior de la ventana y muestra el elemento activo del diagrama, el tamaño de la rejilla y el factor de zoom.

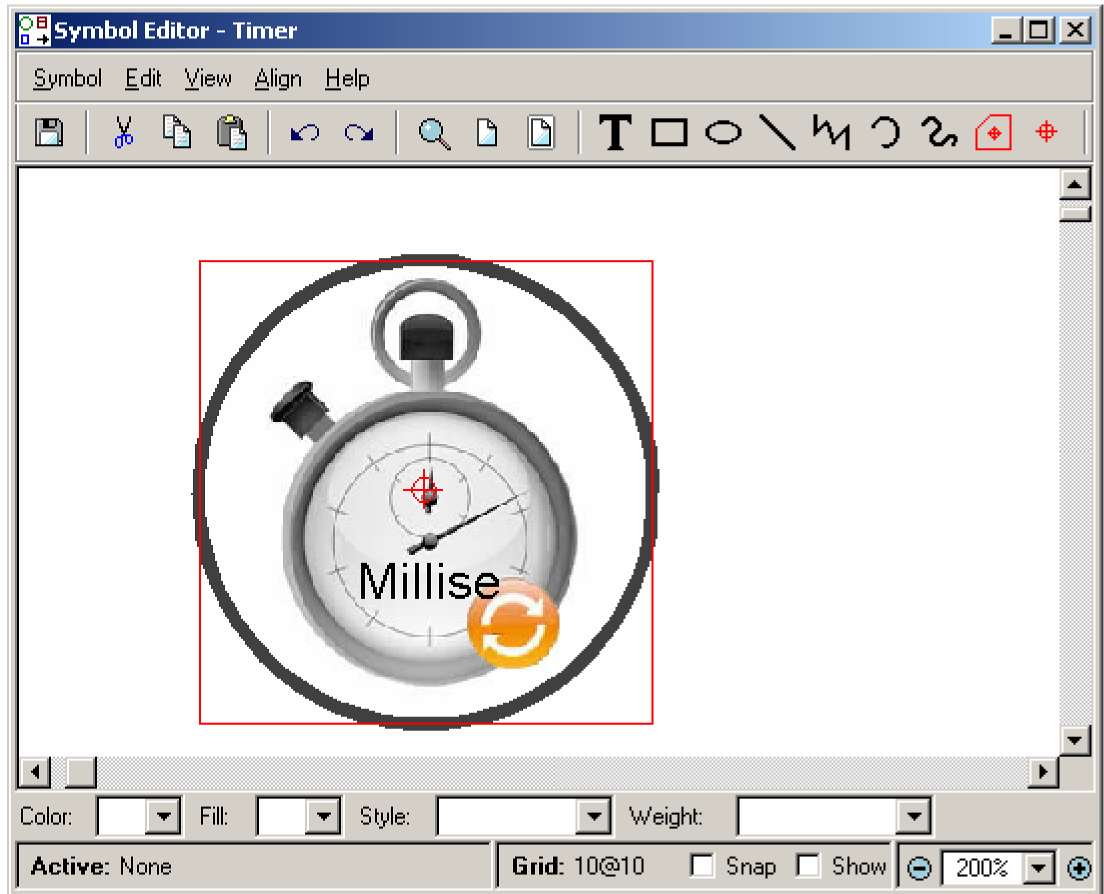


Figura 57: Editor de símbolos aplicado al objeto Timer

La barra de herramientas contiene como hemos comentado antes, las acciones más frecuentes a la hora de trabajar con los símbolos, y que será suficiente en muchos casos. Dentro de ésta, nos disponemos ahora a comentar los que corresponden a la edición de símbolos propiamente dicha, con el fin de adentrarnos en las posibilidades. Se puede ver la barra en la Figura 58.

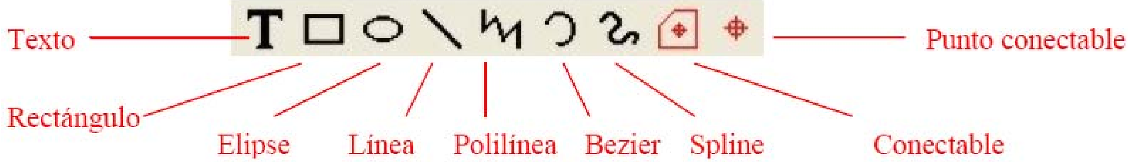


Figura 58: Barra de edición de símbolos

Las formas, líneas y curvas que se pueden utilizar para construir símbolos son:
Rectángulo: permite crear rectángulos (y cuadrados). Además, con el ratón se puede alterar para que los bordes no son sean ángulos rectos sino curvas, incluso indicando mayor o menor curvatura.
Elipse: para las elipses (y círculos), parábolas y otros arcos generados a partir de la abertura de una elipse.
Línea: para dibujar una línea recta.
Polilínea: se utiliza para crear figuras a partir de rectas cuyos extremos están unidos. Es más práctico para dibujar polígonos que hacerlo línea a línea.
Bezier: una *curva Bezier* es una única línea recta modificada para hacerla una curva basándose en dos puntos de control cuya posición se pueden alterar.

Spline: es una polilínea a la que se curvan las rectas que la componen para formar una curva. Por ejemplo, dibujando una uve se transformaría en una parábola con las ramas hacia arriba.

Para todas ellas, es posible modificar mediante el menú contextual: su tamaño y posición, el color y grosor del borde, y es posible especificar que se dibuje la figura solamente cuando la propiedad especificada tome determinado valor. Para los valores boléanos, *T* representa el valor verdadero y *F* el falso. Incluso se puede recurrir a comodines sencillos que son "#" para cualquier carácter y "*" para indicar cero o más caracteres. Los rectángulos y elipses además tienen la opción de especificar el color de relleno de la forma. Para las líneas y polilíneas también se puede indicar el estilo de las líneas.

El botón *Texto* permite introducir en las figuras compuestas texto que puede ser fijo, que corresponde al valor que toma alguna de sus propiedades, o la salida de un informe. El texto tiene borde y se le pueden modificar los atributos que comentábamos antes para las figuras, incluido el color de relleno. También se permite modificar el tipo de fuente, tamaño, color, alineación respecto del borde, el estilo (regular, cursiva, negrita, ambas) y el efecto (subrayado, tachado). El texto puede ser una de las propiedades del objeto, de modo que cuando se representa, dicho texto toma el valor que toma la propiedad especificada para la instancia que se muestra. Otra alternativa es que el texto sea la salida de un informe. En dicho informe se suprimen las palabras clave de inicio (*report <nombre_informe>*) y cierre de informe (*endreport*), y permite a los informes utilizar bucles para navegar desde el elemento actual (no propiedad) a otros objetos, relaciones, roles o puertos.

Los *conectables* son tipos especiales de elementos de los símbolos que definen áreas alrededor del símbolo a los que se pueden conectar los roles de las relaciones entrantes. Consisten en dos partes: el *punto de destino* que es la posición a la que apuntan las líneas de los roles, y el *borde conectable* que suele ser una polilínea roja alrededor de la figura en la que se conectan las relaciones al símbolo del objeto. Los bordes conectables son opcionales y si no se definen, las líneas de las relaciones llegan hasta el punto de destino del objeto. Estos bordes conectables hacen de tope a la hora de dibujar las relaciones, de modo que las líneas llegan hasta los *puntos conectables* de dicho borde. Se pueden incorporar varios puntos conectables a un mismo borde conectable, de forma que las relaciones se puedan enganchar al símbolo en más de un punto. Existe también la posibilidad de generar automáticamente un conectable a partir de una figura, de modo que se puede enganchar relaciones a los bordes más externos del símbolo. Cuando incorporamos a una figura más de un punto conectable (y es conveniente hacerlo aunque sólo se tenga uno), es necesario especificar un puerto ya definido para ligarlo a ese punto. Así podremos identificar posteriormente a qué puntos del objeto queremos que se puedan enganchar ciertas transiciones. Con todo ello, vemos que es posible para un símbolo creado por nosotros, definir concretamente en qué puntos (ligados a puertos) queremos que se puedan enganchar determinados roles de las transiciones, tarea que se llevaría a cabo con la herramienta de grafos.

Todo lo comentado anteriormente es válido para el editor de símbolos que surge al editar los símbolos de objetos y relaciones. Hay una restricción adicional en el caso del editor de símbolos para relaciones, que impide introducir más de un borde conectable y punto conectable en el borde. Cuando lo que intentamos dibujar es el símbolo de un rol para el extremo de una relación, el editor presenta una pequeña variación que facilita la tarea. Veamos el aspecto que presenta en la Figura 59.

Como vemos, en la parte en la que solemos dibujar el símbolo aparece una línea de la que solamente vemos un extremo, que representa al extremo de la relación al que

se aplicaría el rol que estamos editando. Por tanto, en este caso lo que tenemos que hacer es dibujar el símbolo en base a dicho extremo, por ejemplo pintando una punta de flecha en el extremo visible de la recta. Apréciese de que aquí no se tiene la opción de indicar bordes conectables o puntos de conexión como ocurría con los objetos o relaciones.

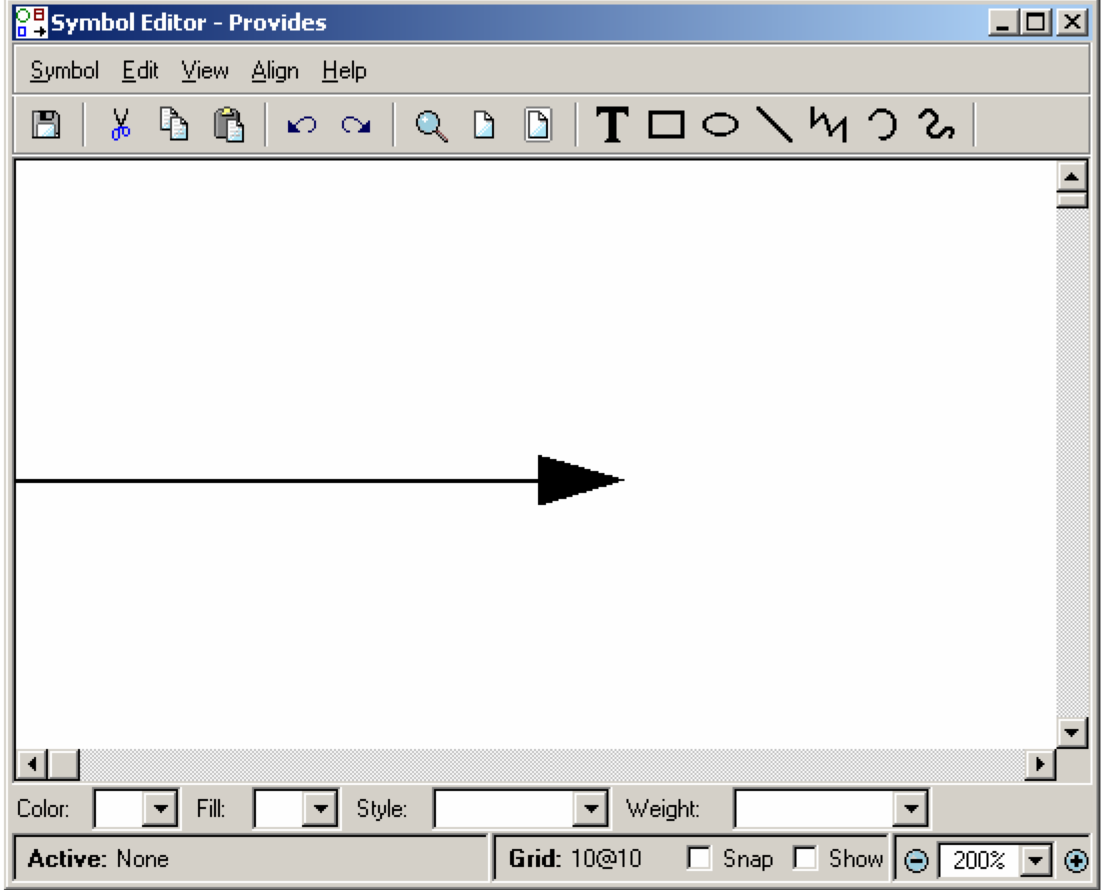


Figura 59: Editor de símbolos para el rol "Provides"

IV. Editores de los modelos

Estos editores son el principal medio para crear, mostrar y editar la información de los modelos. Vamos a presentar los tres editores que incluye Metaedit+, que son el *editor de diagrama*, el *editor de matriz*, y el *editor de tabla*. Cada editor es conforme con el metamodelo elegido y muestra los modelos diseñados. Esos modelos se denominan *grafos*, y hay tres maneras de mostrar un grafo, que se corresponden con los tres editores de que se disponen. Las instancias de los modelos se crearán utilizando alguno de estos editores, siendo posible incluso que un mismo modelo se visualice con más de un editor, comportándose así como si tuviésemos varias vistas del mismo modelo.

i. Editor de diagramas (Diagram Editor)

El editor de diagramas es una herramienta para crear, gestionar y mantener grafos (modelos) como diagramas. Con un editor de diagramas se puede visualizar y editar modelos, así como realizar *explosiones* y *descomposiciones* entre varios grafos. Excepto por la notación del metamodelo seleccionado, la herramienta siempre tiene un aspecto idéntico. A continuación se incluye una figura del editor de diagrama de un ejemplo de modelo de proceso de valor (*Model Process Value*). (Véase Figura 60) La ventana del editor de diagramas se divide en cuatro partes. La primera es la barra de menú que permite realizar todas las acciones habituales sobre los diagramas, como pueden ser crear un nuevo diagrama, copiar un elemento, hacer zoom del diagrama, o insertar una nueva relación de uno de los tipos existentes. La segunda parte es la de las barras de herramientas (Véase Figura 61). Se compone de la barra de acciones para realizar una serie de acciones de las más útiles a la hora de crear los diagramas, la barra de tipos de objetos para crear instancias de algún tipo de objetos (recuérdese que son los metatipos del metamodelo), y la barra de tipos de relaciones para instanciar tipos de relaciones del metamodelo. La primera de las barras es común a todos los lenguajes, mientras que las otras dos, como es lógico, dependen del lenguaje de modelado elegido. Todas estas acciones se pueden realizar también mediante la barra de menú.

La tercera parte es el área de dibujo, que es donde se representan las instancias de los elementos del metamodelo. La última parte es la barra de estado que se encuentra en la parte inferior de la ventana y muestra: el elemento activo del diagrama, sus subgrafos, el tamaño de la rejilla y el factor de zoom.

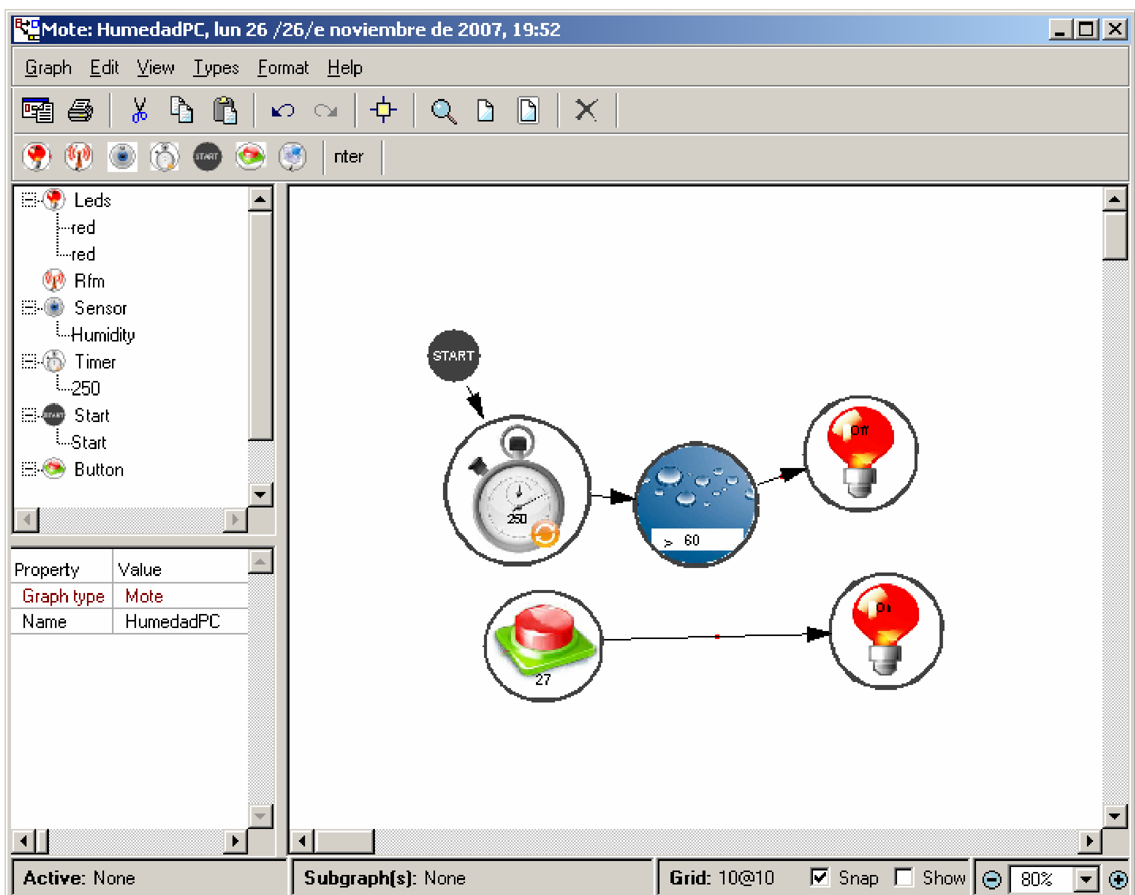


Figura 60: Editor de diagrama para un caso de modelo de mote

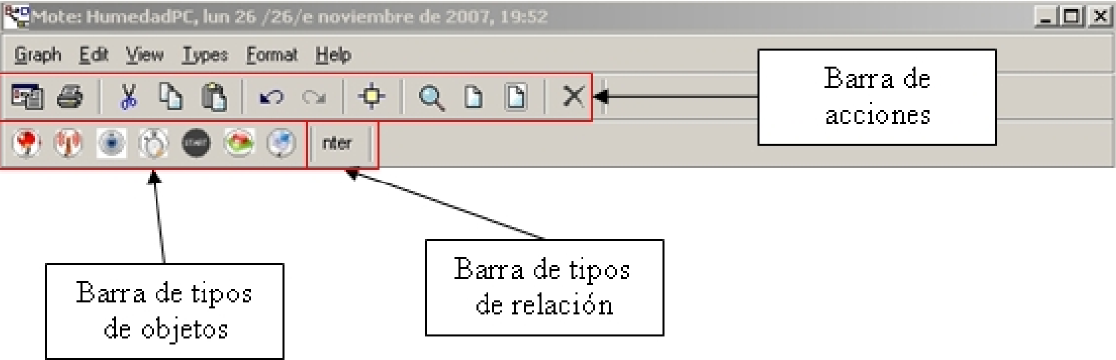


Figura 61: Barra de herramientas del editor de diagramas

ii. Editor de matrices (Matrix Editor)

Permite manejar los modelos (grafos) representándolos como matrices con dos ejes. Cada elemento, tanto si está en una celda como en un eje, dispone de diálogos para

añadir, visualizar y editar la información del elemento. Las modificaciones realizadas en un elemento mediante esta vista, se almacenan en el repositorio y se reflejan en otras herramientas. El editor de matrices es capaz de representar y editar cualquier modelo de cualquier lenguaje, incluso si el modelo se creó originalmente con el editor de diagramas o el de tablas. Se puede utilizar para mostrar vistas de los modelos que originalmente se crearon como diagramas gráficos, como por ejemplo los flujos de datos, y trabajar con los metamodelos específicos basados en matrices, como la planificación de sistemas de negocios, que se basa casi totalmente en matrices. Este editor ofrece funciones especiales necesarias para trabajar con matrices basadas en grafos, como la diagonalización, descomposición en subsistemas y visualización. La Figura 62 muestra una vista de un editor de matrices para un caso de ventas e inventario.

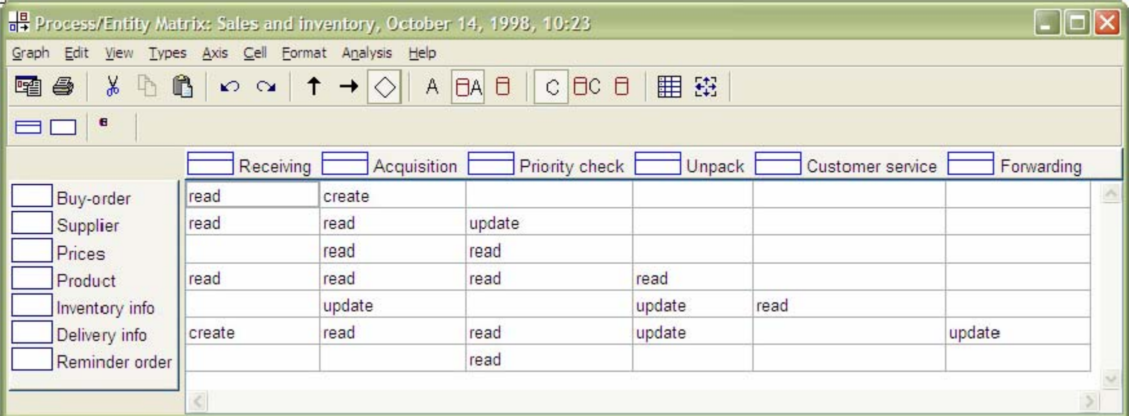


Figura 62: Vista del editor de matrices para un caso de ventas e inventario

La ventana del editor de matrices consiste en tres partes: la barra de menú, el área de barras de herramientas, y la matriz propiamente dicha. Las dos primeras son similares a las del editor anterior. El *área de la matriz* consiste en un eje horizontal, un eje vertical y una matriz entre ellos. Los ejes contienen la representación de los objetos, y cada celda muestra las relaciones binarias o roles entre los objetos correspondientes en los ejes. Dado que una matriz simplemente tiene dos ejes, sólo se pueden representar relaciones con dos roles, mientras que las relaciones n-arias y sus roles no se podrían mostrar.

Al tratarse de una matriz, disponemos en la barra de acciones de opciones para visualizar los datos de la matriz de la forma más idónea dependiendo del caso. Dependiendo de la representación de la matriz podríamos desear que se mostrase la relación o uno de los roles en una celda: el rol ligado al objeto de la columna de la celda, o el ligado a la fila de la celda. Esto se hace con los botones de opciones de visualización de relaciones y roles. Si el DSL que utilizamos tiene símbolos gráficos, podemos elegir si mostrar el texto del objeto, el símbolo o ambos. Esto se hace con los botones de mostrar/ocultar los símbolos de los objetos de los ejes. Lo mismo podríamos realizar en el caso de los símbolos de relaciones/roles en las celdas con los botones para mostrar/ocultar los símbolos de las relaciones o roles de las celdas. Todos estos botones se muestran en la figura adjunta (Véase Figura 63).

El tipo de texto mostrado por cada rol o relación se puede modificar, permitiéndose mostrar sólo el nombre, sólo el tipo, ambos, la primera letra del tipo, o simplemente una X. Otras opciones permitidas son cambiar los ejes, ocultar columnas, ocultar relaciones duplicadas, cambiar el tamaño de columnas y filas, o cambiar las fuentes.

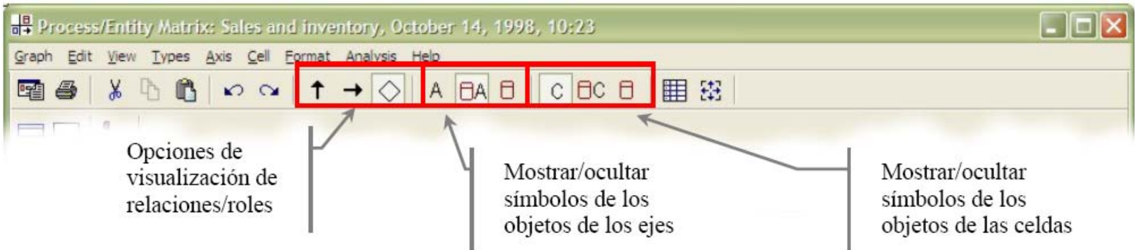


Figura 63: Barra de herramientas del editor de matrices

Otras operaciones de formateo son la diagonalización, que aplicado al eje seleccionado reorganiza los elementos de modo que las relaciones entre elementos forman una diagonal, y la ordenación de los elementos en los ejes vertical u horizontal siguiendo cierto criterio, como puede ser alfabéticamente.

iii. Editor de tablas (Table Editor)

Es el editor más simple de los tres. Proporciona una vista de tabla de los objetos de un tipo dado en un grafo. Los objetos del grafo se representan como filas y las propiedades son las columnas. De esta forma se dispone de una manera clara y compacta toda la información de los objetos del grafo. Se pueden visualizar y editar propiedades de los objetos así como realizar o explorar descomposiciones y explosiones entre varios grafos. Especificaciones de requisitos o listas de problemas no son adecuados para representarlos en diagramas o matrices (excepto las matrices de trazabilidad de los requisitos), y la forma idónea es mediante una tabla. La Figura 64 muestra el editor de tablas para un sistema de inventario.

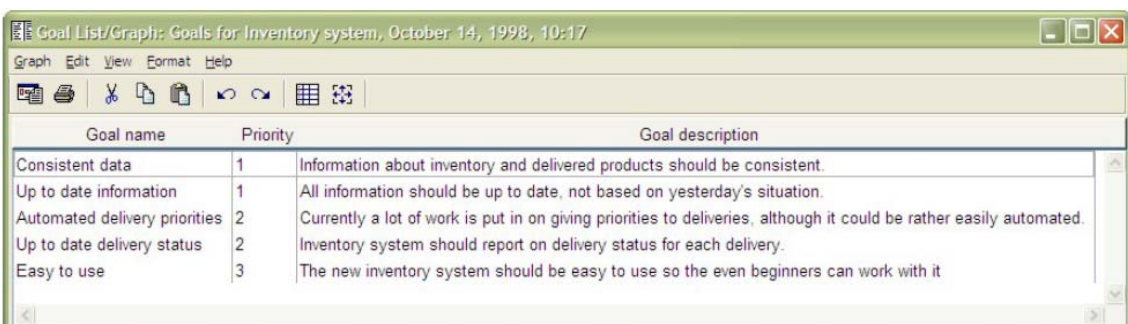


Figura 64: Editor de tablas reflejando las metas de un sistema de inventario

La ventana de los editores de tablas se compone de una barra de menú, la barra de herramientas y la tabla en sí, siendo este último donde se representa la información de las propiedades de los objetos. En este caso, la barra de herramientas es más pequeña que en los editores anteriores y sólo consta de una barra de acciones sencilla. Entre las acciones permitidas para las tablas se encuentran las de visualización y formato, como seleccionar las propiedades de los objetos que se mostrarán, ordenar la tabla hasta por tres criterios de ordenación simultáneamente, modificar el ancho de las columnas, autoajustar, mostrar la tabla completa en la ventana, o modificar las fuentes.

V. Informes y generación de código

En este apartado analizaremos las facilidades que proporciona Metaedit+ para generar documentación o código de algún tipo a partir de un modelo, o para escribir código que realice alguna comprobación sobre un modelo. En Metaedit+, estas tareas se realizan por medio de los *informes* (*reports*). Los informes, al ejecutarse, acceden a información depositada en el repositorio y la transforman en salidas basadas en texto.

i. Informes independientes del DSL (predefinidos)

Metaedit+ dispone de algunos informes predefinidos que pueden ser útiles en numerosas situaciones, y que son independientes del lenguaje, con el fin de evitar tener que definir los nuestros propios. Son los que siguen:

- **Lista de objetos (*Object List*)**: ofrece una lista que describe los objetos incluidos en los grafos. Muestra el identificador de los objetos especificando al lado de cada identificador el tipo de objeto. Se genera en una ventana de salida y puede guardarse como texto y otros formatos.
- **Lista de propiedades (*Property List*)**: es parecido al informe de lista de objetos, pero incluye además el nombre del tipo de la propiedad y el valor de ésta para todas las propiedades de los objetos enumerados.
- **Diccionario (*Dictionary*)**: para imprimir todos los objetos con documentación y definiciones relacionados (sacados del campo de documentación del objeto). Es similar a la lista de objetos, pero además incluye el valor de la propiedad "Documentation" (nombre local de la propiedad) para los objetos que la poseen.
- **Conexiones de objetos (*Object Connections*)**: basado en la lista de objetos, pero que además indica para cada objeto en qué relaciones toma parte. Para cada una especifica el identificador, el tipo de relación, el identificador del objeto destino y su tipo. Si algún identificador de relación estuviese vacío, indicaría como si fuese el identificador el nombre del tipo de relación.
- **Comprobaciones (*Checkings*)**: lista los objetos que no tienen relaciones con otros objetos (huérfanos), y cualquier propiedad vacía en los objetos. Al igual que en informes anteriores, se puede guardar (o imprimir) como texto o en formatos.
- **Exportar el modelo a HTML (*Export graph to HTML*)**: produce un fichero HTML para ser interpretado con un navegador. Muestra cualquier representación del grafo como una imagen GIF a la que se puede hacer clic con el botón izquierdo para ir a la definición del elemento. Más abajo aparece un diccionario de los elementos del grafo, de forma que aparece una lista de los identificadores de los objetos, su tipo y su documentación asociada. Debajo aparecen una serie de tablas para cada objeto en las que se especifica el identificador del objeto, su tipo, los tipos de propiedades que tiene y su valor, las relaciones en que toma parte incluyendo: roles, tipo de relación e identificador del objeto del extremo, y las explosiones o descomposiciones en que toma parte. El fichero generado es una página web que puede ser tratada como tal.
- **Exportar el modelo a Microsoft Word (*Export graph to Word*)**: incluye la misma información de diseño que el informe HTML, pero produce el fichero (con extensión *.rtf*, o *.doc*) para Microsoft Word. El informe usa una plantilla de Word denominada *Metaedit+.dot*. Por defecto, este fichero de plantilla se

localiza en el subdirectorio *reports* bajo el directorio raíz de la aplicación Metaedit+. El diagrama se representa como una imagen vectorial editable. Esta opción, además de generar un informe con la misma información que el HTML, incluye una portada muy sencilla para el informe.

- **XML:** genera un fichero XML con la información de los objetos y sus propiedades, pero no incluye datos relativos a relaciones. Además de la posibilidad de documentar individualmente los grafos, también se puede generar documentación para el proyecto completo. Con este propósito, los grafos que deseamos documentar en el proyecto deben ser definidos como un *modelo de proyecto* (*Project Model*). El modelo de proyecto es una técnica de modelado específica para ilustrar qué grafos pertenecen a cierto proyecto, las relaciones entre grafos, cualquier subproyecto del proyecto. Los subproyectos se pueden descomponer en otros modelos de proyectos. Los modelos del *modelo de proyecto* no se crean automáticamente, sino que se deben mantener manualmente. Se puede disponer de varios modelos de proyecto para generar diferentes conjuntos de documentación. La documentación del proyecto se puede construir en formato HTML o documento de Word. Incluye: nombre de proyecto y descripción, estado actual, gestor del proyecto, personal involucrado, detalles de los contactos, imágenes de los diagramas, un diccionario de los grafos que pertenecen al proyecto, además de la documentación de cada uno de los grafos que en el caso de construirse en formato HTML se generarían en páginas web distintas y en caso de fichero de Word se generaría como parte del fichero del proyecto completo.

ii. Informes dependientes del lenguaje: el navegador de informes

Cuando necesitamos informes más complejos que simplemente información de objetos, sus propiedades y relaciones, es necesario construir informes por nosotros mismos, tarea que se lleva a cabo con la herramienta conocida como el *navegador de informes* (*report browser*). El navegador de informes es un editor de textos especial para crear, editar y gestionar los informes. Permite consultar, editar y ejecutar las definiciones de informes disponibles, así como crear nuevos informes según nuestras necesidades. Véase en la Figura 65 una vista del generador de informes.

Consta de seis partes. La primera de ellas es la barra de menús en la que se encuentran las opciones de que se dispone para manejar los informes tales como crear un nuevo informe, grabar en un fichero uno dado o buscar palabras en el seleccionado. La barra de herramientas ofrece la posibilidad de acceder de forma más directa a las utilidades más frecuentes que se encuentran en los menús. El *área de informes* (*Report box*) contiene una lista de los informes disponibles para diagrama activo, de la cuál se puede seleccionar uno para editarlo. Las partes del *área de conceptos* (*Concept box*) y la del *área de elección* (*Choice box*) están relacionadas. En la primera elegimos la categoría y en la segunda nos aparece el listado de instancias o comandos de esa categoría. Así se nos permite acceder al nombre y propiedades de las instancias de grafos, objetos, relaciones, puertos y roles definidos en el modelo. También nos ofrece la sintaxis del lenguaje de informes propio de Metaedit+ que será descrito en el siguiente apartado. De esta forma siempre podremos consultar los comandos e instancias, lo que supone una ayuda considerable para escribir el código de los informes. Por último, el *área de edición* es la parte en la que escribimos el informe, ya

sea completamente manual o bien ayudándonos en ocasiones con el área de conceptos y elección.

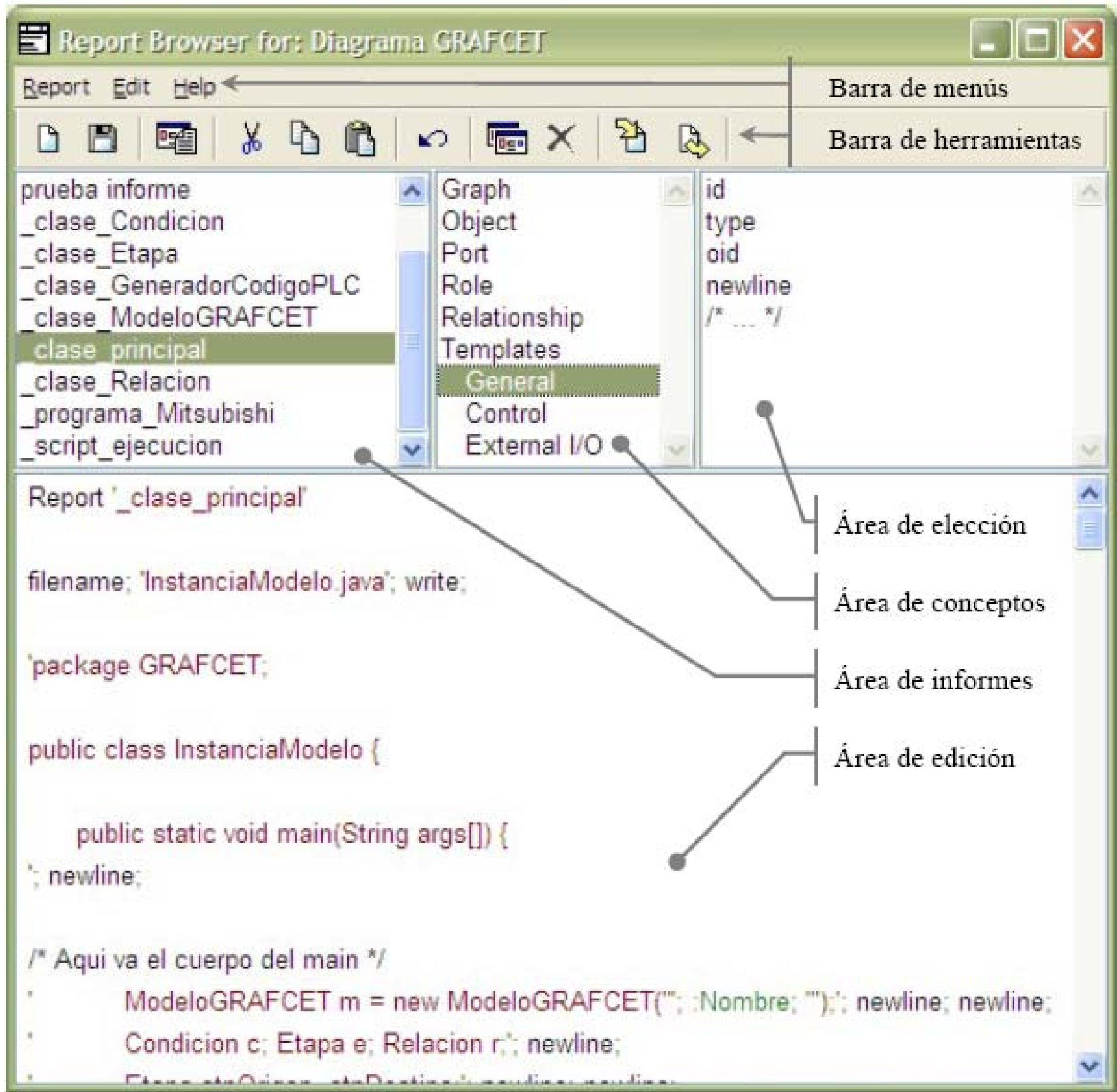


Figura 65: Navegador de informes

Cuando nos encontramos en la tesitura de escribir informes, puede ocurrir que escribir todo el código en un único y extenso informe sea ilegible y en cierto modo costoso de mantener, o que necesitemos generar diversos ficheros de código del mismo o diferente tipo. En estos y otros casos la opción más conveniente es fragmentar el informe en sub-informes. Para ejecutar otro informe desde uno dado se recurre al comando **subreport**. Los sub-informes se ejecutan como si todo el contenido de la definición del sub-informe estuviese incluido en la definición del *informe padre* (informe que lo invoca). Los sub-informes heredan el mismo conjunto de ficheros abiertos, la misma pila de informes que el informe padre. Existe un modo de definir informes que no están disponibles en la herramienta de diagramas a la hora de utilizar nuestro lenguaje gráfico, que consiste en que el nombre del informe vaya precedido de "_". Esto solamente debe usarse con sub-informes que son invocados por otros que se dividen por su complejidad, y que no interesa que el usuario los pueda ejecutar de forma independiente. De todos modos, existe la posibilidad de verlos todos presionando la tecla *SHIFT* mientras se va a consultar la lista para ejecutar uno. Crear nuestros propios

informes permite tener ventajas propias de los principios orientados a objetos, como la herencia. Si nuestros informes siguen enlaces de modelos a un tipo de modelo diferente que engloba al anterior, lo mejor sería crear un sub-informe en ese tipo de grafo, en vez de definirlo en el original. De forma similar, si un nuevo informe creado se puede aplicar a más de un lenguaje, no hay necesidad de duplicarlo para cada uno de ellos, sino que simplemente se define en un supertipo común, o en el supertipo de todos los lenguajes (grafos), en el supertipo *Graph*, lo que sería similar al principio de herencia.

El navegador de informes es el entorno integrado que Metaedit+ nos ofrece escribir informes que generan código o documentación, campos de texto basados en informes en los símbolos, identificadores de informe para elementos del grafo que no sean propiedades, o comprobación de restricciones, pero la piedra angular de los informes es el código que se escribe en el área de edición. Los informes para los campos de texto en los símbolos se introduce en el editor de símbolos, y los informes para los identificadores de los elementos que no son propiedades se introducen en las herramientas de metamodelado, mientras que la generación de documentación, código o restricciones se hace con el navegador de informes. En el siguiente apartado vamos a analizar el lenguaje que nos ofrece Metaedit+ para escribir los informes, y lo trataremos desde el punto de vista del navegador de informes.

iii. El lenguaje para escribir informes

Se trata de un lenguaje propietario, diseñado especialmente para la redacción de informes, cuya ejecución genera algún tipo de salida en base al tratamiento realizado sobre los elementos del modelo. No es por tanto un lenguaje de propósito general, sino un lenguaje sencillo y relativamente potente. Un informe se inicia con Report <nombre_informe> y se finaliza con la sentencia endreport. Cada comando del informe debe acabar en un ";", excepto la construcción foreach. Se permiten comentarios que comienzan en "/*" y finalizan en "*/", y que pueden ocupar varias líneas de texto. Las cadenas de texto deben comenzar y finalizar en el carácter "'".

Para referenciar a los elementos del grafo, se utiliza el nombre del elemento (que podría componerse de varias palabras) precedido de cierto carácter y seguido de ";" o fin de línea. Las propiedades van precedidas de ":", los objetos de ".", las relaciones de ">", los roles de "~", y los puertos de "#". En este lenguaje no hay soporte para variables. Cuando nos encontramos dentro de un bucle anidado y deseamos acceder a propiedades del elemento que estamos recorriendo en otro bucle externo, podemos acceder a dichas propiedades especificando el número de bucles externos en que se buscará la propiedad. Por ejemplo, :Documentation; 1; buscará la propiedad cuyo nombre local es *Documentation* en el bucle inmediatamente externo al que nos encontramos. Esto se puede utilizar también con id, oid, type. No es posible navegar a un objeto, relación o rol a partir de un puerto, sino que debe navegarse primero al rol, y a partir de ahí obtener la información del puerto, y del objeto ligado. Todas las instancias del metatipo (objeto, propiedad, relación, rol) pueden accederse con "()" en vez de con el nombre del tipo de propiedad, lo que es útil para recorrer instancias de un metatipo específico, pero del que no se desea especificar el nombre de un tipo concreto. Entre los paréntesis se pueden indicar patrones sencillos para los nombres de tipos, separándolos por una barra ("|") en caso de desearse especificar varios. Como ejemplo, podríamos considerar foreach (Etapa*) {…}, que nos serviría para acceder a todas las instancias del modelo cuyo nombre de tipo comenzase por *Etapa* como podrían ser *Etapa* y *Etapa Inicial*. Para podernos hacer una idea más precisa de la potencia del lenguaje, vamos a mostrar una tabla en la que aparecen las palabras clave del lenguaje.

Palabra reservada	Descripción
report '<nombre>'	Token de inicio del informe.
endreport	Token de final del informe.
newline	Introduce un salto a la siguiente línea en el informe de salida.
foreach <tipo_arg> { <cuerpo> }	Itera sobre los elementos del modelo especificados por <tipo_arg>, ejecutando <cuerpo> para cada uno. Solamente puede aplicarse a modelos (grafos).
do <tipo_arg> { <cuerpo> }	Navega a los elementos especificados en <tipo_arg> que están relacionados con el actual objeto, relación, o rol, y ejecuta <cuerpo> para cada uno de ellos.
dowhile <tipo_arg> { <cuerpo> }	Similar a *do* pero omite la última cadena de texto fijo del cuerpo en la iteración final.
if <condición> then <cuerpo_then> else <cuerpo_else> endif	Si se cumple <condición> ejecuta los comandos en <cuerpo_then> y en otro caso ejecuta las órdenes de <cuerpo_else>. <condición> puede contener "=" (igual), "=~" (igual, soporta comodines) o "<>" (distinto).
descompositions	Usado como argumento de un bucle *do* permite navegar al modelo de descomposición asociado al objeto actual.
explosions	Usado como argumento de un bucle *do* permite navegar a los grafos de explosión asociado al objeto actual.
id	Escribe en la salida el identificador del elemento actual.
type	Escribe en la salida el tipo del elemento actual.
oid	Escribe en la salida el identificador de objeto único del elemento actual.
subreport; <nombre informe>; run;	Ejecuta el informe cuyo nombre es igual a <nombre_informe>, con la pila de informes y salida de informe actual.
external; <comando>; execute;	Ejecuta un comando externo definido por <comando>, como si se fuese a ejecutar en la línea de comandos.
external; <comando>; executeBlocking;	Similar al anterior, pero mantiene la ejecución del informe hasta que el comando externo se haya completado.
filename; <nombre>; write; <cuerpo>; close;	Escribe en el fichero denominado <nombre> la salida del informe generada por los comandos en <cuerpo>, sobrescribiendo el fichero si ya existía.
filename; <nombre>; append; <cuerpo>; close;	Escribe en el fichero denominado <nombre> la salida del informe generada por los comandos en <cuerpo>, añadiéndolos al final del fichero si existe.
filename; <nombre>; read;	Añade el contenido del fichero denominado <nombre> a la salida del informe.
filename; <nombre>; print;	Exporta el diagrama del modelo actual en un fichero denominado <nombre> o en la impresora. <nombre> debe ser "lpt1" para la impresora, o tener una extensión *.gif* o *.pct* para generarlo en bitmap o gráfico vectorial respectivamente.
sep	Escribe en la salida el carácter separador de nombres de rutas, como "\" en Windows y "/" en Unix.
prompt; <prompt_arg>; ask;	Muestra un diálogo cuyo mensaje es <prompt_arg> y que solicita una cadena de texto que se escribe en la salida del informe actual.

Figura 66: Tabla de palabras reservadas

En la tabla que hemos mostrado se han incluido todas las palabras reservadas del propio lenguaje y como hemos podido comprobar, se trata de un lenguaje sencillo y que se ha diseñado ad hoc para la generación de informes. Posee algunas características avanzadas que pueden ser útiles en algunos casos, como la de mostrar un diálogo que solicita una cadena que se introduce en la salida (comando prompt), o la posibilidad de exportar la imagen del diagrama en un fichero GIF o PCT mediante el comando filename; <nombre>; print;.

Con el objetivo de mostrar el uso de las sentencias más habituales al programar informes, incluimos a continuación un breve código de ejemplo. Vamos a suponer que se dispone del tipo de objetos *Persona*, con un atributo *Nombre* indicado como

identificador y un atributo *Profesión*, y que se relaciona mediante la relación *Posesión* con objetos de tipo *Cuenta Bancaria*, que tienen un atributo *Número Cuenta* que es el identificador. La relación *Posesión* es uno a uno, y sus dos roles son *Titular* y *Cuenta*.

```
Report 'Ejemplo_Persona'
foreach .Persona{
      oid; ': '; id; ' , '; :Profesión;
      do ~Titular>Posesión~Cuenta.()
      {
            ' # '; id;
      }
      newline;
}
endreport
```

El informe anterior recorre todas las instancias del tipo *Persona* y para cada una de ellas muestra el identificador de objeto (*OID*), el valor de la propiedad *Nombre* (que es el identificador del tipo *Persona*), el valor de la propiedad *Profesión*, y el número de cuenta (que es el identificador de *Cuenta Bancaria*). Un ejemplo de la ejecución de este código podría ser:

```
      28-4348: Pedro Gallego , Contable # 3055-0143-91-3519795217
      28-4352: Remedios Pérez , Secretaria # 3055-9232-91-7651844172
```

Pero a pesar de estas características avanzadas y estar diseñado para generar código, no es todo lo potente que podríamos desear. Las sentencias de iteración do y foreach son el núcleo del lenguaje para navegar a través de los elementos, pero dado un elemento actual no permiten ir más allá de los elementos relacionados a éste. Si nos encontramos navegando cierto objeto, no es posible obtener información más allá de los objetos conectados por relaciones directas a éste. Para poder navegar a partir de los elementos relacionados a partir del objeto que antes comentábamos, habría que introducir un bucle anidado que recorriese los objetos a que esta relacionado. El problema no se halla en dichas construcciones, que sí son adecuadas. El problema que surge es que no hay otra manera de navegar por los elementos, es decir, no disponemos de funciones, recursividad y variables que en ocasiones se convierten en la manera fundamental de generar código. Si deseásemos navegar un grafo partiendo de un objeto y realizar un recorrido para acabar en otro distinto, no podríamos sin conocer el grafo de antemano, pues la solución sería anidar tantos bucles como longitud del camino (menos 1) tuviésemos a partir del objeto origen. Si pudiésemos implementar funciones recursivas, podríamos hacer que recursivamente se fuesen visitando los nodos vecinos de uno dado. Otra alternativa sería hacerlo iterativamente, pero sería necesario poder disponer de variables, de forma que habría una que contendría el nodo actual que se está navegando.

Intentaremos aclarar más lo comentado anteriormente. La Figura 67 muestra un ejemplo de máquina de estados en la que existe un ciclo. Para simplificar el ejemplo y que sea más claro, suponemos que solamente puede partir una única transición de cualquier etapa.

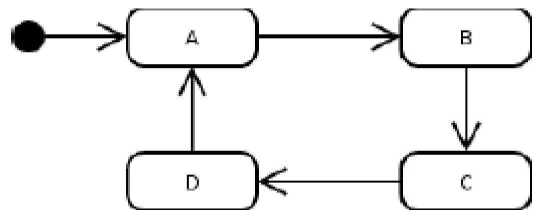


Figura 67: Ejemplo de ciclo en una máquina de estados

Si pensamos en cómo se podría comprobar la existencia de ciclos expresada en pseudocódigo, de modo que podría ser trasladado a cualquier lenguaje de propósito general como Java o C, se nos puede ocurrir algo parecido al siguiente código.

```
estado_actual = get_estado_inicial());
mientras (exista_transición(estado_actual)){
      estado_actual = transitar(estado_actual);
      si (estado_actual == get_estado_inicial()) devuelve verdadero;
}
devuelve falso;
```

El código se introduciría en una subrutina que devolvería verdadero en caso de encontrar un ciclo y falso en caso contrario. Se ha optado por un enfoque iterativo, pero de igual modo se podría haber resuelto mediante un programa recursivo. Veamos ahora cuál sería la manera de comprobar en Metaedit+ que existe un ciclo.

```
Report 'comprobar ciclos'
foreach .EstadoInicial{
      do ~OrigenTransición~DestinoTransición.(){
            /* Ahora estamos en el estado A */
            if type = 'Estado' then 'Hay ciclos'; endif;
            do ~OrigenTransición~DestinoTransición.(){
                  /* Ahora estamos en el estado B */
                  if type = 'Estado' then 'Hay ciclos'; endif;
                  do ~OrigenTransición~DestinoTransición.(){
                        /* Ahora estamos en el estado C */
                        if type = 'Estado' then 'Hay ciclos'; endif;
                        do ~OrigenTransición~DestinoTransición.(){
                              /* Ahora estamos en el estado D */
                              if type = 'Estado' then 'Hay ciclos'; endif;
                              do ~OrigenTransición~DestinoTransición.(){
                                    /* Ahora estamos en el estado A */
                                    if type = 'Estado' then 'Hay ciclos';
                                    endif;
                              }
                        }
                  }
            }
      }
}
endreport
```

Claramente vemos que este código funcionará correctamente porque se ha escrito conociendo el diagrama sobre el que se va a aplicar, pero no funcionaría para cualquier diagrama. Este es el problema del lenguaje de informes de Metaedit+, que al no disponer ni de variables ni de funciones y recursividad, es imposible escribir un programa que de forma genérica comprobase los ciclos de cualquier máquina de estados del modo en que lo expresamos antes en pseudocódigo. Por tanto, en el lenguaje de informes proporcionado por Metaedit+ no permite generar código del que se necesita

navegar los elementos en cierto orden. Comprobaciones tan útiles como asegurarse de la no existencia de ciclos en el grafo es imposible realizarlo para cualquier grafo. La solución que nos aporta el framework no es otra que recurrir a herramientas externas que nos cubran las carencias que esta herramienta tiene. Existe un comando (orden external) que permite ejecutar sentencias de línea de comandos, de forma que el sistema operativo se encarga de ejecutarlos. De este modo, sería posible ejecutar compiladores, intérpretes u otros programas que a partir de una entrada que les proporcionamos, nos generarían la salida que Metaedit+ no puede proporcionar.

En apartados anteriores citábamos que el lenguaje de informes que se utilizaba en el navegador de informes era el modo de generar documentación, código e informes de comprobación. Nos queda puntualizar algunos detalles de ésta última posibilidad. Las restricciones sintácticas se refieren a qué elementos se pueden conectar con qué otros, en qué medida y en qué puertos es posible especificarlas con la herramienta de grafos.

Hay algunas restricciones semánticas como pueden ser la de asegurar que el campo que usamos de identificador de algún tipo de objetos sea único, que se pueden especificar en el entorno gráfico del framework. Pero es imposible disponer de un medio gráfico de especificar restricciones semánticas que se aplican a todo el grafo, porque cada lenguaje de modelado es diferente y no está sujeto a normas.

Una manera viable de incorporar estas restricciones a nuestro modelo es especificarlas mediante código. En Metaedit+, utilizamos el lenguaje de informes también para la especificación de restricciones semánticas que no es posible indicar con las herramientas gráficas de que dispone (son muy pocas las que se pueden indicar de esta manera). Pero no es posible especificar código asociado a un grafo y que se ejecute directamente cuando lo estamos editando o cuando pulsamos algún botón para tal fin. La manera que disponemos de hacerla es ejecutar un informe escrito expresamente para comprobar restricciones del grafo. En este informe deberíamos escribir en la salida estándar (o en un fichero) las violaciones semánticas que se encuentran o bien podríamos recurrir a la orden prompt, de forma que cada vez que se halla alguna incorrección, surja una ventana indicándonoslo. Esta última manera solicita una cadena que aparecerá en la salida estándar. Aún con todo ello, nos volvemos a encontrar con las mismas limitaciones del lenguaje de informes que indicábamos anteriormente. Hay restricciones semánticas que no es posible comprobar con éste lenguaje, como puede ser la comprobación de inexistencia de ciclos, y cuya solución es recurrir a un lenguaje externo que nos realice la comprobación.

VI. El repositorio

Un *repositorio* o base de datos es la mayor unidad de datos en Metaedit+. Es aquí donde se almacena toda la información relacionada con el metamodelo y los modelos. Pueden existir varios repositorios, pero los datos de uno de ellos no pueden ser referenciados directamente desde otro diferente. Un repositorio se compone de *áreas*, que corresponde con el concepto de *proyecto* que es perceptible por el usuario de Metaedit+. En un entorno multi-usuario, cada repositorio requiere de un *servidor Metaedit+* propio ejecutándose. El servidor para Metaedit+ es un programa aparte con su propia interfaz de usuario.

Los ficheros del repositorio se disponen generalmente en una jerarquía de un único directorio. El directorio del nivel más alto contiene el fichero *artbase.roo* que contiene los nombres y rutas de todos los repositorios, y un subdirectorio para cada uno de esos repositorios. En el directorio de un repositorio hay dos ficheros, *manager.ab* y *trid*. El primero de ellos contiene los nombres y rutas de todas las áreas del repositorio, los nombres y contraseñas cifradas de todos los usuarios de ese repositorio y cualquier correspondencia de nombre de disco necesario para acceder a él. El directorio de un repositorio contiene además varios subdirectorios: *areas*, que contiene un directorio por cada área física en el repositorio; *users*, que contiene un directorio por cada usuario del repositorio; *backup*, que contiene una copia de seguridad del repositorio en el estado en que quedó al ejecutar el último *commit* con éxito; *comm*, sólo se crea cuando se trata de un repositorio multi-usuario, y contiene información de acceso multi-usuario mediante fichero de comunicación; y *counters*, que se crea únicamente cuando se trata de un repositorio multi-usuario y normalmente está vacío.

Metaedit+ no almacena la información en ficheros ordinarios, sino como objetos. De este modo, toda la información de un modelo creado por un usuario no está en un fichero, sino representada como objetos. Esto permite una reutilización eficiente de datos entre modelos, pero entraña diversas diferencias de uso comparado con herramientas cuyo almacenamiento está basado en ficheros. Nos estamos refiriendo a que Metaedit+ emplea un enfoque transaccional en la que es necesario confirmar (*commit*) y abandonar (*abandon*) los cambios en el repositorio. Al ejecutar el programa Metaedit+ y entrar en el repositorio (*log-in*) comienza una *transacción*. Todos los cambios que se realicen sobre los datos del modelo son parte de una transacción, y no se actualizan en el repositorio de forma inmediata. Para almacenar los cambios en el repositorio de forma permanente, es necesario confirmar (*commit*) explícitamente la transacción. Una vez confirmada una transacción, la transacción finaliza y comienza otra nueva. Es posible abandonar (*abandon*) la transacción, lo que significa que se cancelan los cambios y se retorna al estado previo en que se confirmó la última transacción. Los botones para confirmar y abandonar están accesibles en la barra de herramientas de la ventana principal como muestra la Figura 68.

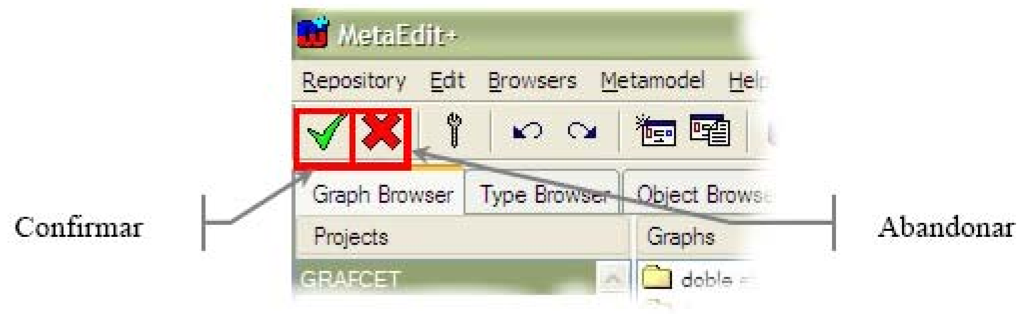


Figura 68: Commit / Abandon

La exportación de la información de un proyecto almacenado en un repositorio con el fin de utilizarlo en otro repositorio se realiza en Metaedit+ mediante el mecanismo de parches. Mediante la herramienta *Type Manager* es posible crear un parche (extensión *.pat*) en el que se incluye la definición completa de todos y cada uno de los tipos creados mediante el lenguaje de metamodelado GOPPRR para los proyectos seleccionados. La herramienta *Graph Manager* nos permite del mismo modo construir un parche en el que se incluyen los modelos junto con todas sus instancias, además de los informes para los proyectos seleccionados. Instalando estos dos parches mediante la opción *File in Patches* que nos aparece en el primer diálogo al ejecutar el programa, se reconstruye completamente el metamodelo y los modelos para los proyectos deseados en un repositorio en el que no fueron creados.

VII. Posibilidades avanzadas de Metaedit+

Como delata el título del apartado, nos disponemos a describir brevemente el resto de las funcionalidades (no tan básicas) que nos ofrece el framework de metamodelado. No entraremos en profundidad pues son opciones que implican otras tecnologías de software y que no viene al caso para el objetivo de este proyecto. Simplemente conviene conocer qué otras facilidades se nos aportan y que no son fundamentales para el metamodelado, con la finalidad de conseguir una visión global de la potencia de la herramienta. Nos disponemos pues a esbozar un par de facilidades como son la importación/exportación en XML y la API para acceder a Metaedit+ desde nuestros propios programas.

i. Importación y exportación en XML

Aquí tratamos la exportación o importación (extensión .gxl) de los datos del diseño entre Metaedit+ y otras aplicaciones. La estructura del fichero XML refleja la estructura del lenguaje de metamodelado GOPPRR, por lo que cualquiera familiarizado con dicho lenguaje no debería tener problemas para entenderlo. La estructura del XML omite el metamodelo y la información de representación, con el objetivo de facilitar a los usuarios la tarea de crear ficheros XML correctos que de otro modo se volvería difícil. Por tanto, el usuario es el responsable de asegurar que el metamodelo en el que se basa el fichero XML concuerda con el que hay en el repositorio cuando se importa. El uso del formato XML no permite la actualización de modelos que han sido importados previamente, y cada vez que importamos un modelo en XML se crea un nuevo modelo de datos completo. En la Figura 69 mostramos un ejemplo de la estructura de estos ficheros.

```
1  <gxl>
2    <graph type="WatchApplication">
3      <slot name="Name">
4        <value>
5          <string>Stopwatch</string>
6        </value>
7      </slot>
8      <object>...
9      </object>
10     <binding>
11       <relationship>...</relationship>
12       <connection>
13         <role>...</role>
14         <port>...</port>
15         <object>...</object>
16       </connection>
17     </binding>
18   </graph>
19 </gxl>
```

Figura 69: Ejemplo de la estructura del XML

El fichero se inicia con la etiqueta **<gxl>** que se debe cerrar al finalizar el fichero.

Debajo de esta vemos la etiqueta **<graph>** y que contiene todos los elementos del grafo.

Bajo la etiqueta del grafo se encuentran las definiciones de objetos (**<object>**) y enlaces (**<binding>**). La definición de los enlaces contiene definiciones para las relaciones (**<relationship>**) y conexiones (**<connection>**). Deben existir al menos dos conexiones en cada enlace. La definición de conexión contiene definiciones para los roles, puertos y objetos, denotados por sus respectivas etiquetas **<role>**, **<port>** y **<object>**. Si no hay puertos definidos para la conexión, se puede omitir la etiqueta. **<graph>, <object>, <relationship>, <role>, <port>** tienen un atributo denominado *type* que indica el tipo del elemento. Con la etiqueta **<slot>** podemos especificar el nombre local de las propiedades del elemento. Dentro de ésta, se anida la etiqueta **<value>** en la que se indica el valor de la propiedad, encerrado en etiquetas que indican el tipo (en el ejemplo anterior, en la línea 5 se aprecia el tipo *String* para la propiedad *Name*). Los tipos pueden ser **<string>**, **<text>**, **<int>**, **<float>**, **<bool>**.

Dentro del slot también puede ir una etiqueta **<property>** para referirse a la propiedad (no es necesario para importar). Los tipos de atributos son obligatorios en las etiquetas de propiedades de grafo, objeto, elación, rol o puerto, a menos que se use un atributo *href* que permite que no haya otros contenidos. El valor de un tipo de atributo puede ser el nombre del tipo visible por el usuario o el nombre del tipo interno y único. Si un elemento se referencia más de una vez, se exportará con un atributo extra denominado *id* cuyo valor debe ser único. Una referencia posterior al mismo elemento no tendrá otro contenido que no sea el atributo *href*. A modo de ejemplo, si tenemos que **<object type=”Button” id=”id27”>**, posteriormente se puede volver a referenciar como **<object href=”#id27”/>**. Este mecanismo se utiliza para asociar los subgrafos (ya sea por explosión o descomposición) a los elementos (sólo objetos en la descomposición).

ii. API de acceso

Es el mecanismo más potente de integración proporcionado por Metaedit+. Provee un modo de acceder a la información de los modelos desde nuestras propias aplicaciones. El API de interfaz de Metaedit+ se implementa como un servidor de Servicio Web SOAP. Una aplicación que pretenda utilizar la API Metaedit+ debe implementar un cliente SOAP que se encarga de establecer la conexión y realizar las llamadas a Metaedit+. SOAP es un estándar abierto y está ampliamente extendido, por lo que existen frameworks disponibles que manejan estos aspectos de forma transparente. El API puede utilizarse para acceder y cambiar en Metaedit+ los elementos del metamodelo, es decir, la sintaxis abstracta (grafos, objetos, relaciones, roles, propiedades y puertos). No es posible, sin embargo, acceder o cambiar elementos de representación del metamodelo (sintaxis concreta), como podría ser cambiar la disposición de los elementos del diagrama. También, debido a que las funciones del API operan a un nivel más bajo que las operaciones de interfaz de usuario, y que con el API no hay posibilidad de realimentación, las funciones que crean o modifican datos no comprueban que los datos son legales y acordes con el metamodelo actual. El framework proporciona algunas características adicionales que pueden ser accedidas o controladas por el API, como resaltar o animar elementos del diagrama para propósitos de simulación o trazas. Con el objetivo de soportar llamadas mediante SOAP, se proporciona un servidor SOAP y el fichero WSDL que contiene las definiciones de los comandos del API para las aplicaciones externas. Se puede ver en la Figura 70 la *herramienta de la API* (*API Tool*) para controlar el servidor.

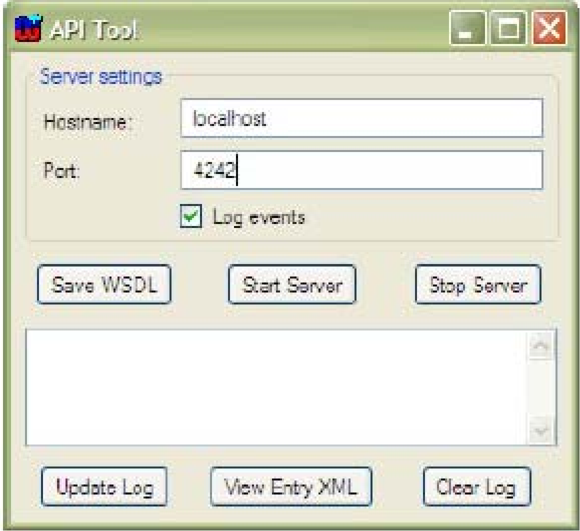


Figura 70: Herramienta de la API

Si el registro (*log*) de eventos está activado, los registros de peticiones SOAP entrantes y procesadas se muestran en el área de texto en la parte baja de la ventana de la herramienta de la API. Se puede actualizar la vista del registro pulsando el botón *Update Log*, limpiarlo con *Clear Log*, y ver la fuente XML completa para la entrada de petición SOAP con *View Entry XML.*

Además de los tipos estándar como cadenas de caracteres y enteros, el API recurre a cuatro tipos específicos: *MEOop* es un manejador de una instancia conceptual en un repositorio que puede ser un grafo, objeto, relación, rol, puerto, propiedad o enlace (*binding*), *MEAny* para representar un valor que puede ser de uno de varios tipos, *METype* representa un tipo del metamodelo y *MENull* para el valor nulo. Los métodos son numerosos y se clasifican en: control (órdenes de línea de comandos, animación, apertura de argumentos en la representación del grafo, y obtención de objetos copiados), gestión de propiedades (crear, establecer o consultar), lectura de los elementos del grafo, navegación por los enlaces de los contenidos del grafo, tratamiento de subgrafos (descomposición y explosión), introducción y consulta de objetos y enlaces, borrado de grafos, manejo de tipos, y otro genérico de impresión en una cadena.

Para implementar el acceso al servidor de Metaedit+ desde nuestras aplicaciones, se necesita tener instalado un framework SOAP para el lenguaje de programación o plataforma. Hay numerosos frameworks para tal fin. En el caso de considerar el cliente en Java 2, podríamos recurrir al parser XML *Xerces2* de Apache y a la librería SOAP *Axis* de Apache. El cliente que se implemente, accederá al servidor mediante la librería SOAP y le solicitará la ejecución de diversos métodos de los que dispone (especificados en el fichero WSDL) para gestionar los datos del modelo que yacen en el repositorio de Metaedit+. Las operaciones disponibles sobre los datos siempre estarán limitadas por los servicios que el servidor nos ofrece.

Capítulo 4
Manual de usuario

I. Introducción

Este capítulo pretende introducir al usuario a la programación de los motes usando el DSM diseñado en el presente proyecto. No se incidirá en el lenguaje final para programarlos (nesC) sino en aspectos de dominio del problema que son las redes de sensores inalámbricas. Precisamente los conceptos del dominio son los únicos que aparecerán en el modelo de cualquier aplicación generada con esta herramienta y son los pasos a seguir para un correcto modelado los que se incluyen aquí.

En primer lugar es necesario comprender cuál es el dominio del problema que nos atañe. Como se ha comentado en capítulos anteriores el DSM eleva el nivel de abstracción del lenguaje a conceptos que realmente aparecen en el dominio y esconde herramientas de programación innecesarias para describir la solución (accesos a memoria, punteros, variables, métodos…). DSM no sólo ofrece dichos conceptos de dominio, quedarnos aquí sería quedarnos en un DSL con notación textual ya explicado en capítulos anteriores, sino que además los expresa con una notación gráfica, con objetos visibles y conectores a modo de diagrama. Este diagrama es el llamado modelo de la solución.

Para el caso de las redes de sensores, más concretamente de los nodos, los conceptos del dominio no son otros que vayan más allá de los diferentes sensores de los que disponga el dispositivo en cuestión. Evidentemente, para poder expresar la solución hace falta algo más que un simple sensor. Por ejemplo, si lo que se desea es medir la temperatura, no nos basta con colocar un sensor de temperatura en el centro del gráfico, sino que también hay que expresar de algún modo qué hacer con la medida ya que las posibilidades son infinitas: enviarla por radio a un nodo determinado, mandarla al PC, encender el led verde si supera los 40 grados… es por tanto aquí donde se definen las reglas de diseño: los componentes con sus respectivos símbolos, los diferentes parámetros que aceptan, la forma en que se pueden conectar, etc.

Para comenzar a usar la aplicación y generar el primer modelo específico de dominio, es necesario dar a conocer, a modo de manual de instrucciones, la interfaz de usuario. En primer lugar se enumeran los distintos objetos con una breve explicación junto con las propiedades asociadas. Seguidamente se exponen las reglas de conexión entre componentes y el significado que tienen según los parámetros que posean los componentes que las forman. Una vez llegado a este punto sólo resta generar el código (pulsar un botón) y programar el mote.

II. Gráfico

Metaedit+ permite realizar el modelado de una aplicación (gráfico) tanto en vista matriz, vista de tabla o vista diagrama. Este DSM esta diseñado para crear los modelos en vista diagrama. El gráfico tiene únicamente una propiedad que es el nombre de la aplicación. Este nombre no debe llevar espacios ni caracteres especiales, a continuación se muestran los pasos a seguir junto con los diálogos para crear un nuevo gráfico una vez abierto el proyecto TinyOS en el repositorio de Metaedit+.

1. Se debe pinchar con el botón derecho del ratón en la zona del programa de dedicada a los gráficos (*Graphs*) y seleccionar la opción *Create Graph*.

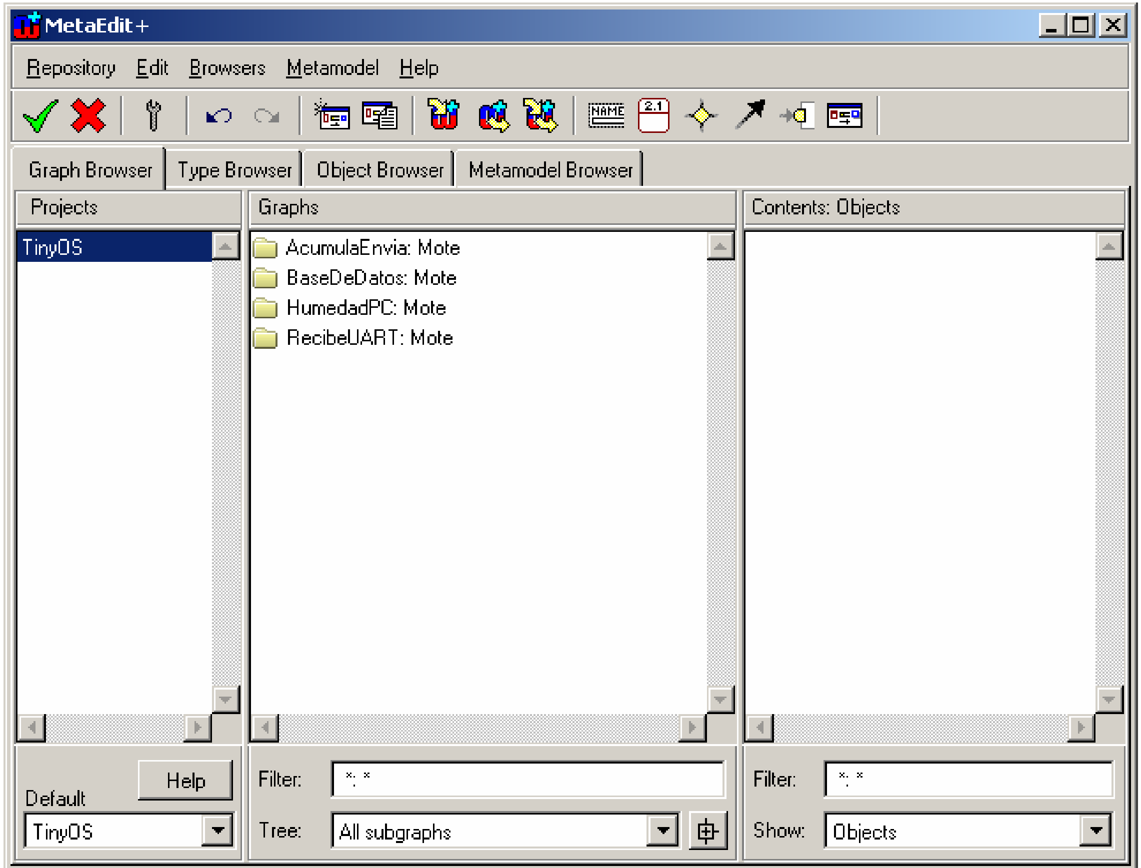


Figura 71: Vista principal Metaedit+

Figura 72: Opción Create Graph

2. En el diálogo que aparecerá (Figura 73) Se debe seleccionar el tipo de gráfico *Mote* y abrirlo como diagrama ya que está pensado para modelarlo así y no como matriz o tabla.

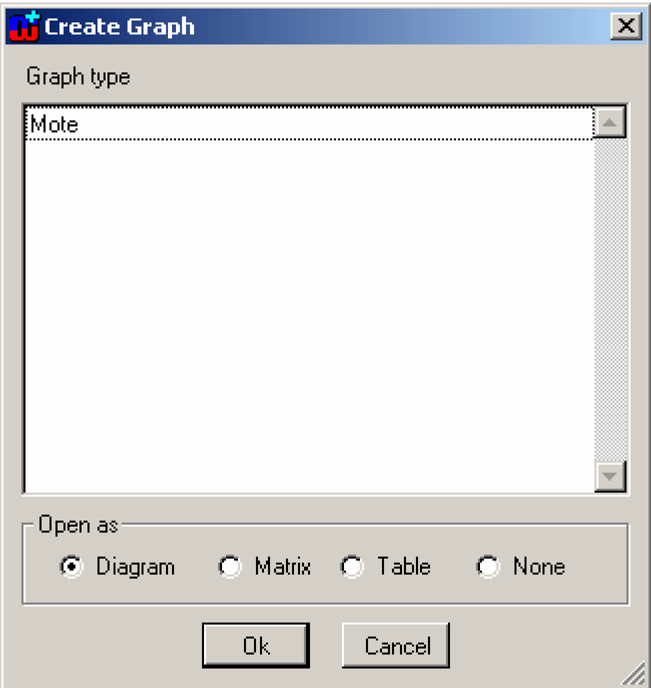


Figura 73: Diálogo Create Graph

3. Ahora aparece un nuevo diálogo que muestra las propiedades de este nuevo gráfico. Como se ha comentado, únicamente tiene la propiedad de nombre (*Name*) que dará nombre a nuestra aplicación. El nombre no debe contener espacios ni caracteres especiales ya que de ser así el código generado no podrá ser compilado con TinyOS. La Figura 74 muestra dicho gráfico.

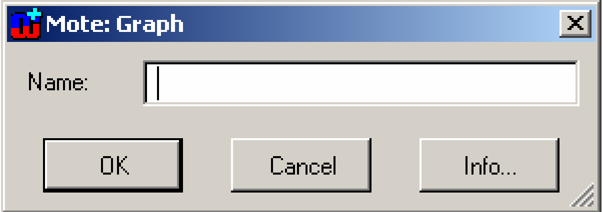


Figura 74: Diálogo de propiedades de gráfico

4. Una vez llegado a este punto se podrá modelar la aplicación usando los objetos y relaciones destinados a ello. En la barra de objetos podremos seleccionarlos y colocarlos libremente en el espacio dedicado a la creación del diagrama. La Figura 75 muestra la ventana de modelado junto a la barra de objetos mencionada.

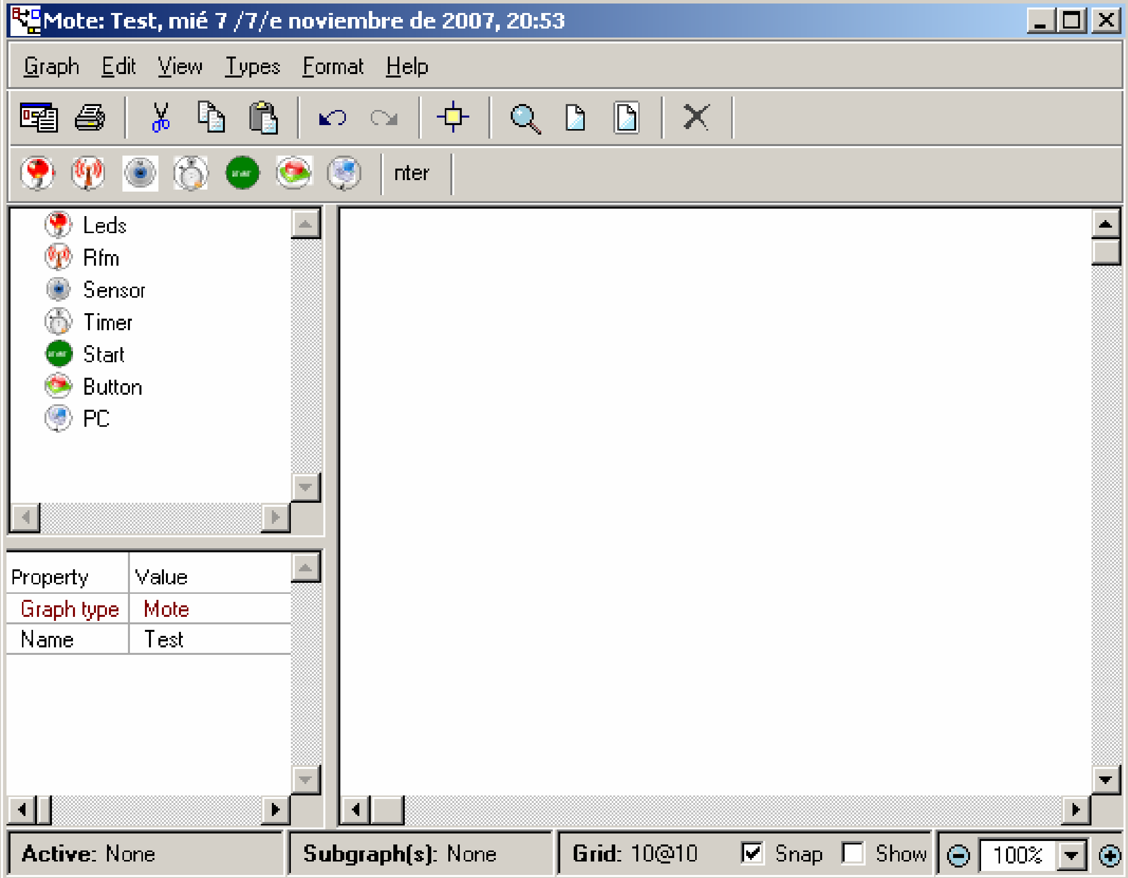


Figura 75: Ventana de modelado

III. Objetos

Los distintos objetos que se pueden usar en el modelado de la aplicación son mostrados a continuación, junto con sus propiedades y el significado de cada una de ellas. También se muestra el símbolo que toma según las propiedades seleccionadas. Estos Objetos pueden ser referenciados en el gráfico llamado Mote como se ha explicado en el punto anterior.

i. Start

Marca el comienzo de la aplicación. Sólo es ejecutado una vez cada vez que comience la aplicación. A este objeto no le podrán llegar conexiones desde otros objetos ya que no tiene sentido semántico una llegada de un evento o un dato a un objeto que marca el comienzo de la aplicación. El símbolo que lo representa se muestra en la Figura 76 y carece de propiedades.

Figura 76: Objeto Start

ii. Timer

Hace referencia a un temporizador. Es necesario que al menos le llegue un evento (Si no le llega jamás comenzará el temporizador) y que al menos sea origen de una relación (si no fuera origen de una relación, aunque concluya el temporizador no se pasará el evento a ningún objeto). En la Figura 77 so observa el diálogo correspondiente a este objeto donde podemos especificar sus propiedades. El valor de las propiedades condicionará la representación del objeto según podemos ver en las figuras siguientes.

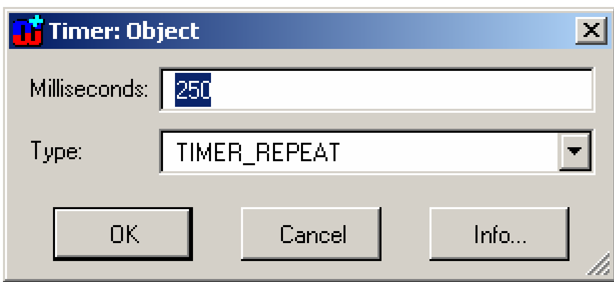


Figura 77: Diálogo Timer

- La propiedad *Milliseconds* especifica el tiempo de ciclo del temporizador en milisegundos. Su valor se mostrará en el símbolo
- La propiedad *Type* especifica si el temporizador será repetitivo o se por el contrario sólo comenzará cuando le llegue el evento que lo dispare. El valor se de esta propiedad modificará el símbolo como se ve en las siguientes figuras:

Figura 78: Timer One Shot

Figura 79: Timer Repeat

iii. Sensor

Hace referencia a la medida de un determinado fenómeno (temperatura, humedad, luminosidad...) que quedará especificado en su propiedad *Type*. Es necesario que al menos le llegue un evento (Si no le llega, jamás intentará medir) y que al menos sea origen de una relación (si no fuera origen de una relación, aunque concluya la medición no se pasará el dato o evento a ningún objeto). En la Figura 80 se observa el diálogo correspondiente a este objeto. En él se puede escoger el valor de las propiedades que harán cambiar la representación del símbolo.

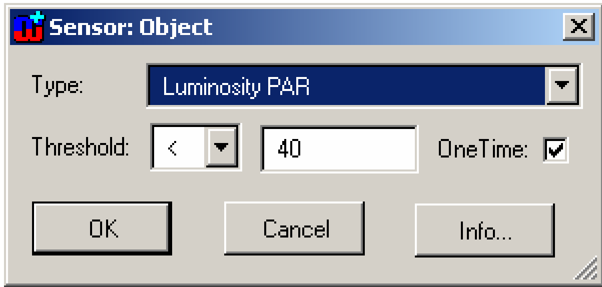


Figura 80: Diálogo Sensor

- La propiedad *Type* hace referencia al tipo de medida que se desea tomar. Los valores posibles de esta propiedad son: *Temperature* (temperatura), *Luminosity PAR* (luminosidad fotosintética), *Luminosity TSR* (luminosidad solar total incluido el infrarrojo), *Humidity* (humedad) e *Internal Temp*. (temperatura interna del chip). Y los símbolos correspondientes a cada propiedad se muestran en las siguientes figuras.
- La propiedad *Thresold* hace referencia al umbral. Se compone de dos campos. El primero es una lista desplegable con todos los posibles valores que pueden condicionar un umbral además del valor *Value* (Value,>,<,==,!=,>=,<=). Si se selecciona en este campo el valor *Value* se entenderá que lo que se desea es medir el valor y pasar su valor al siguiente objeto de la relación. Si se selecciona un valor distinto de *Value* se entenderá únicamente se pasará el valor (el cual podrá ser usado como evento) cuando se cumpla la condición. El otro campo corresponde al valor que debe cumplir la condición y la magnitud serán medida según el tipo de sensor escogido (ºC para temperatura, luxes para luminosidad, % de humedad relativa para la humedad).
- La propiedad *OneTime* sólo se tendrá en cuenta si se ha especificado un umbral. Cuando está seleccionada el sensor mandará el valor de la medida sólo una vez por cada vez que se realice el salto de umbral. Esto

es importante para el ahorro de baterías aunque en ocasiones es preferible que mande la medida cada vez que esté por encima del umbral.

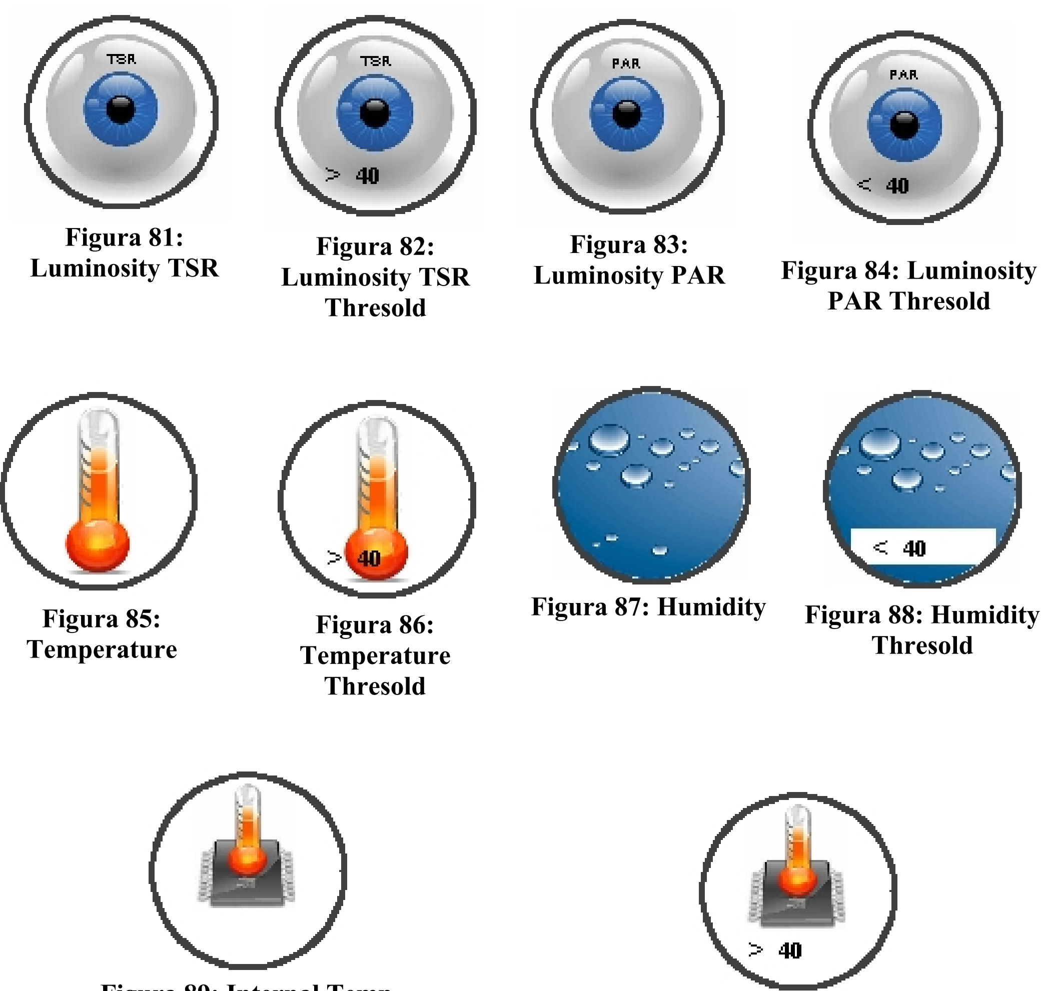


Figura 81: Luminosity TSR

Figura 82: Luminosity TSR Thresold

Figura 83: Luminosity PAR

Figura 84: Luminosity PAR Thresold

Figura 85: Temperature

Figura 86: Temperature Thresold

Figura 87: Humidity

Figura 88: Humidity Thresold

Figura 89: Internal Temp.

Figura 90: Internal Temp. Thresold

iv. Rfm

Hace referencia al acceso al canal radio. Dependiendo de si le llega una relación o sale de él actuará como un emisor o un receptor. El diálogo correspondiente a este objeto se muestra en la Figura 91 donde se pueden ver las diferentes propiedades. Las propiedades *Handler ID* y *Store* son mostradas en el símbolo. Este objeto tiene además la propiedad *Channel* para futuras ampliaciones y que como de momento carece de sentido se ha eliminado tanto del diálogo como del símbolo. El símbolo correspondiente a este objeto se muestra en la Figura 92.

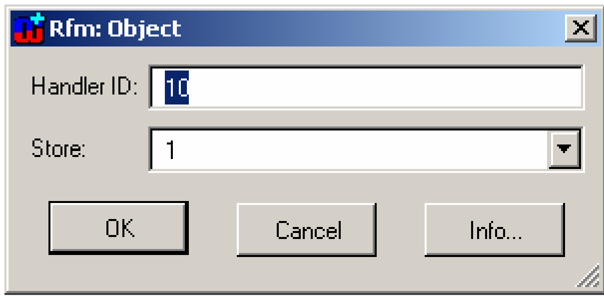


Figura 91: Diálogo Rfm

- La propiedad *Handler ID* hace referencia al identificador del mensaje. Este identificador proporciona un filtro en los mensajes que se quieren procesar. Así, auque de momento no exista dirección de envio de paquete se podría simular con este identificador.
- La propiedad *Store* especifica el número de datos o eventos que se enviarán en cada paquete. Estos datos se irán ordenando en el paquete según vayan llegando así que habrá que tener en cuenta el orden en que llegan mediante temporizadores.

Figura 92: Objeto Rfm

v. Button

Hace referencia al *user button* del que disponen los motes del tipo TelosB. Este objeto generará un evento cada vez que se pulse el mencionado botón. Este objeto no podrá ser el destino de ninguna relación ya que carece de sentido semántico conectar a un botón un evento o un dato. El diálogo correspondiente a este objeto se muestra en la Figura 93 en el cual se puede observar la única propiedad de la que dispone. El símbolo correspondiente a este objeto se muestra en la Figura 94 en el que se puede observar el valor de su variable *Port.*

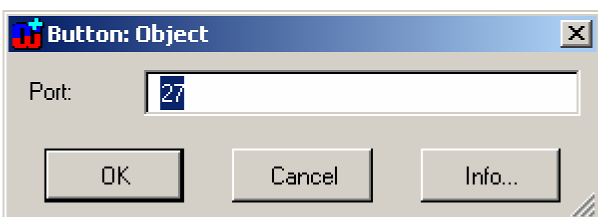


Figura 93: Diálogo Button

- La propiedad *Port* hace referencia al número de puerto sobre el que se sensoriza. El valor por defecto es 27 ya que corresponde al botón *user button* ya instalado en la placa del mote.

Figura 94: Objeto Button

vi. PC

Hace referencia al envío de datos hacia el PC. Siempre aparecerá como objeto terminal en una cadena de proceso. Es decir, desde este objeto no podrán nacer más relaciones. Esto es debido a que por el momento no se encuentra implementada la comunicación desde el PC hacia los motes. En la Figura 95 se observa el diálogo donde se pueden editar las propiedades de este objeto. Dichas propiedades son visibles en el símbolo del objeto como se aprecia en la Figura 96.

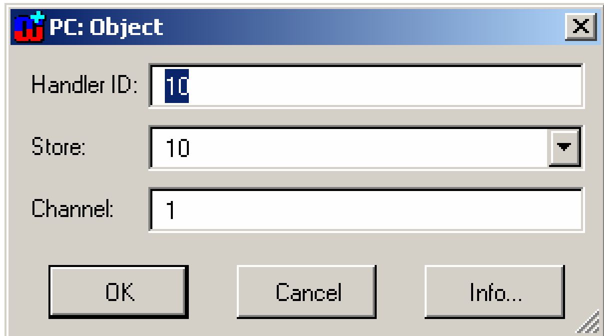


Figura 95: Diálogo PC

- La propiedad *Handler ID* hace referencia al identificador del mensaje. Esta propiedad puede ser usada para atender el mensaje en la aplicación destino según corresponda.
- La propiedad *Store* especifica el número de datos o eventos que se enviarán en cada paquete. Estos datos se irán ordenando en el paquete según vayan llegando así que habrá que tener en cuenta el orden en que llegan mediante temporizadores.
- La propiedad *Channel* es añadida para hacer compatible el envío de datos al PC con la herramienta *Oscilloscope* que se incluye con la instalación del sistema operativo TinyOS.

Figura 96: Objeto PC

vii. Leds

Como su propio nombre indica, simboliza el hecho de realizar alguna acción sobre alguno, o todos los leds instalados en la placa del mote TelosB. Al igual que ocurre con el objeto PC, este objeto siempre aparecerá como objeto terminal en una cadena de proceso, es decir, que desde este objeto no nacerá ninguna relación. Esto es así porque carece de sentido el hecho de que un led provoque un evento o genere un dato. En la Figura 97 se puede observar el cuadro de diálogo en el que se dan valores a las propiedades del objeto. Las propiedades del objeto hacen cambiar la forma de representarlo de acuerdo al color que se escoja. La propiedad *Action* será representada en el símbolo como se ven en los ejemplos de las figuras siguientes.

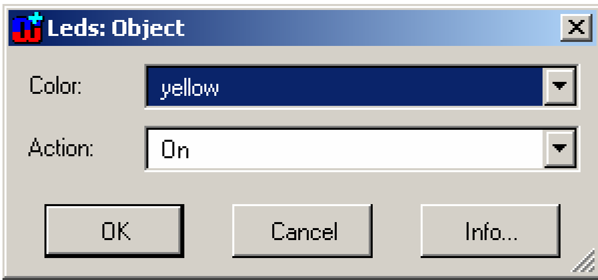


Figura 97: Diálogo Leds

- La propiedad *Color* especifica el color del diodo que se quiere encender. Se debe tener en cuenta que en algunas versiones de TelosB el color amarillo se sustituye por un led azúl aunque en el sistema operativo se le sigue denominando *yellow*. Si se escoge como color la opción *Value* el símbolo cambia al representado en la Figura 100 y se encenderán los leds según correspondan a los tres bits más significativos del dato que se le transfiera al objeto. Con esta opción no se tiene en cuenta el valor de la propiedad *Action.*
- La propiedad *Action* especifica la acción a realizar con el diodo seleccionado. Las opciones posibles son *On*, *Off* y *Toggle* que corresponden a encendido, apagado y conmutado del led respectivamente.

Figura 98: Leds Yellow On

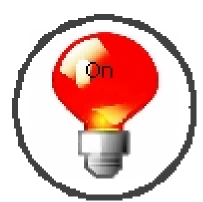


Figura 99: Leds Red On

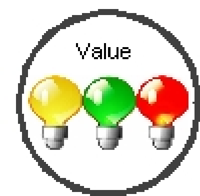


Figura 100: Leds Value

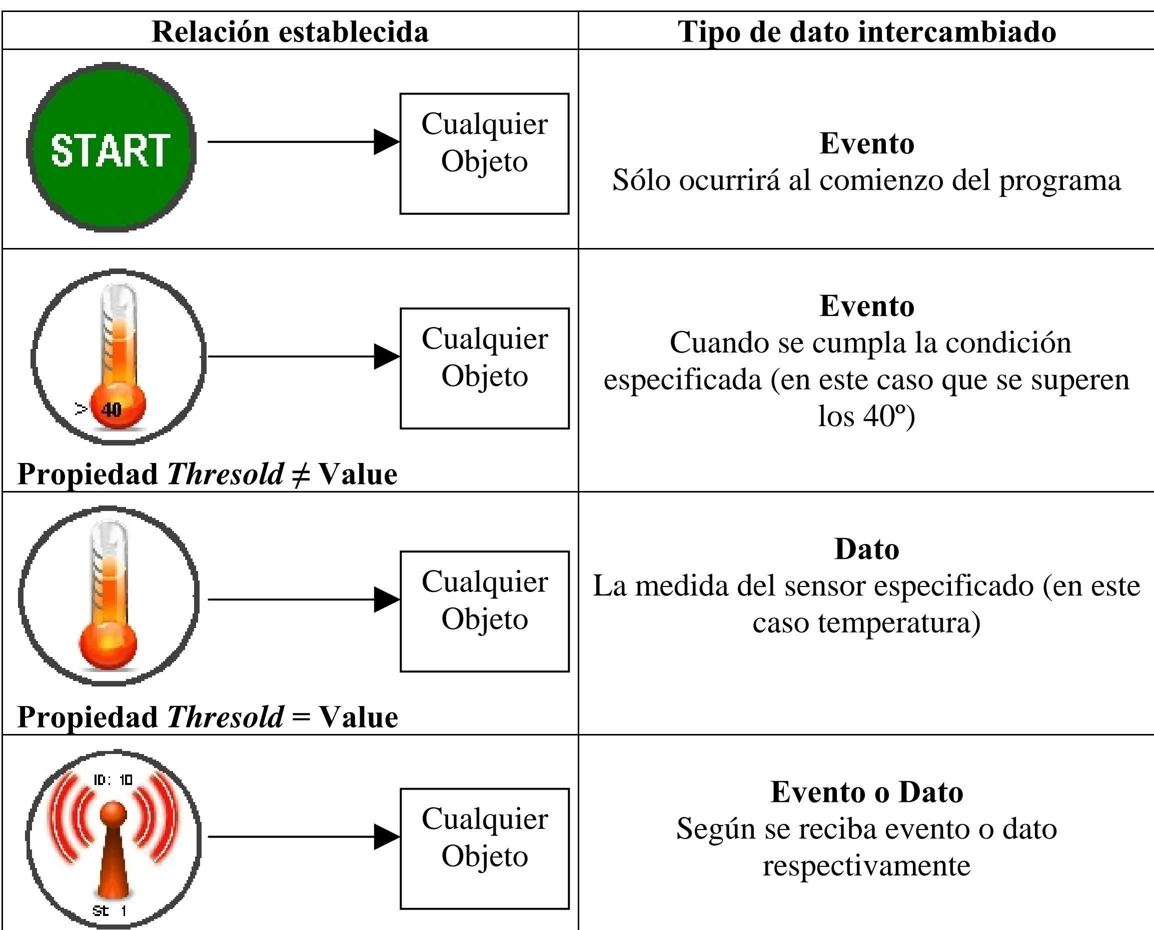
Figura 101: Leds Green Toggle

IV. Relaciones

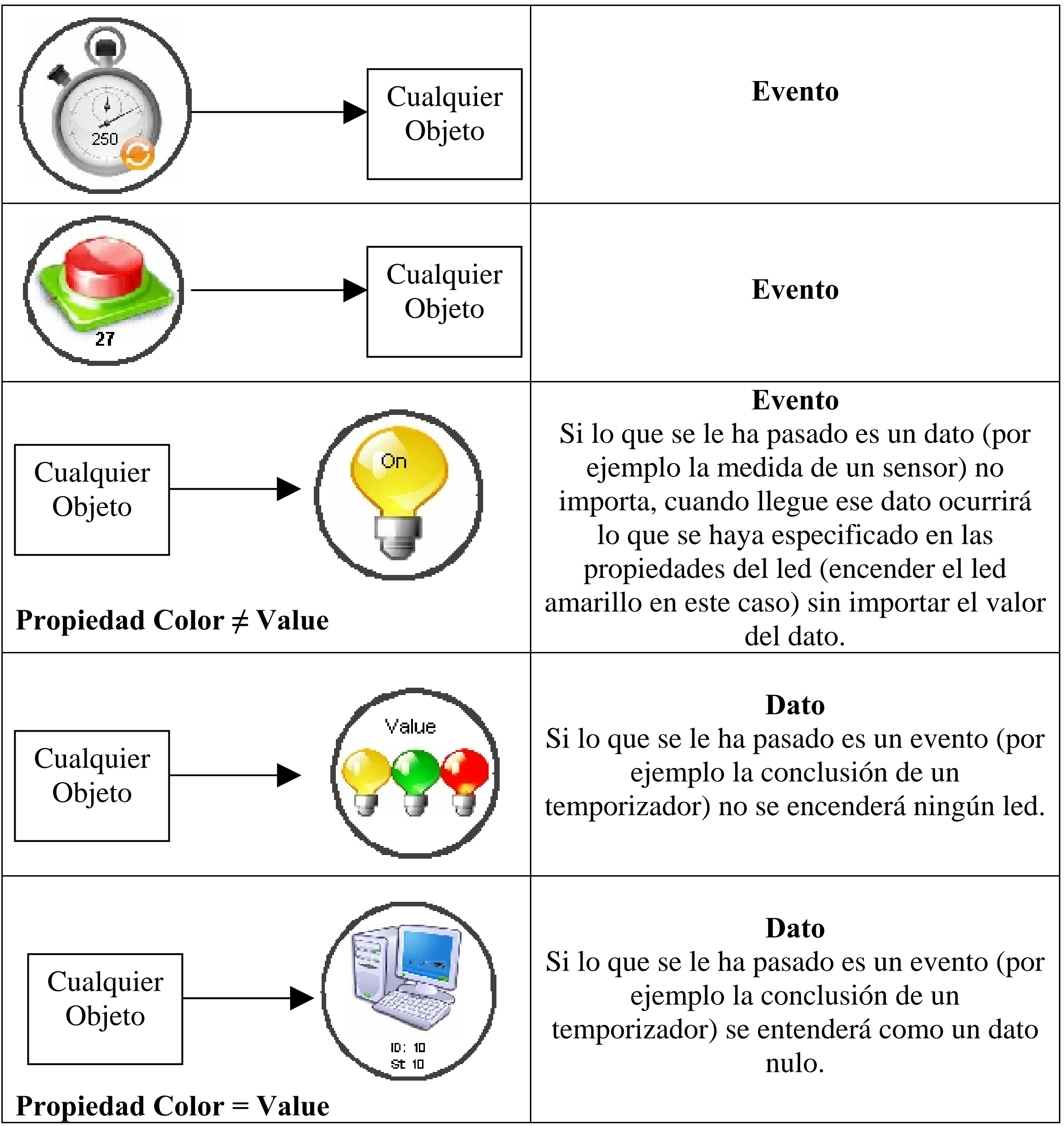
La unión de un objeto cualquiera objeto1, con un segundo objeto cualquiera objeto2 indicará que el cuando se produzca un determinado evento en el objeto1 se actuará de una determinada forma sobre el objeto2 (Ej. cuando se supere un umbral de temperatura se mandará por radio una señal a una dirección). Dicha unión se realiza en Metaedit+ mediante una relación. La única relación existente en el DSM es la llamada interfaz (*Interface*). Para crear una relación entre un objeto y otro (ya existentes en el modelo) se ha de pulsar sobre el botón *Interface* situado a la izquierda de la barra de objetos. Una vez seleccionada la relación, se deben seleccionar los objetos que se desean conectar. Esta relación interconecta los objetos en relaciones uno a uno. Los tipos de datos que se pueden pasar entre dos objetos interconectados son dos:

1. Dato: Por ejemplo el dato recibido por radio o el dato medido.
2. Evento: Por ejemplo el hecho de que un temporizador acabe su ciclo o que un sensor supere un umbral establecido.

Los tipos de datos a transmitir (Dato o Evento) se pasarán en una determinada relación dependiendo del contexto. Esto provoca una importante mejoría para la facilidad de uso ya que con las propiedades de los objetos que se interconectan junto con el sentido de la conexión se obtienen los datos necesarios para establecer el tipo de datos que se intercambiarán. Así en la siguiente tabla podemos ver los tipos de datos que se pasarán entre las distintas conexiones y según las propiedades seleccionadas.



Relación establecida	Tipo de dato intercambiado
START → Cualquier Objeto	**Evento** Sólo ocurrirá al comienzo del programa
> 40 → Cualquier Objeto **Propiedad *Thresold* ≠ Value**	**Evento** Cuando se cumpla la condición especificada (en este caso que se superen los 40º)
→ Cualquier Objeto **Propiedad *Thresold* = Value**	**Dato** La medida del sensor especificado (en este caso temperatura)
ID: 10 St 1 → Cualquier Objeto	**Evento o Dato** Según se reciba evento o dato respectivamente



250 → Cualquier Objeto	**Evento**
27 → Cualquier Objeto	**Evento**
Cualquier Objeto → On **Propiedad Color ≠ Value**	**Evento** Si lo que se le ha pasado es un dato (por ejemplo la medida de un sensor) no importa, cuando llegue ese dato ocurrirá lo que se haya especificado en las propiedades del led (encender el led amarillo en este caso) sin importar el valor del dato.
Cualquier Objeto → Value	**Dato** Si lo que se le ha pasado es un evento (por ejemplo la conclusión de un temporizador) no se encenderá ningún led.
Cualquier Objeto → ID: 10 St 10 **Propiedad Color = Value**	**Dato** Si lo que se le ha pasado es un evento (por ejemplo la conclusión de un temporizador) se entenderá como un dato nulo.

Figura 102: Tabla de relaciones

No existe un número de objetos limitados en el gráfico, es decir, se podrán incluir y relacionar tantos objetos como se desee. Sin embargo existen algunas restricciones o reglas para las relaciones creadas entre objetos las cuales se enumeran a continuación:

1. Al objeto *Start* no podrá llegarle ninguna relación. Es decir, todas las relaciones en las que intervenga un objeto *Start* comenzarán desde el mismo ya que carece de sentido el que le podamos pasar un dato o un evento al "comienzo de la aplicación".
2. Las relaciones que lleguen al objeto *Rfm* se entenderán como una transmisión vía radio del evento o dato a transmitir.
3. Las relaciones que tengan su origen en el objeto Rfm se entenderán como una recepción de un evento o dato según lo que se reciba.
4. El objeto *Leds* no podrá ser origen de ninguna relación ya que carece de sentido que sea el objeto *Leds* quien origine un evento o dato.

5. El objeto *Button*, al igual que ocurre con el *Start* no puede ser el destino de ninguna relación ya que este objeto no puede procesar ningún dato ni evento de llegada.
6. Al objeto *Timer*, para que sea "útil", es necesario que al menos le llegue un evento (Si no le llega jamás comenzará el temporizador) y que al menos sea origen de una relación (si no fuera origen de una relación, aunque concluya el temporizador no se pasará el evento a ningún objeto).
7. Al objeto *Sensor*, para que sea "útil", es necesario que al menos le llegue un evento (Si no le llega jamás intentará medir) y que al menos sea origen de una relación (si no fuera origen de una relación, aunque concluya la medición no se pasará el dato o evento a ningún objeto).
8. El objeto PC no podrá ser origen de ninguna relación ya que actualmente no se encuentra implementada la recepción de paquetes desde el ordenador. Por lo tanto el objeto PC siempre que se encuentre en una relación aparecerá en el rol de destino (*Provides*).

Capítulo 5
Implementación del DSM

I. Introducción

En los capítulos anteriores se ha explicado cómo se estructura una aplicación en nesC y la manera de diseñar las reglas de modelado en MetaEdit+. Ahora se procede a explicar la forma en que se han diseñado las reglas de modelado para poder generar código nesC automáticamente con la ayuda de la herramienta MetaEdit+.

Como se ha comentado en capítulos anteriores, el diseño de DSM ha intentado salir del contexto de la programación en nesC y se ha elevado el nivel de abstracción del lenguaje al dominio del problema. Este hecho, además de las numerosas ventajas comentadas a lo largo del documento conlleva algunos inconvenientes. El DSM diseñado no puede contemplar todas las posibilidades de programación que permiten los motes. De hecho de lo que se trata a grandes rasgos es de reducir las posibilidades de programación a las realmente útiles dentro del dominio del problema incrementando así la rapidez y la sencillez de desarrollo.

En el momento de crear reglas de diseño es constante el enfrentamiento entre la versatilidad de la aplicación frente a la practicidad o facilidad de uso. Por ejemplo, en el supuesto de que se quisiera usar la medida de un sensor de temperatura para determinar cuál es el tiempo de ciclo de un determinado temporizador. Esta posibilidad encadenaría una serie de problemas. En primer lugar aparecería una ambigüedad en la conexión entre el objeto *Sensor* (rol *Uses*) y el objeto *Timer* (rol *Provides*) ya que esta conexión se puede entender más como trigger que active el temporizador cuando se supere una determinada temperatura. En el caso de que se solventase esta ambigüedad entre conexiones diferenciando una de otra por la aparición del parámetro umbral en el objeto sensor, todavía habría una falta de significado en la conexión por sí sola, ya que rara vez (por no decir nunca) se va a usar la medida de un sensor, ya sea en grados, luxes o porcentaje de humedad para determinar el tiempo en segundos que va a tener un temporizador. Seguramente se aplicaría una fórmula de conversión entre unidades lo cual complicaría aún más el modelo (dependencias de otras variables unidas a las limitaciones en el herramienta de modelado…). Es por esto por lo que en numerosas ocasiones es conveniente optar por la opción práctica con la que se cubren la gran mayoría de casos en los que ocurre una determinada conexión semántica. Para casos tan aislados como el expuesto en el ejemplo será necesario recurrir al código generado por la herramienta y hacer las modificaciones pertinentes.

El siguiente paso conlleva la generación de código a partir del modelo de la aplicación. Se ha usado la herramienta de generación de código mediante *scripts* proporcionada por MetaEdit. La generación de código se hace totalmente automática a partir del modelo sin preguntar absolutamente nada y sin dejar ningún campo en blanco que deba ser programado. Esto ha resultado una tarea difícil pero el hecho de que todos los datos necesarios se puedan sacar directamente del modelo ha inspirado a esta conclusión final.

Así el primer paso para crear el DSM es el diseño del metamodelo el cual se explica en el apartado II del presente capítulo. El siguiente capítulo resumirá los *Reports* necesarios para la generación del código.

II. Metamodelo

Las reglas de modelado en Metaedit+ no se representan con un metamodelo gráfico como ocurre con otras herramientas de diseño de DSM como GME. Metaedit+ emplea el lenguaje GOPPRR (Graph-Object-Property-Port-Role-Relationship). Las reglas de diseño son simplemente especificadas en los cuadros de diálogo explicados en el capítulo III del presente documento.

La programación del diseño consiste en especificar objetos, propiedades, puertos, relaciones, roles y gráficos que se van a usar así como las relaciones entre ellos.

Para comenzar se crearon cada uno de los objetos enumerados en el punto III del capítulo 4 junto a sus propiedades. La forma de crear un objeto y sus propiedades queda bien explicada en el capítulo 3, sin embargo es necesario incluir la posibilidad de añadir dependencias de un elemento del símbolo con las propiedades de un objeto. Esta herramienta es usada asiduamente en el proyecto ya que numerosas ocasiones la representación del símbolo depende del valor de sus propiedades. Por ejemplo, el objeto sensor mostrará un símbolo u otro dependiendo del valor escogido para su propiedad *Type*. Para conseguir este efecto primero se sitúan en el editor todas las imágenes posibles y seleccionando una a una con el botón derecho del ratón se escoge la opción *Format* y aparecerá el cuadro de diálogo mostrado en la Figura 103. En este caso la imagen seleccionada es la de la Figura 87 y se expresa con este cuadro de diálogo la condición de que no aparezca a no ser que se haya seleccionado *Humidity* en la propiedad *Type*.

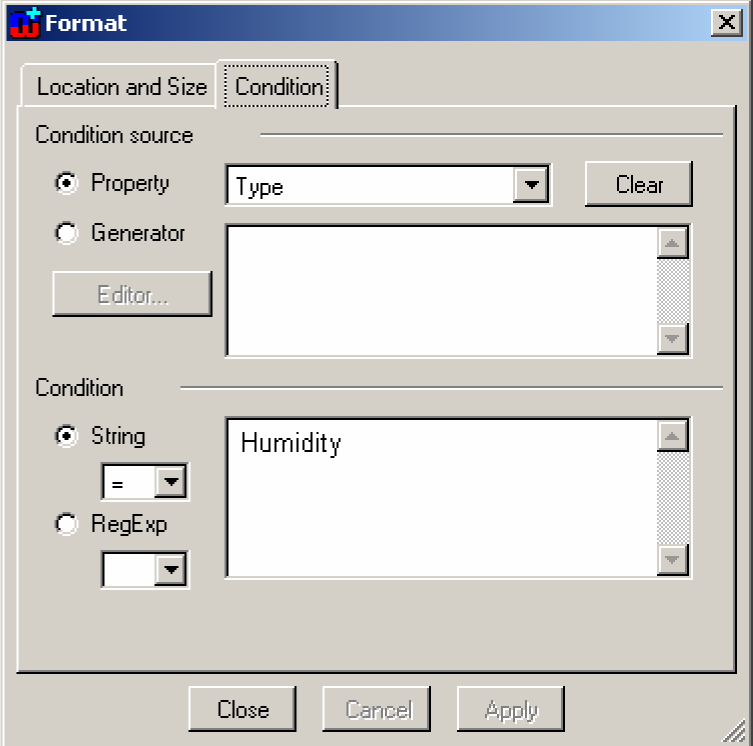


Figura 103: Dependencias de símbolos

Además de estos objetos y propiedades se ha creado la relación *Interface* la cual no tiene ninguna propiedad y únicamente sirve para conectar dos de los objetos mencionados. La Figura 104 muestra el cuadro de diálogo para la creación de dicha relación.

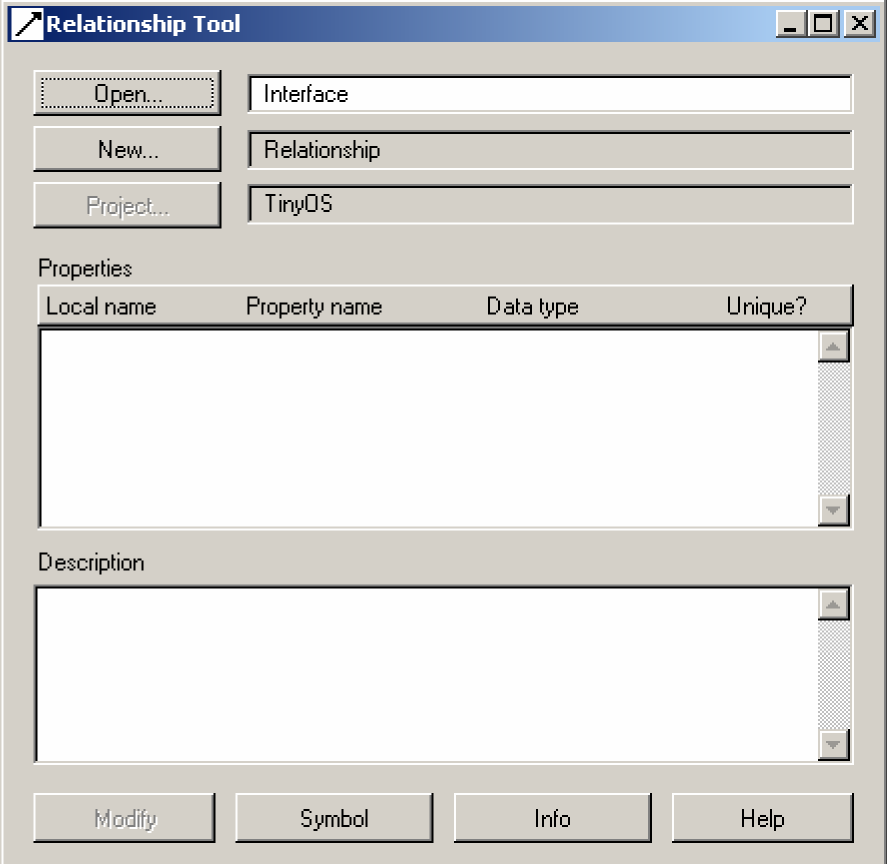


Figura 104: Creación de Interface

Son necesarios además los dos roles que dan cohesión a la relación. Siguiendo la notación de nesC, aunque su significado no sea exactamente el mismo, el nombre que se le da a los roles es *Provides* y *Uses*. Estos roles no tienen ninguna propiedad al igual que ocurre con la relación *Interface* vista. Solamente se usan para saber el origen y el destino de una relación entre dos objetos. Simplemente con el orden en el que se hace una conexión entre dos objetos a la hora de modelar se asignan los roles: *Uses* para el primer objeto que se seleccione y *Provides* para el segundo. Sin embargo, para dar más legibilidad a los modelos se ha optado por modificar el símbolo que da por defecto MetaEdit+ a los roles añadiendo una flecha al final del rol *Provides* como se muestra en la Figura 105.

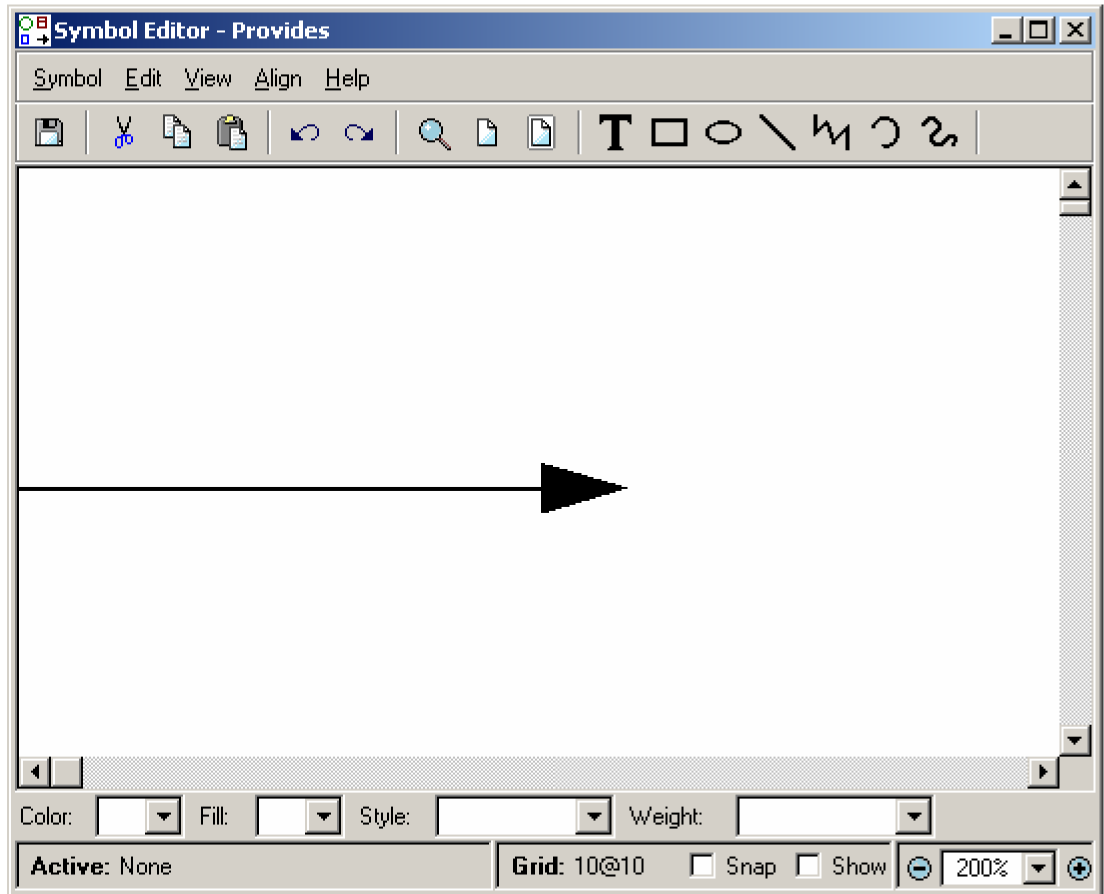


Figura 105: Símbolo Provides

En el proyecto no se han usado puertos diferentes evitando así restricciones de representación al programador. Por lo tanto usan todos los objetos el mismo tipo de puerto genérico que rodea la figura al completo. De esta forma la conexión a un objeto puede ser llevada a cualquier punto del objeto entendiendo el significado de esta únicamente por el rol usado y no por el puerto.

Por último se ha creado el gráfico que es donde realmente se encuentran las reglas de conexión de objetos. El orden de asignación de roles en una relación también se especifica en este cuadro de diálogo. El nombre dado al gráfico en cuestión es mote ya que se recrea en los aspectos de dominio del nodo sensor en sí y no en las redes automáticas. El cuadro de diálogo principal se muestra en la Figura 106 donde se aprecia que la única propiedad necesaria para este gráfico es la de *Name* la cual da nombre a la aplicación final generada desde el modelo. En el gráfico primero se especifican los objetos, relaciones y roles que se usarán en el modelado como muestra la Figura 107.

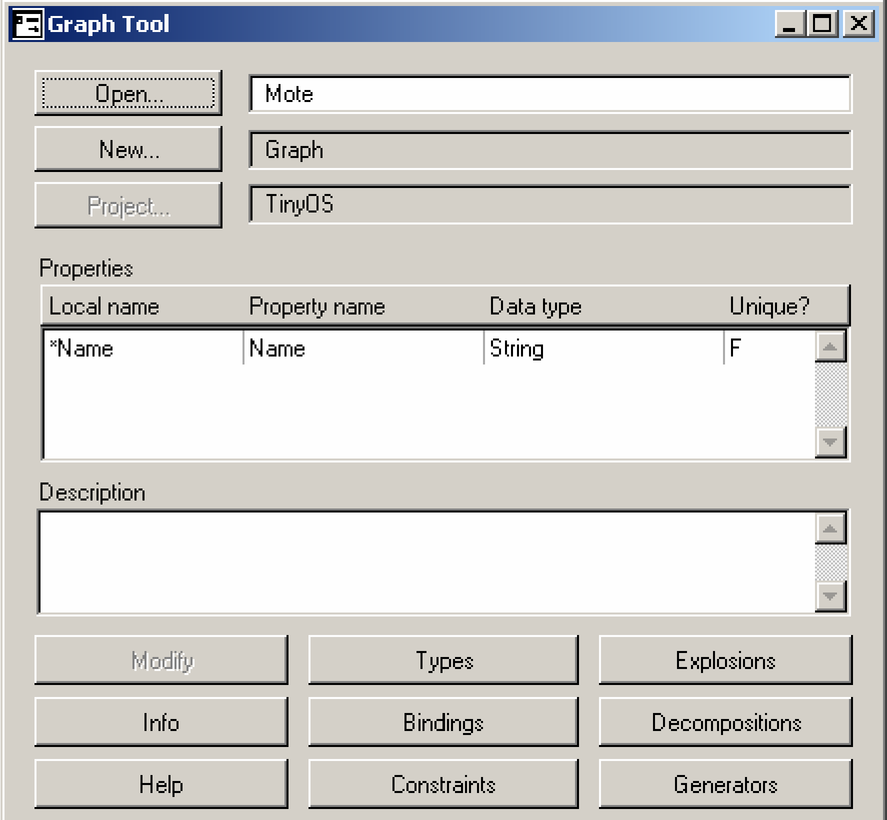


Figura 106: Creación de Mote

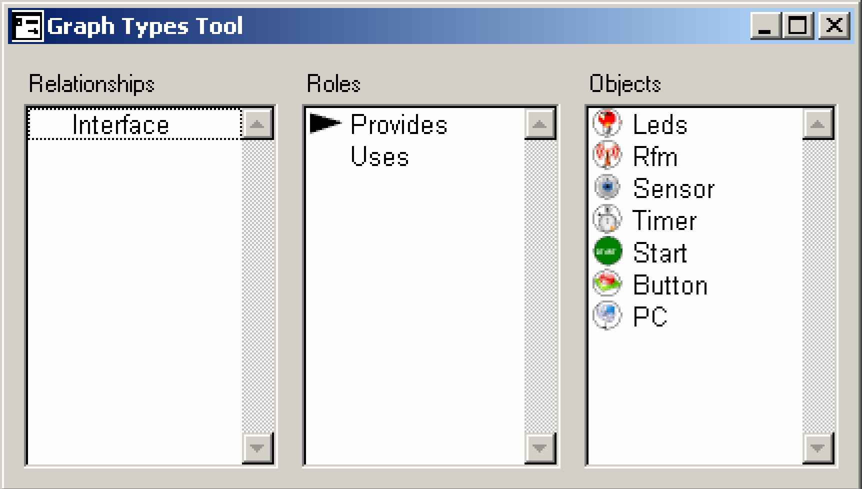


Figura 107: Graph Types

El siguiente paso consiste en definir las relaciones que existirán entre los diferentes objetos y a través de qué roles se efectuarán. Para comenzar a definirlas es necesario pulsar el botón *Bindings* del cuadro de diálogo principal. El proceso consiste en seleccionar la relación que se desea programar (en este caso la única llamada *Interface*) y acto seguido incluir los roles que van a intervenir (*Provides* y *Uses*). El orden en que se introducen los roles es importante ya que a partir de este orden se irán asignando a los diferentes componentes según se vayan seleccionando al crear una relación. Así, los roles que se han introducido aquí se han introducido en el orden lógico

Uses y *Provides*. La Figura 108 y la Figura 109 muestran dicho orden introducido. Además hay que especificar qué objetos podrán actuar en qué rol como se muestra en la cuarta columna de las citadas figuras. De esta forma se observa cómo el objeto *Leds* no se encuentra dentro del rol *Uses* ya que como hemos dicho anteriormente carece de sentido que este objeto sea el origen de un evento o dato. Ocurre lo mismo con el objeto PC debido a que queda sin implementar la recepción de paquetes desde el PC. En el caso contrario tenemos los objetos *Start* y *Button* que no se encuentran en el conjunto de objetos que pueden ser usados en el rol de *Provides*.

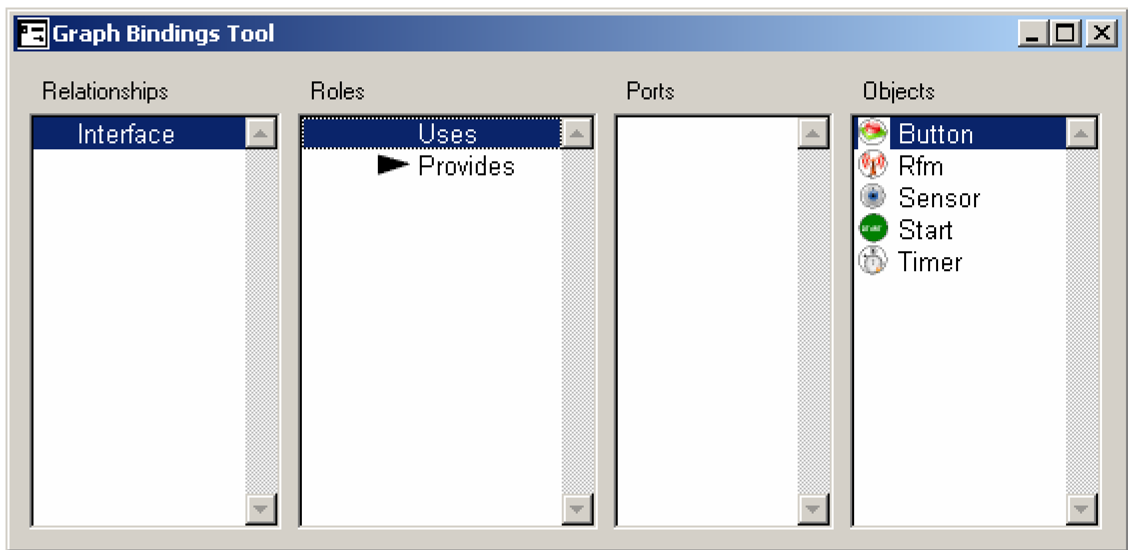


Figura 108: Graph Bindings Role Uses

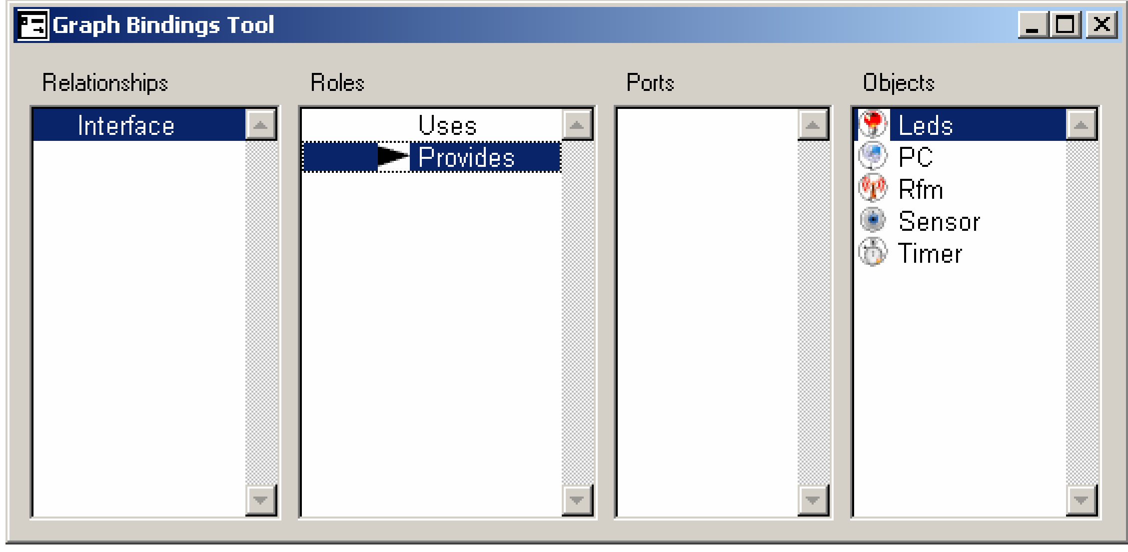


Figura 109: Graph Bindings Role Provides

III. Generación de código

Como se ha comentado se usa la generación de código mediante el lenguage de *Scripts* proporcionado por la herramienta. Aunque en un principio el objetivo del proyecto era crear únicamente el archivo de configuración de la aplicación nesC, se ha conseguido generar absolutamente todo el código ya que la descripción de la aplicación quedaba perfectamente definida en el modelo.

La filosifía a seguir es la de crear un componente nesC por cada objeto que se encuentre en el diagrama. Esto es así excepto para los objetos de radio en los que se tiene en cuenta el rol que toman para crear el componente TinyOS. Por ejemplo, si de un mismo componete radio actua en los dos roles posibles (*Uses* y *Provides*) como se muestra en la Figura 110 se crearán dos componetes, uno para cada rol. Una vez creados todos los componentes, sus interfaces se unen en el componente aplicación que tomará de nombre el que le hayamos puesto a la propiedad del gráfico. Entre todas las interfaces que proveen y usan los componetes toma especial relevancia en este proyecto la de *IntOutput* cuyo código se encuentra a continuación. Con esta interfaz se realizan las transferencias de datos o eventos entre los distintos componentes según hayan sido conectados en el modelo.

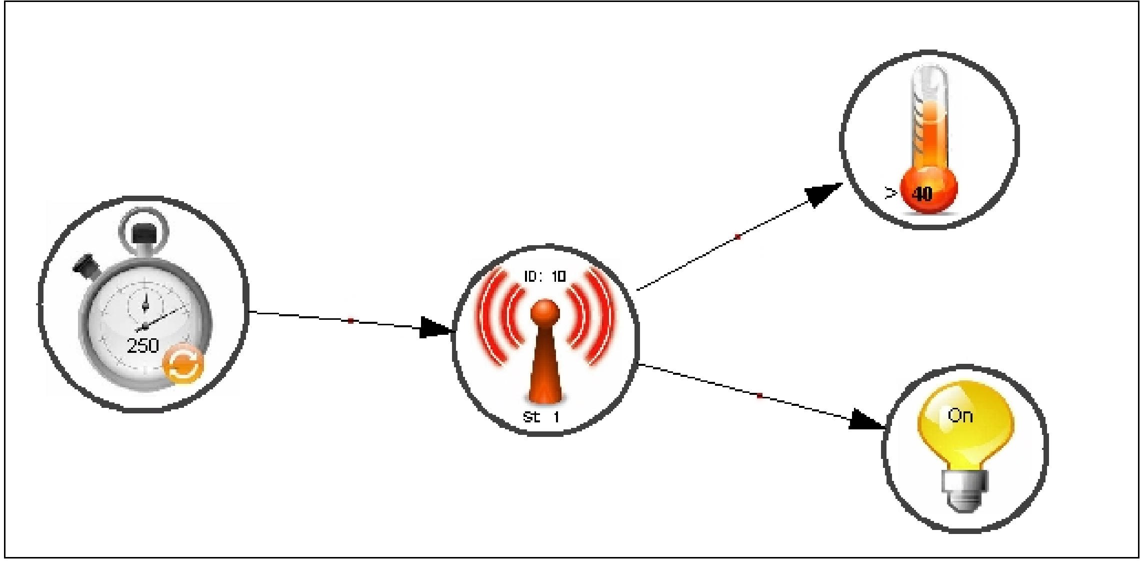


Figura 110: Rfm en ambos roles

IntOutput.nc

```
/**
 * Interface to an abstract ouput mechanism for integers. Two
examples
 * of providers of this interface are IntToRfm and IntToLeds.
 *
 * @author Jason Hill
 * @author David Gay
 * @author Philip Levis
```

```
 * @author Nelson Lee
 */

interface IntOutput {

  /**
   * Output the given integer.
   *
   * @return SUCCESS if the value will be output, FAIL otherwise.
   */

  command result_t output(uint16_t value);

  /**
   * Signal that the ouput operation has completed; success states
   * whether the operation was successful or not.
   *
   * @return SUCCESS always.
   *
   */
  event result_t outputComplete(result_t success);
}
```

Para la creación de los mencionados componentes se llama a un *script* principal (Application) que a su vez va llamando a los *scripts* correspondientes a los diferentes componentes. También se llama a otro *script* para la creación del archivo *Module* de la aplicación principal. Por último se llama al *script* que crea el archivo *Makerules* necesario para la programación en TinyOS del mote. A continuación se ofrece una tabla resumen de los distintos scripts creados para el proyecto.

Application	
Componente:	Componente aplicación con el nombre dado al gráfico x.
Archivos que genera:	*Configuration* de la aplicación principal: x.nc
Interfaces que provee:	-
Interfaces que usa:	-
Descripción: Es el script principal que va llamando al resto. Su principal complejidad reside en la correcta declaración de componentes necesarios y su conexión según el modelo	

descrito.

Module0	
Componente:	Módulo de la aplicación.
Archivos que genera:	*Module* de la aplicación principal: xM.nc
Interfaces que provee:	*StdControl*
Interfaces que usa:	*IntOutput*
Descripción: Es la generación del módulo de la aplicación principal por defecto. Es necesario para poder inicializar los sensores de humedad y temperatura implementados en el archivo *HumidityC*. Normalmente es sobreescrito por el script *Module* que se ejecuta cuando el modelo creado contiene el objeto *Start*.	

Module	
Componente:	Módulo de la aplicación.
Archivos que genera:	*Module* de la aplicación principal: xM.nc
Interfaces que provee:	*StdControl*
Interfaces que usa:	*IntOutput*
Descripción: Sobreescribe el archivo creado por Module0 en caso de que el modelo contenga el objeto *Start*. Además en su comando *StdControl.start* llama al comando *IntOutput.output*. De esta forma se crea el evento de inicialización expresado en el model mediante el objeto *Start*. La interfaz *IntOutput* es conectada en el archivo de configuración al componente que proveea dicha interfaz según el modelo creado.	

ToTimerTo	
Componente:	ToTimerTox_xxxx
Archivos que genera:	ToTimerTox_xxxx.nc ToTimerTox_xxxx.nc
Interfaces que provee:	*IntOutput* as *SignalProvides* *StdControl*
Interfaces que usa:	*Timer* *IntOutput* as *SignalUses*
Descripción: Se genera un componente por cada objeto *Timer* que se encuentre en el modelo. Los componentes creados se distinguirán en el nombre gracias al identificador único de objeto (oid) proporcionado por Metaedit+ con un formato numérico del tipo x_xxxx. Su principal misión es llamar al comando Timer.start con los parámetros especificados	

en las propiedades del objeto modelado. Llamará a dicho comando cuando se produzca la llamada al comando SignalProvides.output. Cada vez que acabe un ciclo de temporización se llamará al comando SignalUses.output (genera el evento).

ToSenseTo	
Componente:	ToSenseTox_xxxx
Archivos que genera:	ToSenseTox_xxxx.nc ToSenseTox_xxxxM.nc
Interfaces que provee:	*IntOutput* as *SignalProvides* *StdControl*
Interfaces que usa:	*IntOutput* as *SignalUses*
Descripción: Se genera un componente por cada objeto *Sensor* que se encuentre en el modelo. Los componentes creados se distinguirán en el nombre del mismo modo que lo hacen los componentes *ToTimerTo*. Su principal complejidad reside en el cálculo del umbral ya que es función de Metaedit+ la transformación de unidades expresadas en las propiedades del objeto (temperatura, luxes o % de humedad) a voltaje comparable con el proporcionado por los distintos sensores. La función de este componente es la de intentar tomar la medida especificada cuando su comando *SignalProvides.output* sea invocado y llamar al comando *SignalUses.output* cuando la medida esté lista y cumpla con el umbral establecido.	

ToRfm	
Componente:	ToRfmx_xxxx
Archivos que genera:	ToRfmx_xxxx.nc ToRfmx_xxxxM.nc IntMsgx_xxxx.h
Interfaces que provee:	*IntOutput* *StdControl*
Interfaces que usa:	-
Descripción: Es ejecutado por la aparición del rol *Provides* de cada objeto *Rfm*. Por lo tanto genera un componente (con su correspondiente estructura de mensaje) por cada vez que se ejecute diferenciandolos con el oid de Metaedit+. Su principal misión es enviar por radio los datos introducidos a través del comando *IntOutput.output* encapsulados en el paquete definido según las propiedades del objeto.	

ToUART	
Componente:	ToUARTx_xxxx

Archivos que genera:	ToUARTx_xxxx.nc ToUARTx_xxxxM.nc IntMsgx_xxxx.h
Interfaces que provee:	*IntOutput* *StdControl*
Interfaces que usa:	-
Descripción: Genera un componente (con su correspondiente estructura de mensaje) por cada vez que aparezca un objeto *PC* en el modelo diferenciandolos con el oid de Metaedit+. Su principal misión es enviar por a través de la UART los datos introducidos por el comando *IntOutput.output* encapsulados en el paquete definido en IntMsg.h según las propiedades del objeto.	

RfmTo	
Componente:	RfmTox_xxxx
Archivos que genera:	RfmTox_xxxx.nc RfmTox_xxxxM.nc IntMsgx_xxxx.h
Interfaces que provee:	*StdControl*
Interfaces que usa:	*IntOutput*
Descripción: Es ejecutado por la aparición del rol *Uses* de cada objeto *Rfm*. Por lo tanto genera un componente (con su correspondiente estructura de mensaje) por cada vez que se ejecute diferenciandolos con el oid de Metaedit+. Su principal misión es recibir por radio y desencapsular los datos según el paquete definido con las propiedades del objeto. Una vez se obtengan los datos éstos se mandarán a través del comando *IntOutput.output*. Si existen más de un dato en el mensaje, éstos se mandarán uno a uno.	

ToLeds	
Componente:	ToLedsx_xxxx
Archivos que genera:	ToLedsx_xxxx.nc ToLedsx_xxxxM.nc
Interfaces que provee:	*IntOutput* *StdControl*
Interfaces que usa:	-
Descripción: Genera un componente por cada objeto *Leds* en el modelo diferenciandolos con el oid de Metaedit+. Su objetivo es encender el led que se haya especificado en las	

propiedades del objeto o el indicar mediante los tres leds el valor de los tres bits más significativos del dato introducido a través del comando *IntOutput.output*.

ButtonTo	
Componente:	ButtonTox_xxxx
Archivos que genera:	ButtonTox_xxxx.nc ButtonTox_xxxxM.nc
Interfaces que provee:	*StdControl*
Interfaces que usa:	*IntOutput*
Descripción: Genera un componente por cada objeto *Button* encontrado en el modelo diferenciandolos con el oid de Metaedit+. Se llamará al comando *IntOutput.output* cada vez que haya una interrupción en el puerto especificado en las propiedades del objeto. Por defecto es el puerto 27 ya que corresponde al puerto usado por el *user button* situado en la placa del Mote TelosB.	

Makefile	
Componente:	-
Archivos que genera:	Makefile
Interfaces que provee:	-
Interfaces que usa:	-
Descripción: Genera el archivo necesario para compilar la aplicación. El código es el siguiente:	

```
filename; 'Metaedit\Makefile'; write;
'
COMPONENT=':Name'
include ../Makerules
'
close;
```

Símplemete se especifica el nombre de la aplicación con la propiedad *Name* del gráfico o modelo en cuestión.

Capítulo 6

Ejemplo de aplicación práctica

I. Escenario

Como ejemplo de aplicación práctica se va a suponer que una empresa desea monitorizar la temperatura de un determinado lugar (lugar 2) donde también se encuentra un trabajador y la humedad de otro (lugar 1) totalmente aislado al acceso humano. Los datos serán registrados en un PC situado en el centro de control (lugar 3). Además, en el caso de que la humedad relativa supere el 50% en el lugar inaccesible se ha de avisar a la persona del lugar 1 mediante el encendido de un led rojo para que ésta actúe oportunamente. La persona también podrá comunicarse con el centro de control (lugar 3, donde se encuentra el PC) conmutando un led amarillo allí mediante un botón. Los datos registrados en el centro de control se representarán en tiempo real en una gráfica donde aparezcan temperatura y humedad en colores distintos. El registro en el PC deberá tener una muestra de cada medida (temperatura y humedad) por cada 250 milisegundos. La Figura 111 muestra un diagrama de la situación y las comunicaciones necesarias.

No será de especial interés el tener los datos de las muestras en el centro de control conforme se vayan tomando sino que bastará mientras no superen 2,5 segundos de retardo (si las muestras se deben tomar cada 250ms no pueden retardarse más de 10 muestras). Sin embargo, sí que es de crítica importancia el hecho de que el trabajador del lugar 1 se entere inmediatamente de que en el lugar inaccesible (lugar 2) se haya superado el 50% de humedad.

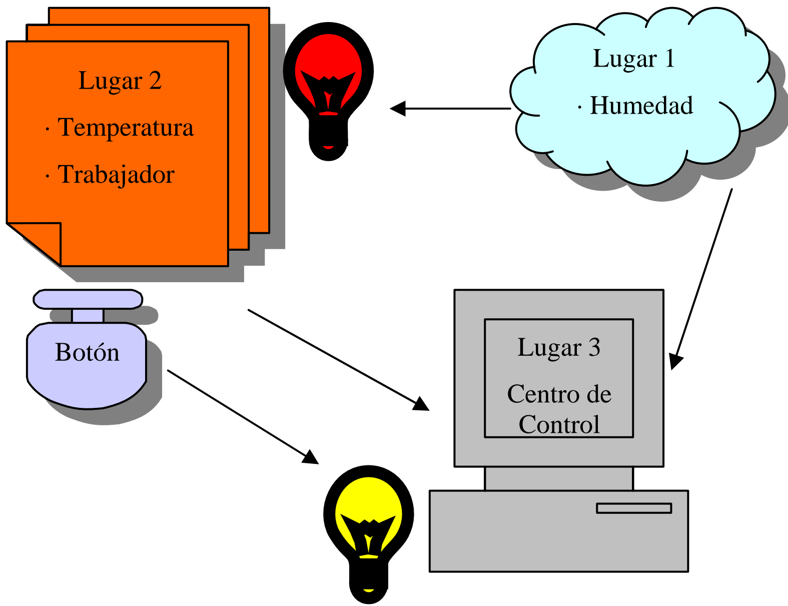


Figura 111: Escenario ejemplo

II. Solución

Es obvio que para la aplicación práctica propuesta se hará uso de tres nodos o motes, uno en cada lugar. Por lo tanto serán necesarios crear tres modelos diferentes, uno para cada nodo. Los modelos serán numerados del mismo modo en que están los lugares del escenario. A continuación se detalla el procedimiento para cada modelo.

i. Modelo 1

Para el modelado de la aplicación de este nodo antes es necesario saber qué elementos van a intervenir. En principio parece obvio que se necesitará un sensor de humedad que vaya tomando medidas. Es necesario indicar a este sensor cuándo se tomarán las medidas, y como se requiere, será necesario un temporizador que se repita cada 250 milisegundos y que comience a la puesta en marcha del nodo (objeto *Start*). Además es imprescindible un objeto *Rfm* para mandar las medidas tomadas al centro de control.

Con los objetos descritos hasta este punto se completa la primera cadena de proceso con la que quedarían cubiertos los requisitos para el registro de datos en el centro de control. Sin embargo, aún resta por cubrir el requisito de avisar al trabajador cuando se supere el 50% de humedad. Para este requisito serán necesarios dos nuevos objetos *Sensor* de humedad (uno para encender el led cuando se supere y otro para apagarlo) con el umbral al 50%. La propiedad *OneTime* en este case se deberá marcar ya que además de ser necesario el ahorro de baterías, no tiene sentido seguir mandando la orden de encender un led cuando ya está encendido. Unidos a estos objetos hacen falta sendos objetos *Rfm* que envíen el evento (encendido y apagado) al lugar 2 donde se encuentra el trabajador. Es importante diferenciar bien los identificadores de mensaje en las propiedades de los objetos *Rfm*. En éste caso los identificadores se han seleccionado de forma secuencial, es decir, 1 para las medidas hacia el centro de control, 2 para cuando sobrepase el umbral y 3 para cuando vuelva a estar por debajo. Es importante que estos identificadores junto con la propiedad *Store* coincidan en origen y destino.

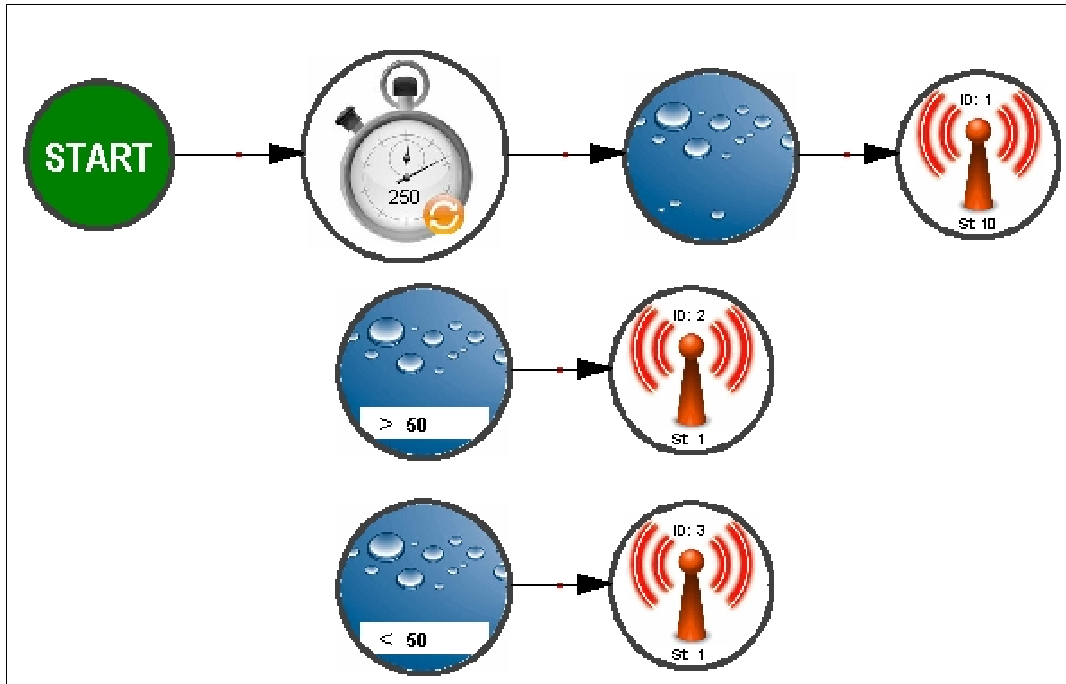


Figura 112: Modelo 1

Se ha establecido la propiedad *Store* del primer objeto *Rfm* a 10 ya que es el retardo máximo de muestras para enviar al PC según los requisitos del escenario. De esta forma, se consigue un sustancial ahorro de batería ya que no se enviarán mensajes hasta que no se hayan obtenido las 10 muestras que caben en su estructura de datos. Puesto que no era de especial importancia la monitorización de los datos en el centro de control en tiempo real, se ha dado prioridad al ahorro de batería. El resto de objetos *Rfm* sí que mandan la medida inmediatamente sea recibida ya que en este caso la batería no se verá afectada porque rara vez se cumplirá la condición de umbral. Además, era crítico el aviso al trabajador cuando se supere el 50% de humedad según los requisitos por lo que no se carecería de sentido acumular 10 muestras que sobrepasen el umbral para enviarlas.

ii. Modelo 2

Para comenzar a modelar el nodo 2 del ejemplo se comenzará con una cadena de proceso igual a la primera del modelo uno, aunque esta vez el sensor será de temperatura. Con esta primera cadena de proceso quedan cubiertos los requisitos de toma de medidas y registro de datos en el PC. Además de cambiar el sensor de humedad por uno de temperatura se ha cambiado el identificador de mensaje dándole el valor de 5 para poder diferenciar las medidas de temperatura y humedad en el centro de control.

El siguiente paso será procesar los paquetes enviados por el nodo 1. Para esto es necesario recibirlos con los mismos parámetros que los objetos *Rfm* con los que fueron enviados. Con ésto se deben situar en el modelo dos objetos *Rfm* con los identificadores 2 y 3 respectivamente. Puesto que el mensaje con identificador de paquete 2 se enviaría cuando se sobrepase el nivel umbral, la recepción de un paquete de este tipo se asocia al encendido del led rojo. Del mismo modo, como el envío de un paquete con ID 3 se realizaría cuando el porcentaje de humedad volviera a estar por debajo del umbral, la recepción de un paquete de este tipo se asocia al apagado del led rojo.

Por último, según los requisitos explicados en el escenario, cuando el trabajador situado donde está este nodo pulse un botón, se conmutará un led amarillo en el centro de control. Para hacer más legible la comunicación entre el trabajador y el centro de control, se encenderá el mismo led en su propio nodo (de esta forma sabrá cuando ha encendido el led del centro de control y cuando lo ha apagado). Para cumplir este requisito es necesario un objeto *Button* con la propiedad de puerto fijada a 27 (*user button*) y un nuevo objeto *Rfm* para poder transmitir el evento. Este nuevo objeto *Rfm* tendrá las propiedades de *Handler ID* y *Store* fijadas a 4 y 1 respectivamente. Además se debe añadir el objeto *Leds* correspondiente con la propiedad de color a *yellow* y la propiedad de Action a *Toggle*. La Figura 113 muestra el estado final del modelo.

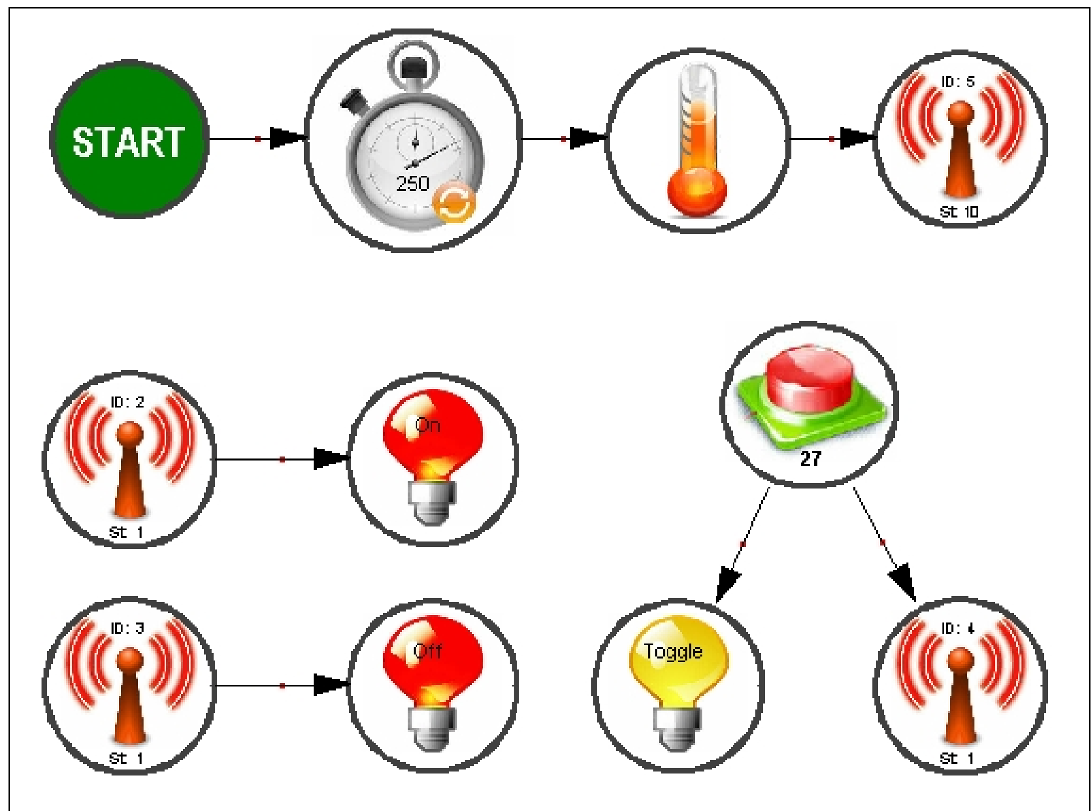


Figura 113: Modelo 2

iii. Modelo 3

Éste es el modelo más sencillo de los tres ya que lo único que debe hacer es tomar los datos recibidos de medidas y pasarlos al PC. Además debe con el encender el led amarillo cuando se reciba el evento de botón pulsado del modelo 2. Para esto sólo son necesarios 3 receptores de señal (objetos *Rfm*), dos objetos *PC* y un objeto led. Los objetos PC tendrán en la propiedad *Channel* los valores 1 y 2 para las antenas con ID 1 y 5 respectivamente (esto es *Channel* 1 para humedad y *Channel* 2 para temperatura). La propiedad *Channel* no es crítica para el modelado de una aplicación cualquiera y simplemente existe para hacer compatible la estructura de mensaje enviado a la UART con la aplicación *oscilloscope* proporcionada en las herramientas de TinyOS. Por tanto aquí, la aplicación *oscilloscope* representará como channel 1 las muestras llegadas de humedad y como channel 2 las muestras llegadas de temperatura.

Es importante establecer las propiedades de los objetos *Rfm* de forma idéntica a sus análogos emisores para garantizar el correcto funcionamiento de la comunicación. Por lo tanto aquí las propiedades *Store* de los objetos *Rfm* correspondientes a las medidas se establecen en 10 como la lo esta en los emisores.

Por último se añade un objeto *Leds* con las mismas propiedades que el led amarillo del modelo anterior. Es decir, con la propiedad *Color* fijada en *yellow* y la propiedad *Action* fijada en *Toggle*. Con todos estos objetos, el modelo final queda representado en la Figura 114.

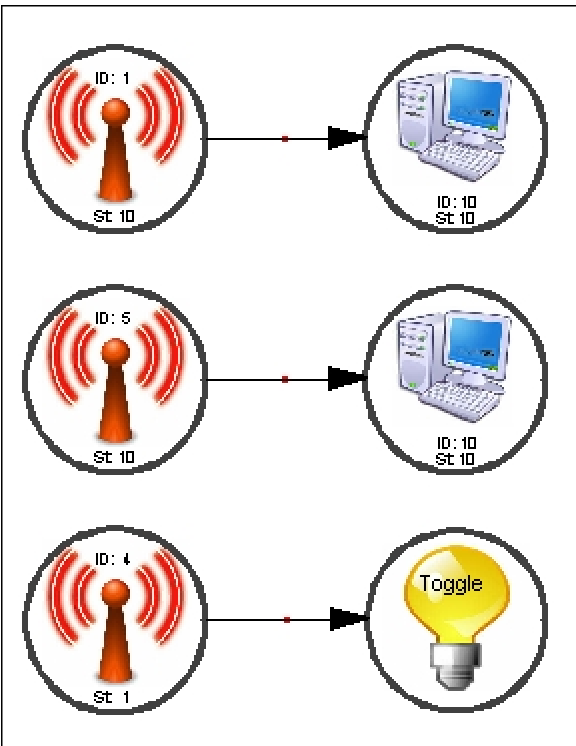


Figura 114: Modelo 3

III. Puesta en marcha

Para la puesta en marcha del sistema se programan cada uno de los nodos con los modelos correspondientes. El código será generado automáticamente con la herramienta y únicamente habrá que compilarlo e instalarlo en el nodo para lo cual se ejecuta el comando correspondiente en consola:

```
Make install telosb
```

En el PC donde será conectado el nodo 3 se hace uso de las herramientas *SerialForwarder*, y *Oscilloscope* proporcionadas en la instalación de TinyOS. Estas herramientas facilitan el registro y representación de datos procedentes de un nodo sensor conectado al puerto USB. En la Figura 115 se puede ver la representación de los datos que se van obteniendo en tiempo real en el centro de control. La gráfica roja muestra la humedad que ha ido detectando el sensor del lugar 1 y la gráfica verde muestra la temperatura que hay desde el lugar 2. Se observan dos picos de humedad en los que se ha encendido el led rojo en el lugar 2 como se especificaba en los requisitos. Una vez informado el trabajador, éste ha actuado de forma conveniente para corregir la humedad (ha elevado la temperatura en el lugar 2 en este caso) y el diodo led rojo se ha apagado. Además el diodo led azul (en este caso, ya que se usan dispositivos TelosB que sustituyen el diodo amarillo por uno azul) se ha encendido y apagado correctamente cuando se pulsa el botón del nodo del lugar 2.

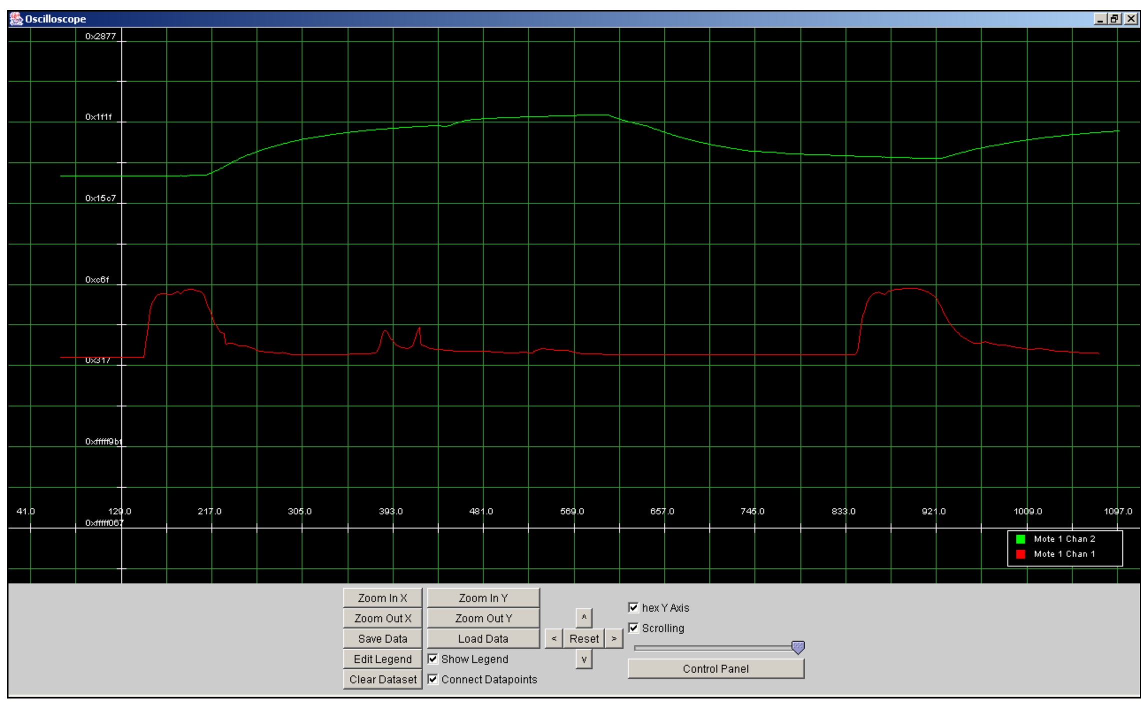


Figura 115: Centro de control

Se concluye por tanto en que la aplicación es un éxito y que la facilidad y el ahorro de tiempo para su desarrollo suponen un gran avance en este tipo de tecnologías. Además la ampliación de la aplicación para posibles cambios de escenario es tan sencillo como modificar el modelo de la forma conveniente.

Capítulo 7
Conclusiones y líneas de futuro

I. Conclusiones

Con el modelado específico de dominio aplicado a las redes de sensores inalámbricas se reduce considerablemente el esfuerzo en la construcción de aplicaciones. Este avance en la ingeniería software se consigue a costa de reducir las posibilidades de programación a las mínimas comunes del dominio filtrando las que nunca o casi nunca se vayan a usar. A partir de la reducción de posibilidades de programación es relativamente fácil crear el lenguaje específico de dominio capaz de expresar sin ambigüedades el propósito de una aplicación.

El desarrollo de este proyecto ha sido posible gracias a que el dominio de las aplicaciones considerado en este caso era verdaderamente reducido. Las redes de sensores inalámbricas consiguen aportar infinidad de soluciones en casi todos los campos con el mínimo de herramientas posibles (sensores y emisor/receptor radio). Puesto que el dominio que se ha considerado es muy general, los objetos y relaciones entre ellos son también muy reducidos. En el caso de que se hubiera profundizado sobre un determinado campo dentro de las redes de sensores (por ejemplo: monitorización de aves) el dominio habría cambiado sustancialmente con lo que hubiera sido necesario crear un DSL con aspectos más complejos (para el ejemplo: tipos de aves a monitorizar o algoritmos de encaminado de paquetes).

Respecto a la herramienta Metaedit+ ha resultado verdaderamente útil su sencillez de manejo y la tremenda rapidez en la que se puede crear un DSM. Las herramientas y el método que aporta para la definición de objetos, propiedades, relaciones, roles y puertos son en mi opinión las más acertadas para crear un DSL con notación gráfica. También su uso para la creación de modelos una vez definido el metamodelo resulta relativamente sencillo. Sin embargo no se puede decir lo mismo del lenguaje de script para la generación de código que resulta tosco, pesado y con reducidas posibilidades Los desarrolladores de la herramienta son conscientes de ello y proponen el uso de aplicaciones externas para crear el código a partir de un archivo generado con la información del modelo.

Los objetivos iniciales de este proyecto se han alcanzado en buena medida ya que en un principio se propuso la generación de código semiautomática (únicamente se crearía el archivo de configuración de la aplicación principal) y se ha acabado generando todo el código necesario para programar un nodo. Se ha adquirido una buena experiencia tanto en la herramienta Metaedit+ como en el sistema operativo TinyOS. La documentación presentada puede ser de utilidad para aquellos que quieran introducirse tanto en el terreno de la herramienta usada como en el de la plataforma.

II. Líneas de futuro

Como posibles ampliaciones al proyecto para mejorarlo o complementarlo se proponen las siguientes:

Para el objeto *Rfm* se podrían desarrollar las mejoras expuestas a continuación:

- Emisión de paquetes con selección de dirección (no BroadCast) ya que actualmente como parámetros del objeto antena se toma únicamente el identificador de mensaje.
- Posibilidad de cifrado en la emisión/recepción de paquetes. Esta característica resulta realmente interesante en este tipo de tecnologías inalámbricas.
- Mejoras en el orden de los datos enviados en un solo paquete. Actualmente se pueden enviar varias medidas en un solo paquete pero estas medidas se van ordenando conforme van llegando al objeto. A veces no se puede controlar el orden de llegada de las medidas y por tanto sería interesante seleccionar la posición de la medida dentro de un mensaje.

Como ampliaciones algo más complejas para el proyecto se proponen:

- Diseño de un nuevo tipo de componentes e interfaces de TinyOS en los que exista una distinción real entre evento y dato. Actualmente para mandar un evento entre dos objetos se envía un dato con valor nulo lo cual es un desperdicio de memoria importante.
- Adición de posibilidad de escribir en memoria EEPROM. Actualmente no existe apenas documentación en este campo y el modelado de un objeto así resulta más complejo de lo que en principio aparenta.
- Recepción de paquetes desde el PC. Actualmente el PC sólo puede recibir mensajes desde el nodo sensor. Sin embargo resulta de vital importancia la posibilidad de poder actuar sobre los nodos ya que estos están preparados para ello.
- Adición de sensores y actuadores externos. Esto satisfaría una de las mayores demandas del mercado de esta tecnología.

Capítulo 8

Bibliografía y referencias

[1] Fernando Losilla, Bárbara Álvarez, Pedro Sánchez Palma. "Una experiencia de modelado de los sistemas teleoperados para limpieza de cascos de buques mediante características y casos de uso genéricos" IV Jornadas de trabajo DYNAMICA. Actas del Workshop Archena, Murcia (Spain) 17-18/11/2005.

[2] Pedro José Meseguer Copado. “Programación de redes de sensores inalámbricas para aplicaciones domóticas”. Proyecto final de carrera. Universidad Politécnica de Cartagena

[3] Jun Han, Abhishek Shah, Mark Luk, Adrian Perrig. “Don’t Sweat Your Privacy. Using Humidity to Detect Human Presence” Junio 2007.

[4] Javier Luis Cánovas Izquierdo, Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina. “Utilidad de las transformaciones modelo-modelo en la generación automática de código”

[5] Pablo Gómez Palarea, Óscar Sánchez Ramón. “Herramientas de Metamodelado: Microsoft DSL Tools vs MetaEdit+”. Proyecto final de carrera. Universidad de Murcia.

[6] Janne Luoma, Steven Kelly, Juha-Pekka Tolvanen “Defining Domain-Specific Modeling Languages: Collected Experiences”. MetaCase

[7] Domain-Specific Modeling With MetaEdit+: 10 Times Faster Than UML. MetaCase

[8] Keeping it in the family. Letters from the Front. July-August 2002

[9] www.metacase.com

[10] http://cuartageneracion.blogspot.com

[11] http://www.dsmforum.org/

[12] www.hamamatsu.com

[13] www.sentilla.com

[14] http://www.sensirion.com

[15] http://focus.ti.com

[16] http://www.tinyos.net/

[17] http://dis.um.es/~jmolina/pfc.html