UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Máster Oficial en Sistemas Inteligentes y

Aplicaciones Numéricas en Ingeniería



Trabajo Final de Master

# Entorno de desarrollo integral para el lenguaje de modelado Tara

**Octavio Roncal Andrés**

**Tutores:** José Juan Hernández Cabrera

José Évora Gómez

22 de diciembre de 2014

# Agradecimientos

A mis padres, a mi familia canaria, a José Juan y José por la gran ayuda y por supuesto a mi futura esposa.

# Índice general

# Índice de figuras

# Resumen

Una de las grandes preocupaciones en el sector del desarrollo de software es mejorar la productividad, reducir los tiempos de desarrollo y garantizar que el software pueda evolucionar de forma flexible con el fin de que el producto software satisfaga las necesidades del cliente con el menor coste posible. El Model Driven Engineering (MDE) es una metodología que surge con estos objetivos. Para lograrlo, MDE se apoya en el uso de lenguajes específicos de dominio (DSL) que permiten expresar lógicas de negocio con un lenguaje cercano al ámbito del dominio.

El resultado de este trabajo ha sido un entorno de desarrollo para la creación de DSLs utilizando a su vez un DSL de creación de lenguajes. Este entorno de desarrollo da soporte a la creación y edición del DSL deseado. Además, da soporte a la creación y edición de modelos con los lenguajes creados. Este trabajo contribuye al marco de MDE así como a los lenguajes específicos de dominio.

# Capítulo 1

# Motivación y contextualización

Hoy en día, uno de los grandes retos del negocio de desarrollo de software es mejorar la productividad y la flexibilidad a la vez que se reducen los riesgos implícitos en el reto. Estos ambiciosos objetivos se ven diluidos en la mayoría de los desarrollos de software que se dan hoy en día. Esto se debe a que los sistemas de software modernos están alcanzando niveles de complejidad muy altos. Los sistemas pueden incluir millones de líneas de código, y cualquiera de ellas puede ser la causante del colapse de todo el sistema. Según la ley de la Complejidad Creciente de Lehman [1]: "*A medida que un programa en evolución cambia, su estructura tiende a ser cada vez más compleja*".

En la ingeniería de software uno de los puntos claves para que un producto llegue a buen término es identificar los requisitos de usuario. Estos requisitos suelen ser cambiantes debido al progresivo aumento del conocimiento a cerca del problema a medida que se acomete. Durante el proceso se puede querer introducir variables o cambiar partes en el desarrollo de la aplicación. Esto crea una incertidumbre en el proceso de desarrollo. Esta incertidumbre creada o bien por la inseguridad del cliente a cerca de sus requisitos, o bien debido a un problema de comunicación entre el cliente y los

desarrolladores. Si este problema no es considerado desde el inicio podría conllevar un incremento en los costes de mantenimiento y mejora de un sistema de software.

Un aumento de costes de mantenimiento y mejora impide dedicar recursos a su propia evolución obteniendo así un decremento de la calidad y por tanto de satisfacción del usuario[1]. La consecuencia final de la disminución de la satisfacción de los usuarios supone poner directamente en riesgo la inversión realizada sobre el mismo. La Figura 1.1 muestra la situación habitual en la creación de aplicaciones mediante desarrollos poco flexibles a los cambios de requisitos. En ella se muestra la desalineación entre los requisitos esperados por el cliente y los realizados por el equipo de desarrollo.

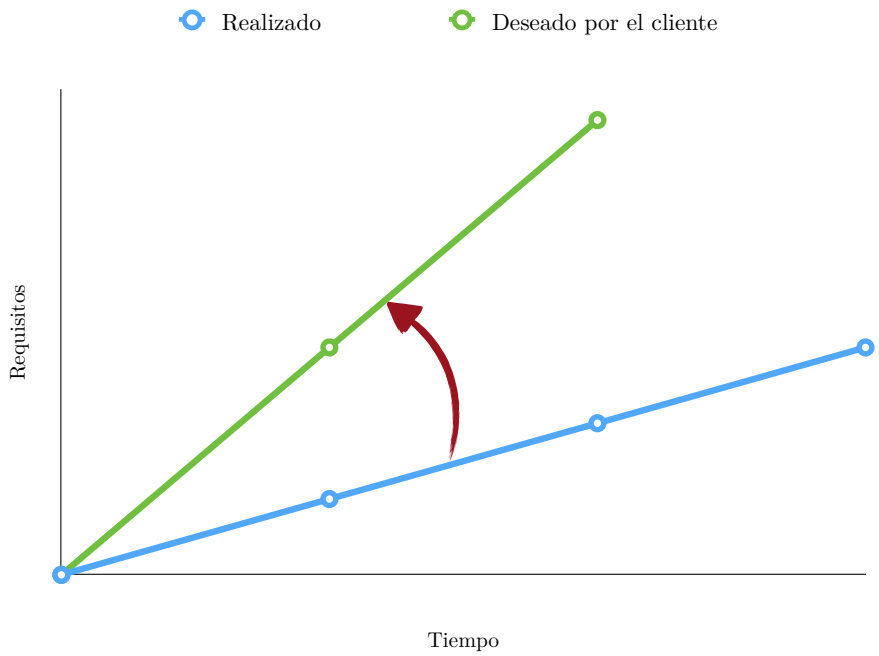


Figura 1.1: Resultado habitual en un desarrollo de software

*¿Cómo vamos a hacer frente a este aumento de la complejidad sin perder la flexibilidad necesaria?*

Existen múltiples enfoques al respecto, como el uso de frameworks de desarrollo, los cuales tratan de simplificar el desarrollo y pueden aportar

flexibilidad tecnológica, pero no flexibilidad entendida como la capacidad de adoptar nuevos requisitos de usuario. Otro enfoque es el Model-Driven-Engineering (MDE) que surge al identificar el modelo como parte esencial dentro del diseño software. El concepto de modelo esta presente en los dos patrones de diseño de software que son mayormente adoptados en la actualidad, el Model-View-Controller (MVC)[2] y el Model-View-Presenter (MVP)[3] debido a que ayudan a simplificar el desarrollo de software elevando el nivel de abstracción sobre el cual se crea el software.

En la actualidad, las implementaciones más significativas de MDE son el Model-Driven-Architecture (MDA) de Object Management Group (OMG)[4], el Software Factories de Mircrosoft[5][6] y Model Integrated Computing (MIC)[7]. En este trabajo se han descartado estos enfoques debido a su lenta curva de aprendizaje y poca flexibilidad.

Dado que el este trabajo está insertado en la linea de investigación en MDE del SIANI, iniciada en 1997, se ha optado por el enfoque de la ingeniería dirigida por modelos interpretados: Generated Models Driven Engineering (GMDE)[8]. A diferencia de otros enfoques de MDE, GMDE tiene como principal característica el desacoplamiento entre el modelo y el metamodelo. Con este enfoque el motor que interpreta el modelo no tiene el requisito previo de conocer el metamodelo ya que ambos comparten una misma estructura profunda que sería el metametamodelo.

Se nos plantean las siguientes preguntas

*¿Cuál es la mejor manera de especificar ese modelo?*

*¿Cómo este modelo puede ser entendible de manera sencilla por alguien cercano al dominio?*

Existen múltiples lenguajes para describir un modelo, desde lenguajes de marcado, como XML, lenguajes gráficos como UML[9] o lenguajes de propósito general. Además de los anteriores, se encuentran los Domain Specific Language (DSL) los cuales aportan mayor nivel de abstracción que los

lenguajes genéricos y expresan conceptos específicos del dominio en un nivel de representación más alto obteniéndose un lenguaje más natural y cercano al cliente. En consecuencia, el cliente o experto del dominio, puede involucrarse en mayor medida en el desarrollo del sistema consiguiendo un mayor ajuste entre el resultado final y el esperado por el cliente. En la Figura 1.2 se muestra el resultado esperado, en el que los requisitos del cliente están mas alineados con el resultado final tanto en número como en tiempo empleado.

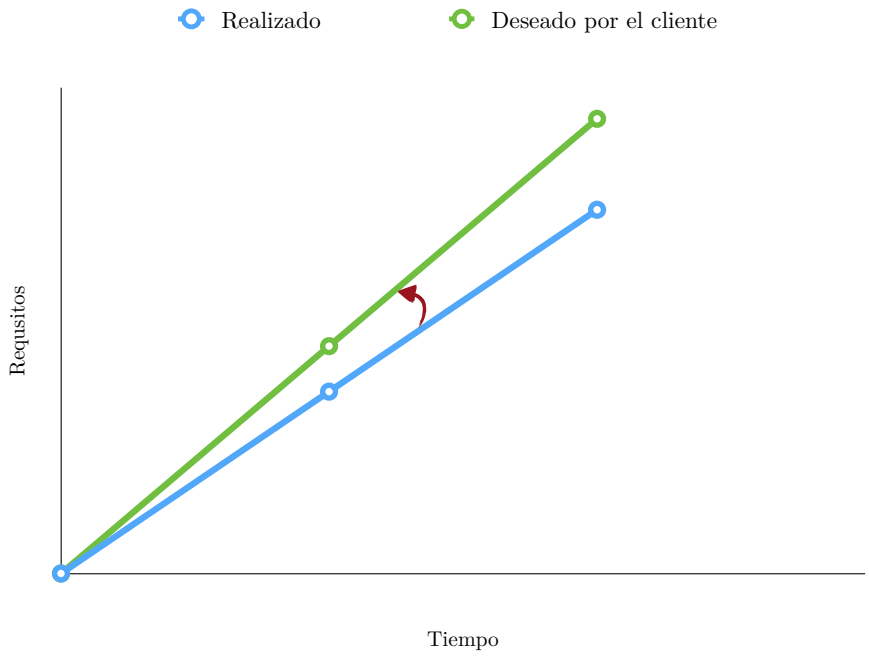


Figura 1.2: Resultado esperado en un desarrollo de software

En el mundo de la creación de software existen compañías involucradas en el desarrollo de aplicaciones de diferentes ámbitos de dominio, estas pueden ser motores de simulación, sistemas de información, cuadros de mando, definición de bases de datos, sensorización, eBusinesses... Estas compañías necesitan un DSL para cada dominio del problema. La creación de lenguajes de este tipo es una tarea compleja y pueden estar sometidos a frecuentes cambios por los expertos del dominio.

Debido a que éste es un problema creciente debido al aumento del uso de DSLs, en este documento se analizan soluciones a la siguiente pregunta,

*¿Es posible mejorar la productividad en la creación y mantenimiento de diferentes DSL?*

# Capítulo 2

# Estado del arte

La forma habitual de gestionar la complejidad del desarrollo de software es mediante el uso de la abstracción, la descomposición del problema y la separación de asuntos (separation of concerns)[10]. El desarrollo de software dirigido por modelos (MDE) es una aproximación que enfoca la gestión de la complejidad mediante el uso de otro tipo de lenguajes. En vez de usar lenguajes próximos a la tecnología (3GL)[11], la idea es usar lenguajes mas expresivos y cercanos al dominio del problema, Lenguajes Específicos de Dominio (DSL). Con estos lenguajes se pueden realizar descripciones precisas y detalladas del mundo real con el que el software tiene que tratar. Estas descripciones son modelos que representan algún aspecto del mundo. Por ello, también se les llama lenguajes de modelado. Ahora bien, a pesar de que estos lenguajes pueden representar fielmente la realidad, no expresan bien como operativamente deben funcionar. El reto realmente consiste en que estos modelos puedan convertirse en productos software que resuelvan las necesidades de usuarios reales.

A continuación se describen los conceptos mas relevantes referentes al problema y el estado actual de los mismos.

## 2.1. Modelos y metamodelos

De manera general un modelo se describe como una abstracción de la realidad que la representa de una manera simplificada[12]. Los modelos permiten representar una parte de la realidad vista por las personas que utilizan ese modelo para entender, cambiar, manejar y controlar esa parte de la realidad[13].

En el contexto de la ingeniería de software, el modelo de dominio es un modelo conceptual que representa los elementos que existen en la realidad de un problema específico. Este modelo describe las diferentes entidades, sus atributos, roles y relaciones entre ellas, así como las limitaciones que gobiernan el dominio del problema.

El concepto de modelo está presente en diversos patrones de diseño como el Model-View-Controller[2] o el Model-View-Presenter[3] que ayudan a simplificar el desarrollo de software.

Un metamodelo es descrito como un modelo que define otro modelo. Los metamodelos representan y especifican modelos, es decir, describen los elementos válidos de un modelo. De manera más precisa, un metamodelo enuncia qué puede ser expresado en los modelos válidos de un cierto lenguaje de modelado[14].

Por tanto, un metamodelo constituye las especificaciones de un lenguaje de modelado[14]. Un lenguaje, descrito por un metamodelo, puede tener un propósito específico o dominio en donde se aplica. De este modo, es posible capturar la semántica de todos los modelos disponibles para un determinado ámbito de conocimiento a través de un lenguaje[15].

La noción de metamodelo se basa en la arquitectura por niveles adoptada por el consorcio OMG en la especificación del Meta-Object-Facility[16].

## 2.2. Arquitectura de niveles

Un metamodelo identifica qué puede ser expresado en los modelo válidos de un cierto lenguaje de modelado[14]. El concepto de metamodelo está basado en la arquitectura de metadatos adoptada por el consorcio OMG en la especificación del Meta-Object Facility (MOF)[16].

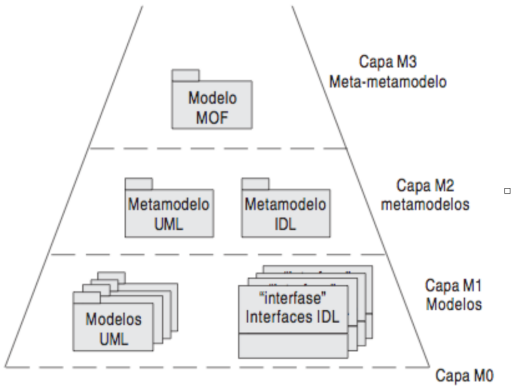


Figura 2.1: Ilustración de Meta-Object Facility

MOF está diseñado como una arquitectura de cuatro capas o niveles: información, modelos, metamodelos y meta-metamodelos. La figura 2.1 representa la arquitectura.

El nivel M3 constituye el meta-metamodelo. Este nivel es utilizado para construir los metamodelos, nivel M2. Un ejemplo es el metamodelo UML (Unified Modeling Language), que define al propio UML. Los elementos de M2 describen los modelos descritos del nivel M1. Ejemplos de modelos son, por ejemplo, diseños específicos escritos en UML. El último nivel es M0, donde se describen los objetos del mundo real.

La figura 2.2 ilustra esta arquitectura de niveles para el dominio concreto sistemas de información. En el nivel de metamodelo (M2) se definen agentes y entidades. Asimismo, los conceptos de M1 permitirían, por ejemplo, realizar un censo de quién vive en qué edificio.

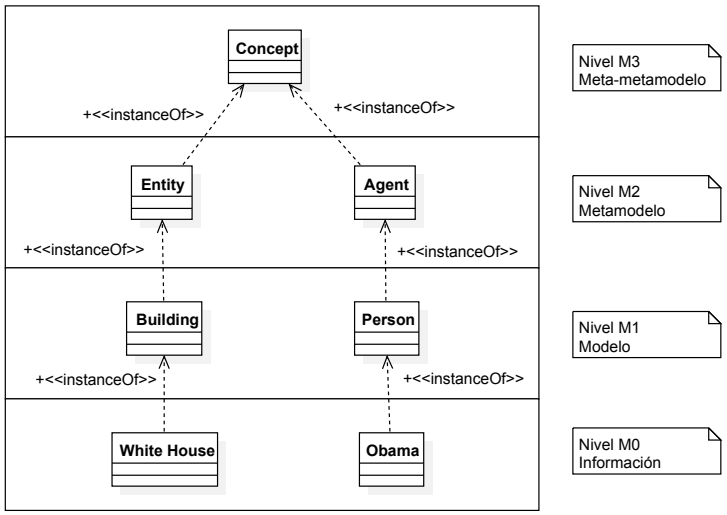


Figura 2.2: Ejemplo UML de la arquitectura de niveles

La especificación MOF es análoga en el caso de los lenguajes. De este modo habría un lenguaje primigenio y lenguajes derivados de él en niveles inferiores.

## 2.3. Model Driven Engineering

El diseño de sistemas software en ciertos ámbitos de conocimiento puede resultar una tarea ardua, especialmente, cuando el dominio del problema tiene una especial complejidad y está en continua evolución.

En este sentido el Model Driven Engineering[17] (MDE) ofrece mecanismos para afrontar estos problemas. MDE es una metodología para el desarrollo de software basada en modelos. Es decir, la funcionalidad del software es descrita en modelos y el código es generado en base a dichos modelos.

MDE utiliza diversos niveles de abstracción entre el diseño del software y su implementación. Estos niveles permiten diseñar el software en un lenguaje de alto nivel donde se describen los conceptos con una semántica cercana al dominio del problema. En este sentido, no solo los desarrolladores, sino

también los expertos en el dominio pueden tener un papel importante en el desarrollo del software.

Asimismo, la combinación de diversas tecnologías MDE permiten solventar el problema de la complejidad en los sistemas [17]. Principalmente, MDE se basa en los siguientes conceptos:

- Domain-specific modelling language (DSML). Lenguajes que permiten formalizar la estructura, comportamiento y requisitos del software en un dominio particular[18]. Los DSMLs son descritos utilizando metamodelos que definen los conceptos y las relaciones entre ellos para un dominio concreto. La idea principal es describir los elementos de manera declarativa, en lugar de imperativa.

- Generadores o motores (engines) que analizan los modelos escritos en un determinado DSML para generar el código fuente o representaciones alternativas del modelo.

Los DSMLs son utilizados en la actualidad para dotar de mayor expresividad sintáctica y semántica del dominio al software. De este modo, se reduce la curva de aprendizaje y facilita a los expertos a asegurarse que el sistema cumple los requisitos [14]. Además, las herramientas de MDE imponen restricciones al dominio que permiten detectar y prevenir errores en las etapas iniciales de desarrollo.

### 2.3.1. Dominios donde se aplica

Existen ámbitos de conocimiento muy diversos en los que se utiliza MDE. Un ejemplo es el campo de la simulación. El framework Tafat[19] usa MDE para la creación de simuladores de sistemas complejos. Su objetivo es acelerar el proceso de construcción de simuladores. Concretamente, el uso de esta metodología ha sido explotado en Tafat para la construcción de simulaciones

de “Smart Grids” [20], facilitando así la toma de decisiones en la implantación de estrategias para la eficiencia energética de las redes eléctricas.

Otro ejemplo es la aplicación de MDE en eBusinesses. En esta investigación [8] se orienta la ayuda de pequeñas organizaciones que disponen de medios limitados para definir su proyección organizativa y tecnológica. La investigación concluye que esta aproximación ayuda a este tipo de organizaciones y a administraciones públicas a modelar sus estructuras de servicio.

El MDE también se ha adentrado en el campo de la bioinformática. En [21] evalúan su uso para resolver problemas surgidos en el estudio de los caminos de señales (reacciones en una célula como reacción a un estímulo). Los autores afirman que MDE aporta una mayor eficiencia en el desarrollo, mayor claridad de los datos y un manejo muy intuitivo de los mismos.

En el ámbito industrial hay empresas que aplican técnicas de MDE. Es el caso de Motorola[22] que lleva cerca de veinte años utilizándolas. A lo largo de estos años han visto como la introducción de técnicas de coordinación y control han supuesto un gran aumento en términos de calidad y productividad.

Una investigación realizada en [23] evalúa la posibilidad de utilizar MDE para el desarrollo de aplicaciones en redes de sensores. Los autores buscan un enfoque completamente diferente al resto de propuestas de su mismo ámbito. Como resultado obtienen una aplicación que reduce sensiblemente el esfuerzo y tiempo de producción.

En el campo de desarrollo de aplicaciones móviles también se plantea el uso de MDE. En concreto para predecir el rendimiento de un determinado diseño[24], donde esta metodología posibilite a los desarrolladores entender rápidamente las consecuencias de decisiones arquitectónicas.

Otro estudio introduce el uso de MDE en el diseño de sistemas multiagentes[25]. A partir de los resultados extraídos de un caso de uso concluyen que MDE aporta principalmente flexibilidad.

### 2.3.2. Investigación

MDE constituye un campo de investigación en auge para investigadores de todo el mundo. Resulta interesante conocer qué otras entidades u organizaciones trabajan con MDE, así como el ámbito de sus investigaciones.

A continuación se detallan algunas de las instituciones cuyos investigadores publican un mayor número de trabajos sobre MDE.

A. *Institut National de Recherche en Informatique et en Automatisme* (INRIA). Algunas de sus investigaciones comprenden el diseño de lenguajes de modelado[26] o la creación de frameworks basados en MDE para ámbitos específicos como la paralelización de sistemas empotrados[27].

B. *Universite Lille nord de France*. Investigadores de esta universidad buscan la manera de garantizar la interoperabilidad entre modelos de diferentes plataformas[28].

C. *Universidad de Murcia*. Estudian aspectos como la creación de lenguajes específicos de dominio basados en los principios de MDE[29] o la posible aplicación de MDE en pequeñas empresas de desarrollo software[30].

D. *Universite de Toulouse*. Trabajan con MDE para crear modelos de sistemas multiagentes adaptativos[31], optimizar de sistemas de tiempo real[32] o para el desarrollo de sistemas empotrados distribuidos[33].

E. *Universidad de Oviedo*. Una de sus líneas de investigación trata la construcción de un framework para el desarrollo de todo tipo de aplicaciones[34][35]. Para ello mezclan diversos enfoques del MDE.

La figura 2.3 recoge la localización de estas instituciones. Se ha incluido el Instituto Universitario SIANI[36] (punto F), en el cual desarrollamos nuestra investigación entorno al MDE.

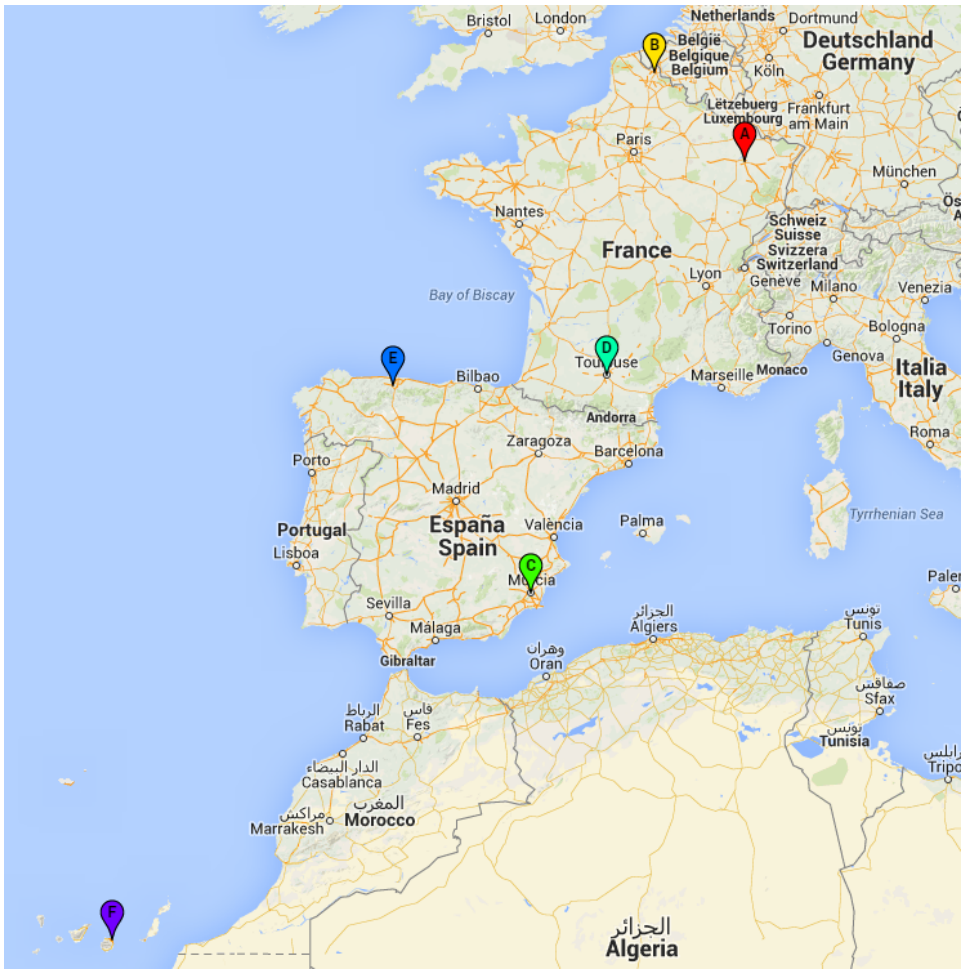


Figura 2.3: Mapa de instituciones que investigan MDE.

Toda esta información se ha obtenido de las bases de datos de investigación WoS[37] y Scopus[38], buscando por los términos "Model-Driven Engineering" y "MDE" publicaciones recientes (últimos 5 años).

## 2.4. Línea de investigación de GMDE SIANI

En general, cualquier aplicación opera con algún modelo de la realidad (M0). Normalmente, este modelo es persistente en algún tipo de base de datos u otro medio de almacenamiento. Así, todo software se diseña para in-

terpretar un tipo modelo, por tanto podemos decir que el software incorpora un metamodelo (M1). Es decir, un modelo del modelo (figura 2.4).

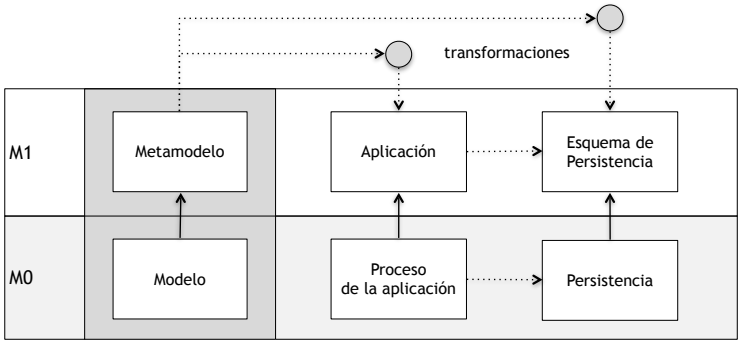


Figura 2.4: Paradigma clásico en MDE.

En el MDE clásico, un metamodelo expresado en un lenguaje específico de dominio (DSL) se transforma en un software. Conceptualmente es correcto, dado que una aplicación puede concebirse como un metamodelo capaz de interpretar un modelo. Para poder llevar a cabo este enfoque, deben existir herramientas que soporten la transformación automática de los metamodelos. Incluso, estas herramientas deben, no sólo ofrecer la posibilidad de aplicar transformaciones predefinidas, sino ofrecer también un lenguaje que permita a los desarrolladores definir sus propias transformaciones y ejecutarlas bajo demanda. Existen diversas aproximaciones a la transformación de modelos:

- Direct Model Manipulation (Pull transformation). Estas herramientas ofrecen al usuario acceso a la representación del modelo y la capacidad para transformarlos mediante APIS.

- Intermediate Representation. El modelo se exporta a una representación estándar (XML) que puede ser usada por otra herramienta para transformarlo.

- Transformation Language Support. Se ofrece un lenguaje que permite expresas y componer transformaciones. Podemos afirmar que para realizar estas transformaciones es necesario que estos procesos interpreten a su vez de alguna forma el metamodelo. En cierto modo, el conocimiento para interpretar el metamodelo y transformarlo en software está incorporado en estos procesos de transformación.

No obstante, cabe la posibilidad de plantear otra alternativa que permita usar estos metamodelos para producir software que no se base estrictamente en la transformación. Concretamente, la línea de investigación en la que está inserto este trabajo la visión es producir un software a partir de modelos sin realizar estas transformaciones del metamodelo.

La idea está inspirada en la teoría de la gramáticas generativas de Chomsky. Esta teoría afirma que al dominar un lenguaje, somos capaces de comprender un número indefinido de expresiones que no hemos oído anteriormente y que no tienen parecido a las expresiones que constituyen nuestra experiencia lingüística. Chomsky explica que en todos los lenguajes subyace una gramática generativa que puede derivar diferentes estructuras sintácticas. Por ello, frases construidas con estructuras sintácticas distintas pueden ser perfectamente interpretadas por una persona, ya que conoce la estructura profunda de la frase. Inspirados en esta teoría, la idea consiste en definir una gramática que contemple la derivación de diferentes metamodelos. Así, al derivarse de una misma gramática, todos los metamodelos pertenecerían a una misma familia y por tanto compartirían la misma estructura profunda. A esta línea de investigación la hemos llamado Ingeniería dirigida por Modelos Generativos (Generative Models Driven Engineering - GMDE)

Si se concibe que cualquier aplicación software se desarrolla para manipular modelos representados con un metamodelo, la visión consiste en construir un motor de propósito general que pueda manipular modelos representados con metamodelos de una misma familia.

La clave está en el motor de propósito general que puede operar directamente con un modelo sin necesidad de conocer el metamodelo específico ya que la estructura profunda del modelo es la misma. No obstante, para la ejecución de operaciones o comportamientos específicos del dominio, el propio motor podría invocar la ejecución de comportamientos particulares del metamodelo. Para especificar la gramática generativa del metamodelo se propone la formalización de un metametamodelo (M2). Así, el metametamodelo contiene los componentes y la gramática generativa que permiten especificar diferentes metamodelos. A su vez, el metametamodelo se formaliza mediante los componentes y gramática definidos en el metametametamodelo (M3) (figura 2.5).

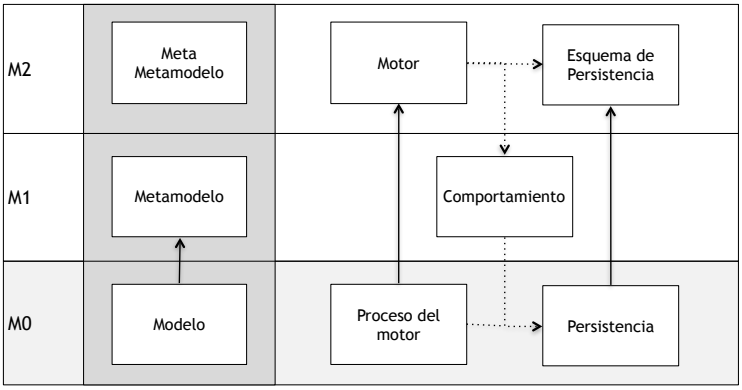


Figura 2.5: Visión del GMDE.

## 2.5. Domain Specific Languages

En contraste con los lenguajes de propósito general, los lenguajes específicos de dominio (Domain Specific Languages) son lenguajes de programación o lenguajes de especificación ejecutables que ofrecen, a través de anotaciones apropiadas y abstracciones, expresividad[39]. Por lo general se enfocan a un dominio de problema particular. Algunos ejemplos de DSL pueden ser el

lenguaje R para estadística, Mathematica para cálculos matemáticos o SQL para bases de datos relacionales.

Los DSL son generalmente declarativos. Como consecuencia, pueden ser vistos como lenguajes de especificación, así como lenguajes de programación. Entre sus ventajas destacan que permiten expresar soluciones en un idioma cercano al problema y que mejoran la productividad, mantenimiento y portabilidad. Por contra, requieren un coste tanto de diseño e implementación, como de educación a los usuarios finales del lenguaje.

Los DSLs se pueden clasificar según la construcción del lenguaje en:

- Internos: Utilizan un lenguaje anfitrión para darle la apariencia de otro lenguaje.
- Externos: Disponen de su propia sintaxis y es necesario un reconocedor (parser) para poder procesarlos.

Desde el punto de vista del dominio del problema en:

- Horizontales: el usuario final que utilizará el lenguaje no pertenece a ningún dominio específico de dominio.
- Verticales: el usuario que utilizará el lenguaje pertenece al mismo dominio del lenguaje. Por ejemplo, un lenguaje para la definición de simulaciones de redes eléctricas, donde los usuarios finales sean expertos encargados de especificar dichas simulaciones.

En el ámbito del Model Driven Engineering los DSL son ampliamente utilizados para diseño de modelos expresivos y cercanos al dominio de aplicación.

# Capítulo 3

# Hipótesis

El marco de desarrollo de la ingeniería dirigida por modelos es aplicable a productos de diversos ámbitos del conocimiento, como los sistemas de información, simuladores, creación de portales web, modelado de bases de datos, etc. Esto implica que para cada ámbito sea necesario un lenguaje en el que se puedan expresar los modelos. Es posible utilizar lenguajes de propósito general para esta tarea, pero ninguno aporta la semántica y enfoque de un lenguaje específico del dominio. De esta forma, lo ideal sería tener un lenguaje específico para cada dominio que se presente. Sin embargo este enfoque requiere mucho trabajo y un esfuerzo por parte diseñador del lenguaje en conocer el dominio al que va dedicado el lenguaje. Por otro lado, si un usuario quisiera desarrollar modelos para diferentes dominios, debería aprender los lenguajes en los que se especifica cada uno.

A partir de esta reflexión, nos planteamos las siguientes preguntas

*¿Es posible expresar un DSL con un meta-DSL?* y *¿Podría tener el meta-DSL la misma gramática que el DSL?*

Estas preguntas nos dirigen a plantear la siguiente hipótesis:

**Si los dominios pueden expresarse como un conjunto de conceptos parametrizables y asociados entre sí por relaciones de composición, agregación, abstracción y concreción debiera ser posible diseñar un meta DSL capaz de crear DSL para los diferentes dominios.**

## 3.1. Estructura de los lenguajes

Un lenguaje de modelado, de forma básica, está compuesto por un léxico, una sintaxis y una semántica [40]. De esta forma, el léxico se puede definir como un conjunto de tokens que dependen del dominio al que esté destinado el lenguaje; la sintaxis como un conjunto de reglas que definen como esos token se relacionan y la semántica como la definición del significado de cualquier sentencia sintácticamente correcta.

En la figura 3.1, se puede ver la relación existente entre los componentes de un lenguaje. Tanto la sintaxis como la semántica tienen su base léxico.

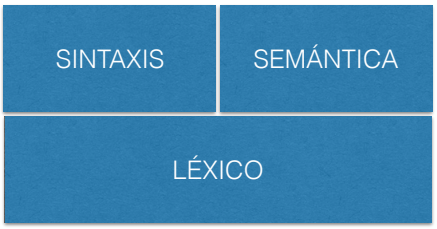


Figura 3.1: Componentes de un lenguaje de programación.

En un lenguaje, el léxico, la sintaxis y la semántica, podrían ser descritas en un modelo. Ese modelo debería expresar el conjunto de tokens que se pueden expresar en el lenguaje, la relación entre ellos (sintaxis) y el significado de esas relaciones (semántica). Para definir este modelo, se puede usar un lenguaje, concretamente, un meta-lenguaje[41]. Si las meta-relaciones entre

los meta-tokens y su meta-significado son los mismos que los del lenguaje, entonces solo es necesario cambiar el conjunto de tokens para crear un lenguaje nuevo. Este proceso es descrito en la Figura 3.2. Se aprecia como todos les lenguajes derivados comparten la misma sintaxis y semántica al conjunto de las cuales se les ha denominado Proteo.

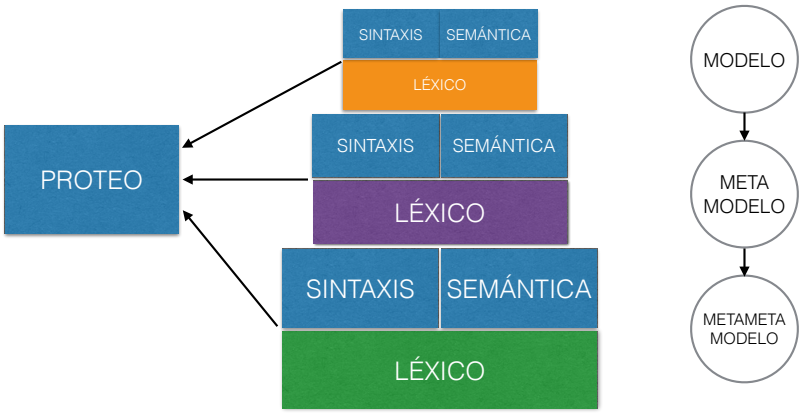


Figura 3.2: Creación de lenguajes a partir de modelos.

## 3.2. Tara. Definición del lenguaje

Tara pretende ser un meta-DSL que permita especificar lenguajes específicos de dominio. En los ámbitos de dominio descritos en la problemática se pueden identificar conceptos pertenecientes al domino. Por ejemplo, en los sistemas de información existen los conceptos de formulario, campo, tesauro, resumen, etc.

En los entornos de simulación de "Smart Cities" se encuentran los conceptos de simulación, edificio, dispositivo, planta de energía, calle, o transformador entre otros. En los cuadros de mando se encuentran indicadores, cubos, hechos... Estos conceptos pueden necesitar ser parametrizados, como por ejemplo un edificio esta parametrizado por su área, su volumen o

su localización. En un sistema de información, un campo de tipo fecha tiene como parámetro la precisión de dicha fecha.

Tara es un DSL para la creación de otros DSL, es declarativo[42] y está basado en conceptos. Estos conceptos se relacionan entre sí, pueden tener una dirección (address), contener variables, facetas y anotaciones. La Figura 3.3 muestra una porción de código escrito en Tara en el que se describe un ejemplo de la estructura de un formulario en un sistema de información.

```
Concept Form
    has Field

Concept Field is component named addressed
    var boolean enabled = true is terminal

    Concept Look is property
        var string label

    sub Text
        var string value is terminal
        Concept Look extends Look
            Concept Format is property
                var word value > Url; Email; Password
            Concept Length is property
                var natural min = 0
                var natural max
```

Figura 3.3: Ejemplo de la definición de un campo de un formulario.

Hasta el momento se han identificado las siguientes relaciones entre conceptos:

1. Composición. Dos conceptos están estrictamente limitados por una relación complementaria. El constituyente no se entiende sin el constituido, es decir, separado no tiene sentido. El tiempo de vida de cada uno de los objetos es dependiente. En la Figura 3.4 se puede ver la representación en UML de la relación y en la Figura 3.5 se ven dos formas de expresar la misma relación, el libro está compuesto de capítulos.

2. Agregación (Acumulación). Dos conceptos se consideran usualmente como independientes y aun así están ligados. Se puede decir que es de tipo todo/parte. El tiempo de vida de cada uno de los objetos es

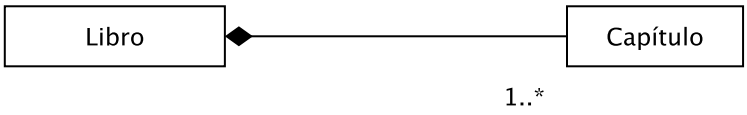


Figura 3.4: Representación UML de la relación de composición.

```
Concept Libro
    Concept Capítulo


Concept Libro
    has Capítulo

Concept Capítulo
```

Figura 3.5: Descripción de una composición de conceptos en Tara.

independiente. En la Figura 3.6 se muestra una la representación UML de la relación y en la Figura 3.7 su correspondiente representación en Tara.

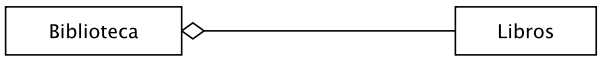


Figura 3.6: Descripción de una agregación de conceptos.

3. Especialización. La especialización es una relación existente entre dos o más conceptos. Marca una relación de jerarquía entre conceptos, en la cual, un concepto principal (padre) puede ser heredado por otros secundarios (hijos), los cuales adquieren "por herencia" los atributos del primero (concepto principal).

   Por ejemplo: El presupuesto tradicional (Padre). Un presupuesto extendido, que emita los recibos de pagos correspondientes, calculando el montante de adelanto y su resto.

4. Generalización. La acción opuesta de la especialización es generalización. La especialización es obtenida completando algunos detalles. La

```
Concept Biblioteca
    Concept Libro is aggregated

Concept Biblioteca
    has Libro is aggregated

Concept Libro
```

Figura 3.7: Descripción de una agregación de conceptos.

generalización es obtenida eliminando algunos detalles. En la Figura figura 3.8 se puede ver que el juego es una abstracción de parchís y ajedrez, y a su vez ellos son especializaciones de juego. En la Figura 3.9 se muestra las dos formas de representar esta relación entre conceptos. Por un lado utilizando la palabra reservada, la cual expresa en este caso que *ajedrez* extiende, o lo que es lo mismo, es una especialización de *juego*. La segunda forma consiste en expresar que *parchís* es un "sub" de juego. El resultado es el mismo, permitiendo esta última una forma más compacta de expresión.

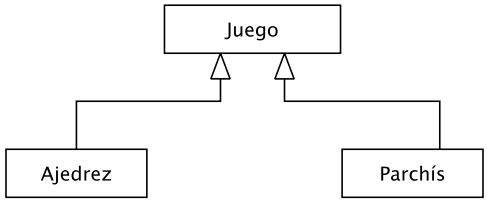


Figura 3.8: Descripción de una generalización entre conceptos

```
Concept Juego
    var integer numeroDeJugadores

Concept Parchis extends Juego
Concept Ajedrez extends Juego


Concept Juego
    var integer numeroDeJugadores
    sub Parchis
    sub Ajedrez
```

Figura 3.9: Descripción de una generalización entre conceptos escrita en Tara

Aplicar una técnica bajo condiciones específicas es especializar la técnica.

5. Concreción o instanciación. La concreción es la relación existente entre un concepto de un modelo y un o varios conceptos de un modelo de nivel inferior dentro de la jerarquía de modelos. En la Figura 3.10 se muestra como Building es una concreción de Concept y a su vez Tristan Peralta es una concreción de Building.

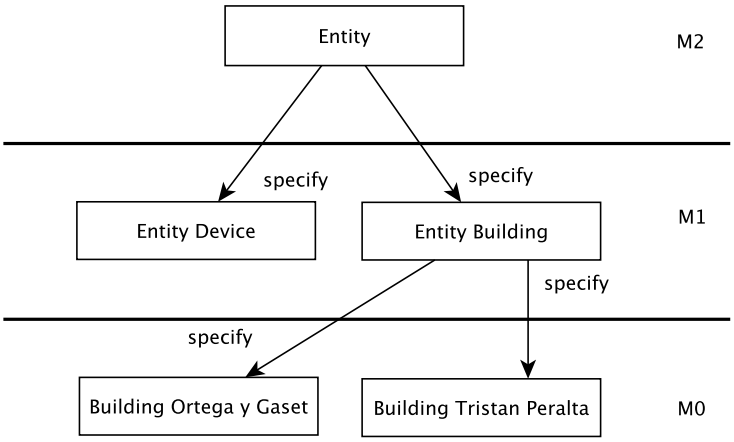


Figura 3.10: Descripción de una concreción entre modelos

6. Abstracción. La abstracción es el procedimiento inverso al de la concreción. Dado un concepto de nivel n, este es posible abstraerlo dentro de un concepto del modelo de nivel superior n+1.

### 3.2.1. Direcciones

Las direcciones (addresses) son un mecanismo de localización de conceptos que permite encontrar de manera única a conceptos utilizando una dirección. Esta dirección es en formato IPv4, como por ejemplo, '192.168.1.100'. Estas direcciones permiten al motor que interpreta el modelo buscar instancias de conceptos a partir de su dirección. Además son una ventaja para el

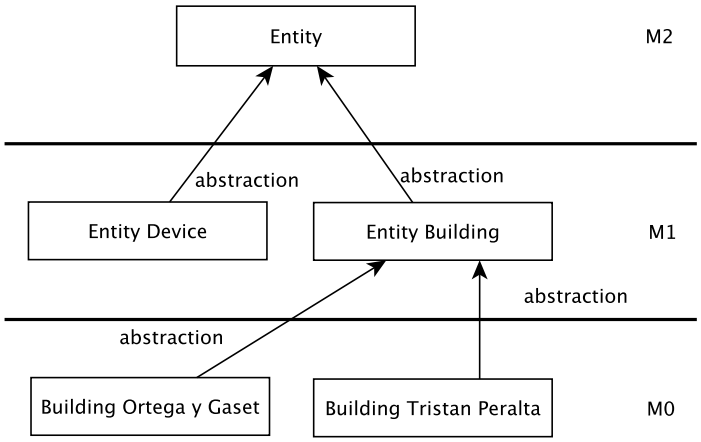


Figura 3.11: Descripción de una abstracción entre modelos

modelador, el cual si no es necesario puede evitar ponerle nombre al concepto. Estas direcciones, como se ha visto antes, pueden ser variables. De esta forma un concepto puede tener la referencia a otro objeto del modelo utilizando una dirección pudiendo navegar de un concepto a otro.

### 3.2.2. Variables

Las variables pueden ser de los siguientes tipos:

- Primitivos: Natural, Integer, Double, Boolean, String.
- Enumerados (Word).
- Recursos (resources) del proyecto.
- Direcciones.
- Referencias a otros conceptos

Las variables pueden tener valores por defecto. Las referencias sólo pueden tener el valor por defecto de “empty” que aportan un valor pero nulo. Las variables que no tienen valores por defecto deben inicializarse en el constructor de la instancia o asignándole valor en el cuerpo de la misma. Las

variables que sí tienen valores por defecto pueden inicializarse en el constructor o ser modificadas en el cuerpo del concepto.

Todos los tipos de variables pueden tener valores múltiples. Esto implica que la instancia de esa variable puede tener múltiples valores. Además las variables de tipo Double, pueden ser listas acotadas a un tamaño fijo.

### 3.2.3. Facetas

Las facetas son mecanismos que permiten aportar funcionalidades o comportamientos a otros conceptos. No son instanciables per se, sino que se aplican sobre las instancias de otros conceptos.

Inicialmente se declara un concepto como faceta. Estos conceptos pueden tener:

- Variables
- Conceptos componentes.
- Targets (on). Al menos uno. Se especifica cual es el objetivo de la faceta. Cada target puede tener sus propias variables y componentes.

En el siguiente nivel, las instancias de los conceptos pueden implementar alguna de las facetas definidas en el nivel superior. Esto implica que el modelador debe describir, usando un lenguaje de propósito general, cómo es ese comportamiento.

Un concepto al que se le ha aplicado una faceta ya no es extensible.

La Figura 3.12 muestra un ejemplo gráfico del uso de facetas. Ésta muestra una simple representación de edificios que contienen radiadores y aire acondicionado. Estos últimos son eléctricos.

En las Figuras 3.13, 3.14 y 3.15 se describe el uso de facetas usando Tara. En la Figura 3.13 se describe un modelo en el que existen entidades las cuales pueden agregar más entidades y comportamientos.

En la Figura 3.14 se crea una instancia de Entidad que será un edificio que agrega radiadores y aires acondicionados. Además se define un comportamiento eléctrico que se puede aplicar sobre los radiadores y aires acondicionados. Este comportamiento está parametrizado por la potencia para ambos y el modo para el radiador que por defecto está apagado.

En la Figura 3.15 se crea la ultima concreción, en la que existe un edificio llamado Bécquer que contiene un radiador Ferroli cuya potencia es de 100W.

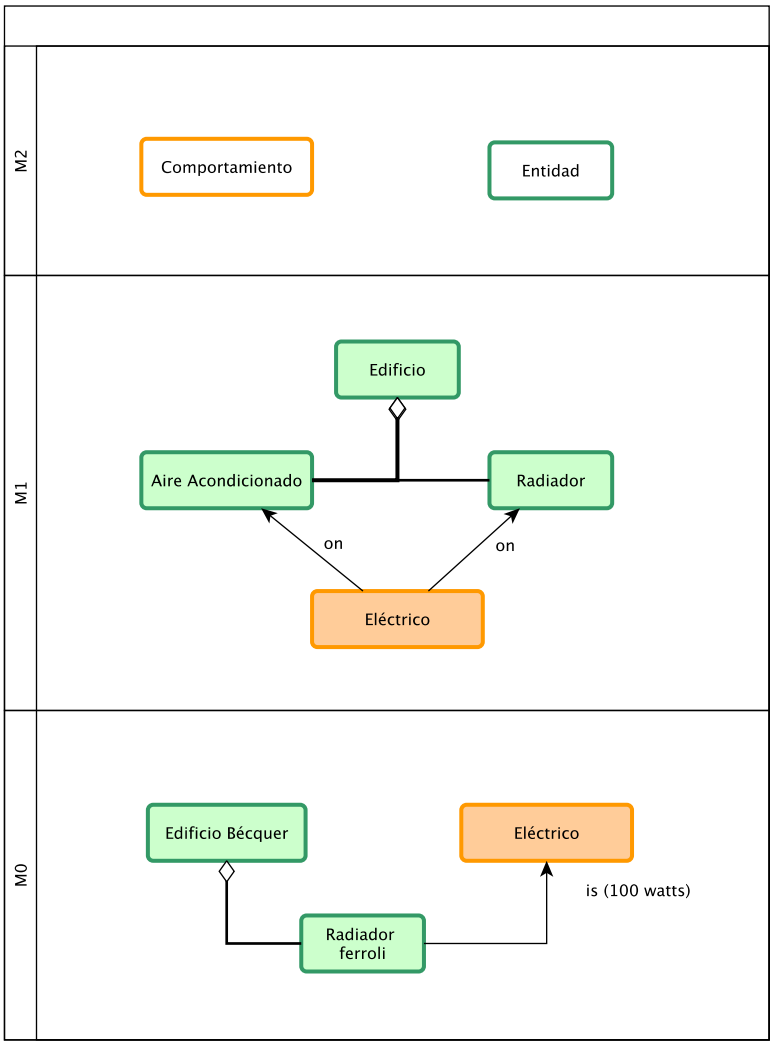


Figura 3.12: Descripción de uso de las facetas.

```
use prueba.m1 as metamodel


Edificio Bequer
    Radiador Ferroli
        is Electrico(power = 200, mode = OFF)
```

Figura 3.13: Descripción de uso de las facetas en Tara en el nivel M2.

```
use prueba.m2 as metamodel


Entidad Edificio
    Entidad AireAcondicionado is aggregated
    Entidad Radiador is aggregated


Comportamiento Electrico is facet
    var double:W potencia
    on Edificio.AireAcondicionado
    on Edificio.Radiador
        var word mode > ON; OFF*
```

Figura 3.14: Descripción de uso de las facetas en Tara en el nivel M1.

### 3.2.4. Anotaciones

Todo concepto puede tener anotaciones, que modifican su comportamiento para el siguiente nivel de modelado. Las anotaciones recogidas son:

- Terminal. Mediante esta anotación el modelador restringe la instancia del concepto para que solamente y de forma obligatoria sea instanciable en el nivel terminal.
- Abstract. Este concepto no es instanciable en el siguiente nivel. Es escrita para ser extendida.
- Named. Obliga a que todas sus instancias tengan nombre.
- Single. Restringe a uno el número de instancias del concepto anotado.
- Required. Obliga a que en todos los modelos derivados de este deban tener al menos una instancia de este concepto.

```
use prueba.m1 as metamodel

Edificio Bécquer
    Radiador Ferroli
        is Electrico(potencia = 200 W)
```

Figura 3.15: Descripción de uso de las facetas en Tara en el nivel M0.

Las anotaciones single y required restringen por tanto la cardinalidad de la instancia del modelo.

- Component. Restringe la instanciación del concepto a ser componente de otro concepto. Es decir, la instancia de este concepto no podrá ser concepto raíz del modelo.
- Aggregated. Permite especificar la relación de agregación entre dos conceptos del modelo. Por defecto es de composición.
- Property. Tiene la propiedad de component y single, además un concepto property no puede ser aggregated.
- Addressed. Obliga a que la instancia tenga una dirección en formato IPv4
- Intention. Una intención es una declaración de una funcionalidad que puede tener una instancia de un concepto. No tienen contenido, pero permiten especialización.
- Facet. Anota un concepto como faceta.

Todas las anotaciones se heredan, excepto abstract.

Las variables también pueden tener ciertas anotaciones. Estas son:

- Terminal. Permite no dar valor a la variable hasta el nivel terminal.

- Local. Esta anotación es solo aplicable a las variables de tipo referencia. Limita el espectro de la referencia a componentes del concepto raíz en el que se encuentra el concepto que se está instanciando.

# Capítulo 4

# Marco Tecnológico

Los lenguajes programación, independiente del paradigma al que pertenezca debe ser interpretado o compilado. Para este proyecto, se ha elegido la opción de compilación debido a que no se posee una maquina que sea capaz de interpretar directamente el código Tara.

## 4.1. Compilador

A grandes rasgos, un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto. Como parte de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente[40].

De esta manera un programador puede diseñar un programa en un lenguaje mucho más cercano a como piensa un ser humano, para luego compilarlo a un programa más manejable por una computadora.

### 4.1.1. Partes de un compilador

La construcción de un compilador involucra la división del proceso en una serie de fases que variará con su complejidad. Generalmente estas fases

se agrupan en dos tareas: el análisis del programa fuente y la síntesis del programa objeto.

#### Análisis

Se trata de la comprobación de la corrección del programa fuente, e incluye las fases correspondientes al Análisis léxico (que consiste en la descomposición del programa fuente en componentes léxicos), Análisis sintáctico (agrupación de los componentes léxicos en frases gramaticales ) y Análisis semántico (comprobación de la validez semántica de las sentencias aceptadas en la fase de Análisis Sintáctico).

#### Síntesis

Su objetivo es la generación de la salida expresada en el lenguaje objeto y suele estar formado por una o varias combinaciones de fases de Generación de Código (normalmente se trata de código intermedio o de código objeto) y de Optimización de Código (en las que se busca obtener un código lo más eficiente posible).

### 4.1.2. Etapas del proceso

El proceso de traducción se compone internamente de varias etapas o fases, que realizan distintas operaciones lógicas. Es útil pensar en estas fases como en piezas separadas dentro del traductor, y pueden en realidad escribirse como operaciones codificadas separadamente aunque en la práctica a menudo se integren juntas. Fase de análisis

#### Análisis léxico

El análisis léxico constituye la primera fase, aquí se lee el programa fuente de izquierda a derecha y se agrupa en componentes léxicos (tokens),

que son secuencias de caracteres que tienen un significado. Además, todos los espacios en blanco, líneas en blanco, comentarios y demás información innecesaria se elimina del programa fuente. También se comprueba que los símbolos del lenguaje (palabras clave, operadores, etc.) se han escrito correctamente[43]. Como la tarea que realiza el analizador léxico es un caso especial de coincidencia de patrones, se necesitan los métodos de especificación y reconocimiento de patrones, se usan principalmente los autómatas finitos que acepten expresiones regulares. Sin embargo, un analizador léxico también es la parte del traductor que maneja la entrada del código fuente, y puesto que esta entrada a menudo involucra un importante gasto de tiempo, el analizador léxico debe funcionar de manera tan eficiente como sea posible.

**Análisis sintáctico**

En esta fase los caracteres o componentes léxicos se agrupan jerárquicamente en frases gramaticales que el compilador utiliza para sintetizar la salida. Se comprueba si lo obtenido de la fase anterior es sintácticamente correcto (obedece a la gramática del lenguaje). Por lo general, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico (AST)[44]. La estructura jerárquica de un programa normalmente se expresa utilizando reglas recursivas. Por ejemplo, se pueden dar las siguientes reglas como parte de la definición de expresiones:

1. Cualquier identificador es una expresión.
2. Cualquier número es una expresión.
3. Si expresión1 y expresión2 son expresiones, entonces también lo son:
   - expresión1 + expresión2
   - expresión1 * expresión2
   - ( expresión1 )

Las reglas 1 y 2 son reglas básicas (no recursivas), en tanto que la regla 3 define expresiones en función de operadores aplicados a otras expresiones.

La división entre análisis léxico y análisis sintáctico es algo arbitraria. Un factor para determinar la división es si una construcción del lenguaje fuente es inherentemente recursiva o no. Las construcciones léxicas no requieren recursión, mientras que las construcciones sintácticas suelen requerirla. No se requiere recursión para reconocer los identificadores, que suelen ser cadenas de letras y dígitos que comienzan con una letra. Normalmente, se reconocen los identificadores por el simple examen del flujo de entrada, esperando hasta encontrar un carácter que no sea ni letra ni dígito, y agrupando después todas las letras y dígitos encontrados hasta ese punto en un componente léxico llamado identificador. Por otra parte, esta clase de análisis no es suficientemente poderoso para analizar expresiones o proposiciones. Por ejemplo, no podemos emparejar de manera apropiada los paréntesis de las expresiones, o las palabras begin y end en proposiciones sin imponer alguna clase de estructura jerárquica o de anidamiento a la entrada.

### Análisis semántico

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los elementos que serán necesarios para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones. Un componente importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si cada operador tiene operandos permitidos por la especificación del lenguaje fuente. Por ejemplo, las definiciones de muchos lenguajes de programación requieren que el compilador indique un error cada vez que se use un número real como índice de una matriz. Sin embargo, la especificación del lenguaje puede imponer restricciones a los

operandos, por ejemplo, cuando un operador aritmético binario se aplica a un número entero y a un número real.

### Fase de síntesis

Consiste en generar el código objeto equivalente al programa fuente. Sólo se genera código objeto cuando el programa fuente está libre de errores de análisis, lo cual no quiere decir que el programa se ejecute correctamente, ya que un programa puede tener errores de concepto o expresiones mal calculadas. Por lo general el código objeto es código de máquina re-localizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asignación de variables a registros.

### Generación de código intermedio

Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar esta representación intermedia como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes; debe ser fácil de producir y fácil de traducir al programa objeto.

Esta da cabida a todos los tipos de relaciones que posibilita el lenguaje de modelado. Al final, esta representación, gráficamente se puede representar como un grafo en el que los nodos se relacionan entre si de diversas formas. Este grafo debe posibilitar su representación en cualquier otro lenguaje, ya sea uno de propósito general o específico. Optimización de código

La fase de optimización de código consiste en mejorar el código intermedio, de modo que resulte un código máquina más rápido de ejecutar.

Esta fase de la etapa de síntesis es posible sobre todo si el traductor es un compilador (difícilmente un intérprete puede optimizar el código objeto). Hay mucha variación en la cantidad de optimización de código que ejecutan los distintos compiladores. En los que hacen mucha optimización, llamados «compiladores optimizadores», una parte significativa del tiempo del compilador se ocupa en esta fase. Sin embargo, hay optimizaciones sencillas que mejoran sensiblemente el tiempo de ejecución del programa objeto sin ralentizar demasiado la compilación.

## 4.2. Entorno de desarrollo integral (IDE) Intellij IDEA

Intellij Idea[45] es un IDE de java comercializado por Jetbreains, disponible una versión bajo la licencia Apache 2 y otra de pago. En un informe de Infoworld en 2010, IntelliJ obtuvo la puntuación más alta en pruebas realizadas a las cuatro herramientas de programación Java más usadas: Eclipse, IntelliJ IDEA, NetBeans, y Oracle JDeveloper.

La primera versión de IntelliJ IDEA fue presentada en enero de 2001, y en su momento fue uno de los primeros disponibles IDE de Java con las capacidades de refactorización de código y navegación avanzada integradas. Google ha desarrollado Android Studio, un nuevo IDE para desarrollar en Android, basado en la edición de la comunidad de código abierto de IntelliJ IDEA. Soporta entre otros lenguajes Java, Python, Groovy, Ruby o Scala

Entre sus características se encuentran:

- Auto-completado inteligente de código.
- Análisis del código “on-the-fly”.
- Integración con sistemas de control de versiones.
- Modular y ampliable usando un amplio set de plugins.

- Fiabilidad y robustez muy superior a otros entornos.
- Herramienta de refactorización inteligente.
- Soporte para diversos lenguajes conocidos y posibilidad de integrar lenguajes personalizados.
- Proyecto y sus componentes

### 4.2.1. Proyecto

En IntelliJ IDEA, un proyecto encapsula todo el código fuente, librerías, instrucciones de construcción en una sola unidad organizativa. Todo lo que haces en IntelliJ IDEA, se realiza en el contexto de un proyecto. Un proyecto define algunas colecciones denominadas módulos y librerías. Dependiendo de los requisitos lógicos y funcionales al proyecto, puede crear un único módulo o un proyecto multi-módulo.

### 4.2.2. Módulo

Un módulo es una unidad discreta de la funcionalidad que se puede ejecutar, probar y depurar de forma independiente. Incluye módulos de cosas tales como el código fuente, scripts de creación, pruebas unitarias, los descriptores de despliegue, etc. En el proyecto, cada módulo puede utilizar un SDK específico o heredar SDK definida a nivel de proyecto (véase la sección SDK más adelante en este documento). Un módulo puede depender de otros módulos del proyecto.

### 4.2.3. Librerías

Una librería es un archivo de código compilado (como archivos JAR) que sus módulos dependen. IntelliJ IDEA es compatible con tres tipos de librerías:

Librería de módulo: las clases de esta librería sólo son visibles en este módulo y la información de la misma se registra en el archivo * .iml módulo. Librerías de proyecto: las clases de librería son visibles dentro del proyecto y la información de la misma se registra en el proyecto * .ipr archivo o en .idea/libraries.

Librería global: la información de la librería se registra en el archivo applicationLibraries.xml en el ¡Usuario Inicio¿/config/IntelliJIdea/. . . /opciones. Librerías globales son similares a las librerías de los proyectos, pero son visibles para los diferentes proyectos.

### 4.2.4. SDK

Cada proyecto utiliza un Software Development Kit (SDK). Para los proyectos de Java, SDK se refiere como JDK (Java Development Kit). El SDK determina qué librería API se utiliza para construir el proyecto. Si su proyecto es multi-módulo, el SDK proyecto de forma predeterminada es común para todos los módulos dentro del proyecto. Opcionalmente, puede configurar SDK individual para cada módulo. Java posee su propio SDK, el Java Development Kit (JDK)[46]. Es un kit de desarrollo de software (SDK) para la creación de programas en Java.

### 4.2.5. Facetas

Una faceta representa cierta configuración, específico para un marco / tecnología particular, asociado con un módulo. Un módulo puede tener múltiples facetas. Por ejemplo, la configuración de Maven para un proyecto se almacena en la faceta Maven del mismo.

La Figura 4.1 ilustra de forma gráfica la estructura de proyectos de IntelliJ.

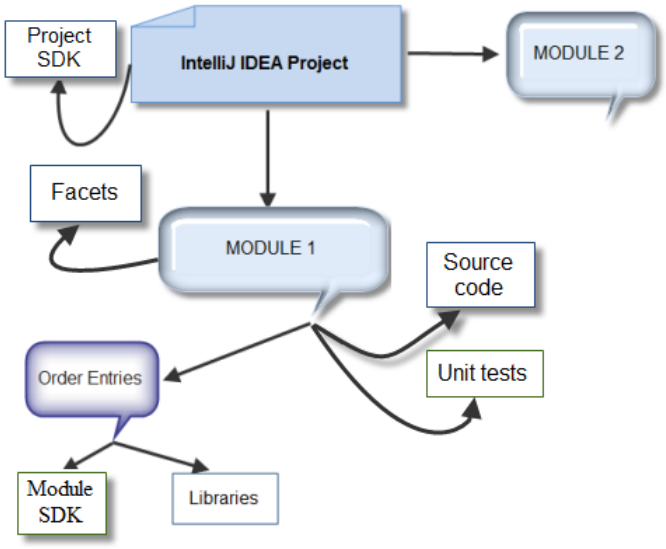


Figura 4.1: Estructura de un proyecto en Intellij IDEA.

### 4.2.6. Plugin

En el contexto de un entorno de desarrollo como Intellij, los plugins son la única manera compatible para ampliar la funcionalidad de IDEA. Un plugin utiliza la API expuesta por IDEA u otros plugins para implementar su funcionalidad. Este apartado se centra en la estructura del sistema de plugins y ciclo de vida del plugin.

Hay 3 maneras de organizar el contenido del plugin:

1. Un plugin consiste en un archivo jar colocado en el directorio de plugins. El archivo debe contener el archivo de configuración (META-INF / plugin.xml) y las clases que implementan la funcionalidad del plugin. El archivo de configuración especifica el nombre del plugin, descripción, versión, fabricante, la versión IDEA apoyado, plugin de componentes, acciones y grupos de acciones, colocación interfaz de usuario la acción.

2. Los ficheros del plugin están localizados en un directorio. El directorio 'clases' y todos los jars ubicados en el directorio 'lib' se agregan automáticamente al classpath.

3. Programas archivos se encuentran en un archivo jar que se coloca en la carpeta lib. Todos los jars del directorio 'lib' se agregan automáticamente al classpath.

Para cargar las clases de cada plugin, IDEA utiliza un cargador de clases separado. Esto permite que cada plugin para utilizar una versión diferente de una biblioteca, incluso si la misma biblioteca es utilizada por IDEA en sí o por otro plugin. Por defecto, el principal cargador de clases carga IDEA clases que no se encontraron en el cargador de clases del plugin. Sin embargo, en el archivo plugin.xml, se puede utilizar el ¡depende¿para especificar que un plugin depende de uno o varios otros plugins. En este caso, los cargadores de clases de estos plugins se utilizarán para las clases que no se encuentran en el plugin actual. Esto permite tener un plugin para hacer referencia a las clases de otros plugins.

**Componentes del plugin**

Los componentes son el concepto fundamental de la integración de plugins. Hay tres tipos de componentes: a nivel de aplicación, a nivel de proyecto y de nivel de módulo. Componentes de nivel de aplicación. Se crean y se inicializan en IDEA de puesta en marcha. Se pueden adquirir a partir de la instancia de la aplicación. Componentes a nivel de proyecto. Se crean para cada instancia de Project en IDEA (pueden crearse incluso para proyectos sin abrir). Se pueden adquirir a partir de la instancia de Project. Componentes a nivel de módulo. Se crean para cada módulo en cada proyecto cargado en IDEA. Los componentes de módulo se puede adquirir a partir de instancia del módulo con el mismo método. Cada componente debe tener

clases de interfaz y de aplicación especificados en el archivo de configuración. La clase de interfaz se utiliza para recuperar el componente de otros componentes y la clase de implementación será utilizado para la creación de instancias de componentes. Tenga en cuenta que dos componentes del mismo nivel (Aplicación, Proyecto o Módulo) no pueden tener la misma clase de interfaz. Clases de interfaz y de ejecución pueden ser el mismo. Cada componente tiene el nombre único que se utiliza para su externalización y otras necesidades internas. El nombre de un componente por devolución de su método.

*Extensiones de plugins y puntos de extensión*

IntelliJ IDEA ofrece el concepto de extensiones y puntos de extensión que permite un plugin para interactuar con otros plugins o con el núcleo de IDEA. Puntos de extensión Si usted quiere que su plugin para permitir que otros plugins para extender su funcionalidad, en el complemento, debe declarar uno o varios puntos de extensión. Cada punto de extensión define una clase o una interfaz que se permite acceder a este punto.

*Extensiones*

Si se pretende usar el plugin para extender la funcionalidad de otros plugins o el núcleo IDEA, en el complemento, se debe declarar una o varias extensiones.

*Acciones*

IntelliJ IDEA ofrece el concepto de acciones. Una acción es una clase derivada de la clase AnAction, cuyo método “actionPerformed” se llama cuando se selecciona el botón de opción de menú o barra de herramientas. El sistema de acciones permite plugins para añadir sus propios elementos para los menús y barras de herramientas de IDEA. Las acciones se organizan en grupos, lo que, a su vez, pueden contener otros grupos. Un grupo de acciones puede formar una barra de herramientas o un menú. Los sub-grupos del grupo pueden formar sub-menús del menú.

*Servicios*

IntelliJ IDEA ofrece el concepto de servicios. Un servicio es un componente plugin cargado bajo demanda. IntelliJ IDEA asegura que sólo se carga una instancia de un servicio a pesar de que el servicio se llama varias veces. Un servicio debe tener las clases de interfaz y de ejecución especificados en el archivo plugin.xml. La clase de implementación de servicio se utiliza para la instanciación servicio. IntelliJ IDEA ofrece tres tipos de servicios: servicios de nivel de aplicación, los servicios a nivel de proyectos y servicios de nivel de módulo. Archivo de configuración del plugin (plugin.xml) Lo siguiente es un ejemplo de archivo de configuración del plugin. Ésta muestra casos de uso y describe todos los elementos que se pueden utilizar en el archivo plugin.xml.

## 4.3. Antlr

ANTLR (ANother Tool for Language Recognition; en español .ºtra herramienta para reconocimiento de lenguajes")[47] es una herramienta creada principalmente por Terence Parr, que opera sobre lenguajes, proporcionando un marco para construir reconocedores (parsers), intérpretes, compiladores y traductores de lenguajes a partir de las descripciones gramaticales de los mismos (conteniendo acciones semánticas a realizarse en varios lenguajes de programación).

ANTLR cae dentro de la categoría de meta-programas, por ser un programa que escribe otros programas. Partiendo de la descripción formal de la gramática de un lenguaje, ANTLR genera un programa que determina si una sentencia o palabra pertenece a dicho lenguaje (reconocedor), utilizando algoritmos LL(*) de parsing. Si a dicha gramática, se le añaden acciones escritas en un lenguaje de programación, el reconocedor se transforma en un traductor o intérprete. Además, ANTLR proporciona facilidades para la creación de estructuras intermedias de análisis (como ser ASTs - Abstract

Syntax Tree), para recorrer dichas estructuras, y provee mecanismos para recuperarse automáticamente de errores y realizar reportes de los mismos.

ANTLR es un proyecto bajo licencia BSD, viniendo con todo el código fuente disponible, y preparado para su instalación bajo plataformas Linux, Windows y Mac OS X.

Actualmente ANTLR genera código Java, C, Python, Perl, Delphi, Ada95, JavaScript y Objective-C. Otros lenguajes como Ruby, php, etc. son generados por medio de extensiones planteadas por la comunidad.

### BNF y Flex

La notación de Backus-Naur, también conocida por sus denominaciones inglesas Backus-Naur form (BNF)[48], Backus-Naur formalism o Backus normal form, es un metalenguaje usado para expresar gramáticas libres de contexto: es decir, una manera formal de describir lenguajes formales. El BNF se utiliza extensamente como notación para las gramáticas de los lenguajes de programación de la computadora, de los sistemas de comando y de los protocolos de comunicación, así como una notación para representar partes de las gramáticas de la lengua natural (por ejemplo, el metro en la poesía de Venpa). La mayoría de los libros de textos para la teoría o la semántica del lenguaje de programación documentan el lenguaje de programación en BNF.

Flex (The Fast Lexical Analyzer)[49] es una alternativa de software libre a Lex. Se trata de un programa informático que genera analizadores léxicos ("scanners.º "lexers"). Estos programas realizan análisis de carácter, y crean cadenas de tokens a través de la utilización de un autómata finito determinista (DFA). Un DFA es una máquina teórica aceptar lenguajes regulares. Estas máquinas son un subconjunto de la colección de máquinas de Turing. DFA son equivalentes desplazando a la derecha las máquinas de

Turing a sólo lectura. La sintaxis se basa en el uso de expresiones regulares. Ver también autómata finito no determinista.

# Capítulo 5

# Trabajo instrumental

La aplicación resultante de este proyecto es un plugin que se integra directamente con el entorno de desarrollo IntelliJ IDEA, en adelante IntelliJ. Este plugin proporciona a IntelliJ, soporte para un el nuevo lenguaje de Tara, consiguiendo así las mismas ventajas que las de desarrollar código de lenguajes de propósito general en en este entorno. La Figura 5.1 describe la situación de Tara IDE dentro del ecosistema de desarrollo. Se muestra cómo el entorno de desarrollo de Tara (Tara IDE) se sustenta sobre Intellij Idea y éste a su vez sobre la maquina virtual de Java. Por otro lado existe un kit de desarrollo de Tara el cual soporta la representación del metamodelo y modelo descrito. En el caso de la figura, el metamodelo especifica un motor de simulación y el modelo, la simulación concreta.

### 5.0.1. Que es y cómo funciona

El plugin arquitectónicamente está compuesto de 3 piezas principales. La principal es el entorno que se integra directamente con Intellij, aportando todas las ayudas que el entorno te permite añadir al editor para interactuar con tu código. Estas ayudas son de gran utilidad para el desarrollador ya que le permiten escribir y detectar errores más rápido.

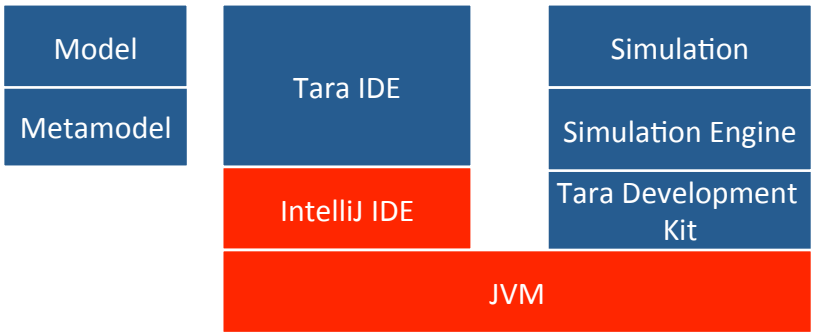


Figura 5.1: Estructura del entorno de desarrollo Intellij IDEA

Otro módulo representa el compilador del lenguaje. Éste es stand-alone, es decir, independiente de la plataforma de Intellij. Esto permite ser ejecutado o incluido desde otra aplicación, o ser ejecutado directamente desde línea de comando.

Por último existe una representación abstracta de los modelos la cual sirve de nexo entre un nivel de modelado y del siguiente. Este proceso es descrito en la Figura 5.2.

Tanto el entorno de desarrollo como el compilador tienen como entrada, código escrito en Tara, el cual es interpretado por la gramática. Una vez comprobado que es sintácticamente correcto. Ambas plataformas crean un abstract syntax tree (AST)[47] sobre el cual en caso del IDE se aplican las extensiones documentadas más adelante y en caso del compilador crea una representación del modelo, analiza semánticamente el modelo y genera el código correspondiente en un lenguaje de programación de propósito general. En este caso, el lenguaje escogido ha sido Java. El proceso es descrito en la Figura 5.3.

### Entorno de desarrollo

El plugin aporta las siguientes funcionalidades al desarrollo del código Tara:

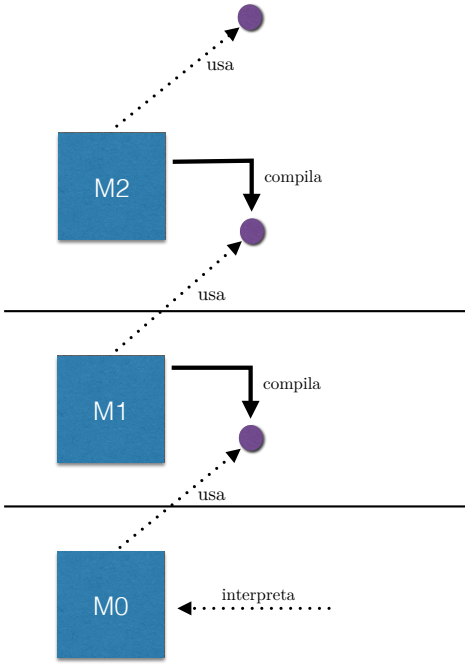


Figura 5.2: Descripción de la dinámica de modelado en Tara IDE.

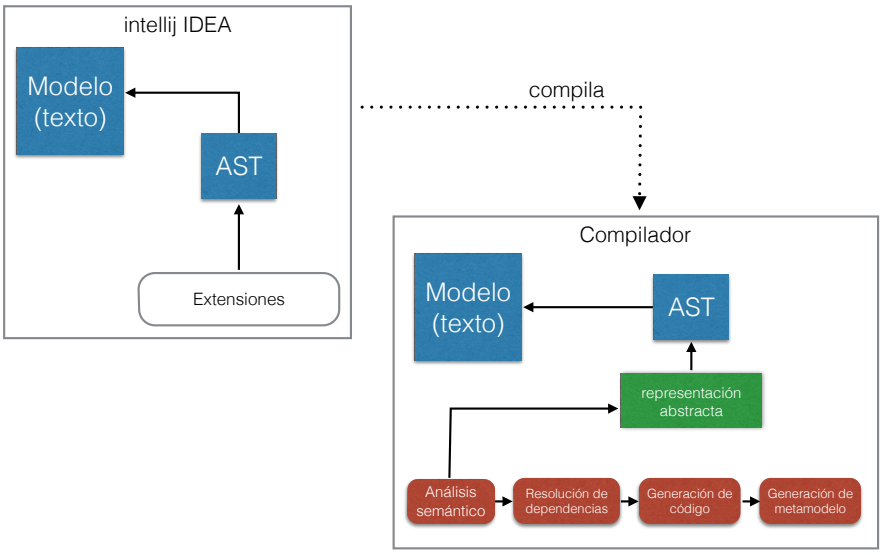


Figura 5.3: Representación del proceso de compilación

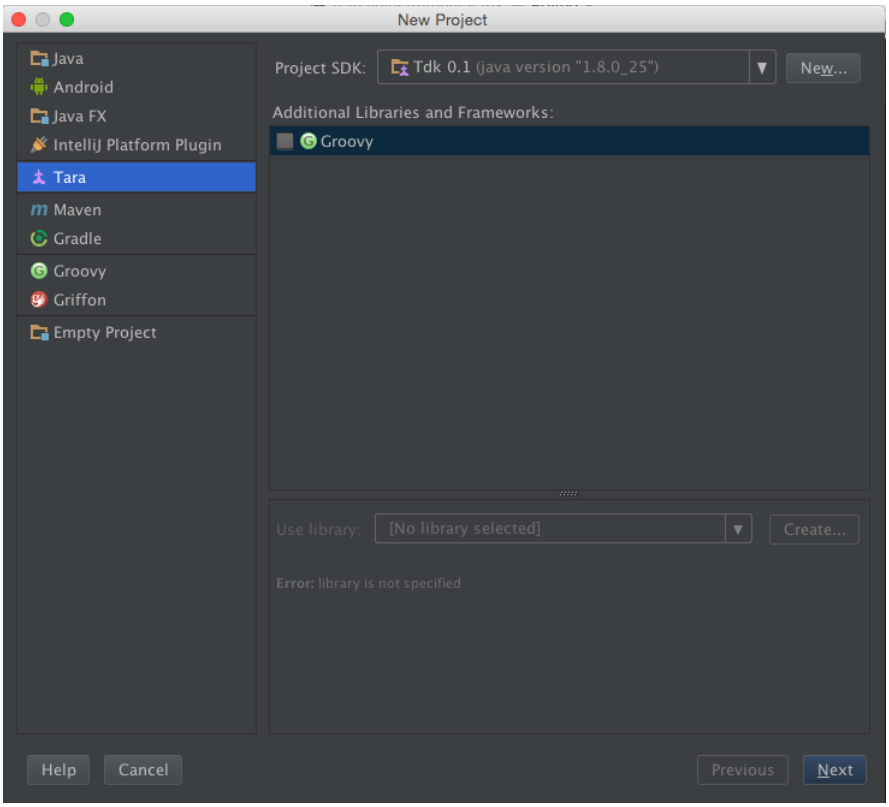


Figura 5.4: Ejemplo de creación de un proyecto.

Proyecto propio. El plugin permite crear un proyecto personalizado en el que se recogen las configuraciones necesarias de un entorno de modelado. Se puede ver un ejemplo en la Figura 5.4.

Modulo propio. Tara tiene como unidad de modelo un módulo de intellij, por tanto, la configuración del modelo está relacionada con el módulo.

Highlighting. En un lenguaje el highlighting permite identificar las partes más importantes del código así como mantener una estructura rápidamente reconocible. Un ejemplo de highlighting se puede ver en la Figura 5.5.

Anotaciones. El plugin provee de un análisis semántico del código *on-fly*, es decir, mientras el modelador escribe, el analizador semántico anota los resultados sobre el código, resaltando los posibles errores como se muestra

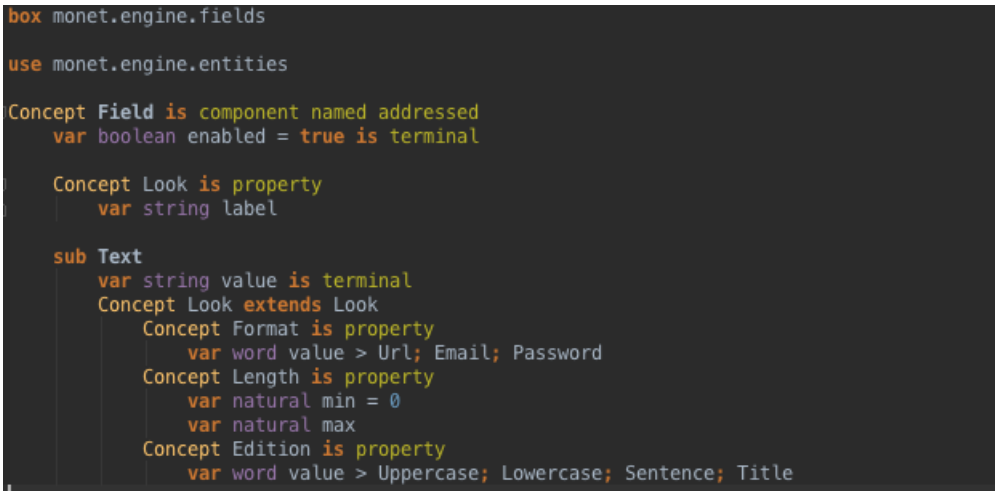


Figura 5.5: Highlighting.

en la Figura 5.6. Además permite realizar arreglos rápidos (Quick Fix) que permiten al desarrollador corregir los fallos rápidamente.

Completado de código (Code Completion). Esta característica permite al usuario que mientras escribe el IDE le sugiera código de forma intuitiva y poco intrusiva de forma que le sea más ágil la escritura.

Navegación entre conceptos. El plugin permite navegar desde una referencia de un concepto a su declaración y viceversa, dada la declaración de un concepto conocer que conceptos hacen le refieren. El la Figura 5.7 se puede ver dado un concepto, las referencias hacia el que se encuentran en el resto del modelo.

Structure. Funcionalidad del plugin la cual permite ver un esquema del modelo.

Refactoring. El plugin permite mover o renombrar conceptos del modelo entre paquetes manteniendo las referencias entre los conceptos así. En la Figura 5.8 se puede ver un ejemplo de renombrado.

LineMarking. Esta característica permite navegar entre las declaraciones de conceptos Tara y el código generado que lo representa. Las Figuras 5.9 y 5.10 muestra los ejemplos de cómo navegar desde Tara y desde Java.

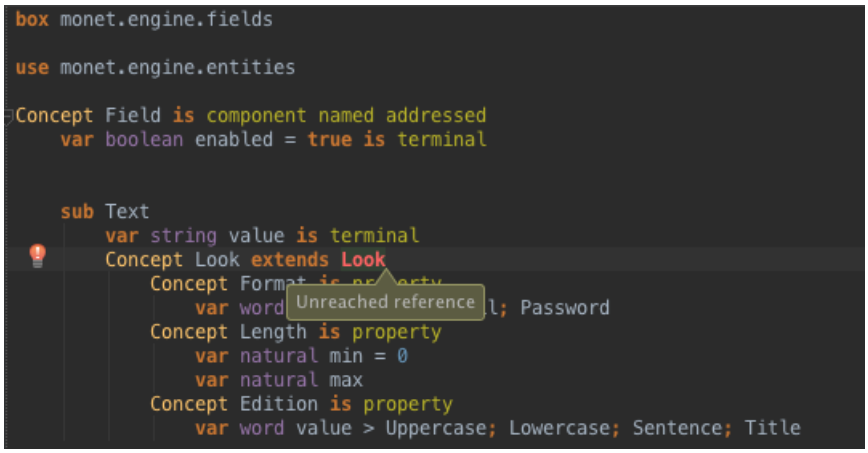


Figura 5.6: Anotaciones.

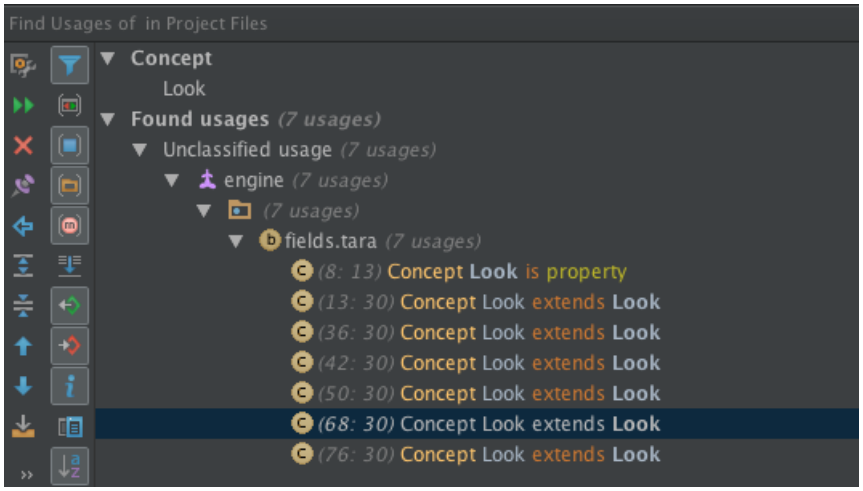


Figura 5.7: Referencias a un concepto en el modelo.

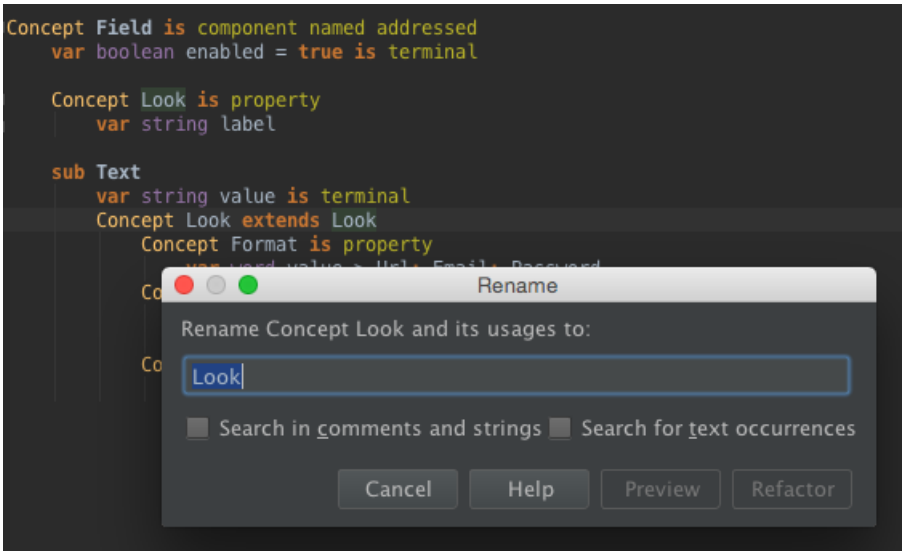


Figura 5.8: Renombrado de un concepto.

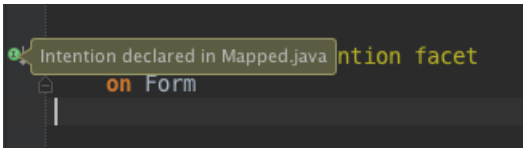


Figura 5.9: Ayuda para el completado de parámetros.


Figura 5.10: Ayuda para el completado de parámetros.


Figura 5.11: Ayuda para el completado de parámetros.

Ayuda al modelado. Esta característica engloba un conjunto de ayudas para que el modelador pueda crear instancias del metamodelo cumpliendo las restricciones de ese. Entre ellas destacan el aporte de información en los parámetros, auto-completado de conceptos o generación de código. Se puede ver un ejemplo en la Figura 5.11 sobre la ayuda en la adición de parámetros a un concepto.

### 5.0.2. Metodología de desarrollo

En el contexto del desarrollo ágil de software, queremos definir los requerimientos de manera rápida, incremental[50], y justo a tiempo (inmediatamente antes de desarrollar). También queremos que nuestras especificaciones sean ejecutables, para que se mantengan siempre sincronizadas con el producto, y nos ayuden en cuanto a detectar problemas de regresión. Es por ello que para definir las especificaciones de la aplicación usamos un proceso de desarrollo de software que se denomina Desarrollo basado en el Comportamiento[51]. Este proceso de desarrollo fue ideado por Dan North como respuesta a los problemas que encontró en el desarrollo orientado a pruebas:

- Por dónde empezar el proceso.
- Qué probar y qué no probar.
- Cuanto hay que probar de una vez.
- Cómo entender la causa de los fallos de las pruebas.

BDD especifica que las pruebas unitarias de software deben ser especificadas en términos del comportamiento deseado de la unidad. BDD utiliza un formato semi-formal para la especificación de comportamiento que se tomó prestado de las especificaciones de la historia de usuario desde el ámbito del análisis y diseño orientado a objetos. Cada historia de usuario debe, de alguna manera, seguir la siguiente estructura:

- Título
- Narrativa
- Criterios de aceptación o escenarios

Las pruebas de aceptación deben ser escritos utilizando el marco ágil estándar de una historia de usuario: "Cómo [rol] quiero [característica] para qué [los beneficios]". Los criterios de aceptación deben estar redactados en términos de escenarios e implementan como clases: Dada [contexto inicial], cuando [evento ocurre] y [garantizar algunos resultados]. Este proceso tiene como artefacto una especificación semi-formal de los comportamientos del sistema a través de historias de usuario.

# Capítulo 6

# Conclusiones

Este trabajo es consecuencia de la existencia de una problemática en el ámbito del desarrollo de software. Uno de ellos es la falta de flexibilidad que los equipos de desarrollo tienen para adoptar cambios de requisitos impuestos por el cliente. Otro problema es la complejidad existente tanto a nivel tecnológico como en las propias necesidades de los usuarios. En estas dos dimensiones es necesario fortalecer la comunicación entre los usuarios y el equipo de desarrollo. El enfoque de este trabajo para aportar soluciones a esta problemática es una metodología llamada Generative Models Driven Engineering. Este enfoque es una línea de investigación del SIANI en MDE, en la que se define un motor de propósito general que es capaz de interpretar modelos de de una misma familia de metamodelos. Este motor puede operar sobre estos modelos ya que comparten la misma estructura profunda definida en un metametamodelo. En general, MDE se apoya en el uso de lenguajes específicos de dominio (DSL) con los que se pueden expresar modelos de un determinado ámbito, con un lenguaje más cercano al dominio.

Por otro lado, existen ámbitos de desarrollo de software variados para los que se pueden requerir diferentes DSL. Cada uno de estos DSL expresarían la problemática de diversos dominios. De esta forma, se plantea si es posible

mejorar la productividad en la creación, mantenimiento y aprendizaje de los diferentes DSL.

Se ha estudiado el estado del arte en el que se recogen las visiones actuales en MDE y DSL. Una vez descrita la problemática existente y analizado las líneas de investigación actuales, se han formulado las siguientes preguntas, las cuales responden a necesidades que se han observado a la hora de crear DSLs: *¿Sería posible expresar los DSL con un meta-DSL?* y *¿Este DSL podría usar la misma gramática que el meta-DSL?*

Como respuesta a las preguntas se ha hipotetizado que: "si los dominios pueden expresarse como un conjunto de conceptos parametrizables y asociados entre sí por relaciones de composición, agregación, abstracción y concreción, debiera ser posible diseñar un meta DSL capaz de crear DSL para los diferentes dominios."

Para comenzar a investigar en torno a esta hipótesis, se ha creado un metalenguaje bautizado como Tara, el cual es capaz de crear DSLs, compartiendo la misma gramática y semántica que el metalenguaje. Este lenguaje es declarativo y está basado en conceptos. Estos conceptos se relacionan entre sí, pueden tener una dirección (address), contener variables, facetas y anotaciones.

Para poder validar la hipótesis se debe poder probar si el metalenguaje es capaz de crear DSLs de cualquier dominio. Esto lleva tiempo y por ello se ha desarrollado una herramienta que facilita la validación de lenguajes. En este caso, se ha realizado para Tara. Esta herramienta es un plugin para el entorno de desarrollo Intellij IDEA. Este plugin da soporte a la edición de código Tara de la misma forma que Intellij, que ya lo hace para lenguajes de propósito general como Java.

Como trabajo futuro se propone la experimentación de la hipótesis planteada a través de la creación de lenguajes específicos para diferentes ámbitos. Este tipo de hipótesis no puede ser totalmente validada, puesto que existen

infinitos dominios para los cuales se pueden crear lenguajes [20]. Sin embargo, lo que se pretende a través de la experimentación, es la confirmación de que Tara es válido para expresar los conceptos de los dominios sobre los que se haya experimentado. Por otro lado, este proceso permitirá mejorar el lenguaje y la herramienta para conseguir modelos más flexibles y fáciles de leer y entender.

Este trabajo aporta una forma nueva de crear modelos, con un lenguaje específicamente diseñado para la creación de nuevos lenguajes de dominio. Esto hace que los modelos sean más expresivos. Se posibilita que los expertos de dominio tengan mayor conocimiento de la representación del modelo mejorando así la comunicación entre los miembros del equipo y el cliente. Además, permite crear modelos y gestionar la herencia entre ellos de un modo más simple, aportando mayor flexibilidad a los cambios. Esto es gracias a la integración del sistema de gestión de DSLs con un mismo entorno de desarrollo, en el cual todos los niveles del MDE son accesibles, modificables y generables. Como complemento, el entorno aporta la seguridad inherente en un analizador sintáctico y semántico que otorga un compilador.

# Bibliografía


[1] M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *Journal of Systems and Software*, vol. 1, pp. 213–221, Jan. 1979.

[2] G. E. Krasner, S. T. Pope, and Others, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[3] C. Brooks, T. H. Feng, E. A. Lee, and R. van Hanxleden, "Multimodeling: A preliminary case study," tech. rep., California Univ. Berkeley Dept of electrical engineerind and computer science, 2008.

[4] O. M. G. (OMG), "MDA - The Architecture Of Choice For A Changing World." http://www.omg.org/mda/.

[5] M. Corporation, "Microsoft Software Factories." http://msdn.microsoft.com/en-us/library/ff699235.aspx.

[6] J. Greenfield, "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools." http://msdn.microsoft.com/en-us/library/ms954811.aspx, Nov. 2004.

[7] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *IEEE Computer*, vol. 30, pp. 110–112, 1997.

[8] J. J. Hernandez, *Implatación del eBusiness en pequeñas organizaciones con una orientación al modelado y la interoperabilidad.* Las Palmas de GC, Spain: ULPGC Scientific Publishing, 2009.

[9] O. M. G. (OMG), "Unified Modeling Language." http://www.uml.org.

[10] P. Tarr, H. Ossher, W. Harrison, and S. S. Jr, "N degrees of separation: multi-dimensional separation of concerns," *Proceedings of the 21st ...*, 1999.

[11] D. Rico, H. Sayani, and R. Field, "History of computers, electronic commerce and agile methods," *Advances in Computers*, vol. 73: Emergi, 2008.

[12] F. Joly, *La cartografía.* Barcelona, España: Oikos-Tau., 1988.

[13] M. Pidd, *Tools for Thinking; Modelling in Management Science.* John Wiley and Sons Ltd, 3rd ed., 2009.

[14] E. Seidewitz, "What models mean," *IEEE software*, vol. 20, no. 5, pp. 26–32, 2003.

[15] R. G. Flatscher, "Metamodeling in EIA/CDIF—meta-metamodel and metamodels," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 12, no. 4, pp. 322–342, 2002.

[16] O. M. G. (OMG), "MOF - Meta Object Facility." http://www.omg.org/mof/.

[17] D. D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[18] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation.* John Wiley & Sons, 2008.

[19] J. Evora, J. J. Hernandez, and M. Hernandez, "Tafat: A framework for developing simulators based on Model Driven Engineering," in *European Simulation and Modelling Conference 2013 (ESM'13)*, 2013.

[20] J. Evora, *A methodological research on software engineering applied to the design of Smart Grids using a Complex System approach.* Las Palmas de GC, Spain: ULPGC Scientific Publishing, 2014.

[21] A. G. Llana, "Recuperaci{ó}n, transformaci{ó}n y simulaci{ó}n de datos biol{ó}gicos mediante ingenier{í}a dirigida por modelos.," Master's thesis, Universidad Polit{é}cnica de Valencia, 2010.

[22] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jezequel, "Model-driven engineering for software migration in a large industrial context," in *MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, PROCEEDINGS* (Engels, G and Opdyke, B and Schmidt, DC and Weil, F, ed.), vol. 4735 of *Lecture Notes in Computer Science*, pp. 482–497, 2007.

[23] C. Vicente-Chicote, F. Losilla, B. Álvarez, A. Iborra, and P. Sanchez, "Applying MDE to the development of flexible and reusable wireless sensor networks," *International Journal of Cooperative Information Systems*, vol. 16, no. 3/4, pp. 393–412, 2007.

[24] C. Thompson, J. White, B. Dougherty, and D. C. Schmidt, "Optimizing mobile application performance with model–driven engineering," in *Software Technologies for Embedded and Ubiquitous Systems*, pp. 36–46, Springer, 2009.

[25] J. M. Gascueña, E. Navarro, and A. Fernández-Caballero, "Model-driven engineering techniques for the development of multi-agent systems," *Engineering Applications of Artificial Intelligence*, vol. 25, no. 1, pp. 159–173, 2012.

[26] J.-M. Jézéquel, O. Barais, and F. Fleurey, "Model driven language engineering with Kermeta," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6491 LNCS, pp. 201–221, 2011.

[27] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, "A Model-Driven Design Framework for Massively Parallel Embedded Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 4, pp. 39:1—-39:36, 2011.

[28] L. Bondé, P. Boulet, and J.-L. Dekeyser, "Traceability and Interoperability at Different Levels of Abstraction in Model-Driven Engineering," in *Applications of Specification and Design Languages for SoCs* (A. Vachoux, ed.), pp. 263–276, Springer Netherlands, 2006.

[29] J. S. Cuadrado and J. G. Molina, "A Model-Based Approach to Families of Embedded Domain-Specific Languages," *Software Engineering, IEEE Transactions on*, vol. 35, pp. 825–840, Nov. 2009.

[30] J. S. Cuadrado, J. L. C. Izquierdo, and J. G. Molina, "Applying model-driven engineering in small software enterprises," *Science of Computer Programming*, vol. 89, Part B, no. 0, pp. 176–198, 2014.

[31] S. Rougemaille, F. Migeon, C. Maurel, and M.-P. Gleizes, "Model Driven Engineering for Designing Adaptive Multi-Agents Systems," in *Engineering Societies in the Agents World VIII* (A. Artikis, G. O'Hare, K. Stathis, and G. Vouros, eds.), vol. 4995 of *Lecture Notes in Computer Science*, pp. 318–332, Springer Berlin Heidelberg, 2008.

[32] O. Gilles and J. Hugues, "A MDE-based optimisation process for Real-Time systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pp. 50–57, IEEE, 2010.

[33] F. Krichen, B. Hamid, B. Zalila, M. Jmaiel, and B. Coulette, "Development of reconfigurable distributed embedded systems with a model-driven approach," *Concurrency and Computation: Practice and Experience*, 2013.

[34] V. García-Díaz, J. Tolosa, B. G-Bustelo, E. Palacios-González, O. Sanjuan-Martínez, and R. Crespo, "TALISMAN MDE Framework: An Architecture for Intelligent Model-Driven Engineering," in *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, vol. 5518 of *Lecture Notes in Computer Science*, pp. 299–306, Springer Berlin Heidelberg, 2009.

[35] V. García-Díaz, H. Fernández-Fernández, E. Palacios-González, B. C. P. G-Bustelo, O. Sanjuán-Martínez, and J. M. C. Lovelle, "{TALISMAN} MDE: Mixing {MDE} principles," *Journal of Systems and Software*, vol. 83, no. 7, pp. 1179–1191, 2010.

[36] I. U. de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería, "SIANI." http://www.siani.es.

[37] T. Reuters, "Web Of Science." www.webofscience.com.

[38] ELSEVIER, "Scopus." www.scopus.com.

[39] A. Van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography.," *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[40] A. Aho, R. Sethi, and J. Ullman, "Compiladores: principios, técnicas y herramientas," 1998.

[41] V. Turchin, "Metalanguage," 1999.

[42] J. Lloyd, "Practical advantages of declarative programming," *Joint Conference on Declarative Programming, GULP- ...*, 1994.

[43] A. V. Aho, *Compiladores: Principios, técnicas y prácticas.* México: Addison Wesley, 2008.

[44] N. Chomsky, "Syntactic structures," 2002.

[45] I. JetBrains, "Intellij idea," *On-line at www. intellij. com*, 2003.

[46] H. P. R. S. Li Gong, Marianne Mueller, "Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2,"

[47] T. Parr, "The definitive ANTLR reference: building domain-specific languages," 2007.

[48] L. Garshol, "BNF and EBNF: What are they and how do they work," *acedida pela última vez em*, 2003.

[49] V. Paxson, "Flex–Fast lexical analyzer generator," *Lawrence Berkeley Laboratory*, 1995.

[50] C. Larman and V. R. Basili, "Iterative and incremental developments. a brief history," *Computer*, vol. 36, no. 6, pp. 47–56, 2003.

[51] D. North, "Introducing bdd," *Better Software, March*, 2006.