UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Máster Oficial en Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería

Trabajo Final de Master

Proteo, lenguaje para la especificación de lenguajes específicos de dominio

Bycor Sánchez Sánchez

Tutores: José Juan Hernández Cabrera

José Évora Gómez

31 de diciembre de 2014

Agradecimientos

A mis padres y hermanos, quienes en todo momento me han brindado su apoyo incondicional, a mis tutores José Juan Hernández y José Évora, por la gran ayuda prestada, y finalmente a mi novia, por alentarme en todo momento a seguir adelante.

A todo ellos, muchas gracias.

Resumen

En el sector del desarrollo software surge la necesidad de mejorar la productividad y garantizar la evolución flexible del software, de tal forma que el producto satisfaga las necesidades del cliente con el menor coste posible.

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 (Domain Specific Language, DSL) para expresar lógicas de negocio con un lenguaje cercano al ámbito de aplicación.

El desarrollo de sistemas complejos ha potenciado el uso de MDE en contextos muy diversos. Por tanto, es necesaria la creación de lenguajes específicos para cada contexto.

En este trabajo se plantea la hipótesis de la existencia de una gramática capaz de representar múltiples DSLs. Esta gramática constituye la base de un super-lenguaje al que llamamos Proteo. En el contexto de este trabajo, un super-lenguaje es un lenguaje abstracto que puede ser extendido para definir DSLs.

Como trabajo instrumental se ha implementado la gramática, abordando la fase de análisis del compilador. Ésta comprende la concepción de un analizador léxico abstracto y los analizadores sintáctico y semántico del super-lenguaje.

Las herramientas desarrolladas permitirán abordar una futura fase de experimentación donde evaluar la hipótesis planteada.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Motivación y contextualización

En la actualidad, uno de los grandes retos del negocio de desarrollo de software es mejorar la productividad y la flexibilidad. En la mayoría de ocasiones estos objetivos se ven diluidos, debido 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 [31]: "*A medida que un programa en evolución cambia, su estructura tiende a ser cada vez más compleja*".

Una de las claves en la ingeniería del software es identificar correctamente los requisitos del usuario. Sin embargo, estos requisitos son a menudo cambiantes por un desconocimiento inicial del problema o por una falta de comunicación con el cliente. Ello hace que durante el proceso se planteen cambios en el software ya desarrollado, lo que conlleva un aumento en los costes de mantenimiento.

Este aumento de costes de mantenimiento impide dedicar recursos para la evolución del sistema, lo que repercute negativamente en la flexibilidad para abordar nuevos cambios. De este modo, disminuye la calidad del producto final y, por tanto, también la satisfacción del usuario[31].

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

Existen múltiples enfoques al respecto. Uno de ellos es 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 identifica el modelo como parte esencial dentro del diseño software. El concepto de modelo está presente en los dos patrones de diseño de software mayormente adoptados en la actualidad, el Model-View-Controller (MVC)[28] y el Model-View-Presenter (MVP)[4]. En ambos casos los modelos 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)[33], Software Factories de Microsoft[8][23] y Model Integrated Computing (MIC)[45]. En la línea de investigación en el que está insertado este trabajo se ha desarrollado el enfoque de la Ingeniería Dirigida por Modelos Generados: Generated Models Driven Engineering (GMDE)[24].

Los diferentes enfoques de MDE requieren especificar los modelos en algún tipo de lenguaje. De manera general, muchos de estos modelos son expresados en los llamados Lenguajes Específicos de Dominio (Domain Specific Languages, DSL). Este tipo de lenguajes aportan un nivel de abstracción mayor que los lenguajes propósito general, lo que permite expresar elementos específicos del dominio al que va dirigido.

Así, se obtiene un lenguaje más natural y cercano que permite involucrar al propio cliente o expertos en el dominio en el desarrollo. Esto se traduce en una comunicación más fluida y, por tanto, un mayor ajuste al resultado esperado por el cliente.

Como todo lenguaje, los Lenguajes Específicos de Dominio requieren una cierta curva de aprendizaje para su correcto uso. Esta curva puede ser suave, sin embargo, la necesidad de utilizar múltiples DSLs obliga al aprendizaje desde cero de todos

ellos. Este problema es cada vez mayor a medida que aumenta la cantidad de lenguajes requeridos.

La problemática descrita es creciente, pues en la actualidad se desarrollan multitud de aplicaciones para diferentes ámbitos de dominio como, por ejemplo, sistemas de información, sistemas de simulación[15], sensorización[49], eBusinesses, etc. Las instituciones y empresas que desarrollan software necesitan un DSL para cada dominio del problema.

En algunos casos, los desarrolladores deben moverse entre múltiples dominios. Por ejemplo, alguien que trabaje con sistemas de simulación y con cuadros de mando que dan información del propio sistema. La curva de aprendizaje inherente a cada uno de ellos requiere un coste en tiempo y recursos que puede llegar a repercutir en la calidad del producto final.

De debido a todos estos problemas, en este documento se analizan soluciones a las siguiente pregunta,

¿Es posible reducir la curva de aprendizaje entre DSLs para diversos dominios?

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)[46].

El desarrollo de software dirigido por modelos (MDE)[43] es una aproximación que enfoca la gestión de la complejidad mediante el uso de otro tipo de lenguajes. En lugar de usar lenguajes próximos a la tecnología (3GL)[41], la idea es usar lenguajes más 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.

En este apartado se describen los conceptos más relevantes, así como 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[26]. Los modelos permiten representar una parte de la realidad vista por las personas que lo utilizan para entender, cambiar, manejar y controlar esa parte de la realidad[38].

En el contexto de la ingeniería de software, el modelo de dominio es un modelo conceptual que representa los elementos que existen en el contexto 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.

Un metamodelo se describe 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[44].

Por tanto, un metamodelo constituye la especificación de un lenguaje de modelado[44]. 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[16].

2.2. Arquitectura de niveles

El concepto de metamodelo se basa en la arquitectura de metadatos adoptada por el consorcio OMG en la especificación del Meta-Object Facility (MOF)[34].

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

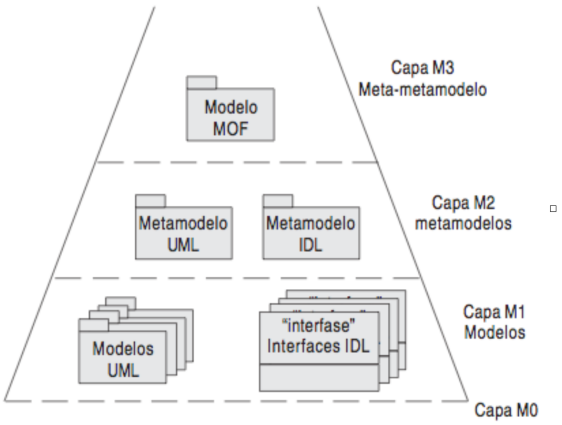

Figura 2.1: Ilustración de Meta-Object Facility.

El nivel M3 constituye el meta-metamodelo. Este nivel es utilizado para construir los metamodelos, nivel M2. Un ejemplo es el metamodelo UML[35] (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 están 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 elementos de M1 permitirían, por ejemplo, realizar un censo de quién vive en qué edificio.

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.

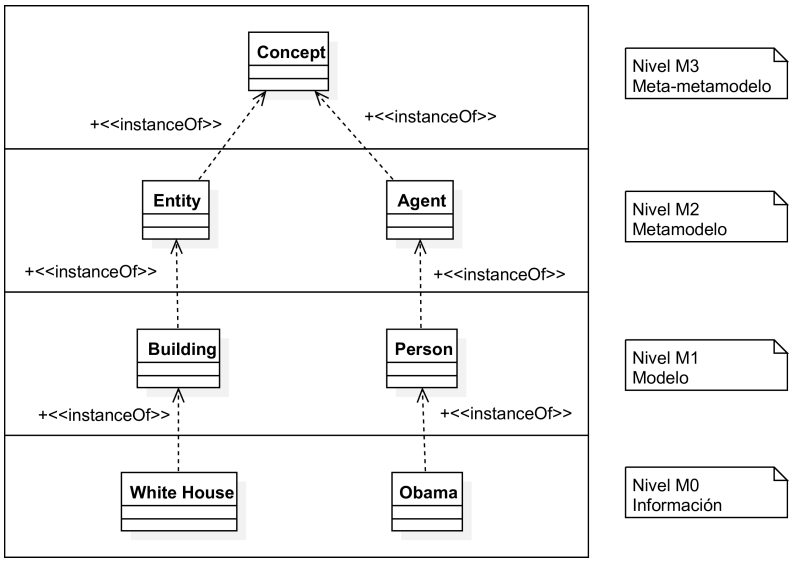

Figura 2.2: Ejemplo UML de la arquitectura de niveles.

En este sentido el Model Driven Engineering[43] (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 elementos 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[43]. 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[27]. 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[44]. 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[15] 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 explotada en Tafat para la construcción de simulaciones de "Smart Grids" [14], 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 la investigación [24] 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 [32] 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[17] 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 [49] 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[47], 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 multi-agentes[21]. 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[25] o la creación de frameworks basados en MDE para ámbitos específicos como, por ejemplo, la paralelización de sistemas empotrados[18].

B. *Universite Lille nord de France*. Investigadores de esta universidad buscan la manera de garantizar la interoperabilidad entre modelos de diferentes plataformas[48].

C. *Universidad de Murcia*. Estudian aspectos como la creación de lenguajes específicos de dominio basados en los principios de MDE[10] o la posible aplicación de MDE en pequeñas empresas de desarrollo software[9].

D. *Universite de Toulouse*. Trabajan con MDE para crear modelos de sistemas multi-agentes adaptativos[42], optimizar de sistemas de tiempo real[22] o para el desarrollo de sistemas empotrados distribuidos[29].

E. *Universidad de Oviedo*. Una de sus líneas de investigación trata la construcción de un framework para el desarrollo de todo tipo de aplicaciones[20][19]. Para ello mezclan diversos enfoques del MDE.

La figura 2.3 recoge la localización de estas instituciones. Se ha incluido el Instituto Universitario SIANI[11] (punto F), en el cual desarrollamos diversas líneas de investigación entorno al MDE.

Toda esta información se ha obtenido de las bases de datos de investigación Web Of Science[40] y Scopus[13], 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 interpretar un tipo

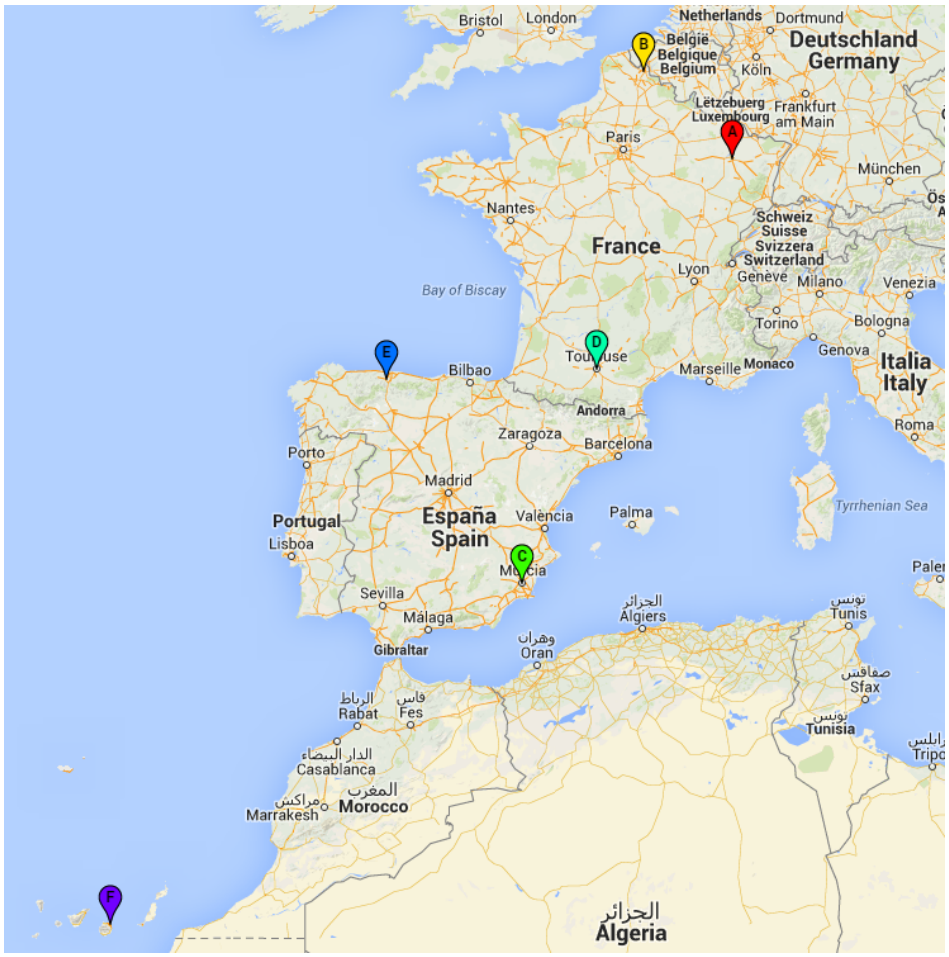

Figura 2.3: Mapa de instituciones que investigan MDE.

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

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

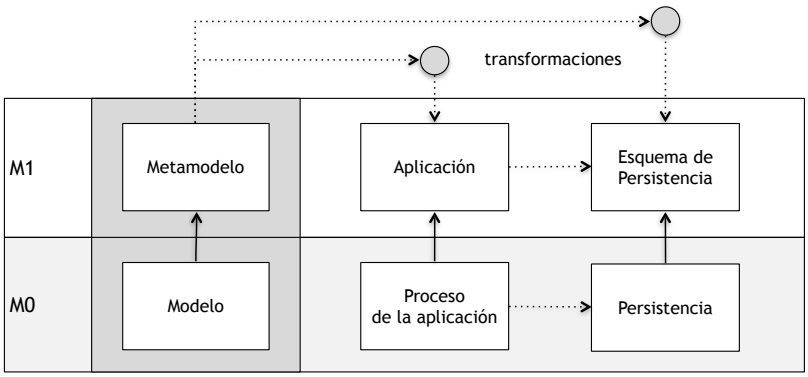

Figura 2.4: Paradigma clásico en MDE.

definir sus propias transformaciones y ejecutarlas bajo demanda. Existen diversas aproximaciones a la transformación de modelos:

- Direct Model Manipulation (Pull transformation). Herramientas que 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 expresar 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 visión de la línea de investigación en la que

está inserto este trabajo es la de 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[7]. 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. 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 (figura 2.5).

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 meta-metamodelo (M2). Así, el meta-metamodelo contiene los componentes y la gramática generativa que permiten especificar diferentes metamodelos.

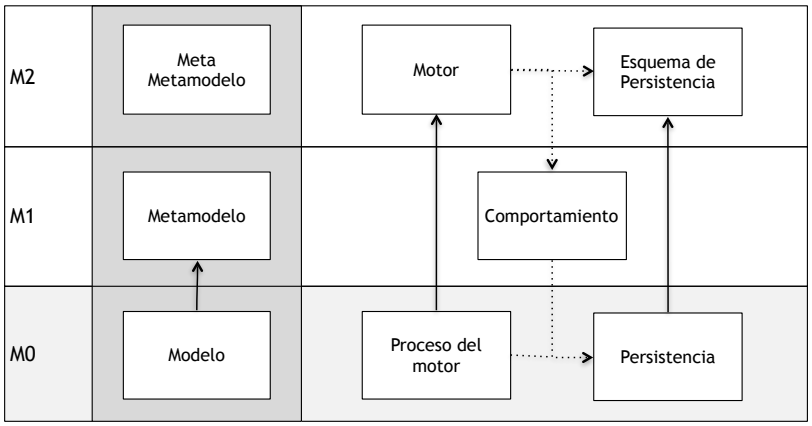

Figura 2.5: Visión del GMDE.

2.5. Domain Specific Languages

En contraste con los lenguajes de propósito general, los Lenguajes Específicos de Dominio (Domain Specific Languages, DSL) son lenguajes de programación o lenguajes de especificación ejecutables que, a través de anotaciones apropiadas y abstracciones, ofrecen expresividad[12]. Por lo general se enfocan a un dominio de problema particular. Algunos ejemplos de DSLs pueden ser el lenguaje R para estadística, Mathematica para cálculos matemáticos o SQL para bases de datos relacionales.

Los DSLs 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 ingenieros eléctricos los encargados de especificar dichas simulaciones.

En el ámbito del Model Driven Engineering los DSLs son ampliamente utilizados para diseño de modelos expresivos y cercanos al dominio de aplicación. Debido a la cantidad de ámbitos donde se aplica MDE (apartado 2.3.1) y la cantidad de investigadores que centran sus trabajos en él (apartado 2.3.2), los DSLs adquieren una gran importancia.

Capítulo 3

Hipótesis

El marco de desarrollo de Ingeniería Dirigida por Modelos (MDE) es aplicable en múltiples ámbitos de conocimiento, algunos de ellos muy diversos entre sí. Algunos ejemplos pueden ser sistemas de información, simuladores o entornos industriales.

MDE es una metodología para el desarrollo de software basada en modelos. De forma general, un modelo se describe como una abstracción de la realidad que la representa de una manera simplificada[26]. 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[38].

Estos modelos necesitan ser expresados en algún tipo de lenguaje. Es posible utilizar lenguajes de propósito general para ello, sin embargo, los Lenguajes Específicos de Dominio (DSL) son capaces de aportar una mayor semántica al contexto del problema.

De manera ideal, sería deseable disponer de un lenguaje específico en cada dominio que se quisiera desarrollar. No obstante, la creación de nuevos lenguajes exige mucho trabajo, así como un conocimiento avanzado del dominio al que va destinado.

Los lenguajes específicos de dominio reducen la curva de aprendizaje con respecto a los lenguajes de propósito general. Ello se debe a que los expertos del

dominio encuentran en él conceptos que les son familiares. Sin embargo, diferentes DSLs disponen de diferentes mecanismos para expresar los conocimientos. Este enfoque obliga a aquellos usuarios que trabajan con múltiples DSLs a invertir tiempo y recursos en aprender cada uno de ellos. Este problema es mayor a medida que aumenta la cantidad de lenguajes a utilizar.

La problemática es cada vez más creciente debido al aumento de instituciones y empresas que aplican lenguajes específicos para el desarrollo software en distintos ámbitos de conocimiento.

Por tanto, se plantea la siguiente pregunta

¿Podría existir un lenguaje abstracto a partir del cual pudieran derivarse lenguajes específicos para cada dominio?

De ser así, este lenguaje definiría mecanismos comunes para el uso de la familia de DSLs derivados de él. Ello reduciría el coste de creación y aprendizaje de DSLs en diferentes dominios, al compartir todos ellos la misma base.

Un ejemplo ilustrativo puede ser un desarrollador que trabaje con sistemas de información y con herramientas de análisis de datos del propio sistema, ambos escritos en DSLs diferentes.

El hecho de que estos dos DSLs pudieran compartir una misma base, permitiría que esta persona solo debiera aprender uno de ellos. Esto se debería a que la sintaxis para expresar las cosas sería la misma, cambiando solo los elementos de cada lenguaje. Por lo tanto, el desarrollador realizaría de manera más eficiente y productiva su trabajo.

Para conseguir una sintaxis común para todos los DSLs, el lenguaje abstracto (concepto al que llamamos super-lenguaje) debe disponer de una gramática genérica. Las gramáticas definen las cadenas de caracteres admisibles por el lenguaje a través de reglas de formación.

Todo ello lleva a proponer el desarrollo de una gramática que permita describir DSLs para cualquier dominio. En este trabajo, se propone una gramática con esta finalidad. Continuaciones a este trabajo deberán validar la capacidad de esta

gramática para describir DSLs de cualquier dominio. La hipótesis que se maneja en esta línea de investigación es, por tanto:

Si los dominios son expresables como un conjunto de elementos instanciables, asociados entre sí por relaciones de composición, agregación, abstracción y concreción; debiera ser posible la existencia de una gramática común a todos los dominios y, por tanto, la creación de un super-lenguaje.

Capítulo 4

Marco Tecnológico

Este apartado introduce los conceptos de compilador, gramática, analizador sintáctico y árbol de sintaxis abstracta. El concepto de compilador define la fase de análisis que seguirá el trabajo. La gramática es un elemento clave, pues definirá la sintaxis del super-lenguaje. Se identifica también los analizadores sintácticos, que utilizan las gramáticas para evaluar la validez de los datos. Finalmente, se describen los Árboles de Sintaxis Abstracta (Abstract Syntax Tree), estructuras para una mejor representación de los datos.

4.1. Compilador

Un compilador es un programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando código interpretable por la máquina[2].

Debido a su capacidad de transformar el lenguaje a código ejecutable, los compiladores permiten a los programadores el uso de lenguajes de alto nivel. Este tipo de lenguajes se asemejan más al lenguaje humano, lo que dota de mayor semántica al programa escrito.

Las etapas involucradas en el proceso de traducción son:

1. **Análisis**. Esta etapa comprueba la validez del programa fuente. Su funcionamiento se divide en tres sub-etapas:

   - *Léxico*. El programa original se descompone en símbolos o "tokens", pequeños fragmentos indivisibles que conforman el lenguaje.
   - *Sintáctico*. Se agrupan los "tokens" en frases gramaticales, de tal forma que se cumplan las reglas definidas por el lenguaje (gramática).
   - *Semántico*. Se comprueba la validez semántica del código fuente.

2. **Síntesis** . Esta etapa genera la salida expresada en el lenguaje objeto (lenguaje de salida del compilador). Se compone de:

   - *Optimización de código*. Trata de mejorar la eficiencia del código.
   - *Generación de código*. Genera el código máquina o código intermedio.

La figura 4.1 ilustra las diferentes etapas.

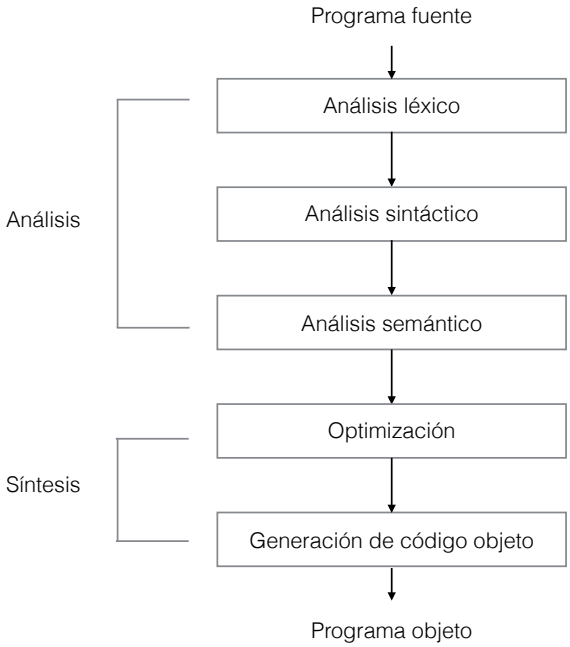

Figura 4.1: Etapas del proceso de compilación.

4.2. Gramática

La gramática comprende el estudio de las reglas y principios que gobiernan el uso de las lenguas y la organización de las palabras dentro de las oraciones. De manera formal una gramática se define como una cuádrupla[5]:

G = (VT, VN, S, P).

donde:

- VT= conjunto finito de símbolos terminales.
- VN=conjunto finito de símbolos no terminales.
- S es el símbolo inicial y pertenece a VN.
- P= conjunto de producciones o de reglas de derivación.

En el ámbito de las ciencias de la computación, cada lenguaje de programación se define a través de una gramática formal[39]. Una gramática formal es una estructura compuesta por un conjunto de reglas de formación que definen las cadenas de caracteres admisibles en un determinado lenguaje. Cabe destacar que estas gramáticas no atienden al significado de las fórmulas bien formadas, sino a la forma en que dichas fórmulas se combinan.

La jerarquía de Chomsky[5] es una clasificación jerárquica de distintos tipos de gramáticas formales que generan lenguajes formales. La tabla 4.1 recoge esta jerarquía.

Tipo	**Tipo de gramática**	**Complejidad**
0	Gramáticas no restringidas	Indeterminada
1	Gramáticas dependientes del contexto	Exponencial
2	Gramáticas libres del contexto	Polinómica
3	Gramáticas regulares	Lineal

Cuadro 4.1: Jerarquía de Chomsky.

La jerarquía comienza con un tipo de gramáticas que pretende ser universal (tipo 0). Aplicando restricciones a sus reglas se van obteniendo los otros tres tipos.

En la clasificación de la tabla 4.1 cada tipo de gramática contiene a todos los tipos siguientes (figura 4.2). Las gramáticas tipo 2 y 3 son utilizadas más frecuentemente, puesto que su menor complejidad permite el desarrollo de analizadores sintácticos más eficientes.

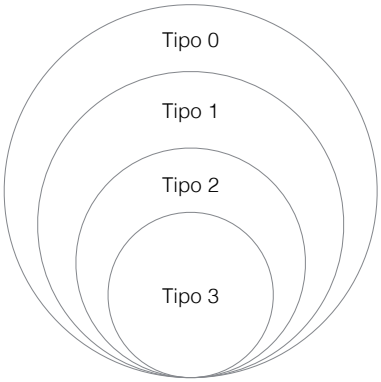

Figura 4.2: Contención de tipos de gramáticas.

4.3. Analizadores sintácticos

El analizador sintáctico o reconocedor (parser) pertenece a la fase análisis sintáctico del compilador. Su objetivo es comprobar si la sucesión de símbolos de entrada pertenecen a un determinado lenguaje[2]. Para ello, parte de la definición sintáctica del lenguaje (gramática). Asimismo, el analizador sintáctico trata de obtener la relación entre los elementos del lenguaje en una estructura en forma de árbol.

Dos de los métodos de análisis, en función de la estructura, son:

- *Descendente*. Estrategia de reconocimiento que parte del nivel más alto del árbol de representación. A partir de ahí, desciende recursivamente por las reglas de derivación de la gramática hasta obtener los símbolos terminales.

  Los analizadores sintácticos LL utilizan esta estrategia, partiendo de izquierda a derecha. El número de símbolos utilizados para predecir a qué rama acceder

puede ser variable. Este hecho se indica como LL(K), siendo K el número de símbolos. Se da también el caso de LL(*), las cuales buscan el mayor número de símbolos posible para desambiguar.

- *Ascendente*. Estrategia de reconocimiento que parte de los nodos hoja del árbol. A partir de ellos trata de escalar por los símbolos terminales hasta llegar a su raíz.

  Los analizadores sintácticos LR utilizan esta estrategia, recorren el árbol de representación de las hojas hacia la raíz. Existen tres tipos de parsers LR: SLR (K), LALR (K) y LR (K) canónico.

El árbol de la figura 4.3 representa la asignación del valor de una suma a una variable. Para este ejemplo, la figura 4.4 enumera el recorrido que realiza un analizador descendente. A partir de la raíz del árbol accede a los nodos hoja. Por su parte, el recorrido seguido por un analizador ascendente (figura 4.5) parte de los nodos hoja y va escalando por el árbol.

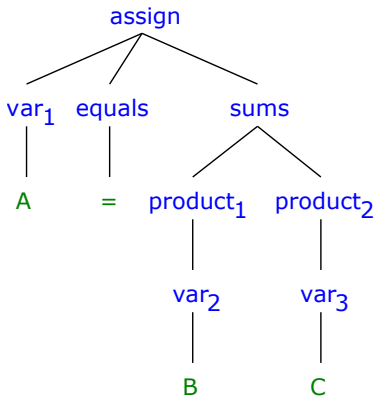

Figura 4.3: Ejemplo árbol, asignación de variables.

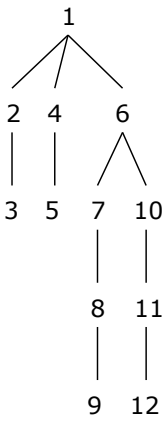

Figura 4.4: Recorrido descendente.

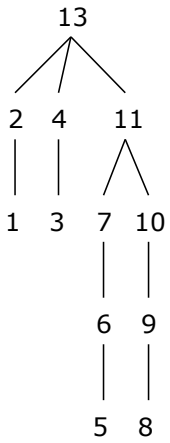

Figura 4.5: Recorrido ascendente.

4.4. Abstract Syntax Tree

Un Abstract Syntax Tree (Árbol de Sintaxis Abstracta) es una estructura en forma de árbol que representan el código fuente de forma simplificada, es decir, omite aquellos detalles poco importantes o irrelevantes.

Este tipo de árboles no debe ser confundido con los Parse Tree (árboles de derivación o árboles de sintaxis concreta). En contraste con los AST, los árboles de derivación representan el código fuente según la estructura sintáctica definida por la gramática, sin omitir ningún tipo de detalle.

Las figuras 4.6 y 4.7 ilustran la sutil diferencia entre estas dos estructuras. En ambos casos se ha representado la asignación de una suma de variables a otra variable.

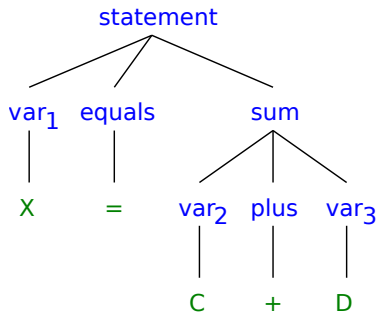

Figura 4.6: Ejemplo Parse Tree.

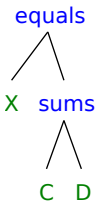

Figura 4.7: Ejemplo AST.

En la figura 4.6 se puede ver un árbol de derivación con todas las reglas que conforman la gramática. Por su parte, el AST de la figura 4.7 representa la misma información omitiendo parte de la información que no es relevante.

Los AST son ampliamente utilizados en los compiladores, pues permiten representar eficientemente la información del código fuente. En concreto son especialmente útiles en la fase de análisis semántico.

Capítulo 5

Definición del super-lenguaje

En el presente trabajo se introduce el concepto de *super-lenguaje*. En este contexto, un super-lenguaje se define como un lenguaje abstracto que puede ser extendido para definir diferentes Lenguajes Específicos de Dominio (DSL).

El super-lenguaje es análogo a una superclase en un lenguaje de programación orientado a objetos, ya que permite la creación de un lenguaje derivado heredando todas las propiedades y estructura del mismo.

De manera general, el estudio de un lenguaje se divide en una estructura de niveles. En el ámbito de la informática, crear un lenguaje de programación requiere centrarse fundamentalmente en tres de estos niveles: morfología, sintaxis y semántica.

La morfología se denota como la identificación, análisis y descripción de la estructura de los morfemas, unidades más pequeñas de un lenguaje (palabras). Dentro del estudio de las palabras se encuentra también la lexicología, que estudia el conjunto de palabras que conforman un lenguaje.

Por su parte, la sintaxis comprende el estudio de los principios y procesos a través de los cuales se construyen las estructuras de un lenguaje[6] (frases). Finalmente, el semántico estudia el significado, centrando su atención en la relación

existente entre los elementos del lenguaje y lo que representan. Cada uno de estos niveles se sustenta con respecto al anterior, como refleja la figura 5.1.

Figura 5.1: Niveles del lenguaje.

En este trabajo se propone la creación de un super-lenguaje, al que se ha denominado Proteo. Proteo permitirá la especialización de lenguajes específicos de dominio para el modelado de sistemas bajo la metodología Model Driven Engineering. En concreto, el enfoque utilizado es el de Ingeniería Dirigida por Modelos Generados (Generated Models Driven Engineering), desarrollado en nuestro grupo de investigación.

Proteo dispone de un léxico abstracto, así como una sintaxis y semántica propia del super-lenguaje. El léxico es esencialmente un catálogo de las palabras que dispone el lenguaje. Proteo proveerá un léxico con ciertas palabras reservadas, sin embargo, cada DSL especificado a través de Proteo deberá incluir los elementos instanciables que pueden ser definidos en él.

A diferencia del léxico, la sintaxis y la semántica de Proteo no cambiará en los DSLs derivados. La gramática comprende las reglas que darán forma a la estructura del lenguaje, definiendo así su sintaxis. Por su parte, la semántica comprueba el correcto significado de los diversos elementos del lenguaje.

Con todo, Proteo define una estructura común para los diversos lenguajes especificados. La figura 5.2 ilustra dicha estructura, donde el léxico especializado para el ámbito de aplicación supone la base para la sintaxis y semántica del DSL. Así, solo sería necesario modificar el léxico para implementar un nuevo DSL.

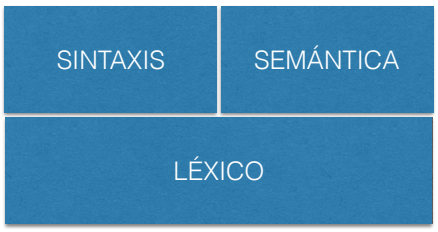

Figura 5.2: Estructura de Proteo.

Debido a que los DSLs definidos a través de Proteo utilizarán MDE, éste incluye ciertos mecanismos que tienen en cuenta la arquitectura de niveles (apartado 2.2).

5.1. Relaciones

Proteo es un super-lenguaje basado en los elementos instanciables de cada disciplina, también llamados en este trabajo *instanciables*. Estos instanciables son definidos en los metamodelos, nivel M2 de la arquitectura de niveles (apartado 2.2).

En ámbitos como los sistemas de información los instanciables pueden ser colecciones, catálogos, formularios o campos. Asimismo, estos elementos pueden contener parámetros. Por ejemplo, un campo de tipo fecha dispone de un formato con el que se expresa dicha fecha.

Los elementos definidos en el lenguaje pueden relacionarse entre sí. Las relaciones identificadas entre ellos son:

1. *Composición.* Elementos estrictamente limitados por una relación complementaria. El constituyente no se entiende sin el constituido, es decir, no tienen sentido de manera separada. La figura 5.3 ilustra esta relación. Un libro está compuesto de capítulos.

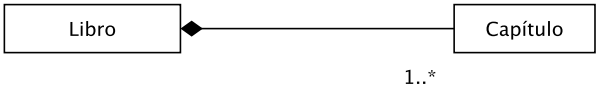

Figura 5.3: Relacion de composición.

2. *Agregación.* Dos elementos se consideran usualmente como independientes y, aun así, están ligados. El tiempo de vida de cada uno de los objetos es independiente. La figura 5.4 ilustra esta relación de agregación.

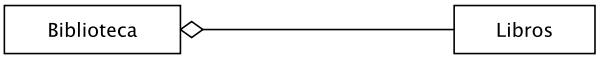

Figura 5.4: Relación de agregación.

3. *Especialización.* Esta relación se produce entre dos o más elementos. Marca una relación de jerarquía entre elementos, donde un elemento principal (padre) puede ser heredado por otros secundarios (hijos), los cuales adquieren "por herencia" los atributos del primero.

4. *Generalización.* Relación opuesta a la especialización. La especialización es obtenida completando algunos detalles. La generalización se obtiene eliminando algunos detalles.

   En la figura 5.5 se puede ver que el juego es una abstracción de parchís y ajedrez. Al mismo tiempo, ambos elementos son especializaciones de juego.

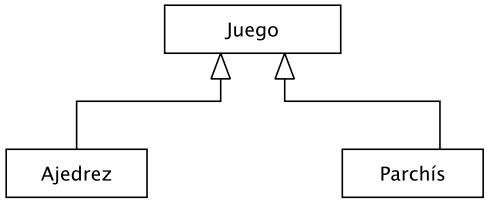

Figura 5.5: Generalización entre elementos.

5. *Concreción* o *Instanciación.* La concreción o instanciación es la relación existente entre un elemento de un modelo y un o varios elementos de un modelo de nivel inferior dentro de la arquitectura de niveles (figura 5.6).

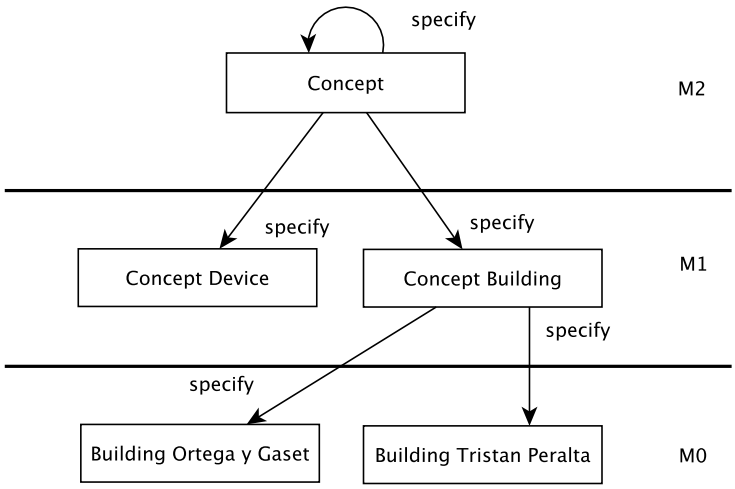

Figura 5.6: Descripción de una concreción entre modelos.

6. Abstracción. La abstracción es el procedimiento inverso al de la concreción. Dado un elemento de nivel n, éste es posible abstraerlo dentro de un elemento del modelo de nivel superior n+1 (figura 5.7).

5.2. Características

Proteo dispone de una serie características aparte de los elementos, como son las direcciones, las variables, las facetas y las anotaciones.

5.2.1. Direcciones

Las direcciones (addresses) son un mecanismo de localización de elementos. Esta dirección se representa con un formato IPv4 como, por ejemplo, '192.168.1.100'. Las direcciones permiten al motor que interpreta el modelo buscar los elementos

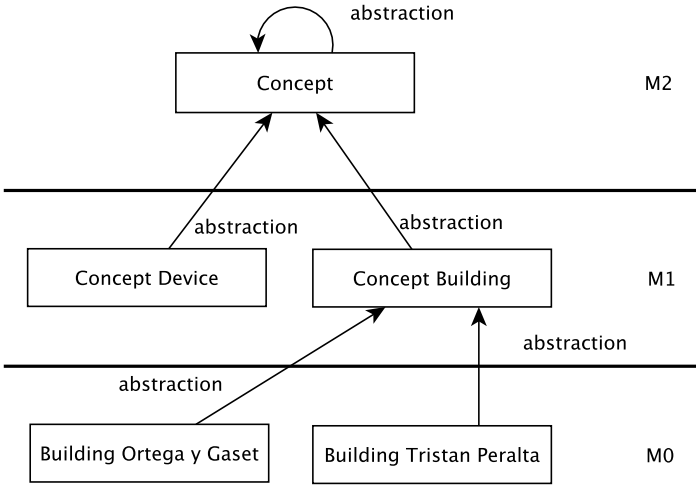

Figura 5.7: Descripción de una abstracción entre modelos.

a partir de su dirección. También permite la declaración de elementos sin nombre (anónimos).

5.2.2. Variables

Las variables son los parámetros que puede disponer un elemento. Proteo permite declarar variables de los siguientes tipos:

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

Las variables pueden tener valores por defecto. Las referencias sólo pueden tener el valor por defecto “empty”. Asimismo, todos los tipos de variables pueden tener valores múltiples. En el siguiente nivel de la arquitectura de niveles se deberán

inicializar las variables, a no ser que ya tengan un valor por defecto o una anotación (apartado 5.2.4) indique lo contrario.

5.2.3. Facetas

Las facetas son mecanismos que permiten aportar funcionalidades o comportamientos a otros elementos. No son elementos instanciables por sí mismos, sino que se aplican sobre otros elementos. Un elemento declarado como faceta puede tener:

- Variables
- Elementos de tipo componente
- Targets (on). Elementos sobre los que son aplicables. Al menos debe ser uno.

En el siguiente nivel de la arquitectura de niveles, las instancias de los elementos pueden implementar alguna de las facetas definidas en el nivel superior. Esto implica que además debe describir, usando un lenguaje de propósito general, su comportamiento.

5.2.4. Anotaciones

Todo elemento puede tener anotaciones que modifican su comportamiento. Las anotaciones recogidas en Proteo son:

- Terminal. Mediante esta anotación en modelador restringe que el elemento del modelo solo sea instanciable a nivel de sistema. Además, resulta obligatoria.
- Abstract. Elemento no es instanciable en el siguiente nivel.
- Named. Obliga a que todas sus instancias tengan nombre.
- Single. Restringe a una el número de instancias posibles del elemento anotado.
- Required. Obliga que en todos los modelos derivados de éste deban tener al menos una instancia de este tipo.

- Component. Restringe la instancia del elemento a ser componente de otro elemento. Es decir, el elemento no podrá ser raíz del modelo.
- Aggregated. Permite especificar la relación de agregación entre dos elementos del modelo. Por defecto es de composición.
- Property. Tiene la propiedad de component y single. Además, un elemento 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 un instanciable. No tienen contenido, pero permiten especialización.
- Facet. Anota un elemento como faceta.

Todas las anotaciones se heredan, excepto abstracto. Por su parte, las variables también pueden tener ciertas anotaciones. Estas son:

- Terminal. Permite no dar valor a la variable hasta el nivel terminal (M0).
- Local. Esta anotación es solo aplicable a las variables de tipo referencia. Limita el espectro de la referencia a componentes del elemento raíz en el que se encuentra el que se está instanciando.

Capítulo 6

Trabajo instrumental

6.1. Herramientas desarrolladas

En este trabajo se plantea la hipótesis de la posible existencia de una gramática capaz de representar múltiples lenguajes para el modelado en diferentes ámbitos de conocimiento. Por tanto, este apartado se centra en la búsqueda y desarrollo de una **gramática** genérica que defina el conjunto de reglas admisible en el lenguaje.

Esta gramática constituye la base del super-lenguaje Proteo. En el contexto de este trabajo, un super-lenguaje es un lenguaje abstracto que puede ser extendido para definir otros lenguajes, heredando todas las propiedades y estructura.

Proteo tiene como propósito ser un super-lenguaje para la especialización de Lenguajes Específicos de Dominio (DSLs), los cuales serán utilizados bajo la metodología Model Driven Engineering[43].

Proteo define una estructura con un léxico abstracto, sobre la cual se sustenta una sintaxis y semántica común para todos los lenguajes derivados (figura 6.1).

El léxico comprende un catálogo de las palabras del lenguaje. Proteo define ciertas palabras reservadas, sin embargo, cada DSL especificado deberá incluir los elementos instanciables que pueden ser definidos en él.

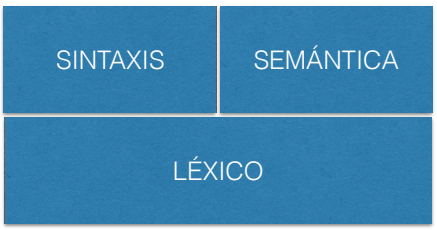

Figura 6.1: Estructura de Proteo.

Por su parte, la sintaxis y la semántica será la misma para los DSLs derivados. La sintaxis comprende la forma que adopta estructuralmente el lenguaje, mientras que la semántica comprueba el correcto significado de los diversos elementos que lo componen.

Bajo este enfoque, la creación de nuevos lenguajes específicos se reduce a completar el léxico abstracto de Proteo con los elementos instanciables en cada ámbito de conocimiento. Estos instanciables son definidos en los metamodelos inherentes a cada lenguaje, nivel M2 de la arquitectura de niveles (apartado 2.2). Por ejemplo, en el contexto de sistemas de información los elementos instanciables podrían ser formularios, colecciones, campos o catálogos.

El super-lenguaje requiere de analizadores que validen la composición estructural, así como la coherencia semántica de los lenguajes derivados. Los siguientes apartados abordan estos aspectos siguiendo la fase de análisis, propia de la construcción de un compilador. Esta fase comprende análisis léxico, sintáctico y semántico para Proteo.

6.1.1. Análisis léxico

El análisis léxico constituye la primera fase de un compilador. Su principal función consiste en leer el modelo escrito por el usuario y, en base a él, elaborar como salida una secuencia de “tokens” o símbolos. Estos tokens indican el conjunto de posibles secuencias de caracteres aceptadas por el lenguaje.

La construcción de un analizador léxico para Proteo requiere la especificación léxica del super-lenguaje. A lo largo de este apartado se muestran las diferentes partes que conforman el léxico abstracto desarrollado.

El código 6.1 contiene la declaración de algunas de las palabras clave que conforman el léxico de Proteo. Su formación es sencilla pues el token, a la izquierda, se corresponde directamente con una cadena de texto, a la derecha.

Código 6.1: Palabras clave.

```
SUB             : 'sub';
VAR             : 'var';
EXTENDS         : 'extends';
ABSTRACT        : 'abstract';
SINGLE          : 'single';
REQUIRED        : 'required';
COMPONENT       : 'component';
AGGREGATED      : 'aggregated';
INT_TYPE        : 'integer';
DOUBLE_TYPE     : 'double';
STRING_TYPE     : 'string';
...
```

Entre la multitud de palabras reservadas se encuentran las anotaciones (single, required, aggregated, etc), los tipos de variables (enteros, reales, cadenas de texto, etc), declaración de herencia entre elementos (sub, extends), entre muchos otros.

Cada lenguaje que se especializa a través de Proteo dispone de sus propios elementos instanciables, propios para su ámbito de aplicación. El léxico dispone de un token genérico que los identifica. Este token toma el nombre de METAIDENTIFIER. El código 6.2 contiene un ejemplo cumplimentado de este token para un DSL en el contexto de los sistemas de información. Nótese que la barra representa un operador booleano OR.

Código 6.2: Elementos instanciables.

```
METAIDENTIFIER      : 'Collection' | 'Form' | 'Catalog';
```

También existen una serie de símbolos en el super-lenguaje, como los definidos en el código 6.3. Éstos son utilizados con múltiples fines en Proteo. Un ejemplo es el símbolo igual (‘=’) para la asignación de valores a variables. El tratamiento léxico de estas reglas es idéntico al de las palabras clave.

Código 6.3: Símbolos.

```
LEFT_PARENTHESIS    : '(';
RIGHT_PARENTHESIS   : ')';
LEFT_SQUARE         : '[';
RIGHT_SQUARE        : ']';
EQUALS              : '=';
...
```

El léxico también identifica los posibles valores a los que puede ser inicializada una variable en el lenguaje. En el código 6.4 quedan reflejados algunos de ellos. Por ejemplo, un booleano puede ser verdadero o falso, un número negativo dispone del símbolo ‘-’ seguido de un uno o múltiples dígitos o que una cadena de texto es todo aquello entrecomillado.

Nótese la notación utilizada, donde el signo de interrogación (‘?’) corresponde a 0 o 1 elementos de ese token, el símbolo de suma (‘+’) es de 1 a n elementos y el asterisco (‘*’) de 0 a n elementos.

Código 6.4: Valores.

```
BOOLEAN_VALUE       : 'true' | 'false';
NATURAL_VALUE       : PLUS? DIGIT+;
NEGATIVE_VALUE      : DASH DIGIT+ ;
DOUBLE_VALUE        : (PLUS | DASH)? DIGIT+ DOT DIGIT+;
STRING_VALUE        : APHOSTROPHE (~'\'')* APHOSTROPHE;
...
```

Otro elemento muy importante en los lenguajes es el identificador (código 6.5). Este token representa el nombre que toma un elemento o una variable en el DSL. Se compone de una serie de letras, números y/o barras, con la restricción de que debe comenzar por una letra.

Código 6.5: Identificador.

```
IDENTIFIER     : LETTER (DIGIT | LETTER | DASH | UNDERDASH)*;
```

En Proteo se identifican los bloques de código. En lenguajes como C++ éstos se definen con llaves (‘{’ y ‘}’). Sin embargo, para mejorar la expresividad, se ha decidido utilizar una estructura similar al lenguaje Python, donde los bloques son definidos por tabulaciones.

Los tokens NEW_LINE_INDENT y DEDENT del léxico (código 6.6) identifican estos bloques. Ninguno de ellos dispone de reglas, sino que son emitidos cuando el nivel de indentación aumenta o disminuye. Estos niveles de indentación son controlados por una clase propia, encargada de evaluar cuándo se cumplen los requisitos y emitir el token NEW_LINE_INDENT cuando comienza un bloque o DEDENT cuando finaliza.

Código 6.6: Indentaciones.

```
NEWLINE: NL+ SP* { newlinesAndSpaces(); };
SP: (' ' | '\t');
NL: ('\r'? '\n' | '\r');

NEW_LINE_INDENT: 'indent';
DEDENT         : 'dedent';
```

Para la creación del analizador léxico se ha utilizado ANTLR[36], herramienta software para el trabajo con lenguajes (apartado 6.2). A partir de una especificación léxica como la anterior, ANTLR genera automáticamente el analizador en Java. Cabe destacar que los fragmentos de código mostrados son solo una pequeña parte del léxico desarrollado.

6.1.2. Análisis sintáctico

La fase de análisis sintáctico trata de evaluar si una sucesión de símbolos, suministrados por el léxico, siguen una determinada estructura sintáctica definida por el lenguaje.

El desarrollo de un analizador sintáctico requiere definir una gramática. Las gramáticas son estructuras con un conjunto de reglas de formación que definen las cadenas de caracteres admisibles en un determinado lenguaje.

La hipótesis de este trabajo se centra en la existencia de una gramática genérica que permita la definición de múltiples DSLs para diversos dominios. Por tanto, esta etapa resulta crucial para su posible validación.

La gramática diseñada corresponde a una gramática libre del contexto, tipo 2 de la jerarquía de Chomsky[5] (apartado 4.2). En este tipo de gramáticas cada regla de producción sigue la forma:

V $\rightarrow$w

siendo V un símbolo no terminal y w una cadena de terminales y/o no terminales.

A continuación se describen las estructuras principales que conforman la gramática del super-lenguaje Proteo. La gramática diseñada comparte las mismas notaciones que el léxico para expresar multiplicidad (‘+’ y ‘*’) y opcionalidad (‘?’).

Variables

Las variables son los parámetros de los que dispone un elemento. En Proteo se han predefinido ciertos tipos primitivos de variables, como los números naturales, enteros o reales, booleanos, cadenas de texto, fechas, words (enumerados) y referencias.

La manera en que son expresadas las variables es mediante la palabra clave “var”, tipo primitivo (como “string”, “integer”, “boolean”, etc), identificador y

la asignación de un valor determinado. La asignación es opcional. El ejemplo 6.7 ilustra una variable de tipo cadena de texto.

Código 6.7: Ejemplo variable.

```
var string nombre = "Antonio"
```

La gramática para todos los primitivos es muy similar, salvo por el token que indica el tipo y el valor que pueden tener. Las únicas excepciones son los enumerados, que requieren múltiples valores, y las referencias, que requieren el nombre de un elemento al que referenciar. La gramática 6.8 detalla la regla para la formación de variables y especifica cómo es la asignación para el caso de las cadenas de texto. La figura 6.2 ilustra el ejemplo 6.7 a través del árbol de derivación.

Código 6.8: Sintaxis variable.

```
variable : VAR (integerAttribute | doubleAttribute | booleanAttribute
    | stringAttribute | dateAttribute | reference | word);

stringAttribute: STRING_TYPE LIST? IDENTIFIER (EQUALS STRING_VALUE+)?;
```

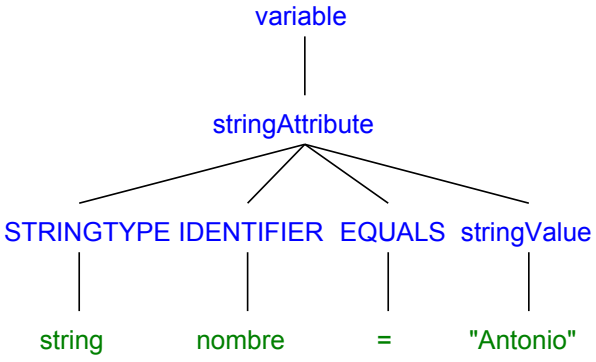

Figura 6.2: Árbol de derivación de una variable.

Elementos

Los elementos son los principales protagonistas del super-lenguaje, pues son el elemento básico de definición (de ahí su nombre). Sintácticamente conforman una estructura bastante compleja. La definición de un elemento en el lenguaje dispone de tres aspectos importantes:

1. Firma (signature)
2. Anotaciones y facetas (annotationsAndFacets)
3. Cuerpo (body)

Gramaticalmente, estos tres aspectos conforman la regla de un elemento, como refleja el código 6.9. De entre las tres reglas, la única obligatoria es la firma.

Código 6.9: Sintaxis elemento.

```
element: signature annotationsAndFacets? body?;
```

- *Firma.* La firma del elemento corresponde a la declaración misma del elemento y sus propiedades. En ella se declaran los siguientes aspectos:

  1. Elemento instanciable.
  2. Identificador (nombre) del elemento declarado.
  3. Inicialización de los posibles parámetros contenidos por el instanciable.
  4. Posible herencia de otro elemento declarado en el lenguaje.

  En el código 6.10 se puede ver el ejemplo de una firma. En concreto, se declara un elemento formulario (instanciable Form) que toma el nombre de FichaTecnica. Asimismo, inicializa su etiqueta (parámetro que tiene en el metamodelo el instanciable Form) con el valor “Formulario 001” y hereda las propiedades de otro elemento llamado FichaGenerica. Gramaticalmente, la firma del modelo queda como en el código 6.11.

Código 6.10: Ejemplo firma.

```
Form FichaTecnica(label="Formulario 001") extends FichaGenerica
```

Código 6.11: Firma elemento.

```
signature: METAIDENTIFIER IDENTIFIER parameters? (EXTENDS
    identifierReference)?;
```

Destacar el token METAIDENTIFIER que indica el tipo de elemento instanciable a utilizar en la declaración del elemento. La regla "parameters" representa a una posible lista de parámetros, mientras que "identifierReference" es el nombre o ruta de un elemento definido en el modelo. El árbol de derivación de la figura 6.3 aclara la firma del ejemplo anterior.

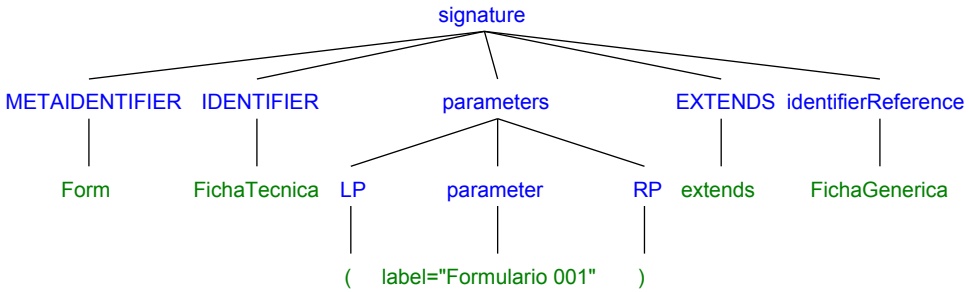

Figura 6.3: Árbol de derivación, ejemplo firma.

- *Anotaciones* y *facetas*. Las anotaciones permiten dotar a un elemento de una determinada propiedad, por ejemplo, que sea de solo lectura. Por su parte, las facetas son elementos no instanciables que aportan funcionalidades o comportamientos a otros elementos. En ambos casos son aplicados con la palabra reservada "is" junto a la firma del elemento. En 6.12 se expone un ejemplo de un elemento con múltiples anotaciones.

Código 6.12: Anotaciones elementos.

```
Form FichaTecnica is named single
```

Con respecto a la gramática, el uso de anotaciones y facetas es muy similar, salvo por el hecho de que las anotaciones están acotadas (son fijas) y las facetas son dinámicas. Así pues, la regla de producción para su aplicación queda como en 6.13.

Código 6.13: Anotaciones y facetas.

```
annotationsAndFacets: IS (annotations | facetApply);

facetApply : METAIDENTIFIER parameters? (WITH METAIDENTIFIER)?
    body?;
annotation: ABSTRACT | TERMINAL | SINGLE | REQUIRED | READONLY |
...
```

Tomando el ejemplo 6.12, la figura 6.4 representa el árbol de derivación generado por la gramática.

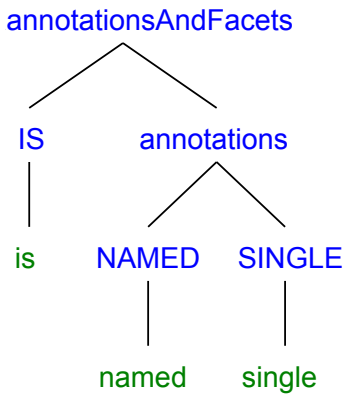

Figura 6.4: Árbol de derivación, ejemplo anotaciones.

- *Cuerpo.* El cuerpo de un elemento se compone de los parámetros (variables) y/u otros elementos con los que se relacione por composición, agregación o herencia. Textualmente el cuerpo se expresa con la utilización de tabulaciones, de un mismo tamaño, en las líneas posteriores a la firma del elemento.

El cuerpo permite expresar de diversas maneras la contención de elemento en otro. Si el elemento existe en el modelo, es posible utilizarlo con la directiva "has" seguida del identificador. Si el elemento extiende del que está siendo definido, se puede declarar con la directiva "sub". También es posible declarar un elemento de la forma habitual, a través de su firma.

En 6.14 se puede observar como el elemento FichaTecnica dispone de un parámetro llamado identificador. Al mismo tiempo, tiene una relación de agregación con el elemento Propietario.

Código 6.14: Cuerpo de un elemento.

```
Field Propietario

Form FichaTecnica
    var string identificador
    has Propietario is aggregated
```

La gramática que define el cuerpo de un elemento es algo compleja, pues intervienen muchos aspectos. Una aproximación algo simplificada es la que podemos ver en 6.15. Se han omitido ciertas reglas para una mejor compresión.

Código 6.15: Gramática del cuerpo del elemento.

```
body: NEW_LINE_INDENT (variable | element | elementReference)+
    DEDENT;

elementReference : HAS identifierReference (IS annotations)?;
```

Los tokens NEW_LINE_INDENT y DEDENT emitidos por el léxico controlan que haya un nuevo nivel de indentación. La única regla que no hemos visto es "elementReference" que posibilita declarar la relaciones con elementos ya existentes en el modelo. La figura 6.5 clarifica la gramática con el ejemplo anterior.

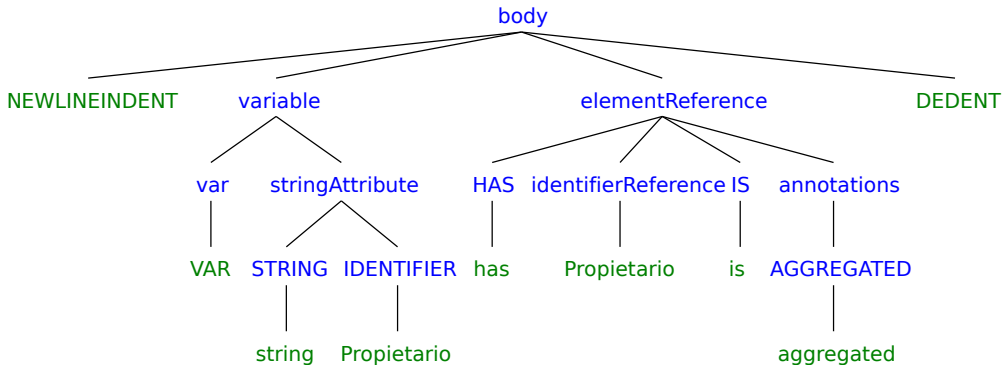

Figura 6.5: Árbol de derivación, ejemplo cuerpo.

Para crear el analizador sintáctico se ha utilizado nuevamente ANTLR[36], herramienta detallada en el apartado 6.2. Mediante el léxico y la gramática definida, ANTLR genera automáticamente un analizador sintáctico LL(*) para el lenguaje. Adicionalmente, crea un árbol de derivación a partir de la gramática y la información del modelo. De este modo, se dispone de una estructura por la que recorrer los datos introducidos.

Los fragmentos gramaticales mostrados a lo largo de este apartado conforman solo la base de la gramática. La gramática final dispone de muchas otras reglas para aspectos más específicos del lenguaje.

6.1.3. Análisis semántico

La etapa de análisis semántico trata de comprobar restricciones e incoherencias que pueda existir en el modelo. Para ello, requiere de algún tipo de estructura que le permita recorrer los datos introducidos fácilmente.

Aprovechando el árbol de derivación (Parse Tree) generado por el analizador sintáctico, se ha creado un árbol de sintaxis abstracta (Abstract Syntax Tree, AST). Este tipo de estructura representa los datos en forma de árbol, omitiendo aquellos detalles desechables o innecesarios.

El analizador semántico desarrollado recorre el AST en profundidad. Su propósito es comprobar algunos aspectos genéricos del lenguaje, como:

- *Elementos duplicados*. Comprueba la existencia de elementos en el lenguaje que tengan el mismo identificador.

- *Referencias correctas*. Se comprueba que los elementos a los que se hacen referencia estén definidos.

- *Comprobación de anotaciones*. Ciertas anotaciones disponen de restricciones. Por ejemplo, un elemento no puede ser componente y agragado al mismo tiempo. Por ello, se comprueba si las anotaciones introducidas son correctas.

- *Comprobación de rutas*. Los elementos son expresables también mediante rutas. Esta etapa comprueba si una ruta es correcta. Un ejemplo de ruta puede ser "FichaTecnica.Fecha", donde Fecha es un elemento definido internamente al elemento FichaTecnica.

En el caso de los modelos es necesario comprobar la concreción o instanciación entre los elementos definidos en el lenguaje con los del metamodelo. Estas restricciones no se evalúan durante el análisis sintáctico. A continuación se ilustra un ejemplo de esta problemática.

Imaginemos un metamodelo como el del código 6.16, donde se describen los elementos instanciables: Colecciones (Collection), Formularios (Form) y Campos (Field). Asimismo, se indica una relación de composición entre los formularios y los campos.

Código 6.16: Metamodelo de ejemplo.

```
Concept Collection is root
Concept Form is root
    Concept Field is multiple
```

A continuación se pueden ver dos modelos de ejemplo. El modelo del código 6.17 es erróneo pues no respeta el metamodelo descrito. Concretamente comete dos errores. En primer lugar, describe un elemento instanciable inexistente en

el metamodelo (Catalog). En segundo lugar, crea una colección compuesta de formularios, sin embargo, dicha relación no existe en el metamodelo.

Código 6.17: Modelo erróneo.

```
Catalog CatalogoCoches
Collection Coches
    Form FichaTecnica
```

Por su parte, el código 6.18 describe un modelo que si respeta los elementos instanciables y las relaciones descritas en el metamodelo.

Código 6.18: Modelo correcto.

```
Form FichaTecnica
    Field Propietario
    Field MarcaCoche
```

Para este tipo de comprobaciones es necesario disponer del árbol sintáctico del metamodelo. Así, se podrá identificar los elementos, parámetros y relaciones entre los elementos allí definidos.

El analizador sintáctico aún está en una fase temprana de desarrollo. Por lo tanto, es muy probable que la cantidad de comprobaciones finales aumente. No obstante, es deseable recorrer lo menos posible el AST para no comprometer el tiempo de respuesta del analizador.

6.2. Recursos

En esta sección se exponen los recursos utilizados para el desarrollo del trabajo de fin de máster. Éstos se dividen en dos categorías: software y hardware.

Recursos Software

Los recursos software utilizados son, en su mayoría, los empleados comúnmente por parte de un desarrollador de software. La única excepción es ANTLR, una herramienta más específica para trabajar con lenguajes.

- *ANTLR*[36] (ANother Tool for Language Recognition). Es una herramienta multiplataforma que facilita el trabajo con lenguajes. Proporciona un marco para construir reconocedores (parsers), intérpretes, compiladores y traductores de lenguajes a partir de las descripciones gramaticales.

  ANTLR es considerado un meta-programa, por ser un programa que escribe otros programas. Partiendo de la descripción formal de la gramática de un lenguaje, genera un programa que determina si una sentencia o palabra pertenece a dicho lenguaje (reconocedor), utilizando algoritmos LL(*) de parsing. La figura 6.6 muestra un ejemplo gráfico de su funcionamiento.

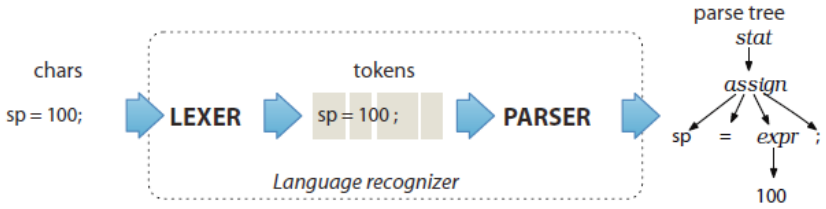

Figura 6.6: Esquema funcionamiento de ANTLR.

  Las gramáticas en ANTLR son expresadas en una notación propia que toma prestados conceptos de notaciones como BNF[3] o EBNF[37]. De manera adicional, la gramática permite añadir acciones escritas en un lenguaje de programación, transformando así el reconocedor en un traductor o intérprete.

  Asimismo, ANTLR también proporciona ciertas estructuras intermedias de análisis (Parse Trees, árboles de derivación) (figura 6.7) para los datos introducidos. Además, provee ciertos mecanismos para la recuperación automática de errores y su comunicación.

- *JVM*, Java Virtual Machine. Es un entorno de ejecución de una plataforma independiente que convierte código Java a lenguaje máquina y lo ejecuta. Una JVM simula un procesador real Java, permitiendo que el código Java sea ejecutado como acciones o llamadas al sistema a cualquier procesador sin importar el sistema operativo.

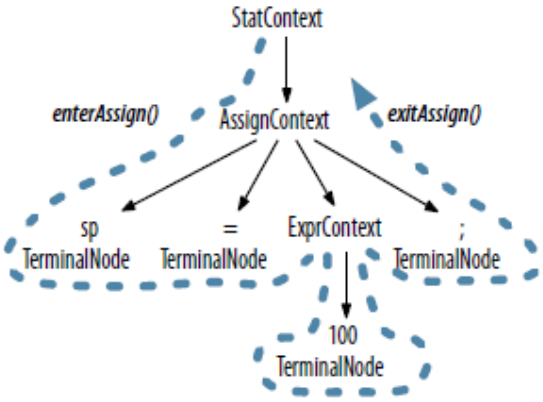

Figura 6.7: Recorrido por el árbol de derivación.

- *JRE*, Java Runtime Environment. JRE incluye la JVM, librerías y componentes que son necesarios para ejecutar programas escritos en Java. JRE está disponible para múltiples plataformas, entre las que se encuentran MAC, Windows y Unix.

- *JDK*, Java Development Kit. Es un kit de desarrollo de software (SDK) para la creación de programas en Java.

- *Intellij IDEA*. Es un entorno de desarrollo integrado (IDE) multiplataforma para la programación en Java. Entre sus características se encuentran: el autocompletado inteligente de código, integración con sistemas de control de versiones, modularidad y posibilidad de ampliación usando plugins.

  Durante el desarrollo de este trabajo se ha utilizado un plugin de ANTLR para Intellij que integra ciertas funcionalidades, como el reconocimiento de gramáticas ANTLR o la generación del reconocedor (parser).

- *Sistema operativo*. El uso de Java, ANTLR e Intellij Idea no condiciona la elección del sistema operativo, pues todos ellos se encuentran disponibles en diversas plataformas. En este trabajo se utilizó un sistema operativo Windows 7 de 64 bits que el autor tenía instalado previamente.

Recursos Hardware

A continuación se expone el hardware empleado para el desarrollo del trabajo. Cabe destacar que el software utilizado no requiere el uso de hardware de alto rendimiento.

- *Equipo informático.* Se utilizó un ordenador portátil HP Envy 14 con las siguientes características:
  - CPU: Intel Core i7-720QM
  - Memoria RAM: 4 GB
  - Disco duro: 500 GB

6.3. Metodologías de desarrollo

Durante el desarrollo del trabajo se han seguido una metodología iterativa incremental, aplicando el enfoque Specification by Example. Se han escogido estas metodologías para seguir un desarrollo paulatino y mediante ejemplos de uso real.

6.3.1. Iterativo incremental

El desarrollo iterativo incremental[30] se compone de un conjunto de tareas agrupadas en pequeñas etapas repetitivas (iteraciones). El comportamiento deseado es que en cada iteración el producto evolucione con respecto al resultado de las iteraciones antecesoras, introduciendo nuevos requisitos y logrando así una mejora paulatina. Las tareas definidas en este trabajo para cada iteración son las siguientes:

1. Definición del super-lenguaje.
2. Desarrollo del analizador léxico.
3. Desarrollo del analizador sintáctico.
4. Desarrollo del analizador semántico.

Finalmente, tras terminar todas las etapas, se evalúa el resultado y deciden nuevos requisitos a introducir en la iteración siguiente.

6.3.2. Specification-by-example

En el entorno de las metodologías ágiles, Specification by Example[1] (SBE) es un enfoque para la definición de los requisitos de software impulsado por usuarios. Se encuadra dentro del proceso de desarrollo software denominado Behaviour-Driven Development (desarrollo basado en el comportamiento).

SBE requiere de expertos en el dominio que proporcionen escenarios realistas de cómo se utilizará el software, así como posibles ejemplos de uso. De este modo, el desarrollo del software se enfoca en ejemplo de uso reales. Este enfoque dispone de dos ventajas principales:

1. Fomenta la comunicación continua entre las compañías de negocio y el equipo de desarrollo software.

2. Ayuda a los desarrolladores a especificar el software con pruebas de uso real.

La aplicación de SBE permite conocer desde el inicio los requisitos del software. Ello evita rehacer trabajo, mejorando así la productividad y aumentando el tiempo de respuesta ante cambios del software[1].

En este trabajo se han utilizado ejemplos de uso reales de sistemas de información y de simulación[15] para la definición del super-lenguaje desarrollado (apartado 5).

Capítulo 7

Resultado

El resultado de este trabajo es un super-lenguaje para facilitar la creación de de lenguajes específicos de dominio. El super-lenguaje es análogo a una superclase en un lenguaje de programación orientado a objetos, ya que permite la creación de un lenguaje derivado heredando todas las propiedades y estructura del mismo.

A continuación es posible ver dos ejemplos de Lenguajes Específicos de Dominio (DSL) implementados a partir del super-lenguaje Proteo, donde ambos disponen de la gramática genérica diseñada. El primero de ellos corresponde a un lenguaje específico para el ámbito de los sistemas de información. El modelo descrito (código 7.1) representa la definición de un formulario de ficha técnica, el cual dispone de un campo nombre, una fecha de nacimiento y una sección en la cual cumplimentar la dirección.

Código 7.1: Modelo de formulario.

```
Form Ficha(label="FichaTecnica")
    TextField Nombre("nombre")
    DateField FechaNacimiento(label="Fecha de nacimiento", precision=
        DAYS)
    CompositeField Direccion(label="Direccion")
        TextField Calle("Calle")
        TextField Numero ("Numero")
```

Por su parte, el segundo DSL pertenece al dominio de los sistemas de simulación. El modelo (código 7.2) define los distintos tipos de lugares que existen, donde cada uno de ellos dispone de un clima.

Código 7.2: Modelo de lugares.

```
Definition Place is named
    var Weather weather = empty

    sub Neighborhood
    sub District
        has Neighborhood
    sub Urbe
        has District
    sub Region
        has City
    sub Country
        has Region
```

Ambos lenguajes toman al super-lenguaje Proteo como base. A partir de él, cada DSL especifica en su léxico los elementos instanciables en su ámbito de conocimiento particular. Por ejemplo, en el lenguaje para los sistemas de información (código 7.1) se habrán definidos los instanciables: Form, TextField, DateField y CompositeField. Por su parte, la gramática y semántica es igual en ambos DSLs.

Capítulo 8

Conclusiones

Este trabajo es consecuencia de la problemática existente en el ámbito del desarrollo de software. Uno de ellos es la falta de flexibilidad de los equipos de desarrollo para adoptar cambios en los requisitos del cliente. Otro problema es la complejidad existente tanto a nivel tecnológico como en las propias necesidades de los usuarios. Estas dos dimensiones requieren fortalecer la comunicación entre usuarios y desarrolladores. 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[11] en MDE, en la que se define un motor de propósito general capaz de interpretar modelos 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 los modelos con un lenguaje más cercano al dominio de aplicación. Estos lenguajes pueden disponer de una curva de aprendizaje suave, no obstante, diferentes DSLs disponen de diferentes mecanismos para expresar los modelos. El coste de aprendizaje de cada uno de ellos conlleva un coste de tiempo y esfuerzo considerable. Debido a esta problemática, en este trabajo se plantea la posibilidad de reducir la curva de aprendizaje entre diferentes DSLs.

Un estudio del estado del arte ha permitido comprobar en qué estado se encuentran los conceptos manejados en el documento. Al mismo tiempo, nos da una visión completa de los diversos enfoques y aplicaciones que realizan otros investigadores que trabajan con MDE. La problemática existente lleva a formular la siguiente pregunta

¿Podría existir un lenguaje abstracto a partir del cual pudieran derivarse lenguajes específicos para cada dominio?

De ser así, todos los lenguajes serían expresables mediante una misma sintaxis, variando solo el catalogo de palabras (léxico) propio de cada dominio. Una gramática genérica da pie a la creación de un super-lenguaje, entendido éste como un lenguaje abstracto que puede ser extendido para definir DSLs.

Por tanto, la hipótesis que sigue esta línea de investigación es "si los dominios son expresables como un conjunto de elementos instanciables, asociados entre sí por relaciones de composición, agregación, abstracción y concreción; debiera ser posible la existencia de una gramática común a todos los dominios y, por tanto, la creación de un super-lenguaje.".

En este trabajo se ha creado un super-lenguaje con el nombre de Proteo. El propósito Proteo es permitir la definición de Lenguajes Específicos de Dominio para el modelado bajo la metodología Model Driven Engineering. Proteo constituye la base común para todos los DSLs. Asimismo, se han desarrollado diversos analizadores para comprobar la validez de los modelos.

La validación de la hipótesis conlleva probar que la gramática construida permita crear DSLs para todos los dominios del mundo. Esto no es posible, al ser el número de dominios infinito. Por ello, se llevarán a cabo futuras experimentos que permitan observar que la hipótesis es válida para aquellos ámbitos aplicados. Sin embargo, nunca se tendrá la certeza de que sea válido para todos. Será misión de otros investigadores encontrar ámbitos en los que no se aplicable y, por tanto, puedan refutar la hipótesis.

Este proyecto aporta la base para construir lenguajes específicos de dominio. A través de estos DSLs, el desarrollador modela la realidad utilizando elementos de su contexto. Ello ayuda a mejorar la comunicación entre el equipo de desarrollo y los expertos en el dominio. Asimismo, este enfoque aporta eficiencia y productividad a aquellas personas que trabajen con múltiples DSL al reducir la curva de aprendizaje.

Como continuación a este trabajo se deberá validar la capacidad de la gramática diseñada para describir DSLs en otros dominios. Ello dará pie al inicio de una investigación en la cual se creen nuevos lenguajes en ámbitos aún no contemplados. De manera complementaria, esto permitirá refinar los mecanismos ya existentes del super-lenguaje e introducir algunos no planteados hasta el momento. Adicionalmente, se tratará de mejorar la expresividad del super-lenguaje para conseguir lenguajes cada vez más sencillos de manejar y entender.

Bibliografía

[1] G Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Pubs Co Series. Manning, 2011.

[2] AV Aho, R Sethi, and JD Ullman. Compilers: Principles. *Techniques, and Tools*, 1986.

[3] JW Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. *Proceedings of the International Comference on ...*, 1959.

[4] C Brooks, TH Feng, EA Lee, and R van Hanxleden. Multimodeling: A preliminary case study. 2008.

[5] N Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 1956.

[6] N Chomsky. Syntactic structures. 2002.

[7] Noam Chomsky. Aspects of the theory of syntax. Technical report, DTIC Document, 1964.

[8] Microsoft Corporation. Microsoft Software Factories. http://msdn.microsoft.com/en-us/library/ff699235.aspx.

[9] JS Cuadrado. Applying model-driven engineering in small software enterprises. *Science of Computer ...*, 2014.

[10] JS Cuadrado and JG Molina. A model-based approach to families of embedded domain-specific languages. *Software Engineering, IEEE ...*, 2009.

[11] Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería. SIANI. http://www.siani.es.

[12] A Van Deursen, P Klint, and J Visser. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices*, 2000.

[13] ELSEVIER. Scopus. www.scopus.com.

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

[15] Jose Evora, Jose Juan Hernandez, and Mario Hernandez. Tafat: A framework for developing simulators based on Model Driven Engineering. In *European Simulation and Modelling Conference 2013 (ESM'13)*, 2013.

[16] RG Flatscher. Metamodeling in EIA/CDIF—meta-metamodel and metamodels. *ACM Transactions on Modeling and Computer ...*, 2002.

[17] F Fleurey, E Breton, and B Baudry. Model-driven engineering for software migration in a large industrial context. *... Languages and Systems*, 2007.

[18] A Gamatié, S Le Beux, and É Piel. A model-driven design framework for massively parallel embedded systems. *ACM Transactions on ...*, 2011.

[19] V García-Díaz. Talisman mde: Mixing MDE principles. *Journal of Systems and ...*, 2010.

[20] V García-Díaz and JB Tolosa. Talisman mde Framework: An Architecture for Intelligent Model-Driven Engineering. *..., Soft Computing, and ...*, 2009.

[21] JM Gascueña. Model-driven engineering techniques for the development of multi-agent systems. *... Applications of Artificial ...*, 2012.

[22] O Gilles and J Hugues. A MDE-based optimisation process for Real-Time systems. *Object/Component/Service-Oriented Real- ...*, 2010.

[23] J Greenfield and K Short. Software factories: assembling applications with patterns, models, frameworks and tools. *Companion of the 18th annual ACM SIGPLAN ...*, 2003.

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

[25] JM Jézéquel, O Barais, and F Fleurey. Model driven language engineering with kermeta. *Generative and Transformational ...*, 2011.

[26] Fernand Joly and Julio Morencos Tévar. La cartografía, 1979.

[27] S Kelly and JP Tolvanen. Domain-specific modeling: enabling full code generation. 2008.

[28] Glenn E Krasner, Stephen T Pope, and Others. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[29] F Krichen, B Hamid, and B Zalila. Development of reconfigurable distributed embedded systems with a model-driven approach. *Concurrency and ...*, 2013.

[30] C Larman and VR Basili. Iterative and incremental development: A brief history. *Computer*, 2003.

[31] MM Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1980.

[32] Abel Gómez Llana. Recuperación, transformación y simulación de datos biológicos mediante ingeniería dirigida por modelos. Master's thesis, Universidad Politécnica de Valencia, 2010.

[33] Object Management Group (OMG). MDA - The Architecture Of Choice For A Changing World. http://www.omg.org/mda/.

[34] Object Management Group (OMG). MOF - Meta Object Facility. http://www.omg.org/mof/.

[35] Object Management Group (OMG). Unified Modeling Language. http://www.uml.org.

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

[37] Richard E Pattis. Ebnf: A notation to describe syntax. *While developing a manuscript for a textbook on the Ada programming language in the late 1980s, I wrote a chapter on EBNF*, 1980.

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

[39] EL Post. Formal reductions of the general combinatorial decision problem. *American journal of mathematics*, 1943.

[40] Thomson Reuters. Web Of Science. www.webofscience.com.

[41] DF Rico, HH Sayani, and RF Field. History of Computers, Electronic Commerce and Agile Methods. *Advances in Computers*, 2008.

[42] S Rougemaille and F Migeon. Model driven engineering for designing adaptive multi-agents systems. *Engineering Societies in ...*, 2008.

[43] DC Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 2006.

[44] E Seidewitz. What models mean. *IEEE software*, 2003.

[45] J Sztipanovits and G Karsai. Model-integrated computing. *Computer*, 1997.

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

[47] C Thompson and J White. Optimizing mobile application performance with model–driven engineering. *Software Technologies for ...*, 2009.

[48] A Vachoux. Applications of Specification and Design Languages for SoCs. 2006.

[49] C Vicente-Chicote and F Losilla. Applying MDE to the development of flexible and reusable wireless sensor networks. *International Journal ...*, 2007.