Tesis doctoral

Propuesta de mecanismos de personalización de meta-modelos en la Ingeniería Dirigida por Modelos

Autor: Santiago Jácome Guerrero
Director: Juan de Lara Jaramillo

UNIVERSIDAD AUTÓNOMA DE MADRID
Escuela Politécnica Superior
Departamento de Ingeniería Informática
Madrid, España

Doctorado en Ingeniería Informática y de Telecomunicación

Junio 2019

Resumen

La Ingeniería Dirigida por Modelos (MDE, de sus siglas en inglés) es una propuesta para el desarrollo de software en la que se atribuye a los modelos el rol principal, frente a las propuestas tradicionales basadas en lenguajes de programación. De esta manera, los modelos son utilizados desde etapas tempranas del desarrollo hasta la implementación de la solución técnica. En MDE los modelos se representan mediante lenguajes de modelado, que deben ser definidos con una sintaxis y semántica precisas para que sean útiles en un dominio específico.

Uno de los hechos ineludibles en la ingeniería de software es que el software siempre está en permanente cambio o evolución debido a varios factores, entre ellos la aparición de nuevos requisitos o los cambios de tecnología. Por tanto, en un entorno de desarrollo de software dirigido por modelos para facilitar la evolución del software, los modelos y sus lenguajes de modelado deben poder cambiar. Es de considerar que, a pesar de este inconveniente, MDE aún no provee de un mecanismo estándar para el proceso de personalización o adaptación de los lenguajes de modelado, que implica modificar sus elementos constitutivos.

El presente trabajo de tesis se centra en proporcionar mecanismos que permitan personalizar la sintaxis abstracta (meta-modelo) de los lenguajes de modelado. Por un lado, se diseña un mecanismo de extensibilidad de meta-modelos, que asegura que la extensibilidad sea realizada de acuerdo a unas normas previstas por el diseñador del lenguaje. Por otro lado, utilizando estos mecanismos de extensión se diseña un mecanismo para el modelado multi-nivel.

En términos concretos, esta tesis hace las siguientes contribuciones:

En el ámbito de extensibilidad de meta-modelos

- descripción de los escenarios donde el control de personalización del meta-modelo resulta de utilidad.
- clasificación de los tipos de personalización.
- diseño de un mecanismo para la definición de personalización de meta-modelos.
- implementación de una herramienta llamada TACO que implementa el mecanismo.
- evaluación del enfoque a través de un estudio de campo donde se muestra la necesidad de extensibilidad, y evaluación a través de la aplicación del enfoque en varios casos de estudio.

En el ámbito de modelado multi-nivel

- caracterización del espacio de diseño para soluciones de modelado multi-nivel a través de un modelo de características.
- clasificación y comparación de las herramientas de modelado multi-nivel existentes.
- diseño de un enfoque conceptual para el modelado multi-nivel integrado dentro de un entorno de modelado de dos meta-niveles.
- implementación de una herramienta llamada TOTEM que implementa estas ideas.
- evaluación empírica y analítica de la propuesta.

Palabras clave:
meta-modelo, personalización, extensibilidad, adaptación, modelado multi-nivel, Ecore, EMF.

Abstract

Model-Driven Engineering (MDE) is a software development paradigm in which models are attributed the main role, compared to traditional proposals based on programming languages. In this way, models are used from the early stages of development to the implementation of the technical solution. In MDE, models are represented by modeling languages, which must be defined with precise syntax and semantics to be useful in a specific domain.

One of the inescapable facts in software engineering is that software is always in permanent change or evolution due to several factors, among them the appearance of new requirements or the changes in technology. Therefore, in a model-driven software development environment to facilitate the evolution of software, models and their modeling languages must be able to change. It is to be considered that, despite this inconvenience, MDE still does not provide a standard mechanism for the process of personalization or adaptation of the modeling languages, which implies modifying its constituent elements.

This thesis focusses on providing mechanisms that allow customizing the abstract syntax (meta-model) of the modeling languages. On the one hand, a meta-model extensibility mechanism is designed, which ensures that the extensibility is carried out according to the rules provided by the language designer. On the other hand, using these extension mechanisms, a mechanism for multi-level modeling is designed.

In concrete terms, the thesis makes the following contributions:

In the scope of meta-model extensibility

- description of the scenarios where the personalization control of the meta-model is useful.
- classification of the types of personalization.
- design of a mechanism for specifying allowed personalizations of meta-models.
- implementation of a tool called TACO that implements the mechanism.
- evaluation of the approach through a field study where the need for extensibility is shown, and evaluation through the application of the approach in several case studies.

In the field of multi-level modeling

- characterization of the design space for multi-level modeling solutions through a feature model.
- classification and comparison of existing multi-level modeling tools.
- Design of a conceptual approach for integrated multi-level modeling within a two meta-level modeling environment.
- implementation of a tool called TOTEM that implements these ideas.
- empirical and analytical evaluation of the proposal.

Keywords:
meta-model, customization, extensibility, adaptation, multi-level modeling, Ecore, EMF.

Tabla de contenidos

1 INTRODUCCIÓN 1

1.1 Motivación 1
1.1.1 Mecanismo de control de la extensibilidad de meta-modelos 2
1.1.2 Refinamiento de meta-modelos mediante modelado multi-nivel 3

1.2 Contribuciones 4
1.2.1 Extensión de meta-modelos 4
1.2.2 Modelado multi-nivel 4

1.3 Publicaciones 5
1.3.1 Revistas internacionales con factor de impacto 5
1.3.2 Conferencias Internacionales 5
1.3.3 Conferencias Nacionales (Ecuador) 6

1.4 Organización 6

1.5 Agradecimientos 6

2 CONCEPTOS PREVIOS 8

2.1 Fundamentos de MDE 8
2.1.1 Conceptos básicos 8
2.1.2 Modelos y meta-modelos 12
2.1.3 Lenguajes de modelado y perfiles UML 15
2.1.4 Añadiendo restricciones al meta-modelo 17
2.1.5 Transformación de modelos y generación de código 18
2.1.6 Modelado multi-nivel 20
2.1.7 Modelado multi-nivel mediante ejemplos 21

2.2 Tecnologías MDE 28
2.2.1 Eclipse Modeling Framework 28
2.2.2 Lenguajes de gestión de modelos 31
2.2.3 Herramientas para la definición de la sintaxis concreta 33
2.2.4 Ejemplo: creación de un DSL sencillo 35

2.3 Resumen y conclusiones 37

3 TRABAJOS RELACIONADOS 38

3.1 Extensión de meta-modelos 38

3.2 Modelado multi-nivel 46
3.2.1 El espacio de diseño de soluciones de modelado multi-nivel. 47
3.2.2 Clasificación de las herramientas de modelado multi-nivel 49

3.3 Resumen y conclusiones **53**

4 MECANISMO DE CONTROL DE EXTENSIBILIDAD DE META-MODELOS **55**

4.1 Introducción **55**

4.2 Escenarios de aplicación de la personalización de meta-modelos **56**
4.2.1 Escenario A: Extensión de meta-modelos 57
4.2.2 Escenario B: Adaptación de DSLs 58
4.2.3 Escenario C: Modelado multi-nivel y el patrón tipo-objeto 58

4.3 Clasificación de tipos de personalización del meta-modelo **60**

4.4 Mecanismo de control de personalización del meta-modelo **64**

4.5 Herramientas de soporte **71**
4.5.1 Herramienta de diseño de extensibilidad 72
4.5.2 Herramienta de personalización 73

4.6 Evaluación **74**
4.6.1 Aplicabilidad del enfoque 75
4.6.2 Caso de Estudio: Definición de reglas de extensión para DI 79

4.7 Resumen y conclusiones **84**

5 MODELADO MULTI-NIVEL BASADO EN EXTENSIÓN **86**

5.1 Introducción **86**

5.2 Enfoque basado en extensión del meta-modelo para modelado multi-nivel **87**
5.2.1 Anotación de meta-modelos estándar con anotaciones multi-nivel 88
5.2.2 Instanciación de modelos multi-nivel mediante extensión. 91
5.2.3 Compilando el modelo multi-nivel 91
5.2.4 Creando el modelo de nivel 0 94

5.3 Arquitectura y herramientas soporte **95**
5.3.1 Herramienta de anotación multi-nivel 96
5.3.2 Herramienta de modelado multi-nivel 99
5.3.3 Extracción de un meta-modelo estándar 100

5.4 Evaluación **101**
5.4.1 Evaluación empírica 101
5.4.2 Evaluación analítica 107

5.5 Resumen y conclusiones **108**

6 DISCUSIÓN Y COMPARATIVA DE ENFOQUES **109**

6.1 Control de la extensibilidad de meta-modelos **110**

6.1.1 Propiedades del mecanismo de extensibilidad 110
6.1.2 Escenarios de uso 111

6.2 Modelado multi-nivel basado en extensión 111
6.2.1 Propiedades del mecanismo de modelado multi-nivel 111
6.2.2 Escenarios de uso 112

6.3 Comparando la extensión con el modelado multi-nivel 113
6.3.1 Un enfoque multi-nivel para DI 114
6.3.2 Mecanismo basado en extensión para el meta-modelo de componentes 116
6.3.3 Discusión 119

6.4 Resumen y conclusiones 119

7 CONCLUSIONES Y TRABAJOS FUTUROS 121

7.1 Conclusiones 121
7.1.1 Extensibilidad de meta-modelos 121
7.1.2 Modelado multi-nivel 122

7.2 Trabajos futuros 122

REFERENCIAS 124

Lista de Figuras

Figura 2.1: Visión simplificada del proceso de desarrollo de software. 9
Figura 2.2: Visión simplificada del proceso de desarrollo de software usando MDE. 10
Figura 2.3: El modelo en el contexto del MDE. 10
Figura 2.4: Definición de un lenguaje de modelado. 11
Figura 2.5: Esquema de transformación de modelos. 12
Figura 2.6: Arquitectura de meta-modelado de cuatro niveles propuesta por la OMG. 14
Figura 2.7: Instanciación de la arquitectura de cuatro niveles. 15
Figura 2.8: Ejemplo de una restricción OCL. 18
Figura 2.9: Resolviendo el ejemplo usando el patrón tipo-objeto. 22
Figura 2.10: Resolviendo el ejemplo usando el enfoque de extensión de meta-modelo. 23
Figura 2.11 Resolviendo el ejemplo utilizando transformaciones de promoción. 24
Figura 2.12: Resolviendo el ejemplo usando el modelado multi-nivel. 26
Figura 2.13: Arquitectura de Clasificación Ortogonal. 27
Figura 2.14: Editor en forma de árbol generado a partir de un meta-modelo. 29
Figura 2.15: Formatos de representación de modelos en EMF. 30
Figura 2.16: Meta-modelo de Ecore. 31
Figura 2.17: Meta-modelo de máquina de estados. 36
Figura 3.1: Lenguaje de Dominio Personalizado. 39
Figura 3.2: Los cuatro niveles de extensión del meta-modelo. 43
Figura 3.3: Caracterización del espacio de diseño de enfoques de modelado multi-nivel. 47
Figura 4.1: Arquitectura del Diagram Definition [122]. 57
Figura 4.2: Meta-modelo para máquina de estados. 58
Figura 4.3: Un sistema de comercio electrónico. 59
Figura 4.4: Refinamiento del meta-modelo. 60
Figura 4.5: Expansión del meta-modelo. 60
Figura 4.6: Restricción del meta-modelo. 61
Figura 4.7: Contracción del meta-modelo. 62
Figura 4.8: Personalización aditiva conservadora. 62
Figura 4.9: Personalización aditiva de ruptura. 62
Figura 4.10: Personalización restrictiva. 62
Figura 4.11:personalizaciones restrictivas seguras. 63
Figura 4.12: Uso de reglas de personalización. 65
Figura 4.13: Meta-modelo de personalización. 67
Figura 4.14: Ejemplos de redefinición. 68
Figura 4.15: Meta-modelo base para modelado de procesos. 70
Figura 4.16: Arquitectura de personalización. 71
Figura 4.17: Modelo de personalización creado con el editor basado en árbol. 72
Figura 4.18: Interfaz gráfica de usuario (GUI) principal de la herramienta de personalización. 73
Figura 4.19: Ventanas de diálogo adicionales de la herramienta de personalización. 73

Figura 4.20: Vista general del asistente de personalización. 74
Figura 4.21: Extracto del meta-modelo EMF-DEVS, y una extensión típica..... 77
Figura 4.22: Meta-modelo CloudML. 77
Figura 4.23: Extracto del meta-modelo CloudML 78
Figura 4.24: Extracto del meta-modelo Diagram Interchange (DI). 79
Figura 4.25: Herramienta de personalización. 82
Figura 4.26: Extracto del meta-modelo DI extendido para UML. 83
Figura 4.27: Uso del perfil de extensibilidad DI para diagramas tipo grafo. 84
Figura 5.1: Enfoque de extensión del meta-modelo para el modelado multi-nivel. 88
Figura 5.2: Meta-modelo para crear anotaciones multi-nivel. 89
Figura 5.3: Meta-modelo para el ejemplo con anotaciones multi-nivel. 90
Figura 5.4: Instanciación multi-nivel del modelo. 91
Figura 5.5: Compilación de clases con anotaciones de potencia. 92
Figura 5.6: Compilación de atributos. 92
Figura 5.7: Compilación de referencias. 93
Figura 5.8: Compilación del modelo multi-nivel de la Figura 5.4. 94
Figura 5.9: Modelo de nivel 0. 95
Figura 5.10: Arquitectura de TOTEM. 96
Figura 5.11: Herramienta de anotación multi-nivel. 97
Figura 5.12: Uso de la herramienta de modelado multi-nivel. 99
Figura 5.13: Extracción de un meta-modelo estándar. 100
Figura 5.14: Evaluación empírica. 102
Figura 5.15: Modelo ejemplo usando Melanee. 104
Figura 5.16: Propiedades de los elementos del meta-modelo en Melanee. 105
Figura 6.1: Meta-modelo ProductType. 112
Figura 6.2: Meta-modelo Diagram Interchange (DI). 114
Figura 6.3: Reglas de extensibilidad de TACO aplicadas al meta-modelo DI. 114
Figura 6.4: Proceso de extensibilidad de DI a través de TACO. 115
Figura 6.5: Anotaciones multi-nivel de TOTEM aplicadas al meta-modelo DI. 115
Figura 6.6: Proceso de modelado multi-nivel de DI a través de TOTEM. 116
Figura 6.7: Meta-modelo Component con anotaciones multi-nivel. 117
Figura 6.8: Anotaciones multi-nivel de TOTEM aplicadas al meta-modelo Component. 117
Figura 6.9: Instanciación del meta-modelo Component. 117
Figura 6.10: Reglas de extensibilidad aplicadas al meta-modelo Component. 118
Figura 6.11: Proceso de extensibilidad de Component a través de TACO. 118

Lista de Tablas

Tabla 2.1: Ventajas y desventajas de enfoques que emulan modelado multi-nivel..... 25
Tabla 3.1: Enfoques de extensibilidad de lenguajes de modelado/meta-modelado. 46
Tabla 3.2: Clasificación de herramientas de modelado multi-nivel. 50
Tabla 4.1: Compatibilidad de los tipos de modificación del meta-modelo..... 61
Tabla 4.2: Reglas de personalización soportadas..... 65
Tabla 4.3: Meta-modelos extensibles. 76
Tabla 4.4: Reglas de extensión para DI. 80
Tabla 4.5: Perfil de extensibilidad *Diagrama de gráfico simple* para DI. 83
Tabla 5.1: Especificación de los valores de las anotaciones. 98
Tabla 5.2: Comparación de las tres herramientas, en base al ejemplo. 106
Tabla 5.3: Características de TOTEM..... 107

Listados

Listado 4.1: Modelo de personalización creado con el DSL textual..... 72
Listado 4.2: Reglas de extensión utilizando el DSL de extensibilidad.. 81
Listado 5.1: Anotaciones multi-nivel utilizando el DSL textual. 97
Listado 5.2: Ejemplo en MetaDepth..... 102

Glosario

API	Application Programming Interface
ATL	Atlas Transformation Language
BPMN	Business Process Management Notation
DCL	Domain Customized Language
DD	Diagram Definition
DFD	Data Flow Diagram
DSL	Domain Specific Language
EBNF	Extended Backus Naur Form
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
GPML	General Purpose Modeling Languages
IFML	Interaction Flow Modeling Language
KDM	Knowledge Discovery Metamodel
M2M	Model to Model
M2T	Model to Text
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query/View/Transformations
SPEM	Software Process Engineering Metamodel
SuS	System under Study
TACO	Tool for Meta-model Customization
TOTEM	Tool for Multi-level Modeling
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Introducción

El objetivo de este capítulo es presentar la motivación que ha impulsado la realización de este trabajo. Para ello se identifican los problemas detectados en el ámbito de la extensión y personalización de meta-modelos en la Ingeniería Dirigida por Modelos (MDE, del inglés Model-Driven Engineering). Como contribuciones proporcionadas por la tesis, se señalan las soluciones propuestas para cada problema. También se listan las publicaciones en revistas y congresos del área. Finalmente se incluye una breve descripción del contenido de los restantes capítulos.

1.1 Motivación

El MDE es un paradigma de desarrollo de software que conecta estrechamente el modelo a la aplicación [24, 66]. De esta manera, el modelo no sólo encapsula el diseño de la aplicación, sino que es utilizado para especificar, simular, probar, verificar y generar código para la aplicación que se construirá [109]. Un modelo es una descripción de un sistema, o parte de éste, escrito en un lenguaje bien definido, que se construye con un propósito determinado, como simular un sistema existente, o diseñar un sistema que se ha de construir [160]. Un modelo se puede construir mediante lenguajes de modelado de propósito general, como por ejemplo el Unified Modeling Language (UML) [137], o utilizando un lenguaje especializado, como son los Lenguajes de Dominio Específico (DSLs, del inglés Domain-Specific Languages). Los DSLs utilizan conceptos y una notación concreta para un dominio particular, más cercanos al problema a resolver [113].

En MDE, los lenguajes de modelado, tanto de propósito general como de dominio específico, se especifican a través de su sintaxis abstracta, sintaxis concreta y semántica [66, 159]. La sintaxis abstracta está definida por un meta-modelo, que describe los conceptos, propiedades, relaciones y restricciones relevantes de un determinado dominio. La sintaxis concreta está dada por la notación textual y/o gráfica utilizada para representar los elementos del meta-modelo. Esta es la notación que utilizará el usuario del lenguaje para diseñar modelos. Un modelo es una instancia de su meta-modelo [19]. Por tanto, un meta-modelo determina un conjunto posiblemente infinito de modelos válidos. La semántica del DSL está dada por el significado que expresa el modelo con relación al sistema modelado (sistema real). Un buen DSL es el que permite especificar modelos cuyo grado de abstracción es tan cercano al problema como sea posible, y a su vez facilita el uso de los modelos para el propósito inicial (simulación, generación de código, etc.).

Tomando en consideración que los meta-modelos son la piedra angular en el desarrollo de software en MDE [27] y que el software presenta una evolución permanente debido a que debe atender las nuevas exigencias del medio [90], se presenta la necesidad de contar con plataformas de trabajo (frameworks) que proporcionen soporte metodológico a la evolución de los meta-modelos. Esta evolución permite desarrollar software que se puede adaptar a nuevos dominios de aplicación [27], reutilizando los meta-modelos existentes.

Actualmente, existen varios esfuerzos independientes que abordan la problemática de la evolución del meta-modelo. Algunos autores se concentran en la co-evolución de los artefactos MDE (modelos, editores, transformaciones, restricciones de integridad, generadores de código) cuando el meta-modelo cambia [74, 93, 48, 49, 50]. Sin embargo, el interés de esta tesis se centra en la propuesta de técnicas para la adaptación controlada de meta-modelos para que se ajusten a un dominio más especializado. A nivel de literatura, se han propuesto diversos términos para referirse a la adaptación de meta-modelos, tales como *modificación, extensión*, *personalización*, *mutación* o *especialización*. Braun y colaboradores [27] señalan que una extensión mejora la expresividad de un lenguaje de modelado conceptual, introduciendo nuevos constructos y propiedades o refinando los elementos existentes para representar conceptos para un propósito específico. A pesar de la importancia que tiene la personalización de meta-modelos en el entorno de MDE, aún no se han desarrollado estándares para esta tarea [29].

En el presente trabajo se proponen dos técnicas para especificar la personalización de meta-modelos, así como el diseño e implementación de los mecanismos para guiar al desarrollador a realizar dicha personalización. A continuación, se resumen ambas técnicas, identificando las carencias actuales que estas resuelven.

1.1.1 Mecanismo de control de la extensibilidad de meta-modelos

Si bien los meta-modelos tienden a ser fijos, existen situaciones en las que estos son diseñados para ser personalizados o especializados, es decir los meta-modelos pueden ser reutilizados para generar sistemas más complejos por subclasificación y redefinición, de forma similar a los *Object-*

Oriented Application Frameworks [60]. Existen varios escenarios que requieren la personalización de los meta-modelos existentes. Por ejemplo, los estándares del Object Management Group (OMG) [132] como el Knowledge Discovery Metamodel (KDM) [126] o el Diagram Definition (DD) [124] se basan en la extensión de los meta-modelos base de acuerdo con ciertas reglas. Sin embargo, estas reglas no son operativas, ya que se describen en lenguaje natural, y por tanto no son compatibles con las herramientas automatizadas. Aunque el modelado es una actividad regulada por meta-modelos, actualmente no existen mecanismos comúnmente aceptados para regular la forma en que se pueden extender los meta-modelos.

Los problemas detectados en este escenario son los siguientes:

- Falta de mecanismos estándar para regular la forma en que meta-modelos existentes se pueden extender.
- Existen meta-modelos que han sido diseñados para ser extendidos para ajustarse a un dominio más específico, sin embargo, las reglas de extensibilidad son expresadas en lenguaje natural, provocando que dichas reglas puedan ser difíciles de entender, puedan ser mal interpretadas o puedan llevar a implementaciones erróneas por el diseñador de la extensión.
- Hay la falta de mecanismos y herramientas automatizadas que guíen el proceso de extensión.

1.1.2 Refinamiento de meta-modelos mediante modelado multi-nivel

Otra manera de personalizar un meta-modelo es mediante su instanciación, como propone el modelado multi-nivel [8, 46]. En el modelado tradicional, el ingeniero trabaja con dos meta-niveles: meta-modelos y modelos. Este es el enfoque considerado por los estándares de meta-modelado actual, como el Meta-Object Facility (MOF) [129] de la OMG, o sus implementaciones prácticas, como el Eclipse Modeling Framework (EMF) [156]. El modelado multi-nivel es una extensión conservadora del enfoque de dos niveles, que admite el uso de un número arbitrario de meta-niveles para describir los sistemas [8]. El modelado multi-nivel permite obtener modelos más simples (con menos elementos) con menor complejidad accidental [11], especialmente cuando surge el *patrón tipo-objeto* (type-object pattern) [46, 105].

Por lo tanto, para simplificar el modelado en estos escenarios, recientemente han surgido varios enfoques y herramientas de modelado multi-nivel para aumentar la flexibilidad en el modelado. Entre las principales herramientas se puede mencionar a DeepJava [98], DeepRuby [120], DPF Workbench [100], Melanee [4], MetaDepth [40], ML2 [31], y MultEcore [105]. Si bien estos enfoques proporcionan primitivas avanzadas para simplificar el modelado, se observa la posibilidad de mejorar la interoperabilidad con los enfoques convencionales de modelado estándar de dos niveles.

El problema detectado en este escenario se puede resumir como:

- Incompatibilidad de los enfoques multi-nivel con herramientas y frameworks de modelado a dos niveles.

1.2 Contribuciones

Este trabajo plantea la solución a los problemas de cada escenario planteado anteriormente, como se resume a continuación en las dos siguientes sub-secciones.

1.2.1 Extensión de meta-modelos

Se propone un mecanismo que permite controlar la forma en que se pueden extender o personalizar meta-modelos. El mecanismo permite especificar un conjunto de reglas que hacen referencia a cada uno de los elementos del meta-modelo a personalizar. Cada una de las reglas permiten ejecutar operaciones ya sea de extensión, modificación o reducción, para obtener un meta-modelo que se ajuste a un dominio más específico. Estas reglas constituyen una especificación precisa de las extensiones y adaptaciones permitidas. Así, estas permiten guiar al desarrollador en la extensión, y comprobar que las extensiones se ajustan a la especificación.

Para el efecto se ha desarrollado una herramienta llamada TACO (disponible en http://miso.es/tools/taco.html), que permite especificar reglas de extensión, y personalizar los meta-modelos de acuerdo a dichas reglas. La herramienta se basa en EMF, se ha implementado como un plug-in de Eclipse y se ha validado para orientar la personalización de meta-modelos, incluidos los meta-modelos estándar de la OMG, como KDM y DD.

1.2.2 Modelado multi-nivel

Tras un análisis del espacio de diseño de los enfoques multi-nivel, y una revisión sistemática de las herramientas existentes hoy en día, se propone un enfoque novedoso para el modelado multi-nivel basado en extensión y el uso de anotaciones. El enfoque permite emular el modelado multi-nivel dentro de un meta-modelo estándar de dos niveles, utilizando EMF. Al utilizar EMF es posible aprovechar el gran conjunto de herramientas maduras de MDE, que permiten por ejemplo crear editores, realizar transformaciones de modelos y construir generadores de código.

Para validar el mecanismo propuesto se ha desarrollado una herramienta llamada TOTEM (disponible en http://miso.es/tools/totem.html), como un plug-in de Eclipse, que implementa los conceptos planteados. Las capacidades de la herramienta y sus beneficios en términos de interoperabilidad con los frameworks estándar de modelado se ilustran a través de varios casos de estudio.

1.3 Publicaciones

Esta disertación ha dado lugar a dos publicaciones en revistas internacionales con índice de impacto (una de ellas en revisión), cinco publicaciones en conferencias internacionales, y cuatro participaciones en conferencias nacionales en Ecuador. A continuación, se detalla cada una de ellas.

1.3.1 Revistas internacionales con factor de impacto

1. "Controlling Meta-Model Extensibility in Model-Driven Engineering". Santiago Jácome y Juan de Lara. IEEE Access. Vol. 6, March 2018. Pag. 19923-19939. DOI: 10.1109/ACCESS.2018.2821111. JCR 2017: 3.557 (Q1, Information Systems).
2. "TOTEM: Reconciling multi-level modelling with standard two-level modelling". Santiago Jácome y Juan de Lara. Elsevier, Software Engineering. Computers Standards & Interfaces. JCR 2017: 1.465 (Q2, Software Engineering). En segunda ronda de revisión.

1.3.2 Conferencias Internacionales

1. "Construcción y adaptación de Lenguajes de Dominio Específico por usuarios finales". Santiago Jácome. Proc. IberoAmerican Conference on Software Engineering (CIBSE-2015) (Doctoral symposyum). ISBN: 978-1-5108-0387-9. Lima-Perú.
2. "Un enfoque para controlar la extensión de meta-modelos en el ámbito de la Ingeniería dirigida por Modelos". Santiago Jácome y Juan de Lara. Proc. IberoAmerican Conference on Software Engineering (CIBSE-2017). ISBN: 978-1-5108-5098-9. Páginas 29-42. Buenos Aires-Argentina. (porcentaje de aceptación, 36%).
3. "Towards a mechanism for controlling meta-model extensibility". Santiago Jácome y Juan de Lara. Proc. International Conference on Software Technologies (ICSOFT-2017). Conferencia "B" en el ranking CORE. Vol 1. Pag. 382-387. DOI: 10.5220/ 0006462603820387. Madrid-España.
4. "Proposal of a mechanism of customization of meta-models Ecore". Santiago Jácome. Proc. Jornadas Iberoamericanas de Ingeniería de Software e Ingeniería del Conocimiento (JIISIC-2017). ISBN: 978-1-5108-4396-7. Pag. 143-154. Latacunga-Ecuador.
5. "Software Development Tools in Model-Driven Engineering". Santiago Jácome, Marcelo Ferreira y Alexandra Corral. Proc. International Conference on Research and Innovation in Software Engineering Software (CONISOFT-2017). IEEE. Pag. 140-148. DOI: 10.1109/CONISOFT.2017.00024. Mérida-Yucatán-México.

1.3.3 Conferencias Nacionales (Ecuador)

1. "Ingeniería Dirigida por Modelos - MDE". Santiago Jácome. Expositor en el Seminario Internacional en Desarrollo e Innovaciones Tecnológicas en Automatización, Robótica, Software y Eléctrica en la Universidad de las Fuerzas Armadas ESPE. Latacunga-Ecuador. Noviembre de 2013.
2. "Ingeniería Dirigida por Modelos". Santiago Jácome. Expositor en Workshop de Ingeniería de Software en la Universidad de las Fuerzas Armadas ESPE. Latacunga-Ecuador. Junio 2016.
3. "Software development environments and tools in MDE". Santiago Jácome. Expositor en Simposio Iberoamericano en Programación Informática (SIIPRIN 2017). Escuela Politécnica Superior del Chimborazo. Riobamba-Ecuador. Noviembre 2017.
4. "Desarrollo de software en Ingeniería Dirigida por Modelos". Santiago Jácome. Expositor en International Conference on Information System and Computer Science (INCISCOS 2017). Universidad Tecnológica Equinoccial. Quito-Ecuador. Noviembre 2017.

1.4 Organización

El resto del documento se organiza de la siguiente manera:

- Capítulo 2: Se hace una revisión de los conceptos previos, incluyendo fundamentos de MDE, de meta-modelado y modelado multi-nivel, y tecnologías de soporte.
- Capítulo 3: Se realiza una revisión de los trabajos relacionados con la extensibilidad de meta-modelos y modelado multi-nivel.
- Capítulo 4: Se describe el mecanismo de control de extensibilidad de meta-modelos propuesto.
- Capítulo 5: Se describe el enfoque de modelado multi-nivel mediante extensión de meta-modelos planteado en esta tesis.
- Capítulo 6: Se discuten los resultados de la investigación, identificando los escenarios de uso de cada enfoque propuesto, comparando ambos, y señalando las ventajas y desventajas de cada uno.
- Capítulo 7: Se finaliza el documento proporcionado algunas conclusiones relacionadas a la presente investigación. También se dentifican líneas abiertas para el trabajo futuro.

1.5 Agradecimientos

Este trabajo fue posible gracias a la beca de estudios proporcionada por la Universidad de las Fuerzas Armadas ESPE (Ecuador) para cursar el Programa de Doctorado en Informática y

Telecomunicación en la Universidad Autónoma de Madrid (España). La beca permitió realizar en primer lugar la maestría "Investigación e innovación en tecnologías de la Información y las Comunicaciones" de forma presencial en España durante un año. Posteriormente, la beca permitió realizar tres estancias de investigación de tres meses cada una, durante tres años en España para desarrollar el presente trabajo de investigación. El resto del trabajo de investigación fue completado de forma remota con el apoyo decidido del Dr. Juan de Lara Jaramillo.

2 Conceptos previos

El objetivo de este capítulo es revisar los pilares en los que sustenta la ingeniería dirigida por modelos, tales como modelos, transformaciones entre modelos, lenguajes de modelado, y las tecnologías que los soportan.

2.1 Fundamentos de MDE

En esta sección se describe el rol de los modelos en el proceso de desarrollo de software en MDE, se revisan también conceptos básicos relacionados a modelos, meta-modelado y modelado multi-nivel.

2.1.1 Conceptos básicos

El proceso de desarrollo de software aborda dos perspectivas: 1) la perspectiva que cubre los aspectos relacionados con el dominio de la aplicación (dominio del problema), vinculada a los aspectos que le interesan al usuario final del sistema, y 2) la perspectiva relacionada con la tecnología software utilizada para implementar el sistema (dominio de la solución) [142]. Se considera por lo general que el dominio del problema no tiene nada que ver con la tecnología software. Por tanto, el desarrollo de software básicamente consiste en la capacidad de representar de forma adecuada el dominio del problema, para posteriormente trasformar esta representación en la implementación del código en una determinada tecnología, es decir, en el dominio de la solución [142].

De manera simplificada, el proceso de desarrollo de software puede verse como una transformación hacia la solución técnica de determinada problemática u oportunidad con la finalidad de resolverla [33], como se muestra en la Figura 2.1.

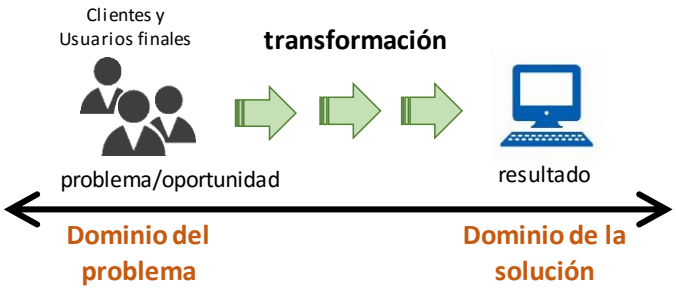


Figura 2.1: Visión simplificada del proceso de desarrollo de software.

El enfoque dirigido por modelos (model-driven) ha recibido una considerable atención desde la anterior década. La Arquitectura Dirigida por Modelos (MDA, del inglés Model-Driven Architecture) de la OMG, el Desarrollo Dirigido por Modelos (MDD, del inglés Model-Driven Development) o la Ingeniería Dirigida por Modelos (MDE) han sido proclamados como la solución para manejar la complejidad a la que se enfrenta la industria de desarrollo de software al proporcionar mejores técnicas de abstracción y facilitar la automatización [152, 110, 151].

El criterio fundamental de MDE es la utilización de la *abstracción***,** que permite construir modelos que representan el sistema a desarrollar. En el contexto de MDE el término abstracción especifica la medida en que un modelo o lenguaje de modelado está orientado a la tecnología de la solución [12]. Bajo nivel de abstracción, significa que está orientado a la solución utilizando tecnologías relacionadas a la implementación de la aplicación. En contra, alto nivel de abstracción significa que está orientado al problema, abordándose el aspecto de *"qué"* es hecho por el sistema, es decir que se centra en la especificación funcional del sistema, sin preocuparse por el aspecto de *"cómo"* se realizará.

Los beneficios del MDE incluyen el de proporcionar una mejor comunicación entre las partes interesadas (stakeholders), proporcionar trazabilidad entre artefactos, aumentar la portabilidad de las soluciones a diferentes plataformas, reducir los errores y el costoso trabajo manual, una mayor productividad al automatizar varias actividades, y mejorar la calidad del software [114, 155]. La literatura reporta mejoras significativas en la productividad al utilizar MDE [89, 157], del orden del 20-30% [161].

En MDE los modelos se utilizan para razonar sobre el dominio de problema y diseñar un resultado en el dominio de la solución. Un enfoque MDE debe proveer mecanismos, técnicas y herramientas para convertir los modelos y transformaciones entre modelos en los artefactos esenciales de todas las fases del proceso de desarrollo de software: análisis, diseño, implementación, despliegue y mantenimiento. De esta manera, se podría definir el MDE como un enfoque para el desarrollo de software que se fundamenta en la separación conceptual entre la especificación funcional de un sistema y su implementación, en el que cobran vital importancia los modelos y las transformaciones entre ellos, como se muestra en la Figura 2.2.

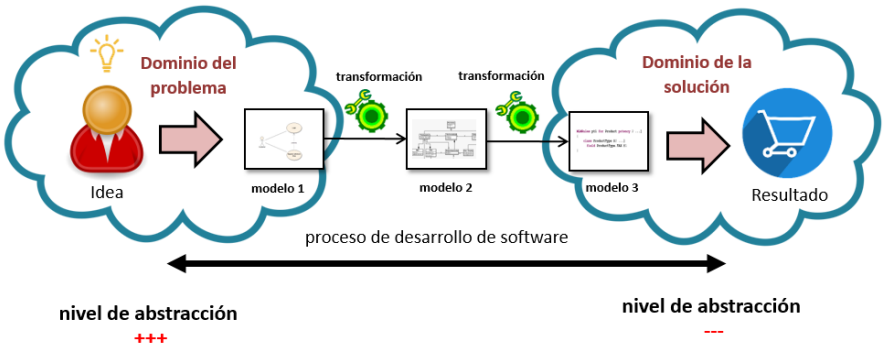


Figura 2.2: Visión simplificada del proceso de desarrollo de software usando MDE.

Bézivin señala que el MDE considera el uso sistemático de modelos en la cadena de producción de software [19]. Un modelo es una descripción de un sistema o parte de éste, escrito en un lenguaje de modelado con sintaxis y semántica precisas, pudiendo ser interpretado automáticamente por un ordenador [95]. MDE considera que se debe mover el foco de desarrollo de software desde el código hacia los modelos, hasta el punto de construir modelos que puedan ser directamente compilados, transformados y ejecutados [95, 110].

Muy frecuentemente, el MDE utiliza DSLs para la creación de modelos. Un DSL es un lenguaje especializado en modelar o resolver un conjunto específico de problemas. Es decir, un DSL se ajusta a una tarea particular en el dominio de la aplicación. La Figura 2.3 muestra el modelo en el contexto del sistema y de un DSL. En contraposición a los DSLs se encuentran los Lenguajes de Modelado de Propósito General (GPML, del inglés General Purpose Modeling Languages), como por ejemplo UML o SysML [88], los cuales están diseñados para representar un rango muy amplio de problemas en distintos dominios [137, 88].

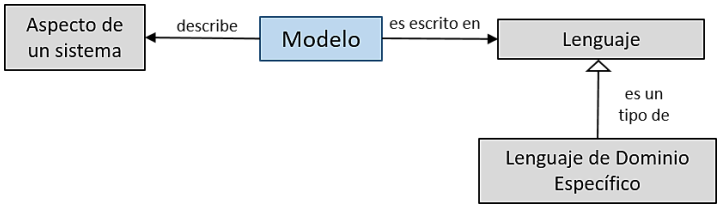


Figura 2.3: El modelo en el contexto del MDE.

La definición formal de un lenguaje de modelado consiste en la descripción de su sintaxis y su semántica [32]. La sintaxis engloba tanto la sintaxis abstracta como la sintaxis concreta [66]. La sintaxis abstracta define los conceptos del lenguaje y las relaciones entre ellos, así como las reglas que establecen cuando un modelo está bien formado. La sintaxis abstracta se describe a través de un meta-modelo, el cual es a su vez un modelo que contiene todos los elementos que permiten representar el dominio de estudio. La sintaxis concreta mapea cada uno de los elementos de la sintaxis abstracta (meta-modelo) a una representación textual o gráfica, que describe cómo se representarán los modelos. La representación textual se realiza a través de un *Lenguaje de Modelado Textual*, como por ejemplo SQL, mientras que la representación gráfica se realiza a través de un *Lenguaje de Modelado Gráfico* como los Diagramas de Flujo de Datos (DFDs). La semántica describe el significado

de cada uno de los elementos del modelo, y a su vez el modelo referencia al sistema o una parte de éste [66].

Al hablar de semántica, para ser más precisos se debe hacer referencia al *dominio semántico* y al *mapeo semántico*. El dominio semántico es típicamente un formalismo matemático, que debe ser bien conocido o entendido, y se refiere al entendimiento pleno de los elementos del domino del problema del sistema real. El mapeo semántico relaciona los elementos de la sintaxis abstracta a los elementos del dominio semántico [32]. La Figura 2.4 muestra los elementos que especifican la definición de un DSL.

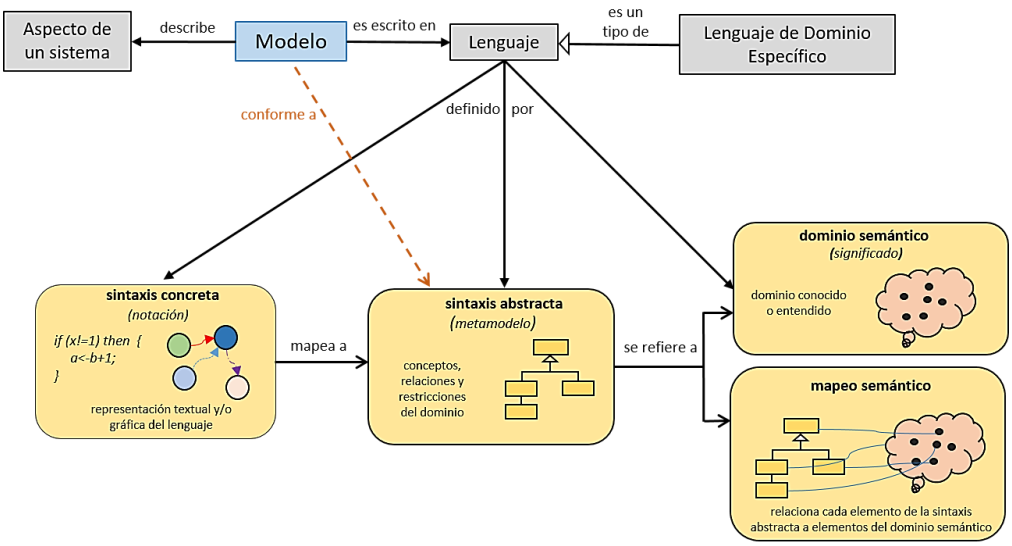


Figura 2.4: Definición de un lenguaje de modelado.

Es de resaltar que el DSL permite desarrollar aplicaciones utilizando elementos del sistema capturados por su meta-modelo, expresando el diseño de la aplicación en forma declarativa, donde se describe el problema sin detallar su solución; en lugar de forma imperativa, donde se deben señalar las instrucciones para realizar una tarea [151]. Normalmente el meta-modelo es expresado en notaciones similares a diagramas de clases utilizando un lenguaje de meta-modelado como MOF [129] o Ecore [52].

Los modelos son valiosos como herramientas para la abstracción, para resumir y para proporcionar perspectivas alternativas [92]. En un proyecto, puede ser necesario el uso de distintos tipos de modelos, cada uno con un fin específico. Como se señaló en la Figura 2.2, para pasar del dominio del problema (alto nivel de abstracción) al dominio de la solución (bajo nivel de abstracción), se requiere de un proceso automatizado de transformación de modelos. Para este caso, MDE considera principalmente dos tipos de transformación, una que transforma un modelo en otro (M2M), y otro tipo que transforma un modelo a texto (M2T). En este último caso, el texto puede representar varios artefactos, como documentación, archivos de configuración, páginas HTML o el código de un lenguaje de programación para que sea ejecutado en una determinada plataforma tecnológica.

Una solución MDE típicamente contiene una especificación de los lenguajes de modelado, las transformaciones entre los modelos, y el proceso utilizado para coordinar la construcción y evolución de los modelos.

Para garantizar los beneficios de la utilización de modelos para desarrollar software, no sólo se requieren herramientas básicas que únicamente permitan editar modelos, sino que se necesita el apoyo de herramientas sofisticadas [92], que permitan manejar de forma automatizada, por ejemplo el proceso de transformación, simulación, verificación y personalización de modelos. La Figura 2.5, muestra los principales elementos que conforman la tecnología MDE.

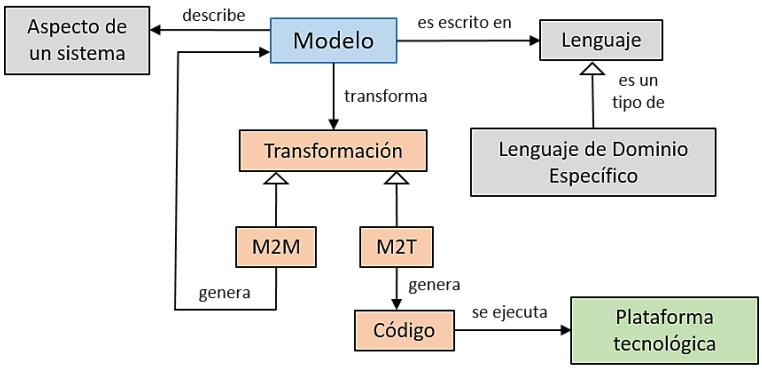


Figura 2.5: Esquema de transformación de modelos.

2.1.2 Modelos y meta-modelos

Los modelos proporcionan abstracciones de un sistema físico (o lógico) que permite a los ingenieros razonar acerca de ese sistema, ignorando detalles accidentales, concentrándose en los relevantes. Todas las formas de ingeniería usan los modelos como elementos esenciales para comprender sistemas complejos del mundo real. Los modelos se utilizan de muchas maneras, como por ejemplo para predecir las cualidades del sistema, para razonar sobre propiedades específicas cuando varios aspectos del sistema cambian, o para poder comunicar las características clave del sistema a las diferentes partes interesadas [12]. Los modelos pueden ser desarrollados como precursores, que guíen la implementación del sistema físico a construir o pueden derivarse de un sistema existente como una ayuda para entender su comportamiento.

Se podría definir un modelo como una abstracción de un sistema o concepto del mundo real [147], que contiene un menor nivel de detalle que éste. En la literatura, se pueden encontrar algunas definiciones de modelo, entre ellas:

- "Un modelo es la descripción o especificación de un sistema y su entorno definido, para cierto propósito. Un modelo es representado habitualmente como una combinación de gráficos y texto" [116].
- "Un modelo es una representación de una parte de la funcionalidad, estructura y/o comportamiento de un sistema" [132].

Según Warner y Kleppe [160], cada modelo describe un único aspecto particular del sistema, para un propósito específico, descrito con el nivel de abstracción adecuado para el problema modelado. La descripción del modelo puede ser gráfica y/o textual [116]. Para crear los modelos se utilizan lenguajes de modelado con un mayor nivel de abstracción que los lenguajes de programación habituales. Esto quiere decir que los lenguajes manejan conceptos más cercanos al dominio de la aplicación, dando la posibilidad de esta manera a que personas sin formación informática puedan desarrollar aplicaciones utilizando las herramientas adecuadas.

En ocasiones se utiliza el término "*modelo*" y "*diagrama*" de modo intercambiable, pero no son lo mismo. Un modelo no es meramente un diagrama o una colección de ellos. Un diagrama es una representación gráfica de un modelo, mientras que un modelo puede ser representado de forma gráfica, por ejemplo, como un diagrama de clases de UML, o de forma textual. En otras palabras, el modelo es independiente de su representación gráfica o textual [66].

Es importante señalar que de un mismo sistema se pueden tener modelos alternativos y complementarios, en función de la información relevante que se quiera resaltar. Por ejemplo, del cuerpo humano se pueden tener diferentes modelos para representar sus diferentes subsistemas, tales como digestivo, respiratorio o circulatorio. De un automóvil, además del sistema de transmisión, son importantes los modelos de los subsistemas de amortiguación, eléctrico y electrónico.

Para ser útiles en ingeniería, los modelos deben proporcionar puntos de vista separados. En cada punto de vista es pertinente determinar qué información es relevante para satisfacer su propósito. El modelo pierde utilidad como representación simplificada si se incluye en él toda la información del sistema. De acuerdo a Selic [152], el modelo debe ser:

- ***Adecuado:*** construido con un propósito establecido, desde de un punto de vista determinado y dirigido para ciertos usuarios.
- ***Abstracto:*** debe ser una versión reducida del sistema que representa, resaltando los aspectos más relevantes para su propósito a la vez que ocultan los aspectos irrelevantes.
- ***Comprensible:*** debe ser expresado de tal forma que pueda entenderse fácilmente por sus usuarios.
- ***Preciso:*** ha de representar fielmente al objeto o sistema modelado.
- ***Predictivo***: se puede utilizar para responder preguntas sobre el sistema e inferir conclusiones correctas.
- ***Bajo costo:*** debe ser más fácil y económico de construir y estudiar que el propio sistema.

En MDE los modelos son generalmente representados a través de DSLs y su meta-modelo especifica los conceptos del lenguaje, las relaciones entre ellos y las reglas estructurales que restringen los posibles elementos del modelo, así como aquellas combinaciones entre elementos que respetan las reglas semánticas del dominio. De esta manera, cada modelo se representa en el lenguaje que define su meta-modelo, quedando establecida la relación entre modelo y su meta-modelo por una relación de dependencia conocida con el nombre "conforme a" [85]. Los meta-modelos son a su

vez modelos, por lo tanto, estarán igualmente escritos en un lenguaje definido por un meta-modelo. El procedimiento recursivo de definir modelos conforme a otros modelos de mayor grado de abstracción termina cuando se alcanza el nivel de meta-meta-modelo, debido a que los meta-meta-modelos pueden considerarse conformes a ellos mismos.

Para poder comprender la relación entre modelos y meta-modelos se utiliza la arquitectura de cuatro niveles, propuesta por la OMG [128].

- ***M0 - Nivel de objeto:*** caracteriza los objetos del mundo real representados por el modelo.
- ***M1 - Nivel de modelo:*** este nivel incluye los modelos, que representan los datos y objetos del nivel M0.
- ***M2 - Nivel del meta-modelo:*** caracteriza los meta-modelos que describen los modelos del nivel M1.
- ***M3 - Nivel de meta-meta-modelo:*** caracteriza los meta-meta-modelos que describen los meta-modelos del nivel M2.

Para poder comprender la relación entre niveles de la arquitectura, se utiliza un paralelismo con los lenguajes de programación y sus gramáticas. Por ejemplo, la ejecución de un programa escrito en Java (proceso del mundo real), es determinada por un programa que obedece a una gramática del lenguaje, que a su vez es definida con un meta-lenguaje como EBNF[1]. En este caso, el nivel M0 corresponde a la ejecución del programa, M1 a los programas, M2 a la gramática del lenguaje, y M3 a los meta-lenguajes como EBNF, como se ilustra en la Figura 2.6 [66].

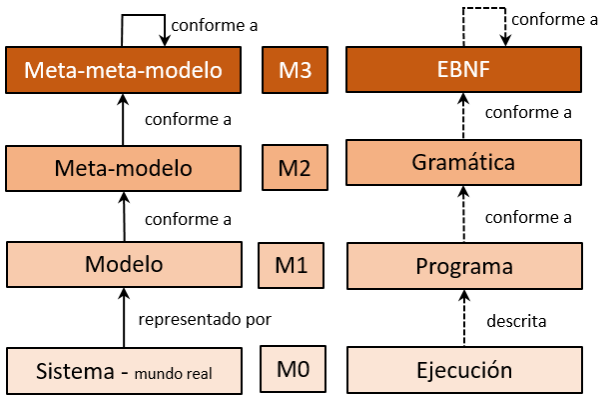


Figura 2.6: Arquitectura de meta-modelado de cuatro niveles propuesta por la OMG.

Tomando como referencia a MOF como lenguaje de meta-modelado, y SPEM (del inglés, Software Process Engineering Metamodel) [16], UML y CWM (del inglés, Common Warehouse Metamodel) [107] como lenguajes de modelado utilizando las facilidades que ofrece MOF, la Figura 2.7 representa una instanciación de la arquitectura de cuatro niveles representada en la Figura 2.6.

[1] Extended Backus–Naur Form es un meta-lenguaje usada para expresar gramáticas libres de contexto.

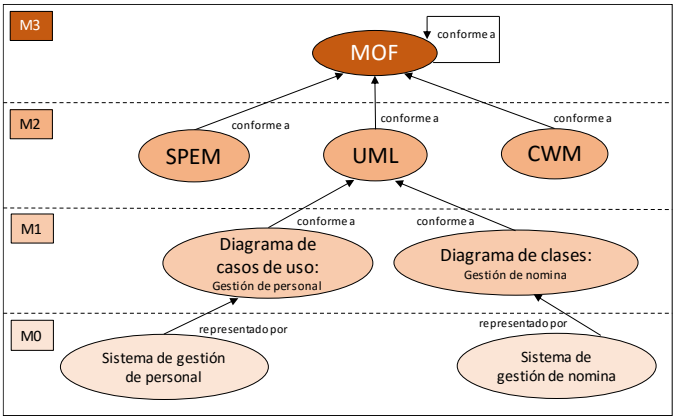


Figura 2.7: Instanciación de la arquitectura de cuatro niveles.

La Figura 2.7 muestra en el nivel M0 dos sistemas que son representados desde diferente perspectiva, el *sistema de gestión de personal* es representado en el nivel M1 por un diagrama de casos de uso, en tanto que el *sistema de gestión de nómina* es representado por un diagrama de clases. Los dos diagramas corresponden a dos de los varios diagramas UML para representar la arquitectura de un sistema. A su vez UML, así como SPEM y CWM del nivel M2 son definidos en base al lenguaje de meta-modelado MOF del nivel M3.

2.1.3 Lenguajes de modelado y perfiles UML

Una de las técnicas utilizadas por la ingeniería de software para el desarrollo de aplicaciones es el "*modelado*", que se remonta a los primeros días de la programación [151]. Las innovaciones más recientes se han centrado en las notaciones y herramientas que permiten a los desarrolladores representar una determinada realidad (cierta perspectiva del sistema), de vital valor sobre todo para los arquitectos y desarrolladores de software. La realidad representada en un modelo se denomina a veces *"sistema bajo estudio"* (SuS, del inglés System under Study). Un sistema bajo estudio puede ser cualquier tipo de sistema, no necesariamente un sistema de software, puede ser un sistema mecánico, vial o de fabricación. Un modelo es una abstracción o simplificación de la realidad. Esta simplificación suprime detalles irrelevantes y retiene aspectos esenciales, siendo utilizada para comprender mejor la realidad que representa. De esta manera, la esencia del sistema se podrá comprender mejor, para poder hacer frente a su complejidad.

La forma más intuitiva y tradicional de representar los modelos es a través del uso de diagramas o representaciones gráficas. Un diagrama puede ser diseñado rápidamente mediante un bosquejo hecho a mano o utilizando sofisticadas herramientas de diagramación [151]. Sin embargo, si no se tiene el suficiente cuidado, estas representaciones pueden ser bastante informales, lo que limitaría su aplicabilidad e interpretación. Estas imprecisiones y ambigüedad pueden provocar que la representación no pueda ser analizada adecuadamente, por lo que no sería posible generar ningún tipo de código para el sistema modelado, o analizar de forma precisa la semántica del modelo.

No obstante, existen notaciones gráficas definidas de forma más precisa, a través de la definición formal de un lenguaje de modelado, como se ha visto en la sección 2.1.1. A continuación, se señalan las características que debe tener un lenguaje de modelado [142]:

- El propósito del lenguaje de modelado tiene que ser claro y estar explícitamente especificado.
- Los conceptos que ofrece el lenguaje deben corresponderse con los del dominio, para poder ser fácilmente reconocidos por los usuarios.
- La notación tanto gráfica como textual debe ser sencilla.
- El lenguaje debe definir un conjunto de reglas que hagan posible el uso de herramientas que validen los modelos y ayuden al usuario en su proceso de construcción.

La semántica del lenguaje debe ser precisa y no ambigua, de tal manera que los usuarios puedan diseñar modelos que otros usuarios o herramientas sean capaces de entender, validar y procesar.

UML es considerado un *Lenguaje de Modelado de Propósito General*. Sin duda es el lenguaje de modelado más utilizado, de los definidos por la OMG, debido a que puede utilizarse para especificar la mayoría de los sistemas basados en objetos o en componentes y modelar aplicaciones de muy diversos dominios de aplicación (financiero, comercial, telecomunicaciones, etc.) en diferentes plataformas tecnológicas (como .NET, CORBA, J2EE) [65].

Al ser de propósito general, UML proporciona una gran flexibilidad y expresividad a la hora de modelar una gran variedad sistemas, pero no es un lenguaje universal, puede haber otras notaciones más apropiadas para modelar cierto tipo de sistemas. Sin embargo, UML es un *lenguaje extensible*, debido que es posible añadir nuevos elementos para abordar aspectos del desarrollo de software no previstos inicialmente en su notación [65]. Para ello UML incluye un mecanismo de extensibilidad en el propio lenguaje, permitiendo definir lenguajes de dominio específico, derivados de UML [153]. Sin embargo, se debe señalar que el mecanismo de extensibilidad propuesto por UML no permite especializar los conceptos propios de su meta-modelo, ya que esto provocaría que la notación extendida no fuese compatible con las herramientas existentes.

La extensibilidad en UML es posible mediante la utilización de los denominados *"perfiles"* (profiles) definidos por el usuario, los cuales no permiten modificar la sintaxis ni semántica original de UML [28]. Los perfiles UML únicamente permiten realizar extensiones de los elementos propios de cualquiera de los diagramas UML, como las clases, asociaciones, propiedades, notas, enumeraciones o actividades. Un perfil permite crear nuevos elementos que no están incluidos como elementos estándar en la notación de UML (para definir el nuevo lenguaje derivado de UML) a través de los denominados *"estereotipos"* (stereotypes). Varios lenguajes adaptados de UML han tenido mucho éxito, e incluso han sido estandarizados por la OMG [101], como por ejemplo SPEM, IFML (del inglés, Interaction Flow Modeling Language) [25] o BPMN (del inglés, Business Process Model and Notation) [39].

La ventaja de los perfiles UML es la capacidad de introducir sistemáticamente otros elementos del lenguaje sin tener que recrear todo el entorno de modelado, como editores, transformaciones y APIs del modelo [101]. Otra de las ventajas de los perfiles, es que se logra tener extensiones dinámicas

del modelo, es decir a diferencia de las extensiones directas del meta-modelo, también los modelos ya existentes pueden ampliarse dinámicamente con información adicional del perfil sin necesidad de recrear los elementos extendidos del modelo.

Los perfiles UML son un mecanismo de extensión ligera que permiten adaptar UML a dominios específicos [65]. Por ejemplo, si se requiere un lenguaje para modelar *portales web*, se podría utilizar un diagrama de clases. No obstante, un diagrama de clases permite representar únicamente clases y sus relaciones, pero no tiene definido como parte de este lenguaje un elemento para representar una *página web*. En este caso, lo que se debe hacer es crear un perfil UML, para poder modelar este nuevo escenario. El perfil deberá hacer uso de un estereotipo para poder representar el elemento *página web*, pero como se señaló anteriormente, el estereotipo es un elemento que debe derivarse de un elemento de UML, en este caso podría ser de una clase (específicamente sería una meta-clase debido a que un estereotipo se enlaza a un elemento del meta-modelo de UML).

Una vez incluido el estereotipo *página web* al perfil y que se encuentre enlazado a una meta-clase a través de una relación de *extensión*, es posible que este nuevo elemento requiera ciertos atributos o propiedades (por ejemplo, el atributo *URL* de la *página web*), si es así, se debe utilizar un *"valor etiquetado"* (tagged value). Además, puede ser necesario incluir ciertas "*restricciones*" (constraints) si se quisiera imponer ciertas reglas, por ejemplo, que no se admitan más de una relación entre una *página web* y una *hoja de estilo*. Esto se podría realizar a través de un lenguaje de restricciones como el Object Constraint Language (OCL) [133] (véase en la siguiente sección).

En ocasiones no resulta adecuado utilizar UML para modelar ciertos sistemas, debido a que en estos casos UML puede resultar demasiado grande o no existe un diagrama adecuado que se ajuste al dominio con el que se trabaja. En estas situaciones resulta adecuado utilizar lenguajes de modelado más específicos que permitan modelar y representar los conceptos de dominios concretos. En este caso, los DSLs tienden a soportar abstracciones de mayor nivel que los lenguajes de modelado de propósito general. Los conceptos definidos en un DSL son más cercanos al dominio del problema que al dominio de la solución, debido a que un DSL sigue las abstracciones y semántica del dominio que representa, explicitando las reglas del dominio como restricciones del propio lenguaje [12].

2.1.4 Añadiendo restricciones al meta-modelo

Un lenguaje de modelado gráfico como UML, y en general, cualquier notación gráfica, no resultan lo suficientemente expresivos como para definir toda la información relevante de un dominio, ni permiten especificar con precisión los aspectos detallados del diseño de un sistema [30]. Para definir de manera precisa y no ambigua los detalles del modelo, los lenguajes de modelado muchas veces se complementan con otros tipos de lenguajes, normalmente lenguajes textuales pero basados en formalismos lógicos. El *Lenguaje de Restricción de Objetos* (OCL, del inglés Object Constraint Language) propuesto por la OMG [133] (desde algunos años estándar del ISO/IEC [80]) es uno de esos lenguajes. Las características principales de OCL son las siguientes:

- Es un lenguaje de definición de expresiones, a partir de las cuales pueden definirse condiciones que un modelo debe satisfacer.
- Es un lenguaje formal, en el sentido que define expresiones con una semántica precisa y no ambigua, y por tanto pueden ser evaluadas mediante herramientas automáticas.
- Es un lenguaje tipado, ya que cada expresión corresponde a un tipo de dato determinado. Solo es posible combinar expresiones de tipos compatibles entre sí.
- Es un lenguaje de especificación, y no de programación.

OCL es utilizado para definir aspectos estructurales (estáticos) y dinámicos (comportamiento) de un modelo. En el ámbito estructural de un modelo, OCL es utilizado para especificar las restricciones de integridad que el diagrama de clases debe satisfacer y también para definir la información derivada de este diagrama. En el ámbito dinámico, OCL es utilizado para definir el efecto de la ejecución de una operación (a través de expresiones que especifican las pre y pos-condiciones de la operación) y para definir consultas sobre la base de información [66].

En el contexto de MDE, OCL se utiliza en el proceso de creación de DSLs. Específicamente, en el momento de definir el meta-modelo del lenguaje, para especificar las reglas que establecen cuándo un modelo del dominio está bien formado. Por esta razón, OCL se ha convertido en un componente clave de cualquier técnica de MDE como lenguaje predeterminado para expresar todo tipo de especificación de requisitos, manipulación y consulta de (meta) modelos [30].

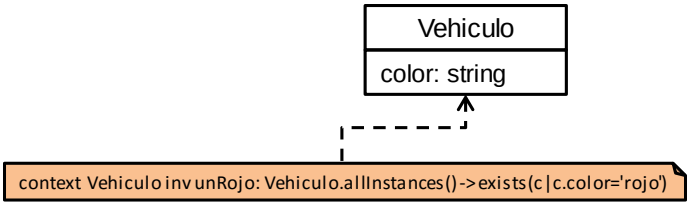


Figura 2.8: Ejemplo de una restricción OCL.

A modo de ejemplo, la Figura 2.8 muestra una restricción OCL que es utilizada para exigir que entre todos los vehículos que corresponden a la clase *Vehiculo* debe existir por lo menos uno de *color* rojo.

2.1.5 Transformación de modelos y generación de código

El proceso de desarrollo de software con MDE involucra un proceso en el que se pasa de modelar en el dominio del problema a obtener modelos y código para el dominio de la solución. MDE trata a los modelos como entidades de primer orden, ya que tiene como objetivo desarrollar, mantener y evolucionar el software realizando transformaciones de modelos [112]. Para ello se debe contar con mecanismos adecuados [66]. En la literatura se pueden encontrar varias definiciones de transformación de modelos:

- Según la guía de referencia de MDA [127], la transformación de modelos es el proceso de convertir un modelo en otro del mismo sistema.
- Kleppe en [94], define la transformación como la generación automática de un modelo de destino a partir de un modelo de origen, conforme a la definición de una transformación. Una definición de transformación es un conjunto de reglas de transformación que describen cómo un modelo en el lenguaje de origen puede transformarse en un modelo en el lenguaje de destino.
- Tratt en [158], señala que una transformación de modelos es un programa que muta un modelo en otro, en otras palabras, algo parecido a un compilador.

La transformación trata de producir diferentes modelos, puntos de vista o artefactos a partir de un modelo basado en un patrón de transformación. En general, la transformación se puede usar para producir una representación a partir de otra, o para cruzar niveles de abstracción o capas arquitectónicas [128].

Cuando una transformación es utilizada para generar una representación a partir de otra, el contenido conceptual del modelo puede permanecer igual, pero ciertamente la información se presenta en otra forma. Por ejemplo, se puede crear documentación HTML para un modelo de información, pero el contenido del modelo de información será el mismo. Por lo general, una transformación combina múltiples modelos de entrada en un modelo de salida. Por ejemplo, en un sistema de información, las clases y objetos de un sistema en J2EE se integran con un conjunto de anotaciones Hibernate para hacer persistencia de sus datos en una base de datos relacional PostgreSQL.

En la medida en que se pueden tener modelos de diferentes niveles de abstracción y que representan diferentes puntos de vista, pero relacionados, la transformación se puede usar para cruzar niveles de abstracción. Sin embargo, es posible que también la transformación se de en el mismo nivel de abstracción. Por ejemplo, se pueden transformar desde diagramas de clase a diagramas entidad relación, o desde diagramas de actividad a un diagrama BPMN, y el modelo resultante no tiene menor nivel de abstracción.

Las transformaciones hacen posible pasar de una representación muy abstracta, y basándose en patrones, a una representación más concreta. Por ejemplo, pasar desde el diseño hasta el código, lo que es conocido en ingeniería de software como "ingeniería directa" [128, 12]. También es posible pasar desde el código hasta el diseño, lo que es conocido como “ingeniería inversa” [112]. En este último caso, es posible pasar de una representación concreta (por ejemplo, un esquema XML) a una representación más abstracta; por ejemplo, un modelo de clases UML.

Existen varios tipos de transformación. Si se transforma un modelo en otro, se denomina transformación *Modelo a Modelo* (M2M*)*. Las transformaciones modelo a texto se utilizan a menudo para generar código a partir de modelos, por lo que a veces se las denomina *generadores de código*. Se debe señalar que la automatización del proceso de generación de código ejecutable libera al programador de la creación y el mantenimiento del código fuente, mejorando significativamente la

productividad del desarrollo, reduciendo los defectos y los errores en los programas, obteniendo una mejor calidad del software. Además, trabajar con modelos del dominio del problema (en lugar de modelos de código) permite elevar el nivel de abstracción y ocultar la complejidad y los detalles específicos de la implementación [142]. Es importante indicar que el código fuente es un modelo ya que es una representación simplificada de las estructuras y operaciones de la máquina de nivel inferior que se requieren para automatizar las tareas del mundo real. El código fuente es un modelo muy útil, ya que le indica a la máquina las acciones que tiene que realizar para mantener el objetivo del sistema [112].

Las transformaciones también se pueden clasificar como *"in-place"*, si cambian un modelo, reestructurándolo u optimizándolo. En este tipo de transformación el modelo origen y modelo destino son los mismos ya que todos los cambios se realizan in-situ (en el sitio) [112]. Finalmente, se pueden mencionar las transformaciones *"out-place"*, que son un tipo de transformaciones que crean elementos de modelo en un modelo basado en las propiedades de otro modelo (independientemente de que ambos modelos se ajusten al mismo meta-modelo) [112].

En MDE las transformaciones son realizadas de forma automática. Para ello existen varios DSLs que facilitan la tarea, entre ellos QVT (Query/View/Transformations) [134], ATL (ATLAS Transformation Language) [63], y Epsilon Transformation Language (ETL) [139], analizado en la sección 2.2.2. QVT es una especificación que define una familia de lenguajes de transformación propuesta por la OMG. La especificación define dos lenguajes para el usuario final (QVT Operational Mappings, un lenguaje imperativo y QVT Relations, un lenguaje declarativo), y un lenguaje de bajo nivel que puede verse como byte-code de QVT (QVT Core). ATL es un lenguaje híbrido (implementa los paradigmas declarativo e imperativo) de transformación de modelos, desarrollado por el grupo de investigación *AtlanMod* como una alternativa a QVT [14]. En su estilo de transformación declarativo la transformación se expresa como reglas que definen cómo se crean los elementos del modelo (o modelos) destino a partir de los elementos del modelo (o modelos) origen.

2.1.6 Modelado multi-nivel

MDE promueve el uso activo de modelos en las diferentes etapas del desarrollo de software [46]. Los enfoques estándar de MDE (como los que se basan en MOF o EMF), promueven un arquitectura de modelado de dos meta-niveles. En este escenario, los DSLs se definen a través de un meta-modelo (en el meta-nivel superior) que es instanciado para construir modelos (en el meta-nivel inferior). Esta situación fuerza la descripción del dominio (meta-modelo) dentro de un solo meta-nivel, en el cual es posible utilizar todas las facilidades nativas de meta-modelado, tales como la creación de clases, referencias, características o definición de tipos de datos. Sin embargo, estas facilidades no están disponibles a nivel de modelo, aunque en algunos escenarios se necesitan [44].

A pesar de las limitaciones del modelado de dos meta-niveles, éste continúa siendo muy exitoso en la práctica. Con la finalidad de superar sus limitaciones desde hace algunos años, varios investigadores han propuesto el *modelado multi-nivel* [7, 69]. Se considera que existen escenarios de

modelado en los cuales el uso de múltiples meta-niveles resulta en modelos más simples con menos complejidad accidental.

El modelado multi-nivel extiende el enfoque estándar de meta-modelado al permitir el modelado en un número arbitrario de meta-niveles, no necesariamente dos [3, 8]. En el contexto de meta-modelado, se puede identificar cuándo un meta-modelo puede ser representado como multi-nivel debido a la presencia de varios patrones. Estos patrones emulan las diferentes facilidades de meta-modelado, como la faceta tipo-objeto de los elementos (patrón tipo-objeto), la relación de instanciación, la creación dinámica e instanciación de referencias y atributos.

Según el estudio realizado por Juan de Lara y colaboradores [46], donde se analiza un corpus de más de 400 meta-modelos existentes, se detecta que un número sustancial de ellos podrían beneficiarse del modelado multi-nivel. Los meta-modelos analizados han sido tomados de diferentes fuentes, como el zoo de meta-modelos de ATL [13], repositorio ReMoDD [144], especificaciones de la OMG [136] y de diferentes dominios (como, arquitectura de software, modelado de procesos, diseño de software, ingeniería web, modernización de software y salud).

En el caso del zoo de meta-modelos de ATM[2], de 296 meta-modelos, 25 (8,4%) incluyen al menos una ocurrencia del patrón tipo-objeto. En el repositorio ReMoDD que contiene 25 meta-modelos, 3 (20%) tienen algún patrón de aparición. En cuanto a los meta-modelos del OMG, de 116 especificaciones, 41 (35%) se consideran que tienen alguna ocurrencia del patrón tipo-objeto. El estudio, concluye señalando que el modelado multi-nivel es aplicable y potencialmente beneficioso en escenarios reales, por tanto, una tecnología relevante dentro del contexto de MDE. En la siguiente sección se realiza una introducción de modelado multi-nivel a través de ejemplos.

2.1.7 Modelado multi-nivel mediante ejemplos

Para motivar la necesidad de modelado multi-nivel y su lógica, esta sección presenta un ejemplo. Supongamos que se tiene la necesidad de crear un lenguaje arquitectónico [108], para que los usuarios del lenguaje (es decir, arquitectos de software) puedan definir tipos de componentes, formados por tipos de puertos (pueden ser puertos de entrada, puertos de salida o ambos). Los tipos de puerto pueden definir los tipos de mensajes que pueden manejar y conectarse mediante tipos de conectores. Se asume que los conectores producen un retraso (*delay*) configurable en los mensajes que transportan. Los tipos de mensaje deben describir la estructura de los datos que llevan los mensajes. Además de los datos definidos, todos los mensajes tienen una marca de tiempo (time stamp). Los tipos definidos (componentes, puertos, mensajes, conectores) se pueden usar para definir la configuración en tiempo de ejecución (run-time) de un sistema. Este es un lenguaje típico basado en componentes [67], del estilo de los lenguajes *componente-conector* (component-connector), y los lenguajes para la simulación basada en componentes [164].

[2] http://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Ecore

A continuación, se resuelve este problema utilizando varios enfoques, que de manera natural conducen a una solución basada en el modelado multi-nivel nativo. Se utiliza esta solución para introducir las nociones básicas del modelado multi-nivel.

2.1.7.1 El patrón tipo-objeto

Un primer intento de describir el ejemplo es definir un meta-modelo que incluya primitivas para crear elementos con faceta de tipo y sus instancias. En la Figura 2.9 se muestra una posible solución que utiliza este enfoque. La figura muestra que el meta-modelo debe contener elementos para representar tipos de componentes e instancias, tipos de puertos e instancias (tanto de entrada como de salida), tipos de conectores e instancias y tipos de mensajes e instancias, así como las relaciones de tipado entre tipos e instancias (referencias *cmpType, pType, mType y cnType*). Esta representación explícita de elementos de tipo e instancia se denomina *patrón tipo-objeto* (type-object pattern) [106]. Se debe tener en cuenta que también debemos definir atributos para los tipos de mensaje (clase *Attribute*) y darles valores (clase *Slot*). Este es un caso particular del patrón tipo-objeto, que se denomina *patrón de características dinámicas* (dynamic features pattern) [46]. En el nivel del modelo, se utilizan objetos para representar tanto elementos de tipo (por ejemplo, *Producer, Consumer, Pin, Pout, MChannel, Packet*), como elementos de objeto (por ejemplo, *p, s, m, c1, op, ci, i1, i2, c1, c2*). Por lo tanto, conceptualmente, el modelo codifica dos meta-niveles dentro de uno, y en la figura se utiliza una línea discontinua para transmitir dicha intención.

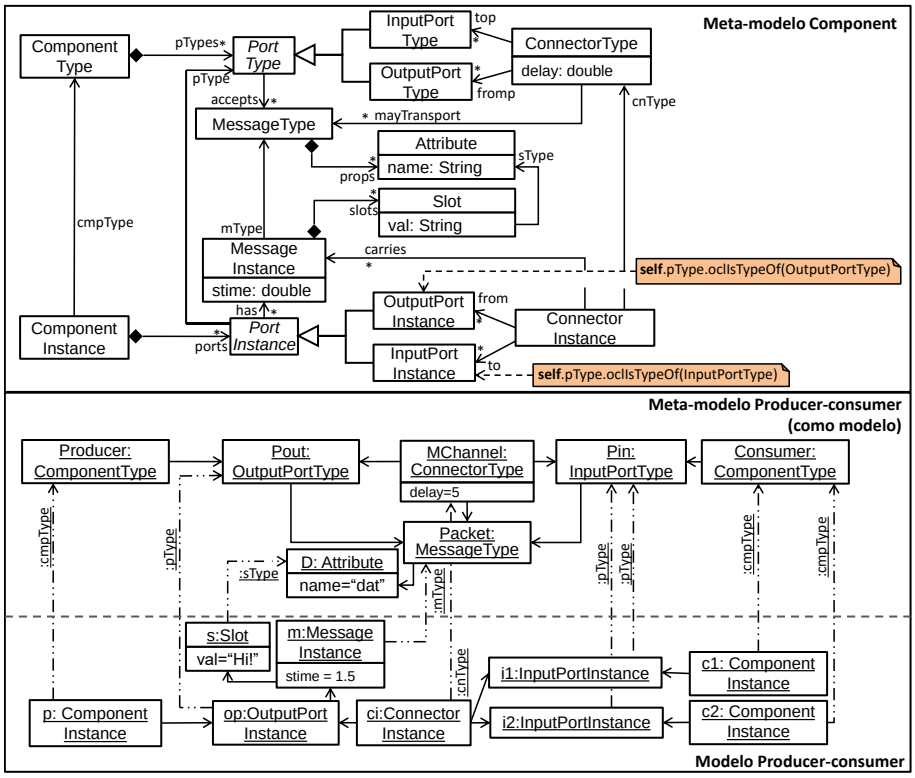


Figura 2.9: Resolviendo el ejemplo usando el patrón tipo-objeto.

En general, si bien esta solución funciona en la práctica, conduce a una gran replicación de elementos en el nivel del meta-modelo, lo que provoca un aumento de la complejidad accidental del meta-modelo [11, 104]. En particular, presenta seis apariciones del patrón de tipo-objeto (*ComponentType/Instance, MessageType/Instance, PortType/Instance, InputPort/Instance, OutputPort/ Instance, ConnectorType/Instance*), y una aparición del patrón de características dinámicas (clases *Attribute/Slot*). Además, si se quisieran definir restricciones de cardinalidad en el nivel del modelo (por ejemplo, en las relaciones entre los objetos *Producer* y *Pout*, *MChannel* y *Pout*, etc.) se tendrían que definir clases y restricciones adicionales en el meta-modelo, utilizando el llamado *patrón configurador de relaciones* (relation configurator pattern) [46].

2.1.7.2 Enfoque de extensión del meta-modelo

Un enfoque alternativo consiste en definir un meta-modelo que solo incluya primitivas para crear elementos y extensiones (a través de la herencia) para crear tipos de componentes específicos, que luego se pueden instanciar en modelos. Este enfoque se llama *estático* (static) en [46]. La Figura 2.10 muestra una solución al ejemplo utilizando este enfoque. En esta aproximación, los tipos necesarios para crear el modelo se definen como subclases del meta-modelo del componente (en lugar de como instancias). Se ha separado con una línea discontinua los elementos del meta-modelo *Component* original de las extensiones.

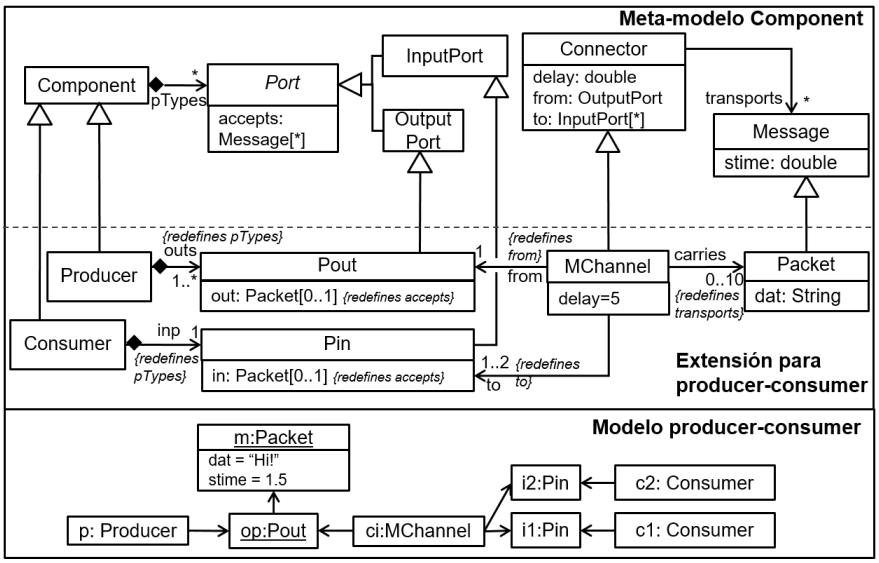


Figura 2.10: Resolviendo el ejemplo usando el enfoque de extensión de meta-modelo.

Esta solución es más natural para modelar elementos con facetas de tipo (como *Producer*, *Consumer*, etc.) en el nivel de meta-modelo. Además, admite la adición de atributos a las subclases (por ejemplo, a la clase *Packet* se le agrega el atributo *dat*, que se tuvo que modelar explícitamente en la Figura 2.9), y la definición de cardinalidades para las relaciones entre las clases extendidas (por ejemplo, para *outs*, *inp*, *from*, *to* y *carrier*).

Sin embargo, se requiere de mecanismos para redefinir asociaciones (como *pTypes*, que se redefine en *Producer.outs* y *Consumer.inp*) y atributos (la característica estática *delay* es redefinida en *MChannel*, asignándole un valor predeterminado). Además, a nivel de modelo, las relaciones como *pTypes* no deben ser instanciadas, sino sólo sus redefiniciones (*outs* e *inp*). Se debe tener en cuenta que muchos frameworks de meta-modelado, como EMF, no soportan estas redefiniciones. Un problema adicional es permitir que el usuario modifique el meta-modelo original (es decir, el meta-modelo *Component* en este caso) puede provocar cambios inesperados (por ejemplo, como crear una clase heredada tanto de *Component* como de *Message*).

2.1.7.3 Transformaciones de promoción

Un enfoque diferente para soportar la doble instanciación requerida por el ejemplo es introducir una transformación que convierte los objetos en tipos, llamada *transformación* de *promoción* (promotion transformation) [46]. La Figura 2.11 muestra una solución del ejemplo utilizando este enfoque. La idea es permitir la creación de instancias del meta-modelo *Component* (ver (a) en la figura), y a través de una transformación modelo a modelo, crear un meta-modelo (ver (c) en la figura) a partir de modelo creado (b). Luego, este meta-modelo se puede instanciar una vez más para crear un modelo *producer-consumer* (ver (d) en la figura).

Por lo tanto, mientras que los enfoques anteriores incorporan dos meta-niveles dentro de uno (ya sea en el nivel del meta-modelo o en el nivel del modelo), las transformaciones de promoción dividen la naturaleza de tipo-objeto del modelo intermedio en dos, y usan una transformación modelo a modelo para crear las facetas de tipo (c) a partir de las facetas de objeto (b).

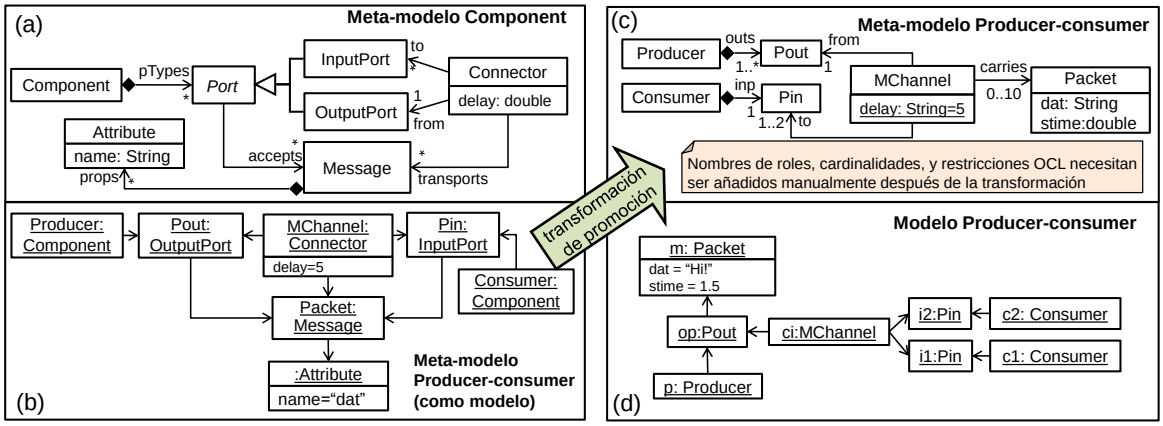


Figura 2.11 Resolviendo el ejemplo utilizando transformaciones de promoción.

Este enfoque tiene varios inconvenientes. Primero debe introducir una transformación modelo a modelo, que agrega complejidad a la solución y mueve algunas decisiones de diseño desde el meta-modelo al código de la transformación. Por ejemplo, el hecho de que las instancias de *Message* en el nivel inferior (por ejemplo, el objeto *m* en el modelo (d)) tengan un atributo *stime*, no se declara en la clase *Message*, sino que la transformación debe crear un atributo *stime* en cada instancia del

Message (por ejemplo, *Packet* en el modelo (c)). En segundo lugar, el enfoque desacopla explícitamente la faceta de objeto de los modelos (es decir, el modelo (b)) de su faceta de tipo (modelo (c)), lo que complica la manipulación del modelo. Por ejemplo, los nombres de rol, cardinalidades y las posibles invariantes de OCL deben agregarse manualmente al meta-modelo (c) una vez que se ejecuta la transformación de promoción. Si el modelo (b) se cambia después, la regeneración de (c) puede sobreescribir dichos cambios manuales. Finalmente, todavía debe incluirse en el meta-modelo (a) elementos que emulan facetas de nivel de tipo. En particular, todavía se debe incluir la clase *Attribute* para emular la definición de atributos en la clase *Message*.

2.1.7.4 Modelado multi-nivel

Los enfoques anteriores han mostrado tres soluciones diferentes para codificar una solución que naturalmente abarca tres meta-niveles, en dos. Las soluciones y sus inconvenientes se resumen en la Tabla 2.1.

Tabla 2.1: Ventajas y desventajas de enfoques que emulan modelado multi-nivel.

Enfoque	Solución	Ventajas y Desventajas
Patrón tipo-objeto	Incluye elementos en el meta-modelo que permiten la creación de tipos y sus instancias en el nivel del modelo.	✗ Replicación pesada de elementos, aumentando la complejidad accidental del meta-modelo. ✓ Compatibilidad media con el meta-modelado estándar de dos niveles: las herramientas estándar se pueden usar para editar y manipular modelos, pero las restricciones no se pueden definir en tipos definidos por el usuario como *Packet* (ya que estos tipos están en el nivel del modelo).
Extensión del meta-modelo	Emula la instanciación con herencia.	✗ El meta-modelo original se ofrece al desarrollador. Se necesitan mecanismos de control de las extensiones. ✗ Necesita emular instancias (por ejemplo, valores de atributos) a nivel de meta-modelo. ✓ Buena compatibilidad con el meta-modelado estándar de dos niveles: las herramientas estándar pueden utilizarse para editar y manipular modelos, y las restricciones pueden definirse en tipos definidos por el usuario como *Packet*.
Transformación de promoción	Crea una transformación que transforma elementos del modelo (objetos) en elementos del meta-modelo (clases).	✗ Necesita crear e introducir una transformación para el modelado. ✗ Desvincula las facetas de instancia y tipo de los elementos. ✗ Todavía es necesario contaminar el meta-modelo con elementos para emular las facilidades de meta-modelado (por ejemplo, la clase *Attribute* en la Figura 2.10). ✗ Se necesitan crear más transformaciones si se utilizan más de tres meta-niveles. ✓ Compatibilidad media con el meta-modelado estándar de dos niveles: se pueden utilizar herramientas estándar para editar modelos, pero es necesaria la integración con una herramienta de transformación.

Las soluciones que se resumen en la Tabla 2.1 no serían necesarias si se permitiera al desarrollador usar varios meta-niveles en lugar de dos. La Figura 2.12 muestra una solución al ejemplo utilizando un enfoque de modelado multi-nivel. Se puede observar que la solución abarca tres meta-niveles. El de más arriba contiene las primitivas del meta-modelo *Component*, y no se requiere que las clases

emulen características de nivel de tipo (por ejemplo, la clase *Attribute*) o características del nivel de instancia (por ejemplo, clases como *ComponentInstance* o *ConnectorInstance* en la Figura 2.9). En el nivel intermedio (meta-modelo *producer-consumer*), se puede utilizar una instanciación normal, y así se puede, por ejemplo, asignar un valor a *MChannel.delay*. Los elementos en este meta-nivel intermedio presentan a la vez facetas tanto de tipo como de instancia, por lo que son llamados *clabjects* (de la concatenación de las palabras *clase* y *objeto* en inglés) [3].

En el nivel superior, es interesante que los clabjects caractericen sus instancias más allá del próximo meta-nivel inmediatamente inferior. Esto se hace con la llamada *potencia* [3], un número natural (o cero) que se asocia a los clabjects, atributos y referencias. En las figuras, las potencias se indican después del símbolo "@". En el ejemplo, al modelo de nivel superior se le asigna la potencia 2, lo que significa que se puede crear una instancia en el siguiente meta-nivel y luego a un nivel inferior. La potencia de un modelo se llama a veces "nivel". Si un clabject no declara explícitamente una potencia, toma la potencia de su modelo contenedor.

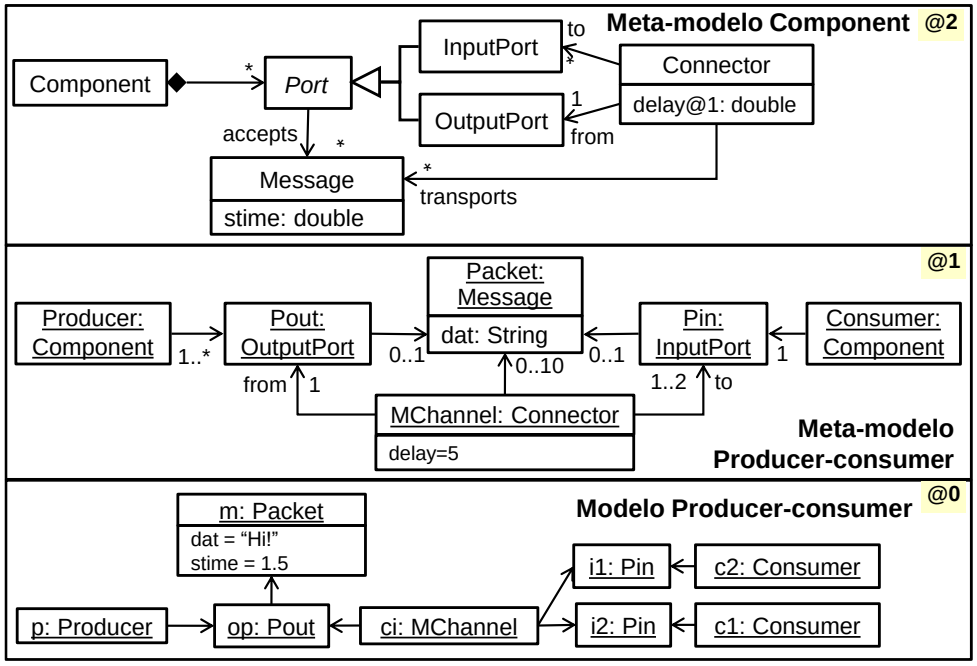


Figura 2.12: Resolviendo el ejemplo usando el modelado multi-nivel.

De esta manera, todos los clabjects en el meta-modelo *Component* reciben potencia 2 (sin embargo, se debe tener en cuenta que, en general, un modelo puede contener clabjects con distinta potencia. Cuando un clabject se instancia, la instancia recibe la potencia del tipo menos uno. Los clabjetcs con potencia 0 no pueden ser instanciados.

Las reglas de potencia que se acaban de describir funcionan de la misma manera para los atributos y referencias. De esta manera, en la figura, *stime* en *Message* recibe potencia 2 de su clabject contenedor. Esto significa que se convertirá en una instancia de atributo (un slot) dos niveles más abajo, por lo que debería recibir un valor en instancias de instancias de *Message* (como *m* en el nivel 0). Esta es una forma de controlar los atributos y referencias que deben tener las instancias indirectas

de un clabject. En contraste, el atributo *delay* en el clabject *Connector* declara explícitamente una potencia de 1, lo que significa que se convierte una instancia en instancias de *Connector* (como *MChannel*).

En el modelado multi-nivel, los elementos (modelos, clabjects, atributos y referencias) generalmente conservan ambas facetas de tipo y de instancia. Por lo tanto, las referencias en el nivel 1 son instancias de referencias en el nivel superior y tipos para referencias en el nivel inferior. De esta manera, al ser instancias, deben obedecer las cardinalidades de sus tipos y al ser tipos, pueden declarar una cardinalidad, que debe ser obedecida por sus instancias. Por ejemplo, en la figura, la referencia *Connector.from* en el nivel 2 declara una cardinalidad de 1. Luego, *MChannel* (instancia de *Connector*) en el nivel 1 obedece dicha cardinalidad, ya que define exactamente una instancia de *from* (que también se llama *from*). La referencia *MChannel.from* define una cardinalidad por sí misma, que luego es obedecida en el nivel 0 por el objeto *ci*.

Si bien la potencia permite una caracterización profunda de instancias en varios meta-niveles inferiores, a veces es difícil considerar todas las características y primitivas posibles, especialmente al diseñar un lenguaje que se va a reutilizar en diferentes dominios. De esta manera, el modelado multi-nivel permite a los clabjects declarar nuevos atributos (como *Packet*, que declara el field *dat*), y también permite crear clabjects sin ningún tipo ontológico. Los atributos, referencias y clabjects sin tipo se denominan *extensiones lingüísticas* (linguistic extensions) [40].

Las extensiones lingüísticas son posibles gracias a la llamada Arquitectura de Clasificación Ortogonal (OCA, del inglés Orthogonal Classification Architecture) [10]. Esta arquitectura distingue dos tipos de tipado: ontológico y lingüístico. El tipado ontológico se refiere a la creación de instancias dentro de un dominio. Por ejemplo, en la Figura 2.12, el *tipado ontológico* de *m es Packet*. El tipado lingüístico se refiere a la primitiva de meta-modelado utilizada para crear el elemento. Por ejemplo, en la Figura 2.12, el tipo lingüístico de *m* es *Clabject*. De esta manera, las extensiones lingüísticas no tienen un tipado ontológico, sino sólo lingüístico.

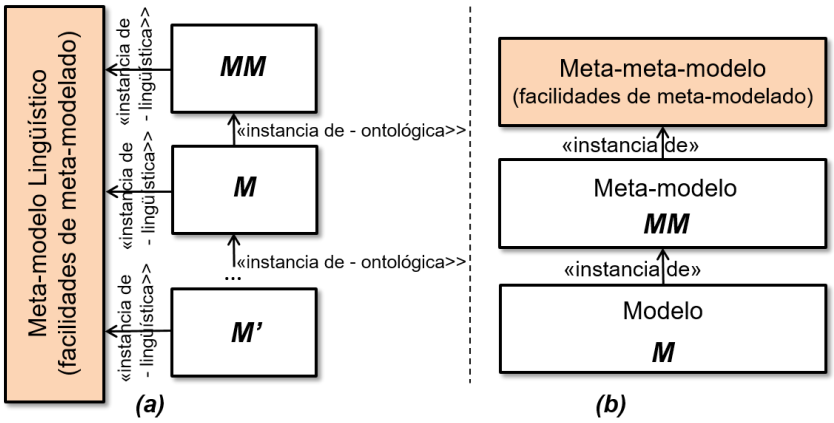


Figura 2.13: Arquitectura de Clasificación Ortogonal.
(a) Arquitectura de clasificación ortogonal; (b) Arquitectura estándar de meta-modelado.

La Figura 2.13 muestra un esquema que compara la OCA (a) con la arquitectura de meta-modelo estándar (b). OCA utiliza un meta-modelo lingüístico que define primitivas de meta-modelado, que están disponibles en todos los meta-niveles. Esto significa que las facilidades de meta-modelado como la herencia se pueden usar en cualquier nivel, incluido el nivel 0. En contraste, en la arquitectura de meta-modelado estándar (Figura 2.13(b)), las facilidades de meta-modelado (como herencia, atributos, referencias y cardinalidades) solo están disponibles en el nivel del meta-modelo, por lo que deben ser creadas de nuevo en el meta-modelo cuando se produce el patrón tipo-objeto. Por ejemplo, en la solución de la Figura 2.9, se tuvo que volver a definir la relación de instanciación, así como la noción de *atributo* e *instancia de atributo (slot)*.

En general, para el ejemplo, el modelado multi-nivel produce la solución con el menor número de elementos, junto con el enfoque de extensión del meta-modelo. Sin embargo, el enfoque de extensión del meta-modelo requeriría mecanismos para emular propiedades a nivel de instancia (slots, links) en el nivel del meta-modelo, un control preciso de cómo se puede extender el meta-modelo y un control sobre meta-niveles inferiores. En comparación con los otros enfoques, la principal desventaja del modelado multi-nivel es la falta de compatibilidad con los enfoques estándar de modelado de dos niveles. Como una de las ventajas del enfoque estático es su compatibilidad con el modelado de dos niveles, en el capítulo 5 se propone la realización y combinación del modelado multi-nivel con el enfoque estático, obteniendo los beneficios de ambos enfoques.

2.2 Tecnologías MDE

En esta sección se hace una revisión de las principales tecnologías y herramientas utilizadas en la Ingeniería Dirigida por Modelos (MDE).

2.2.1 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) es una implementación del estándar de MOF. Es un framework de modelado que facilita la generación de código para construir aplicaciones basadas en modelos de datos estructurados [62]. A partir de una especificación del modelo (*modelo de dominio* o *meta-modelo*) descrita en XMI [138], EMF proporciona herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases que permiten la visualización, edición del modelo, creación y manipulación de instancias del meta-modelo [156].

La Figura 2.14(a) muestra un meta-modelo para un sistema que administra la información de los vehículos que tiene una persona. Al meta-modelo en este caso se representa de manera gráfica, en forma de diagrama de clases. La Figura 2.14(b) muestra una instancia del meta-modelo señalado, es decir un modelo (utilizando un editor tipo árbol) que representa el sistema requerido.

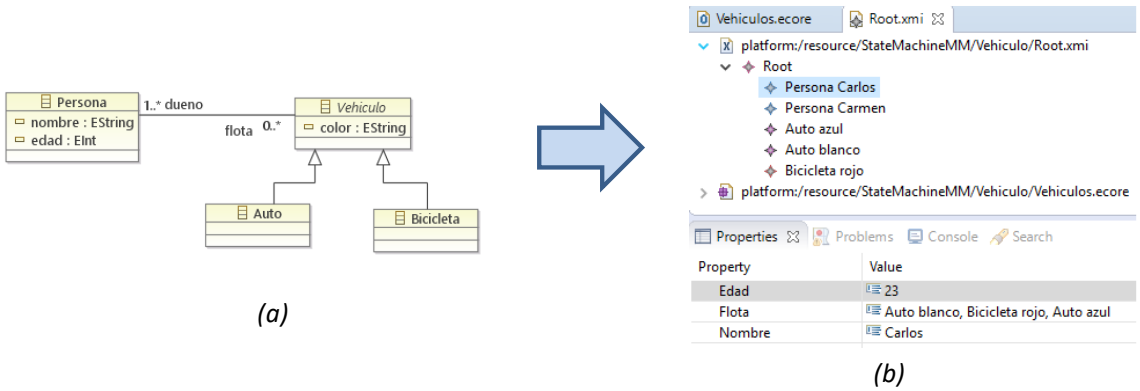


Figura 2.14: Editor en forma de árbol generado a partir de un meta-modelo.

El núcleo de EMF es un estándar común para los modelos de datos, en el que se basan otras tecnologías y frameworks. A continuación, se señalan varias tecnologías que utilizan EMF:

- ***Servidor y persistencia:*** permite la edición de modelos de forma colaborativa, así como el manejo de versiones de modelos. Como ejemplo se puede mencionar a EMFStore[3], un repositorio de modelos para EMF.
- ***Frameworks de interfaces de usuario:*** son aplicaciones con facilidades para mostrar y modificar el modelo de dominio, por lo general de manera gráfica, no necesariamente a través de representaciones típicas de datos como formularios u hojas de propiedades. Un ejemplo de este tipo de framework es EMF Client Platform[4].
- ***Soporte para transformaciones:*** como Acceleo[5], ATL[6] y Epsilon[7].

EMF consta de tres elementos fundamentales:

- ***EMF:*** el framework central de EMF incluye el meta-modelo Ecore para describir modelos. Proporciona soporte en tiempo de ejecución para el modelado, incluye notificaciones de cambios, soporte de persistencia mediante serialización XMI y una API para manipular objetos EMF.
- ***EMF.Edit:*** el framework de EMF.Edit incluye clases genéricas para crear editores de modelos EMF.
- ***EMF.Codegen:*** permite la generación de código para crear un editor completo para un modelo EMF.

Un meta-modelo o modelo en términos generales puede ser representado de diferentes maneras. Por ejemplo, de forma gráfica, serializado, o mediante código en un determinado lenguaje. La Figura 2.15, muestra algunos formatos de representación de meta-modelos en EMF. En la figura, con la

[3] https://www.eclipse.org/emfstore/
[4] https://www.eclipse.org/ecp/
[5] https://www.eclipse.org/acceleo/
[6] https://www.eclipse.org/atl/
[7] https://www.eclipse.org/epsilon/

etiqueta (a) se representan los elementos del meta-modelo en formato gráfico, mostrando su clases y atributos dentro de rectángulos, en tanto que las relaciones son representadas mediante enlaces (como es habitual en los diagramas de clases). En (b) se representa el meta-modelo en XMI, una especificación basada en XML de formatos de intercambio de modelos. En (c) el meta-modelo es representado como un conjunto de clases en lenguaje de programación Java. Finalmente, en (d) el meta-modelo es representado en la sintaxis textual utilizada por el editor OCLinEcore de Eclipse, un editor recomendado para incluir expresiones OCL en el meta-modelo (en las líneas 22 y 23 se puede observar una invariante OCL).

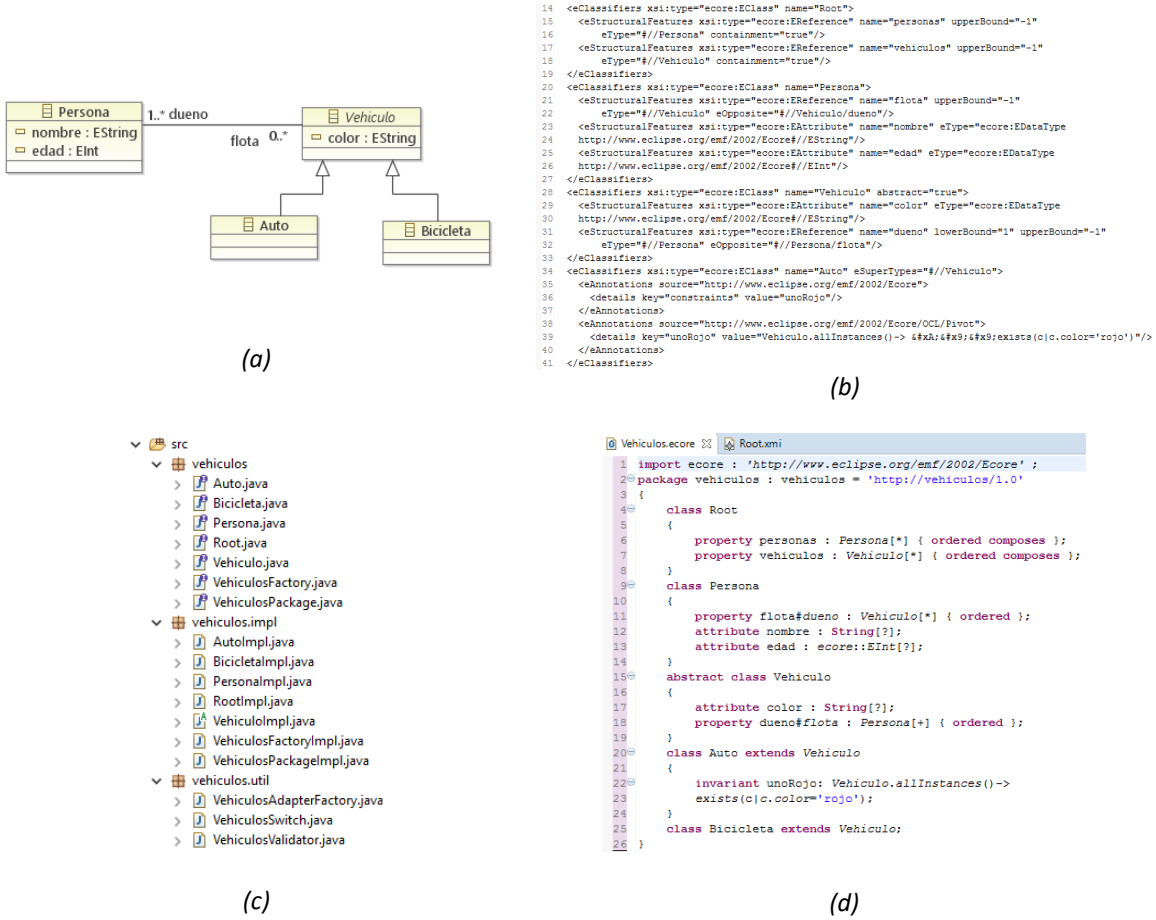


Figura 2.15: Formatos de representación de modelos en EMF.

EMF tiene como elemento central el lenguaje de meta-modelado Ecore, que proporciona la infraestructura básica de Eclipse para crear herramientas y soluciones MDE [156]. Ecore es un subconjunto de MOF, el lenguaje de meta-modelado creado por la OMG para definir formalmente UML. Ecore proporciona los conceptos orientados a objetos para la creación de meta-modelos: los conceptos son expresados con clases, las propiedades con atributos, las relaciones entre conceptos mediante agregación y referencias, mientras que la generalización es empleada para expresar que un concepto es una especialización de otro [66]. Para asegurarse que un modelo se encuentre bien formado se pueden incluir restricciones OCL en el meta-modelo.

La Figura 2.16 muestra la estructura del meta-modelo Ecore, en el cual se pueden identificar todos los elementos para la creación de meta-modelos para crear soluciones MDE.

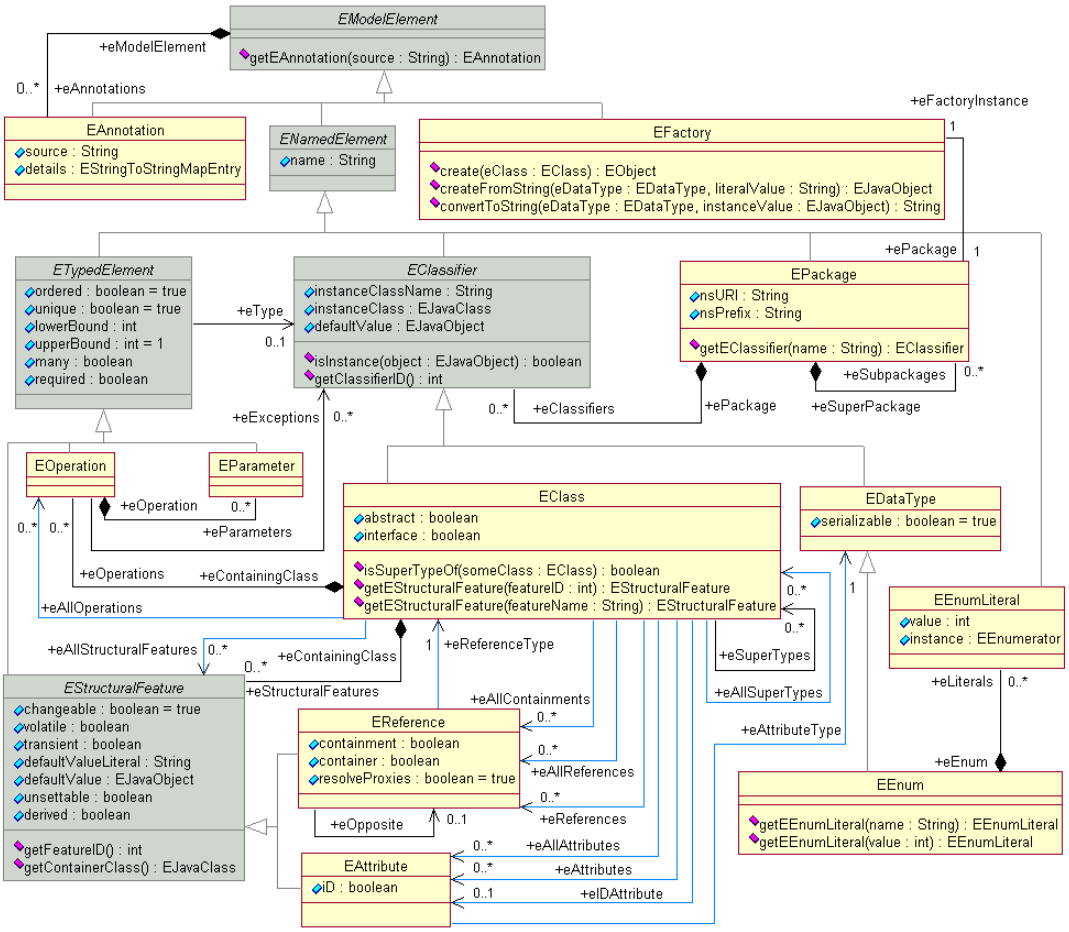


Figura 2.16: Meta-modelo de Ecore.

En la figura se puede observar que el elemento raíz de Ecore es la clase abstracta *EModelElement* que a su vez hereda de la clase *EObject* (equivalente a "java.lang.Object", raíz de la jerarquía de clases del lenguaje de programación Java). Ecore al igual que MOF se basa en elementos de modelado orientado a objetos. Entre los principales elementos se encuentran clases (EClasses en Ecore), atributos (EAttributes), referencias (EReferences), y paquetes (EPackages), los cuales permiten definir un lenguaje de modelado.

2.2.2 Lenguajes de gestión de modelos

La gestión de modelos es la disciplina de gestión de artefactos utilizados en la ingeniería dirigida por modelos (MDE). Un marco de gestión de modelo define e implementa las operaciones (como la edición, comprobación de coherencia (validación), la transformación o la generación de código) necesarias para manipular artefactos MDE. Los enfoques modernos para la gestión de modelos

generalmente implementan estas operaciones a través de lenguajes específicos de dominio (DSLs) [96, 139]. En el núcleo de estas técnicas de gestión de modelos se encuentra un conjunto de servicios para la *navegación* y *modificación* de modelos. Estas operaciones deben ser realizadas de forma automática sobre los modelos. En efecto, uno de los principales retos que deben abordarse para potenciar el papel de los modelos en la ingeniería de software es automatizar dichas tareas para reducir la complejidad que tiene asociada [66].

Actualmente, la mayoría de los entornos de meta-modelado proporcionan una API para la gestión de modelos. Por ejemplo Epsilon [53], es una familia de lenguajes de propósito específico para trabajar directamente con modelos [96]. El núcleo de Epsilon es Epsilon Object Language (EOL) [96], un lenguaje imperativo orientado a modelos que combina el estilo procedimental de Javascript con las capacidades de consulta de OCL. A continuación, se describen algunos lenguajes de Epsilon:

- ***Epsilon Transformation Language (ETL):*** es un lenguaje híbrido de transformación modelo a modelo basado en reglas construido sobre EOL. ETL proporciona todas las características estándar de un lenguaje de transformación, pero también proporciona una flexibilidad mejorada, ya que puede transformar muchos modelos de entrada en muchos modelos de salida y puede consultar/navegar/modificar los modelos de origen y destino.
- ***Epsilon Validation Language (EVL):*** es un lenguaje de validación construido sobre EOL. En su forma más simple, las restricciones EVL son bastante similares a las restricciones OCL, permitiendo definir correcciones automáticas de los errores o personalizar los avisos generados.
- ***Epsilon Generation Language (EGL):*** un lenguaje de plantillas destinado a la generación de código y documentación a partir de modelos.
- ***Epsilon Wizard Language (EWL):*** es particularmente útil para automatizar tareas recurrentes de edición de modelos. Estas incluyen, por ejemplo, refactorizar, aplicar patrones o construir subárboles que consisten en elementos similares. EWL está integrado con EMF/GMF[8] y como tal, los asistentes pueden ejecutarse desde los editores de EMF y GMF (Graphical Modeling Framework). Este último proporcionau un conjunto de componentes generativos e infraestructuras de tiempo de ejecución para desarrollar editores gráficos.
- ***Epsilon Comparison Language (ECL):*** es un lenguaje híbrido basado en reglas para comparar modelos homogéneos o heterogéneos. ECL se puede utilizar para establecer las correspondencias sobre las que se pueden combinar los modelos utilizando el lenguaje de fusión de Epsilon (EML, ver siguiente punto) o para pruebas de transformación.
- ***Epsilon Merging Language (EML):*** es un lenguaje híbrido basado en reglas para combinar modelos homogéneos o heterogéneos. Como un lenguaje de fusión requiere todas las características de un lenguaje de transformación (fusionar el modelo A con un modelo vacío en el modelo B es equivalente a transformar A en B), EML reutiliza la sintaxis y la semántica de ETL y lo amplía con conceptos específicos de fusión de modelos. Antes de poder realizar

[8] Graphical Modeling Framework

la fusión, es necesario establecer correspondencias entre los elementos de los modelos de entrada. Esto se puede lograr usando el lenguaje de comparación de Epsilon (o usando Java).

- ***Epsilon Flock:*** es un lenguaje de migración de modelos creado sobre EOL, para actualizar modelos en respuesta a los cambios del meta-modelo. Flock proporciona un lenguaje de transformación basado en reglas para especificar estrategias de migración de modelos. Un algoritmo de copia conservador migra automáticamente los valores y elementos del modelo que no se ven afectados por los cambios del meta-modelo.

2.2.3 Herramientas para la definición de la sintaxis concreta

Una vez que ha sido definida la sintaxis abstracta (meta-modelo) del DSL, el siguiente paso es definir su sintaxis concreta. Ésta mapea cada uno de los elementos del meta-modelo a una representación textual, gráfica o una combinación de ambos. La sintaxis concreta es la que ve y utiliza el usuario para construir modelos. Como norma general es recomendable utilizar elementos textuales y gráficos con los cuales esté familiarizado el usuario final [81].

Entre las herramientas más relevantes para desarrollar editores gráficos de modelos, se puede mencionar a GMF (Graphical Modeling Framework), Graphiti, Sirius, entre otros. A continuación, se repasan las características más importantes de estos frameworks.

Graphical Modeling Framework (GMF): es un framework de modelado gráfico de Eclipse que proporciona un componente generativo e infraestructura en tiempo de ejecución para el desarrollo de editores gráficos en EMF [54]. Para construir estos editores el diseñador debe definir los elementos gráficos que se mostrarán en los diagramas, la paleta de herramientas del editor, y la relación entre los dos modelos anteriores con el meta-modelo del DSL [66].

Graphiti: es un framework para crear editores gráficos sofisticados basados en Eclipse [55]. Entre los objetivos del proyecto Graphiti se pueden mencionar: a) proporcionar una API de Java simple, bien estructurada y fácil de usar para crear herramientas gráficas, b) proporcionar documentación y tutoriales para hacerlo, y c) proporcionar la capacidad de poder reutilizar algoritmos de layout de un diagrama.

Sirius: el objetivo de Sirius es facilitar la creación de editores basados en la representación de un DSL aprovechando las tecnologías de modelado de Eclipse, incluidos EMF y GMF [122]. Sirius se basa en un enfoque de punto de vista (viewpoint), que se encuentra apoyado por el estándar ISO/IEC-42010 [77], donde se especifica que la misma información puede ser representada de manera sincronizada en diagramas, tablas y árboles. En Sirius a partir del meta-modelo de un DSL, el desarrollador puede especificar el editor de manera completamente declarativa y poder observar los resultados en tiempo real, sin necesidad de generar código. Sirius se apoya en Acceleo, y otros proyectos, para facilitar el establecimiento de las relaciones entre los datos del modelo y su representación gráfica.

Entre las herramientas para crear editores textuales se puede mencionar a Xtext, EMFText y TCS. A continuación, se describen las características más importantes de estos frameworks.

Xtext: es un DSL diseñado para describir lenguajes textuales. XText proporiciona un DSL similar a EBNF para especificar la sintaxis concreta, a partir de la cual se genera un modelo en memoria, conforme a un meta-modelo [58]. Permite implementar lenguajes rápidamente y sobre todo cubre todos los aspectos de una infraestructura completa de lenguajes, desde el analizador sintáctico (parser), el generador de código o intérprete, hasta una completa integración con el IDE de Eclipse [18]. Asocia una representación textual a los conceptos y relaciones del lenguaje especificados en el meta-modelo. Las gramáticas se componen de un conjunto de reglas terminales y no terminales. Adicionalmente, se pueden definir predicados sintácticos, símbolos terminales ocultos y reglas para valores enumerados. Xtext tiene una facilidad para generar una gramática por defecto a partir de un meta-modelo, que se establece a través de reglas, tales como: a) de un paquete del meta-modelo se deriva una nueva gramática, b) de una clase se deriva una regla gramatical, c) de los atributos de la clase se derivan asignaciones simples de una regla, d) de una composición se derivan asignaciones múltiples de una regla (representación de cardinalidad: cero o uno [?], cero o más [*], uno o más [+]), e) la herencia de una clase deriva una regla con alternativas, y f) en relaciones de herencia se produce la des-factorización de atributos de la clase base en las subclases.

EMFText: es una herramienta para definir la sintaxis textual de meta-modelos basados en Ecore [75]. Permite a los desarrolladores definir sus propios lenguajes textuales, ya sean lenguajes de dominio específico o lenguajes de propósito general (por ejemplo, Java). Proporciona un lenguaje de especificación de sintaxis a partir del cual se generan editores y componentes textuales para cargar y almacenar instancias de modelos textuales. Los editores generados por EMFText proporcionan varias características avanzadas, como por ejemplo las que incluye el editor Java de Eclipse. Esto incluye compleción del código (con propuestas de compleción personalizable), sintaxis personalizable y el resaltado (highlighting) de sintaxis, el plegado de código, hipervínculos y los desplazamientos de texto para una navegación rápida, una vista de esquema e informes de errores instantáneos.

Xtext [58] es el framework de modelado textual estándar en Eclipse, pero su núcleo es muy similar al de EMFText. Xtext tiene su propio lenguaje de especificación de sintaxis modular, usa ANTLR (del inglés, ANother Tool for Language Recognition) como generador del parser y generador de editores. Sin embargo, hay diferencias detrás de ambas herramientas. EMFText fue diseñado para ser compacto y declarativo. Sigue el principio de "convención sobre configuración" al proporcionar varios valores por defecto como la derivación automática de una especificación de sintaxis concreta basado en HUTN (del inglés, Human Usable Textual Notation) al estilo Java [75]. Además, EMFText está bien integrado en la interfaz de usuario de Eclipse al proporcionar acciones y asistentes que pueden ejecutarse desde menús contextuales y menús de ficheros. En contraste, el lenguaje de especificación Xtext es más flexible y más cercano a la gramática ANTLR generada. Admite predicados sintácticos y algún tipo restringido de acciones semánticas para influir en la construcción de AST (del inglés, Abstract Syntax Tree).

TCS: Textual Concrete Syntax es un componente de Eclipse/GMT que permite la especificación de una sintaxis textual de un DSLs adjuntando información sintáctica de los meta-modelos [56]. Las especificaciones expresadas con TCS se utilizan para generar automáticamente herramientas para transformaciones (T2M y M2T) [86]. La semántica de TCS se define en relación al lenguaje de meta-modelado KM3, que tiene conceptos similares a los encontrados en MOF [12] pero más simples. TCS ofrece un editor de Eclipse que cuenta con resaltado de la sintaxis, un esquema, e hipervínculos por cada DSL definido. TCS al igual que Xtext también utiliza ANTLR para generar el parser. Como semejanza entre las herramientas señaladas, Xtext, EMFText y TCS son DSLs diseñados para describir lenguajes textuales que necesitan de un meta-modelo para ser parseados eficientemente, al contrario que el estándar XMI (XML Model Interchange) [86].

2.2.4 Ejemplo: creación de un DSL sencillo

La Figura 2.17 muestra algunos de los elementos de un DSL para modelar máquinas de estado. Para ello se ha utilizado EMF como entorno de desarrollo, Ecore para el diseño del meta-modelo y OCL como lenguaje para incluir ciertas restricciones en el diseño del meta-modelo. Para crear un editor textual se ha utilizado Xtext, y Sirius para crear un editor gráfico.

El meta-modelo está compuesto por la clase (EClass en EMF) *StateMachine* que actúa como una clase contenedora raíz de las clases *Transition* y la clase abstracta *AbstractState*. De esta última clase hereda la clase *State* para representar los estados inicial, final y estándar. También de *AbstractState* hereda la clase *Composite* para representar estados compuestos (estados que contienen otros estados).

El meta-modelo también implementa a modo de ejemplo cuatro restricciones OCL para especificar con precisión detalles del diseño de una máquina de estados. Con la etiqueta 1 en la Figura 2.17, se señala la expresión OCL que indica que "todos los estados de una máquina de estados deben tener un valor en el atributo *label* (identificador del estado)". La restricción tiene efecto en el modelo siempre que el atributo *label* tenga cardinalidad [0..1], si *label* tuviera cardinalidad [1..1], la restricción no sería necesaria debido a que internamente EMF consideraría un error debido a que se estaría violando una restricción de integridad en el diseño del meta-modelo. La expresión OCL de la etiqueta 2 señala que "todos los estados deben tener diferente *label*". La expresión OCL de la etiqueta 3, señala que "un estado inicial no puede ser a la vez un estado final e inicial". Finalmente, la expresión de la etiqueta 4 señala que "una máquina de estados únicamente tiene un estado inicial".

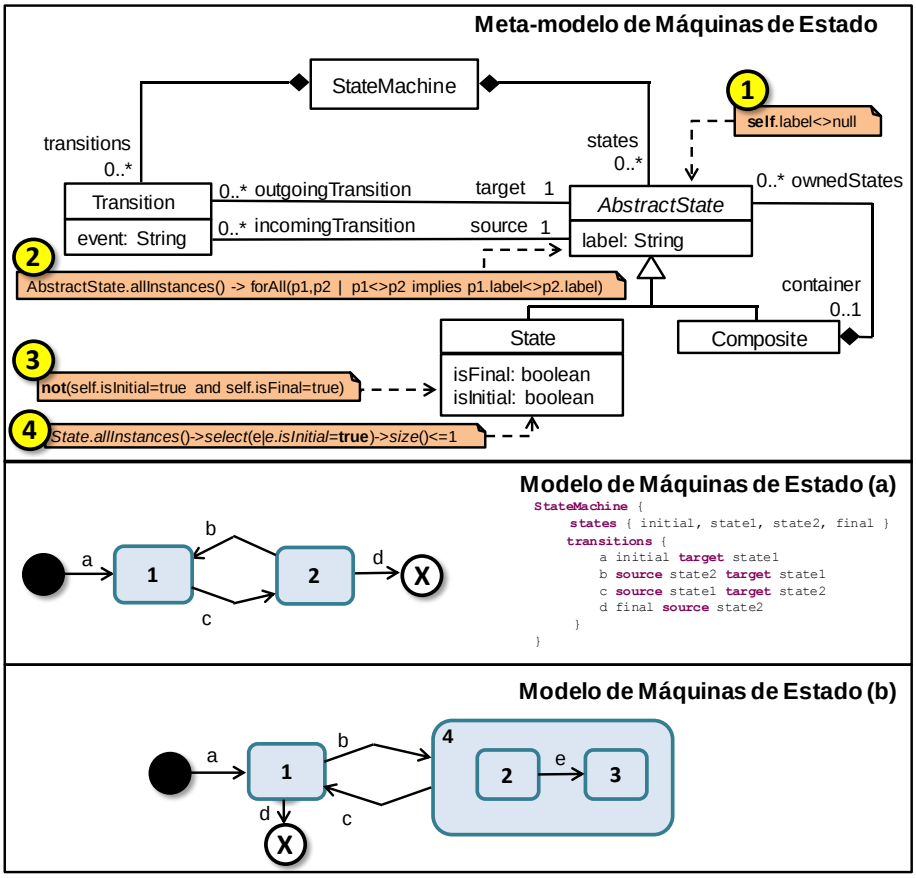


Figura 2.17: Meta-modelo de máquina de estados.

En la segunda sección de la figura, con etiqueta (a), se muestra un modelo de máquina de estados conforme el meta-modelo de la parte superior. En la parte izquierda de esta sección, el modelo se representa con una sintaxis concreta gráfica, donde el estado inicial es representado con un círculo negro, el estado final con un círculo con una *X* en el centro, los estados estándar son representados mediante un círculo con una etiqueta que los identifica, mientras que las transiciones son representadas mediante una flecha con una etiqueta que también las identifica. En la parte derecha de la misma sección, el modelo se representa con una sintaxis concreta textual, donde es fácil interpretar que la máquina de estados está compuesta por los estados *initial*, *state1*, *state2* y *final;* transiciones *a*, *b*, *c* y *d*. De igual manera se puede interpretar que la transición *b* tiene como origen el *state2* y destino el *state1*.

Finalmente, en la tercera sección de la figura, con etiqueta (b), se muestra otro modelo de máquina de estados, donde a diferencia de la anterior, se tiene un estado compuesto (estado 4), en el cual la relación de composición del meta-modelo se muestra como contención física (un estado dentro de otro).

2.3 Resumen y conclusiones

Este capítulo ha ofrecido una visión general sobre los fundamentos de la Ingeniería Dirigida por Modelos (MDE), comenzando por los conceptos esenciales de este paradigma de desarrollo de software. Inicialmente, se resalta que MDE considera el uso sistemático de modelos en la cadena de producción de software, en lugar de programas realizados en un determinado lenguaje de programación. Para entender esta tecnología de desarrollo de software se deben tener nociones sobre modelos, meta-modelos y lenguajes de meta-modelado, así como la arquitectura de cuatro niveles de la OMG.

Debido a que el mayor porcentaje de desarrollo de software dirigido por modelos se realiza en Eclipse, y que para este entorno se encuentran disponibles la mayoría de herramientas, este capítulo ha revisado tecnologías MDE basadas en Eclipse.

Adicionalmente, este capítulo ha hecho una introducción al modelado multi-nivel, una forma de modelado que extiende el modelado estándar expresado en dos meta-niveles, a varios meta-niveles con la finalidad de contar con modelos más simples, especialmente cuando en el modelo se presenta el patrón tipo-objeto o alguna de sus variantes.

3 Trabajos relacionados

En esta este capítulo se analiza el estado del arte de extensión de meta-modelos y modelado multi-nivel. Respecto a modelado multi-nivel, primero se realiza una caracterización del espacio de diseño de los diferentes enfoques, posteriormente sobre esa base se proporciona una comparación detallada de las herramientas existentes, identificando oportunidades de investigación.

3.1 Extensión de meta-modelos

Las diferentes etapas del proceso desarrollo de software generan varios productos. Un producto software de acuerdo al estándar IEEE 610-1990 [78], es un conjunto completo de programas computacionales, procedimientos, documentación asociada, y datos designados para ser entregados al usuario. En MDE estos productos pueden ser representados utilizando lenguajes de modelado. Sin embargo, elegir el lenguaje más adecuado para cada producto específico no es una tarea fácil.

Con respecto al modelado, el estándar por defecto es UML [137, 110]. Este proporciona una notación de modelado orientado a objetos estándar, adecuada para la mayoría de aplicaciones, pero ciertamente no para todas. En este punto, los DSLs logran proporcionar un mejor ajuste a una tarea en particular. Sin embargo, al utilizar una notación particular para una determinada aplicación, esta notación será conocida únicamente por un pequeño grupo de diseñadores y expertos de dominio, dándose el caso de que muchos DSLs puedan ser conceptualmente similares, pero variando considerablemente en términos de notación, creando así una especie de Torre de Babel innecesaria y deshaciendo el logro consolidado por UML [113].

En la práctica cuando un experto de dominio decide utilizar un DSL para resolver algún problema, tiene que utilizar el DSL que se encuentre disponible, pero ¿qué pasa cuando el usuario considera que el DSL no se ajusta plenamente a sus requerimientos?, en este caso será el experto en modelado el que tenga que realizar las modificaciones correspondientes.

La mayoría de DSLs son fijos, es decir, no admiten la posibilidad de personalización para que se ajusten a ciertos requerimientos. No obstante, los DSLs deben evolucionar continuamente para adaptarse a las necesidades cambiantes del dominio que representan [145]. Desafortunadamente, en este momento de la tecnología, la evolución de los DSLs es un proceso costoso y tedioso que requiere cambiar el meta-modelo del DSL y la recreación de todo el entorno de modelado [101]. En este caso es deseable de los DSLs cierta flexibilidad para cubrir estos nuevos requerimientos. Atkinson en [12] señala que un DSL que posea características de flexibilidad se denomina *Lenguaje de Dominio Personalizado* (DCL, del inglés Domain-Customized Language). Un DCL por tanto será un lenguaje derivado cuya definición se basa en un lenguaje base existente. El lenguaje se personaliza para algún propósito específico usando operaciones tales como *modificación, extensión* y *reducción*. Por tanto, un DCL es una subclase de un DSL cuyos miembros tienen relaciones cuidadosamente definidas con respecto al lenguajes base, como se muestra en la Figura 3.1.

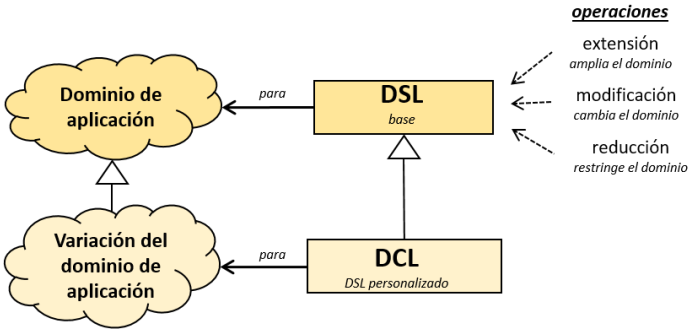


Figura 3.1: Lenguaje de Dominio Personalizado.

Se debe aclarar que la reducción del DSL debe conducir no a una subclase sino a un subconjunto de modelos. Por lo general, el lenguaje base será un estándar ampliamente conocido y la cantidad de personalización se minimizará, en el sentido de que la mayor parte posible del lenguaje base se mantendrá inalterado. El enfoque de la OMG soporta la personalización de UML a través de los *perfiles* [28], lo que se puede considerar como una promoción del concepto de DCL. Sin embargo, un enfoque DCL solo será económicamente viable si el costo de crear un DCL a partir de un lenguaje base (lo que implica adaptar su sintaxis y semántica) supera el retorno de la inversión, es decir, las ganancias de productividad. Por lo tanto, un enfoque DCL necesita hacer que la definición y derivación de lenguajes sea lo más simple e intuitiva posible. De acuerdo a Atkinson [12] solo los enfoques de personalización del lenguaje y las tecnologías que logran minimizar el costo de crear un DCL tendrán un futuro a largo plazo en ingeniería de software.

La personalización del DSL considera varios criterios, como los señalados en la taxonomía de variabilidad de modelos señalados en Cengarle y colaboradores en [32], y que están relacionados a la variabilidad de la sintaxis abstracta, sintaxis concreta y semántica, elementos con los que se define

formalmente un DSL. La variabilidad puede ser especificada mediante reglas en la definición del lenguaje o de forma externa. Estas reglas de variabilidad conducen a que a partir de un DSL origen se puedan obtener varias versiones personalizadas del DSL a través de un proceso de personalización metódico.

En el mismo contexto, se debe considerar que el uso de MDE en proyectos reales implica tratar con diferentes modelos, meta-modelos y herramientas de apoyo. Pero se debe tomar en cuenta que los requisitos son cambiantes, lo que ocasiona problemas, a menudo costosos, en términos de trazabilidad, coherencia o interoperabilidad. Es por ello, que se requieren soluciones que aseguren una mejor adaptabilidad y flexibilidad de las herramientas de modelado [29]. Como los meta-modelos son la piedra angular en tales herramientas, la capacidad de extensión del meta-modelo es fundamental. Sin embargo, los frameworks de modelado actuales no son lo suficientemente flexibles o dinámicos.

Es por eso que, tomando como base los aspectos relacionados con la personalización de los DSLs y considerando que uno de los objetivos del presente trabajo ha sido proponer un enfoque que permita controlar la extensión y adaptación de meta-modelos, se presenta a continuación algunos esfuerzos relacionados a ésta área.

Bruneliere en [29] propone la adaptación de meta-modelos a través de un DSL textual que permite la definición de extensiones del meta-modelo. Las extensiones son creadas por expertos de modelado como anotaciones del meta-modelo en el momento de su desarrollo. El trabajo define un conjunto de operadores de extensión que admiten la creación de nuevas clases desde cero o a partir de especialización y generalización de clases, añadir y filtrar (borrar) propiedades y restricciones, también es posible el filtrado de clases existentes. Los autores lo definen como un mecanismo de extensión ligera (lightweight), con *preservación de compatibilidad* ya que un modelo generado con el meta-modelo extendido se ajusta al meta-modelo original. Los autores consideran que este mecanismo de extensión es transparente, no intrusivo y que permite cierto grado de dinamicidad. A diferencia del enfoque que se presenta en la tesis, estos mecanismos son extensiones concretas de un meta-modelo. El enfoque propuesto en esta tesis define las reglas que debe cumplir cualquier extensión, por tanto, puede verse como complementario a ese trabajo.

Braun en [28] señala que los perfiles UML proporcionan un medio para personalizar modelos UML a dominios y plataformas particulares. Los mecanismos de extensión permiten refinar la semántica estándar de manera estrictamente aditiva, evitando que ésta se contraiga [1]. Por tanto, constituyen un mecanismo de extensión ligera, donde es posible extender el meta-modelo sin sobrescribir los elementos originales [147]. Es decir, se respeta la sintaxis y la semántica original [65]. Los perfiles UML se definen utilizando *estereotipos* (stereotypes), *valores etiquetados* (tagged values) y *restricciones* (constraints) que se aplican a elementos específicos del modelo, como clases, atributos, operaciones y actividades. Un perfil es una colección de dichas extensiones que personalizan colectivamente UML para un dominio en particular (por ejemplo, salud, aeroespacial, financiero) o plataforma (J2EE, .NET) [1]. Langer en [101] adopta la filosofía de extensión de UML en base a *perfiles*, pero en el ámbito de DSLs en el entorno EMF/Ecore. A través del mecanismo de extensión propuesto denominado *EMF Profiles*, es posible extender DSLs de manera no invasiva al definir perfiles de la misma manera que lo

hacen las herramientas UML. Este mecanismo permite definir perfiles genéricos y meta-perfiles para reutilizar perfiles en varios DSLs los perfiles definidos. El mecanismo también es aplicable en otros frameworks de meta-modelado que soporten una arquitectura de meta-niveles similar a EMF. EMF Profiles ha sido implementado como un plug-in de Eclipse para EMF, utilizado GMF como framework de modelado gráfico. Relacionado con el enfoque de la tesis, los perfiles aplicables a UML EMF pueden verse como extensiones concretas de un meta-modelo, pero no como reglas que regulan su extensión.

En [27], Braun y Esswein exploran varios mecanismos de extensión del meta-modelo MOF, basados en una analogía con los principios de la extensión en el campo de la ingeniería del software, tales como: *hooking* (enganche), *aspects* (aspectos), *plug-ins* y *add-ons* (complementos). Para ello, se realiza un análisis de cada uno de los mecanismos señalados.

- ***Hooking:*** también conocido como puntos de extensión (extensión points), consiste en dejar abiertas partes de un programa para definirlas y especificarlas posteriormente con clases, interfaces o métodos que deben ser concretados para la inyección del código. Otras partes del software permanecen fijas, y son llamadas frozen points (puntos congelados) en [99]. El punto de extensión es adaptable, ya que permite definir y formular puntos abiertos en los meta-modelos para su posterior concretización. A través del punto de extensión, es posible completar construcciones específicas únicamente relacionadas con elementos del meta-modelo original, por ejemplo, las clases pueden especializarse a través de subtipos. También es posible extender atributos y expresiones OCL, así como cambiar el nombre de un elemento del meta-modelo para ajustarlo a un contexto particular.
- ***Aspects:*** están relacionados a los principios de la programación orientada a aspectos. Tienen como objetivo centralizar responsabilidades transversales y proporcionarlas dinámicamente en los lugares solicitados [99]. En el contexto de extensibilidad de meta-modelos, un aspecto se entiende como una extensión, que contiene clases de extensión particulares. Esto significa que, si se crea una instancia de una clase original, entonces el aspecto también puede ser instanciado.
- ***Plug-ins:*** son extensiones funcionales de una aplicación a través de interfaces bien definidas [162]. Muchas aplicaciones de hoy en día están hechas enteramente de plug-ins. Por ejemplo, sistemas basados en componentes, como Eclipse soportan extensiones mediante plug-ins. Este campo ha madurado bastante en los últimos años, debido a que antes de reinventar una nueva solución, se construye sobre lo existente para poderlo integrar a la nueva solución, escribiendo solo código especializado. Las nuevas arquitecturas basadas en plug-in son muy atractivas para los desarrolladores porque pueden centrarse en proporcionar funcionalidad modular a los usuarios. Los desarrolladores puede personalizar rápidamente las aplicaciones mezclando y combinando los plug-ins que necesitan [21]. Por lo tanto, un plug-in de un meta-modelo se entiende como un modelo consistente, coherente e independiente, que puede mejorar la expresividad de un lenguaje de modelado. A través de los plug-ins se pueden agregar nuevos elementos conceptuales al meta-modelo

- ***Add-ons:*** son similares a los plug-ins, pero poseen capacidades más bien limitadas y tiene un alcance conceptual más pequeño. Los add-ons tienen la misma arquitectura básica que los plug-ins, exceptuando dos puntos: la complejidad interna de los add-ons es limitada y un add-on no puede ser instanciado o ejecutado solo. Con respecto a los meta-modelos, los add-ons son utilizados para establecer una dependencia fuerte entre los nuevos atributos y conceptos con los conceptos originales, por tanto, son utilizados para agregar nuevos atributos al meta-modelo.

A modo de resumen, se puede concluir señalando que cada mecanismo de extensión propuesto es utilizado para realizar tareas concretas al extender el meta-modelo. Así las técnicas de hooking y add-ons se encuentran entretejidas con el meta-modelo original, mientras que los plug-ins constituyen componentes aislados y definidos por separado. Algunos mecanismos requieren la definición de interfaces específicas o la declaración de elementos extensibles. Esta declaración se realiza en el primer paso del proceso de extensibilidad del meta-modelo. En el siguiente paso, el contenido de la extensión concreta se aplica al meta-modelo. En el último paso, las clases del meta-modelo original y las clases de extensión deben fusionarse (por ejemplo, a través de un hooking) o simplemente conectarse (por ejemplo, plug-ins) para dar forma a una versión unificada del meta-modelo, que constituye la extensión del lenguaje. Finalmente se debe señalar que la propuesta de extensibilidad planteada en esta tesis permite especificar de manera precisa qué extensiones están permitidas para un meta-modelo.

Schleicher [150] señala que UML se utiliza como lenguaje universal para modelar aplicaciones orientadas a objetos para una amplia gama de dominios, pero ciertamente no para todos los dominios. Por esta razón se han realizado adaptaciones a UML, muchas veces superado sus propios mecanismos de extensión. Se señala que, para realizar tareas como la verificación de modelos y generación de código, el meta-modelo de UML debe tener ciertas características de flexibilidad. Se considera que un meta-modelo específico del dominio serviría como una definición formal de una extensión de UML para el dominio de modelado. Ante este escenario, se requeriría agregar más profundidad semántica al meta-modelo estándar. Una forma de hacerlo es especializando los propios mecanismos de extensión estándar de UML. En [17] se presenta un esquema de clasificación de cuatro tipos de uso de estereotipos UML, lo que proporciona una mejor facilidad para meta-modelado. Estos tipos son:

- ***Estereotipo decorativo (decorative):*** utilizado para la manipulación de la sintaxis concreta, puede ser utilizado para sustituir un símbolo de un elemento del modelo. Se utiliza para adaptar la notación de UML a un dominio específico.
- ***Estereotipo descriptivo (descriptive):*** utilizado para introducir nuevos elementos pragmáticos que no cambian la semántica de UML. Son una clasificación secundaria de un elemento válido de UML. Dentro de un modelo UML una clase estereotipada con un estereotipo descriptivo sigue siendo una instancia del elemento original del meta-modelo de UML.
- ***Estereotipo restrictivo (restrictive):*** son nuevos elementos semánticos añadidos a UML. Son miembros de primera clase del nuevo lenguaje e incluyen una definición formal de las

restricciones sintácticas y semánticas de su uso dentro de un modelo. Sin embargo, no cambian el lenguaje base y su semántica solo puede extenderse.

- ***Estereotipo de redefinición (redefining):*** proporciona los medios para reemplazar cualquier elemento del meta-modelo a través de uno nuevo, definiendo un conjunto completamente diferente de restricciones para ello. Esto induce a cambios radicales en el lenguaje original y resulta en la definición de un nuevo lenguaje de modelado.

A diferencia, del enfoque señalado, el enfoque planteado en la tesis podría ser complementario, permitiendo la definición de reglas precisas para la aplicación de los tipos de funcionalidad considerados por cada tipo de estereotipo.

Jiang en [84] señala que los mecanismos de extensión de UML han sido ampliamente utilizados por expertos en modelado durante muchos años, principalmente debido a que el mecanismo de extensión del meta-modelo ha demostrado ser lo suficientemente poderoso, ya que su capacidad de expresión puede variar desde lo muy limitado a algo de muy amplio alcance. Los usuarios pueden usarlo para ampliar tanto la sintaxis como la semántica de UML en general. Por lo señalado, se definen cuatro niveles del mecanismo de extensión del meta-modelo de UML. En dicho trabajo [84], Jiang analiza la legibilidad y y comprensibilidad, el alcance de uso y soporte de herramientas en base a definiciones precisas de niveles de extensión, como se señala en la Figura 3.2.

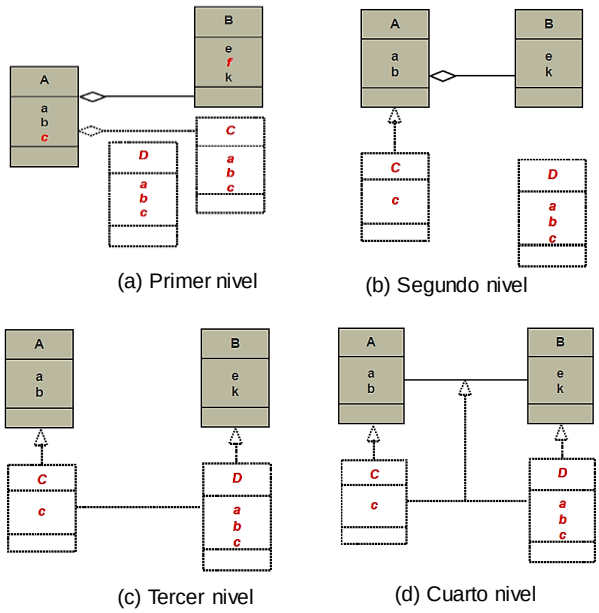


Figura 3.2: Los cuatro niveles de extensión del meta-modelo.

- ***Primer nivel:*** en este nivel de extensión del meta-modelo, los desarrolladores pueden extender el meta-modelo original a su voluntad. Por ejemplo, se pueden agregar atributos, operaciones y relaciones directamente al meta-modelo original sin restricciones.

- ***Segundo nivel:*** los desarrolladores no pueden modificar construcciones en el meta-modelo original, pero puede agregar nuevas construcciones sin ninguna restricción.

- ***Tercer nivel:*** el meta-modelo extendido cumple con las restricciones del segundo nivel de extensión de meta-modelo, y todas las construcciones nuevas deben tener construcciones padres en el meta-modelo original.
- ***Cuarto nivel:*** el meta-modelo extendido satisface las restricciones del tercer nivel de extensión del meta-modelo, y además las nuevas construcciones no deben tener nuevas relaciones entre sí, excepto si son especialización de una relación existente entre las construcciones originales.

Los diferentes niveles establecen restricciones sucesivas, limitando las adaptaciones posibles. Cuando se extiende el meta-modelo UML de acuerdo a unos requisitos de modelado, se puede seleccionar un nivel de extensión apropiado para hacer que la extensión sea más razonable y operativa. Se debe señalar, que la propuesta de extensibilidad planteada en la presente tesis permite establecer normas concretas de extensibilidad, que pueden cumplir cada uno de los niveles mencionados. Además, realiza un número mayor de operaciones de extensibilidad a través de reglas de extensibilidad aplicables a cualquier meta-modelo, no necesariamente el de UML.

Perrouin en [140], ofrece una visión en la que los diferentes tipos de modelos se diseñan como familias en lugar de forma independiente. Toma como referencia las técnicas de la Línea de Productos de Software [143] como estrategia para expresar la variabilidad de un DSL . De esta manera, proponer anotar los elementos del meta-modelo con condiciones de presencia. Estos elementos se activarán de acuerdo a la configuración seleccionada por el usuario, dando lugar así a distintos meta-modelos de manera automática. El objetivo de una LPS no es el desarrollo de un producto, sino el de un conjunto de productos a partir de un repositorio común de activos base (core assets) a través de la configuración de puntos de variación de esos activos [142]. Esos enfoques no admiten una especificación de cómo y si ciertos elementos se pueden ampliar (subclasificar) o refinar (en el caso de referencias). Por lo tanto, son un enfoque *cerrado* para la reutilización (donde se elige un determinado producto de la línea de productos), mientras que la propuesta de esta tesis conduciría a un enfoque *abierto* para la reutilización, donde un meta-modelo base se adapta y luego se extiende de forma controlada. Es de considerar que un repositorio de componentes de DSLs, con indicaciones de cómo estos pueden ser reutilizados (extendidos, restringidos, adaptados) facilitaría la tarea de construir nuevos DSLs, o más generalmente, la reutilización flexible de artefactos asociados a meta-modelos. En este escenario, se requeriría una adaptación conjunta del artefacto asociado. Es decir, la visión de extensión de meta-modelos en base a su variabilidad (LPS), expresa un conjunto cerrado y predefinido de variantes de DSL, mientras que las reglas de extensión propuestas en esta tesis permiten describir un conjunto posiblemente infinito de extensiones, todas las cuales deben obedecer a las reglas de extensibilidad. Por lo tanto, se argumenta que el enfoque constituye un mecanismo abierto pero controlado para la extensión.

Atkinson [5] señala que las capacidades de un lenguaje de modelado pueden extenderse utilizando mecanismos de extensión integrado en los mismos lenguajes, tales como los de UML [137], BPMN [26] y ArchiMate [102]. Sin embargo, existen algunos casos en los que los mecanismos de extensión son externos al lenguaje. Se señala también, que las herramientas de modelado y frameworks disponibles en la actualidad soportan en esencia tres enfoques fundamentales para la extensión de lenguajes de modelado:

- ***Personalización del meta-modelo:*** este enfoque señala que cuando se pretende realizar la extensión de un lenguaje, lo primero que se debe cambiar es el meta-modelo del lenguaje. Sin embargo, en la práctica esta tarea no es admitida por la mayoría de frameworks y herramientas disponibles en la actualidad, ya sea porque los meta-modelos están fuertemente integrados, o porque no son accesibles a cambios en tiempo de ejecución, o quizás porque las extensiones del lenguaje modifican el meta-modelo de tal manera que no se puede utilizar directamente en la herramienta de modelado. En este último caso, una vez que un meta-modelo se ha personalizado, la herramienta modificada necesita volver a compilarse para que sea consciente de los cambios. Esto no sólo es una tarea tediosa y propensa a error, sino que a menudo tiene que ser objeto de mantenimiento para mantener las instancias del modelo en sincronía con el meta-modelo modificado.
- ***Mecanismo incorporado al Lenguaje (built-in):*** este es el caso de UML que cuenta con los *perfiles* como su propio mecanismo de extensión y dentro de éste a los *estereotipos*, *valores etiquetados* y *restricciones*. Estos mecanismos de extensión no modifican la estructura original del meta-modelo de UML, simplemente producen el efecto de personalización de su meta-modelo sin cambiarlo realmente.
- ***Modelo de anotaciones****:* las herramientas de algunos lenguajes de modelado no ofrecen mecanismos de extensión incorporados, o si lo hacen, estos mecanismos no se aplican directamente por las herramientas. En estos casos, se pueden emplear otras técnicas de extensibilidad, como el *modelo de anotaciones,* que es un modelo externo vinculado al modelo a extender a través de referencias y técnicas "weaving", como las descritas en [20].

En [5], autores identifican las fortalezas y debilidades de los tres enfoques de extensibilidad señalados anteriormente y proponen un enfoque alternativo a través del modelado multi-nivel, que eliminaría las debilidades de los mecanismos descritos anteriormente. El enfoque de extensibilidad propuesto en esta tesis, permite emular el modelado multi-nivel en entornos de dos meta-niveles.

En resumen, se observa que existen varios mecanismos utilizados para extender modelos y meta-modelos, resumidos en la Tabla 3.1. Uno de los más importantes, debido a la influencia de UML en el entorno de desarrollo de software, son los perfiles UML. Sin embargo, este mecanismo únicamente permite la extensión de los elementos originales del meta-modelo (extensibilidad ligera). Tomando como base el mecanismo de extensibilidad de UML han surgido también otros enfoques, que han sido trasladados al entorno de DSLs/EMF. También es posible apreciar que existen otros enfoques de extensibilidad, como los que han tomado como base los diferentes conceptos y mecanismos de extensibilidad utilizados en la ingeniería de software, otros han tomado las ideas de línea de productos de software, y también hay enfoques de extensibilidad en base al modelado multi-nivel.

Tabla 3.1: Enfoques de extensibilidad de lenguajes de modelado/meta-modelado.

Propuesta / Criterio	Bruneliere [29]	Braun [28]	Langer [101]	Braun y Esswein [27]	Schleicher [150]	Jiang [84]	Perrouin [140]	Atkinson [5]	de Lara [46]
Tecnología	EMF	UML	EMF	EMF/MOF	UML	UML	Cualquiera	EMF	EMF/MetaDepth
Mecanismos	Extensiones concretas del meta-modelo	Perfiles	Perfiles EMF	Técnicas de extensibilidad utilizadas en ingeniería de software	Estereotipos	Perfiles	Línea de productos de software	Multi-nivel	Emulación de modelado multi-nivel en entornos de dos meta-niveles
Tipos de extensión/ personalización	Operadores de extensión: -creación de nuevas clases desde cero, o a partir de especialización y generalización de clases -añadir, filtrar (borrar) propiedades -añadir y filtrar restricciones -filtrar clases existentes	-Estereotipos: añadir nuevos elementos -valores etiquetados: añadir nuevos atributos -restricciones: a través de OCL	-Estereotipos: añadir nuevos elementos -valores etiquetados: añadir nuevos atributos -restricciones: a través de OCL	-hooking -aspects -plug-ins -add-ons	Tipos de estereotipos: -decorativo -descriptivo -restrictivo -redefinido	Cuatro niveles de extensibilidad	Configuración de la línea de producto para la obtención de un producto	Personalización mediante instanciación	Personalización mediante instanciación
Otras características	A través de un DSL textual se introducen anotaciones en el desarrollo del meta-mo_ delo a ser extendido	Mecanismos de extensión estándar utilizados en UML	Se adopta la filosofía de perfiles de UML, pero en el ámbito de DSLs/EMF	Adapta mecanismos de extensión de la ingeniería de software para extender meta-modelos	Propuesta de clasificación de estereotipos de UML de acuerdo a su entorno de aplicabilidad	Clasificación de la extensión de meta-modelos UML	Se adaptan los métodos de línea de productos. Se da soporte a la variabilidad cerrada de lenguajes	Conceptos implementados en herramienta "Melanee"	Algunos de los conceptos implementados en herramienta "MetaDepth"
Tipo de extensión	Pesada (heavyweight)	Ligera (lightweight)	Ligera	Ligera	Ligera	Pesada (heavyweight)	-	Pesada	Pesada

Es de notar que el modelado multi-nivel permite eliminar ciertas restricciones de los otros mecanismos, como el hecho de que se permiten realizar modificaciones del tipo pesado (heavyweight), donde es posible, por ejemplo, añadir nuevos elementos que no son parte de la sintaxis abstracta del lenguaje de modelado de manera rápida, fácil y dinámica. Otra de las ventajas de este enfoque, es que puede ser aplicado a diferentes entornos de modelado, diferentes a UML.

En la siguiente sección se realiza una revisión de los diferentes enfoques y herramientas utilizadas en modelado multi-nivel.

3.2 Modelado multi-nivel

En esta sección se realiza una caracterización de los enfoques de modelado multi-nivel a través de un modelo de características. Posteriormente, se realiza una clasificación de las herramientas multi-nivel más relevantes.

3.2.1 El espacio de diseño de soluciones de modelado multi-nivel.

Algunos investigadores han propuesto pautas para comparar y clasificar los enfoques de modelado multi-nivel [6, 9]. Sin embargo, la comunidad de modelado multi-nivel no ha propuesto un análisis sistemático de las decisiones de diseño para los sistemas de modelado multi-nivel. Dicho análisis podría servir como base para comparar y clasificar enfoques multi-nivel e identificar omisiones en la investigación actual. La Figura 3.3 muestra un *modelo de características* [87] que representa el espacio de posibles alternativas para realizar soluciones de modelado multi-nivel. A continuación, se describen estas características.

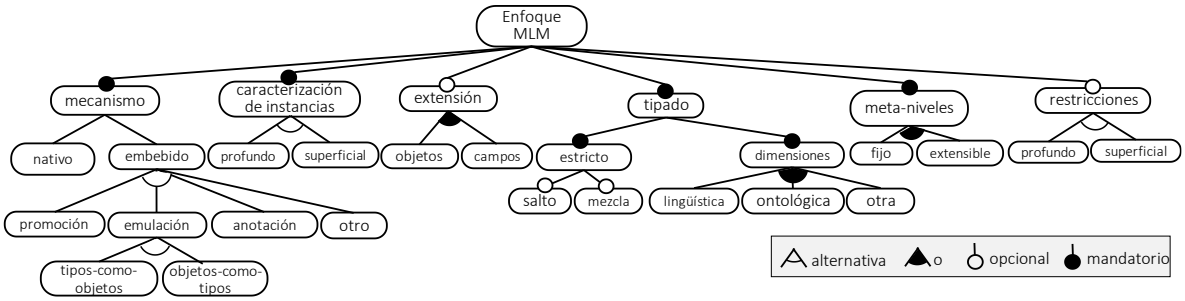


Figura 3.3: Caracterización del espacio de diseño de enfoques de modelado multi-nivel.

Mecanismo: en primer lugar, el mecanismo de modelado multi-nivel puede ser nativo o estar embebido o integrado en una tecnología de dos niveles. Como se vio en la sección 2.1.7.4 (modelado multi-nivel), los mecanismos nativos no incorporan el modelado multi-nivel dentro de un framework de dos niveles, sino que proporcionan primitivas y artefactos diseñados específicamente para el modelado multi-nivel. Por ejemplo, MetaDepth [40] proporciona un mecanismo nativo respaldado por un meta-modelo lingüístico que modela explícitamente la relación de tipado e incluye la noción de potencia.

Con respecto a los enfoques embebidos, hay varias alternativas para emular múltiples meta-niveles dentro de un framework de dos niveles, algunos de los cuales se han analizado en la sección 2.1.7 (modelado multi-nivel mediante ejemplos). Una posibilidad es recurrir a transformaciones de promoción, como se vio en la sección 2.1.7.3. Los enfoques que utilizan transformaciones de promoción pueden ser de dominio específico, por ejemplo, para trazabilidad [51, 2] o de propósito general [105]. Otro enfoque, visto en las secciones 2.1.7.1 y 2.1.7.2, consiste en emular la *faceta de tipo en objetos* (característica *tipos-como-objetos*), o la *faceta de objeto en tipos* (característica *objetos-como-tipos*). El primer enfoque se conoce como el *patrón tipo-objeto* [106], mientras que el segundo es el enfoque de extensión del meta-modelo, también conocido como el enfoque estático [46]. Usando el patrón tipo-objeto, el modelo multi-nivel es embebido en el nivel del modelo, y la relación de instanciación se modela como una relación normal. En cambio, en el enfoque estático, el modelo multi-nivel está integrado en el meta-modelo y la instanciación se emula con herencia. Los enfoques basados en anotaciones (característica "anotación" en el diagrama de la Figura 3.3) anotan el meta-modelo del lenguaje subyacente (por ejemplo, el UML) con información sobre las características que los elementos en el modelo de dominio tienen en diferentes niveles. Por ejemplo, los estereotipos

UML son un mecanismo basado en anotaciones [137]. En [46] se pueden ver varios ejemplos de modelado multi-nivel utilizando estereotipos.

Si bien estos son los mecanismos más comunes para integrar modelos multi-nivel en frameworks de dos niveles según [46], el modelo de características no está cerrado, lo que indica que también pueden existir otros mecanismos (característica "otro" en el diagrama).

Caracterización de instancias: algunos enfoques para el modelado multi-nivel pueden caracterizar instancias en niveles inferiores, no solo en el siguiente (como sucede en el modelado estándar de dos niveles). Esto se denomina *caracterización profunda* (*deep characterization)* [8]. La *potencia* (potency) es uno de esos mecanismos para la caracterización profunda. En contraste, otros enfoques sólo admiten la definición de características para instancias que se encuentran exactamente en el nivel inmediatamente inferior. Es decir, soportan la *caracterización superficial* (*shallow characterization*). Por ejemplo, los enfoques multi-nivel basados en *powertypes* no admiten la caracterización profunda [6]. Un powertype es un tipo, cuyas instancias son subclases de otro tipo [123].

Extensión: como se mencionó en la sección 2.1.7, cuando se diseña un modelo multi-nivel, es difícil, si no imposible, prever todas las posibles primitivas que se utilizarán en el nivel inferior. Esto es especialmente cierto cuando se diseñan lenguajes genéricos (por ejemplo, lenguajes de modelado de procesos), aplicables a varios dominios (por ejemplo, modelado de procesos software, modelado de procesos de producción, etc.). En estos casos, los diferentes dominios pueden introducir nuevas primitivas o añadir nuevos campos a los objetos, no previstos por sus tipos. En el ejemplo de la sección 2.1.7, se requiere agregar nuevos campos (como *dat*) a las instancias de *Message*. De esta manera, algunos enfoques permiten extensiones lingüísticas, es decir, clabjets o nuevos campos sin tipo ontológico.

Tipado: a menudo, los enfoques de meta-modelado abogan por un *tipado estricto* [61], en el que cada elemento en cada nivel (excepto en el nivel superior) es tipado por exactamente un elemento en el nivel superior. Esto implica tres aspectos: la relación de tipado no salta meta-niveles (característica “sin salto” en el diagrama), las instancias y sus tipos no coexisten en el mismo nivel (característica "sin mezcla" en el diagrama) y no existen elementos sin tipo (característica "no extensión"). El meta-modelado *no estricto* a veces se denomina meta-modelado *libre* (*loose*) [61]. En esta tesis, se es más preciso en cuanto a la nomenclatura, al identificar la causa del meta-modelado no estricto. De esta manera, se modela el salto (*jump*) y la mezcla (*mix*) como dos subcaracterísticas del tipado *no estricto*, mientras que el soporte de los elementos sin tipo son modelados por la característica de *extensión*.

Además, los enfoques pueden soportar diferentes clases de tipado, como los tipados ontológicos o lingüísticos (véase la sección 2.1.7). Si bien estas son las dimensiones más comunes, algunos trabajos han propuesto otras, como la *dimensión de realización* (realization) [79]. Se debe tener en cuenta que además de las dimensiones, algunos enfoques permiten varias relaciones de tipado del mismo tipo, normalmente ontológico [41]. Sin embargo, no se capta este hecho en el modelo de características, por su menor relevancia para el modelado multi-nivel.

Meta-niveles: los sistemas de modelado multi-nivel soportan el modelado utilizando más de dos meta-niveles. Sin embargo, en algunos sistemas, el nivel de un modelo tiene que ser decidido cuando se construye dicho modelo, y esta decisión no puede cambiarse cuando se trabaja con modelos en los niveles inferiores. Por ejemplo, este es el caso si se asigna una potencia dada (por ejemplo 3) a un modelo. Otros enfoques son más flexibles y permiten ampliar el número de meta-niveles al modelar en los niveles inferiores. Es el caso, por ejemplo, de los enfoques que permiten la potencia *ilimitada* (*star* o *unbounded*), representada por "*" [40, 91], o la reciente noción de *potencia de caracterización* (characterization-potency) [97].

Restricciones: en el modelado de dos niveles, las restricciones de integridad definidas en los meta-modelos se aplican a los modelos exactamente en un nivel inferior. Estas restricciones incluyen restricciones de cardinalidad, restricciones asociadas a construcciones como composiciones, y otras definidas en lenguajes de restricciones como OCL [133]. En un entorno multi-nivel, se pueden definir restricciones para el siguiente meta-nivel, *restricciones superficiales* (shallow constraints); pero también es útil definir restricciones para modelos en meta-niveles por debajo del siguiente, *restricciones profundas* (deep constraints). Se debe tomar en cuenta que las restricciones profundas incluyen a las superficiales. Algunos enfoques han extendido la semántica superficial de elementos como las cardinalidades a un contexto multi-nivel haciéndolas profundas [5], y otros han extendido los lenguajes de restricción con semántica profunda [35, 40].

3.2.2 Clasificación de las herramientas de modelado multi-nivel

A continuación se analizan algunos enfoques y herramientas prominentes de modelado multi-nivel, utilizando el modelo de características de la Figura 3.3 como base. La Tabla 3.2 resume dicha clasificación. Además de las características del modelo de características, también se han incluido otros criterios, útiles para comprender mejor el objetivo de la herramienta, su propósito y capacidades. Estos criterios adicionales son:

- ***Propósito:*** si el objetivo del sistema es el modelado (en el ámbito de la ingeniería dirigida por modelos o los sistemas de información), la representación de conocimiento o la programación, entre otros. Esta característica es útil para comprender mejor la procedencia de la herramienta y la justificación de sus características.
- ***Lenguaje de gestión de modelos:*** si se proporcionan lenguajes para gestionar el modelo multi-nivel. En el contexto de MDE, los lenguajes de gestión incluyen capacidades para la transformación de modelos o la generación de código. Esta característica ayuda a comprender la utilidad del sistema para las tareas de desarrollo de software.
- ***Compatibilidad con el modelado de dos niveles:*** el grado en que el enfoque es compatible con las herramientas desarrolladas para el modelado estándar de dos niveles, como editores, transformaciones o lenguajes de restricción.

Tabla 3.2: Clasificación de herramientas de modelado multi-nivel.

Herramienta / Característica	*MetaDepth*	*Melanee*	*Multecore*	*DPF Workbench*	*DeepRuby*	*DeepJava*	*DeepTelos*	*OMLM*	*ML2*	*DDI*	*FMMLx*
Mecanismo	Profundidad-nativa (meta-modelo MLM)	Profundidad-nativa (meta-modelo MLM)	Promoción	Profundidad-nativa (especificaciones)	Otro (Meta-programa_ción con Ruby)	Otra (Extensión de Java)	Profundidad-nativa (proposiciones Telos)	Profundidad-nativa (meta-modelo MLM, Flora- 2)	Profundidad-nativa (powertypes)	Profundidad-nativa (proposiciones Telos)	Profundidad nativa (extensión Xcore con característica intrínseca)
Caracterización de instancias	Profunda (potencia, potencia de salto, potencia ilimitada)	Profunda (potencia, durabilidad, mutabilidad, potencia ilimitada)	Profunda (potencia, potencia de rango)	Superficial	Profunda (potencia dual)	Profunda (potencia)	Profunda (instancias más generales)	Profunda (potencia, potencia de salto)	Profunda (atributos de regularidad)	Profunda (potencia dual)	Profunda (atributo de nivel)
Extensión	Si (objetos, campos)	Si (objetos, campos)	Si (objetos, campos)	No	No	Si (objetos, campos)	Si (objetos, campos)	No	No	No	Si (objetos, campos)
Tipado	No-estricto (extensión, salto, sin mezcla) Lingüístico, ontológico	No-estricto (extensión, sin mezcla, sin salto) Lingüístico, ontológico	No-estricto (extensión, salto, sin mezcla) Lingüístico, ontológico	Estricto, ontológico	No-estricto Lingüístico, ontológico	No-estricto Lingüístico, ontológico	No-estricto Lingüístico, ontológico	No-estricto Lingüístico, ontológico, Realización	No-estricto	Estricto Lingüístico, ontológico	No-estricto Lingüístico, ontológico
Meta-niveles	Extensible	Extensible	Extensible	Extensible	Fijo	Fijo	Fijo	Fijo	Fijo	Fijo	Extensible
Restricciones	Profunda (EOL)	Profunda (OCL profundo)	Superficial (solo multiplicida_des)	Superficial	Profunda	Profunda	Profunda	Profunda MULLER/ Flora2	Superficial	Profunda (axiomas y restricciones en ConceptBase)	Profunda (XOCL, una variante de OCL)
Propósito	MDE	MDE	MDE	MDE	Programa_ción	Programación	Representa_ción del conocimiento	Representa_ción del conocimiento	Representación del conocimiento	Representación del conocimiento	Sistemas de información
Soporte para lenguajes de gestión	Si (lenguajes Epsilon, refactorizaciones multi-nivel)	Si (Deep ATL)	Si (Transforma_ciones acopladas multi-nivel)	Si (generación de código con Xpand)	Si (Ruby)	Si (Java)	No	No	No	No	Si (lenguaje de acción XMF)
Compatibilidad con herramien_tas de modelado de dos niveles	No	Media (basada en EMF)	Media (basada en EMF)	No	N/A	N/A	No	No	No	No	No

A continuación, se comentan los principales aspectos de estas herramientas con más detalle.

MetaDepth [40] es un framework para el modelado multi-nivel que utiliza una sintaxis concreta textual. El enfoque se basa en una implementación nativa, utilizando un meta-modelo lingüístico que admite la caracterización profunda a través de la potencia. Se consideran diversas variaciones de potencia, incluidas la potencia de *salto* y la *potencia* ilimitada [43]. La potencia de salto significa que, si a un clabject se le asigna la potencia de salto *n*, será posible crear instancias del clabject *n* niveles más abajo, sin necesidad de crear una instancia del clabject en niveles intermedios. MetaDepth soporta extensiones lingüísticas, donde se pueden crear tanto clabjects no tipados como campos (fields) sin tipo, debido al soporte para un tipado lingüístico y ontológico dual [40]. Debido a la potencia de salto y las extensiones lingüísticas, el tipado ontológico no es estricto, pero un clabject y su tipo no pueden estar en el mismo modelo (*sin mezcla*). El número de meta-niveles es extensible, debido a la potencia ilimitada. Las restricciones de integridad se expresan en EOL [96] (una variante de OCL), y son profundas, ya que se les puede asignar una potencia. El propósito de MetaDepth es su uso para crear soluciones MDE, y para ese propósito se ha integrado con los lenguajes Epsilon para la transformación de modelos (*in-place* y *modelo a modelo*) y la generación de código [139]. También es compatible con otras funciones avanzadas, como refactorizaciones de varios niveles [37]. La herramienta no es directamente compatible con el modelado de dos niveles, y no puede usar herramientas de frameworks de meta-modelado estándar.

Melanee admite el modelado multi-nivel a través de lenguajes definidos por el usuario, en múltiples formatos y con múltiples notaciones [4]. El componente de modelado profundo de la herramienta, respaldado por la arquitectura de clasificación ortogonal e instanciación profunda, admite un número arbitrario de meta-niveles. El concepto de instanciación profunda se implementa a través de distintas clases de potencia, que se asocian a clabjects (*potencia*), atributos (*durabilidad*) y valores de atributos (*mutabilidad*). Melanee también define la noción de potencia ilimitada, que permite un número ilimitado de niveles, y es un medio para representar tipos que tienen instancias de diferentes potencias. Melanee no admite la potencia de salto, pero, debido a que admite clabjects y atributos sin tipo, su tipado ontológico no puede considerarse estricto. La herramienta admite una variante de OCL con semántica profunda y se ha integrado con el Atlas Transformation Language (ATL) para transformaciones de modelos [68]. El objetivo de Melanee es su uso en proyectos de software basados en modelos, y se basa en la plataforma Eclipse utilizando EMF [4]. Sin embargo, aún necesita importadores y exportadores para su compatibilidad con los meta-modelos estándar de Ecore y las arquitecturas de dos niveles (es decir, se requiere adaptación para la interoperabilidad con herramientas de modelado estándar).

MultEcore es un framework de modelado gráfico multi-nivel construido utilizando EMF [104, 105]. Se basa en *transformaciones de promoción repetibles*, que se aplican cada vez que el usuario necesita crear un nuevo modelo, instancia de otro. MultEcore admite la *potencia de rango*, que extiende la potencia de salto con la posibilidad de instanciar un elemento dentro de un rango de meta-niveles, por lo que su tipado no es estricto. Aunque no soporta la potencia ilimitada, permite un número arbitrario de meta-niveles, ya que los modelos no tienen una potencia explícita y por lo tanto siempre se pueden instanciar. MultEcore admite clabjects y atributos sin tipo, pero los tipos de datos se emulan con nodos. La semántica de sus restricciones es superficial, y un lenguaje de transformación dedicado permite expresar transformaciones de modelos. El propósito de la herramienta es su uso dentro de proyectos MDE. Sin embargo, no es directamente compatible con arquitecturas y herramientas de dos niveles. Preservar la semántica del modelo multi-nivel requeriría que un exportador acumule todos los meta-modelos creados en las transformaciones de promoción aplicadas.

DPF Workbench admite el modelado con un número arbitrario de meta-niveles [100], pero no una caracterización profunda. Se basa en el Diagram Predicate Framework, que proporciona una formalización de modelos/meta-modelos y transformaciones de modelos. DPF está basado en la teoría de categorías (category theory) [81]. De esta manera, las restricciones no se especifican en OCL, sino que se usan predicados gráficos, con semántica formal. La herramienta se basa en Eclipse, pero también está disponible una versión web. DPF es una herramienta para crear soluciones MDE, y por lo tanto incluye un generador de código que admite el lenguaje Xpand [57].

DeepRuby es una implementación en Ruby del enfoque Dual Deep Instantiation (DDI) [119]. Para este propósito, el enfoque utiliza las facilidades de meta-programación ofrecidas por Ruby, especialmente la noción de *eigenclass* (clase propia). Al estar integrado dentro de un lenguaje de programación, es compatible con la definición de comportamiento (métodos). Al ser un lenguaje de

programación, no se considera la compatibilidad con los enfoques de modelado de dos niveles en este caso.

DeepTelos extiende el sistema de representación del conocimiento de Telos con una caracterización profunda a través de la noción de las instancias más generales (MGI, del inglés Most General Instances) y no con potencias numéricas [83]. Varios componentes básicos del modelado multi-nivel ya estaban presentes en Telos [118], como el tratamiento uniforme de clases y objetos, la cantidad ilimitada de meta-niveles e ideas similares a la OCA. Se puede clasificar el tipado de DeepTelos como no estricto, ya que no hay niveles de modelado explícitos. El enfoque se implementa en ConceptBase [1].

OMLM es el acrónimo de Framework integrado abierto para el Modelado Multi-Nivel (Open integrated framework for Multi-Level Modeling) [56]. Esta es una herramienta implementada en Flora-2, una implementación de código abierto de F-Logic con numerosas extensiones [163]. La herramienta proviene del campo de la representación del conocimiento. Además de crear modelos multi-nivel (instanciando un meta-modelo multi-nivel nativo), OMLM puede importar modelos de dos niveles existentes. En este último caso, se utiliza un proceso de transformación lingüística y ontológica que considera los pasos de identificación, mapeo y transformación. OMLM soporta potencia única (single) y múltiple (multi). La primera es similar a la potencia de salto, mientras que la segunda corresponde a la potencia estándar. OMLM admite consultas mediante el uso del framework MULTI-LEvel Reasoner (MULLER), que también se puede usar para verificar si la transformación de un modelo de dos niveles dio como resultado un modelo multi-nivel correcto (es decir, conforme al meta-modelo multi-nivel nativo).

ML2 es un lenguaje de modelado textual que soporta la teoría para el modelado conceptual multi-nivel llamada MLT [31, 61]. Incluye un perfil UML para visualización y combina los enfoques basados en powertypes y clabjects. ML2 carece de mecanismos para la caracterización profunda de instancias. Sin embargo, soporta la noción de *característica de regularidad*, que permite restringir los valores de las características en niveles más bajos.

Dual Deep Instantiation (DDI) es un enfoque de modelado multi-nivel que utiliza caracterización profunda a través de la potencia. DDI permite relaciones que conectan objetos en diferentes niveles de instanciación [121]. La potencia puede especificarse por separado para cada extremo de las relaciones. DDI se ha formalizado utilizando F-logic y se ha implementado sobre ConceptBase [82], un sistema de representación de conocimiento basado en la semántica de Datalog y el modelo de datos de Telos [118]. El enfoque no admite la potencia ilimitada, y por lo tanto el número de niveles se fija a priori y no admite extensiones lingüísticas (pero esto se puede emular creando una jerarquía de clabjects diferente). Al estar dirigido a la representación del conocimiento, no está integrado con lenguajes de gestión de modelos y no es directamente compatible con el modelado de dos niveles.

Flexible Meta-Modeling and Execution Language (FMMLx) mejora la expresividad del modelado conceptual tradicional orientado a objetos mediante capacidades multi-nivel. FMMLx es un lenguaje de ejecución y modelado multi-nivel que extiende a XCore [36]. Un modelo multi-nivel

creado con el FMMLx no solo puede incluir objetos en diferentes niveles de clasificación, sino también atributos, operaciones y asociaciones intrínsecas [64]. FMMLx se complementa con un entorno de meta-modelado que amplía XModeler [103]. El framework tiene dos componentes principales. Un editor de modelos genérico permite la creación de (meta-) modelos mediante el uso de una notación genérica similar a UML. En segundo lugar, un editor de sintaxis concreta permite diseñar los símbolos en los que se basa una notación gráfica y widgets adicionales como menús, campos de texto o botones. Los símbolos creados con este editor se asignan mediante correspondencias a sus respectivos elementos del (meta-) modelo utilizando una estructura de árbol genérica que forma parte de XModeler, completando así la definición de un DSL. El enfoque soporta la ejecutabilidad mediante el lenguaje de acción XMF.

En general, se observa una gran variedad de herramientas que soportan construcciones sofisticadas para modelado multi-nivel. Sin embargo, muchas herramientas carecen de integración con lenguajes de gestión de modelos (por ejemplo, para transformaciones de modelos o generación de código). La razón es el alto costo de adaptar la semántica de un lenguaje de transformación de dos niveles a una semántica multi-nivel. También se observa que, si bien algunas herramientas admiten la capacidad de importar modelos de dos niveles a multi-nivel, y algunas herramientas se basan en EMF, generalmente los modelos multi-nivel no pueden interpretarse directamente como modelos estándar válidos. Esto último podría lograrse al *aplanar* mediante una operación de flattening todos los meta-niveles, excepto el nivel 0, dentro de un solo meta-modelo. Esto permitiría tratar un modelo multi-nivel como un modelo estándar de dos niveles, y por tanto la integración con lenguajes de gestión de modelos, como ATL o Acceleo sería inmediata, así como la integración con editores de modelos estándar. Para este propósito, en el capítulo 5 se describe un enfoque que sigue esta filosofía.

3.3 Resumen y conclusiones

En este capítulo se ha realizado un análisis de los principales enfoques y herramientas que hacen posible la personalización de meta-modelos. En cuanto a la extensibilidad de meta-modelos se ha detectado que existen varios enfoques que son utilizados, la mayoría centrados en los principios de extensibilidad de UML en base a perfiles. No obstante, este es un enfoque limitado ya que sólo permite realizar extensiones ligeras. Para mejorar esta situación, se propone un mecanismo para especificar reglas para la extensión de meta-modelos existentes, este mecanismo a su vez deberá guiar de forma automatizada la ejecución de dichas reglas con la finalidad de que no se produzcan modificaciones no deseadas.

En cuanto al modelado multi-nivel, se ha detectado que actualmente existe una falta de interoperabilidad entre las herramientas de modelado multi-nivel y el modelado convencional de dos niveles, situación que debería mejorarse para permitir una migración de soluciones de dos niveles a multi-nivel y viceversa. Esto permitiría la reutilización de meta-modelos existentes como base para crear modelos multi-nivel y permitiría la utilización de herramientas estándar de gestión de modelos

ya existentes. Por ello, se propone diseñar un mecanismo para modelado multi-nivel que emule objetos en el nivel de meta-modelo, donde la creación de instancias se emule con herencia.

4 Mecanismo de control de extensibilidad de meta-modelos

En esta se capítulo se analizan algunos de los escenarios que requieren la personalización de meta-modelos. También se realiza una clasificación de los tipos de personalización de meta-modelos. Finalmente se describe la propuesta de un mecanismo de control de personalización de meta-modelos, su implementación en la herramienta TACO , y una evaluación del enfoque.

4.1 Introducción

En el ámbito del desarrollo de software, existen situaciones en los cuales se requiere personalizar los lenguajes de modelado para que se adapten a nuevos requisitos del usuario. Dependiendo del tipo de personalización que sea necesaria, se podría modificar cualquier componente del lenguaje de modelado. Esta tesis se centra en la personalización del elemento central y vital del lenguaje de modelado, su meta-modelo. No obstante, se debe evitar un cambio descontrolado del meta-modelo, que pueda conducir a incompatibilidades de los modelos con herramientas y servicios existentes. Aunque la actividad de modelado está regulada por el meta-modelo correspondiente, no existen mecanismos comúnmente aceptados que regulen cómo deberían extenderse los meta-modelos. Esto se debe a que los meta-modelos a menudo definen lenguajes, vistas y servicios que suelen estar

integrados (y a veces codificados) en las herramientas y por lo tanto, es menos probable que requieran modificaciones del usuario [5]. Sin embargo, en algunos escenarios, es común diseñar meta-modelos que deben ser extendidos por otros desarrolladores [46]. Por ejemplo, algunas especificaciones de la OMG están destinadas a ser utilizadas extendiendo una cierta parte del meta-modelo. Este es, por ejemplo, el caso del Meta-modelo de Descubrimiento de Conocimiento (KDM, del inglés Knowledge Discovery Metamodel) [126] y los estándares de Definición de Diagrama (DD, del inglés Diagram Definition) [124].

Actualmente, las formas en que se deben realizar estas extensiones se expresan usando lenguaje natural. Sin embargo, este enfoque informal es propenso a errores, más cuando no existe un mecanismo automatizado para verificar las extensiones definidas por el desarrollador con respecto a lo especificado en el estándar, o los medios para guiar al desarrollador en la extensión. Esta situación contrasta con los mecanismos de creación de instancias bien establecidos de los meta-modelos. Por tanto, es de considerar que debería existir un mecanismo similar para establecer reglas para la extensión correcta de un meta-modelo (por ejemplo, clases que se puedan subclasificar, referencias que se puedan redefinir, enumeraciones que admitieran literales adicionales), así como la operacionalización de tales reglas por medio de herramientas.

Para mejorar esta situación, se propone un mecanismo para la especificación de reglas para la extensión de meta-modelos, así como una herramienta que permite su extensión de acuerdo con las reglas definidas. La herramienta, denominada TACO (abreviatura de "**T**ool for met**A**model **C**ust**O**mization"), se ha creado como un plug-in de Eclipse para Eclipse Modeling Framework (EMF), el estándar de meta-modelado de facto en la actualidad [156]. La herramienta contiene un asistente que guía al desarrollador en la extensión del meta-modelo y ha sido validada en diferentes escenarios, incluida la definición de extensibilidad para los estándares KDM y DD, y otros meta-modelos construidos por terceros.

4.2 Escenarios de aplicación de la personalización de meta-modelos

Teniendo en cuenta que los meta-modelos son la piedra angular en el desarrollo de software con MDE [27] y que el software debe evolucionar a lo largo del tiempo [90], es necesario contar con frameworks que proporcionen soporte metodológico a la evolución de los meta-modelos. Esta evolución permite desarrollar software que se puede adaptar a nuevos dominios de aplicación o cambiar los requisitos dentro del dominio [27].

Hay varios esfuerzos independientes que abordan el tema de la extensión del meta-modelo, como se analizó en la sección 3.1, pero aún no se han desarrollado estándares para esta tarea [29]. A continuación, se revisan tres escenarios que motivan la necesidad de especificar las personalizaciones de meta-modelos permitidos, así como los mecanismos para guiar al desarrollador a realizar la personalización de acuerdo con esas reglas.

4.2.1 Escenario A: Extensión de meta-modelos

Hay situaciones en las que los meta-modelos están diseñados para ser extendidos. Estos son meta-modelos básicos, de los cuales se derivan sistemas más complejos por subclasificación y redefinición. Por ejemplo, el estándar DD de la OMG permite la especificación formal de la sintaxis concreta de un lenguaje de modelado que hace posible que las herramientas intercambien diagramas [124]. La idea del estándar es obtener una forma sistemática para intercambiar información de sintaxis concreta entre herramientas, similar a la forma en que se intercambian los modelos entre las herramientas basadas en MOF en XMI. La sintaxis abstracta de un lenguaje de modelado gráfico se define típicamente con un meta-modelo, mientras que su sintaxis concreta (de tipo diagramático en este caso) se define con texto y figuras que el desarrollador considere [59, 22]. No existe un formalismo que señale los elementos textuales o gráficos que se deban utilizar.

El estándar DD considera una arquitectura que permite la especificación del *Diagram Interchage* (DI) y del modelo *Diagram Graphics* (DG). DI es un framework destinado a la extensión en lugar de un componente listo para usar. El extenderlo permite definir aspectos gráficos que el usuario controla, como la posición de los nodos y los puntos de enrutamiento de las líneas. Por lo tanto, se puede ver que DI debe ser refinado para su uso con un lenguaje de modelado particular (por ejemplo, el UML). DG se utiliza para definir los aspectos gráficos que se especifican en el lenguaje de modelado (no controlable por el usuario) [59]. Ambos modelos comparten elementos comunes de un modelo *Diagram Common* (DC). La arquitectura DD espera que las especificaciones del lenguaje definan mapeos entre información gráfica intercambiada y no intercambiada, pero no restringe cómo se hace. La Figura 4.1 corresponde a la arquitectura del estándar Diagram Definition [124].

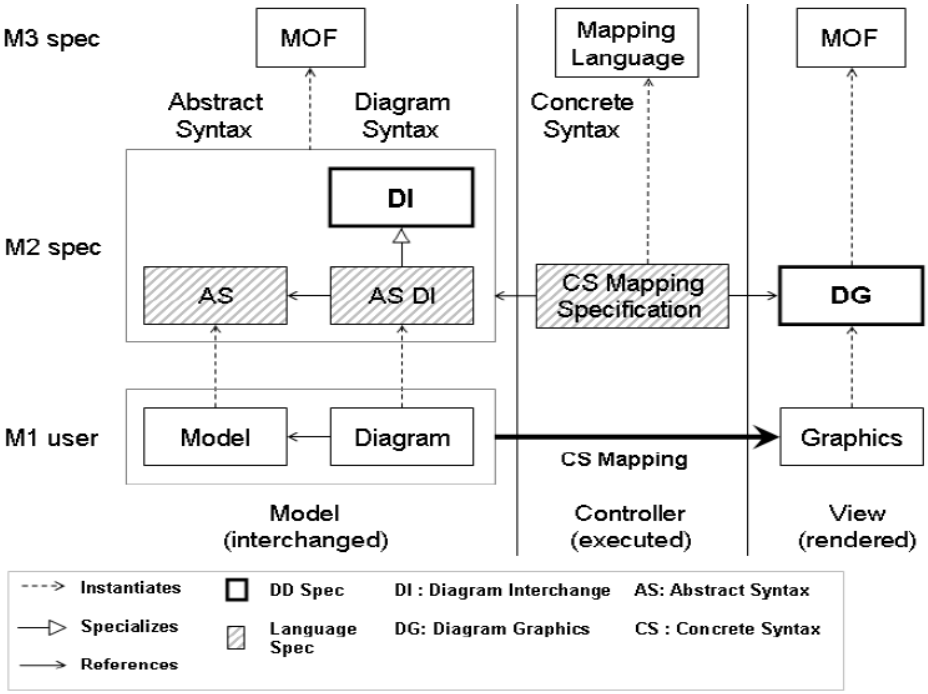


Figura 4.1: Arquitectura del Diagram Definition [124].

Como se señaló, el uso del estándar DD para un lenguaje de modelado específico requiere la extensión de ciertas clases del meta-modelo DI y la manera de extender dicho meta-modelo se describe en lenguaje natural en el estándar. Sin embargo, esto puede conducir a errores, más cuando no hay un mecanismo automático para verificar la corrección de las extensiones, o para guiar al desarrollador en su construcción. La Figura 4.1 muestra la Arquitectura del Diagrama Definition, tomado de [124]. Para su uso, el meta-modelo DI debe extenderse (ver meta-modelo AS DI).

4.2.2 Escenario B: Adaptación de DSLs

Muchos DSLs no están completamente diseñados desde cero, sino que se crean extendiendo o adaptando un lenguaje base a un dominio determinado [141]. Por ejemplo, el diseño de DSL para describir el comportamiento a menudo se basa en lenguajes conocidos, como las máquinas de estados (ver un meta-modelo simplificado en la Figura 4.2) y la adición de extensiones específicas del dominio.

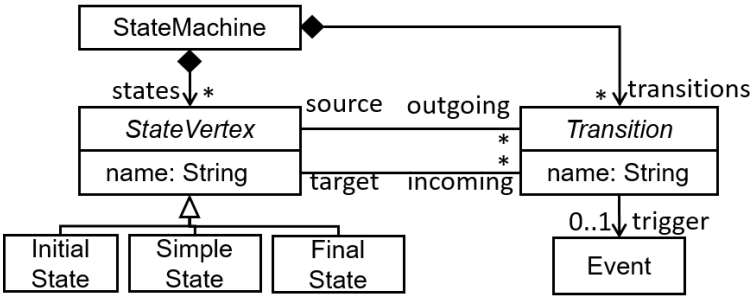


Figura 4.2: Meta-modelo para máquina de estados.

En este escenario, además de extender los elementos existentes, también pueden ser útiles las adaptaciones y simplificaciones (eliminaciones). En el caso de la Figura 4.2, para un dominio específico, un diseñador puede no necesitar máquinas de estados con estados finales o iniciales. Además, para adaptar la máquina de estados a un dominio específico, es posible que necesite actualizar la clase *SimpleState* con nuevas características y definir varias subclases de *Event*. Si necesitamos asegurar una reutilización correcta de artefactos, como por ejemplo las transformaciones de modelos, entonces se requiere una especificación de elementos que no se pueden cambiar (por ejemplo, *StateMachine*, *StateVertex*, *Transition*, *SimpleState*), elementos opcionales que se pueden eliminar (*InitialState*, *FinalState*, *Event*) y elementos extensibles que pueden ser subclasificados o actualizados (*Event*, *SimpleState*).

4.2.3 Escenario C: Modelado multi-nivel y el patrón tipo-objeto

En el meta-modelado tradicional, el ingeniero trabaja con dos meta-niveles: meta-modelos y modelos. Este es el enfoque seguido por los estándares de meta-modelado actuales, como el EMF. El modelado multi-nivel [46] como se señaló en la sección 2.1.6, es una extensión conservadora del enfoque de dos niveles, que admite el uso de un número arbitrario de meta-niveles para describir los

sistemas. El modelado multi-nivel permite obtener modelos más simples (con menos elementos) en algunos escenarios, especialmente cuando surge el patrón tipo-objeto [46].

Como ejemplo, se puede suponer que se quiere describir un sistema de comercio electrónico (e-commerce) que soporte la definición dinámica de tipos de productos (como *Books*) y las instancias de estos. En este dominio, los tipos de productos pueden tener diferentes valores de impuestos (*vat*), mientras que sus instancias tienen un precio (atributo *price)*. Tal sistema se representa en la Figura 4.3(a) usando el patrón tipo-objeto. El meta-modelo contiene clases que modelan tipos de productos, instancias y la relación de tipado entre ambos.

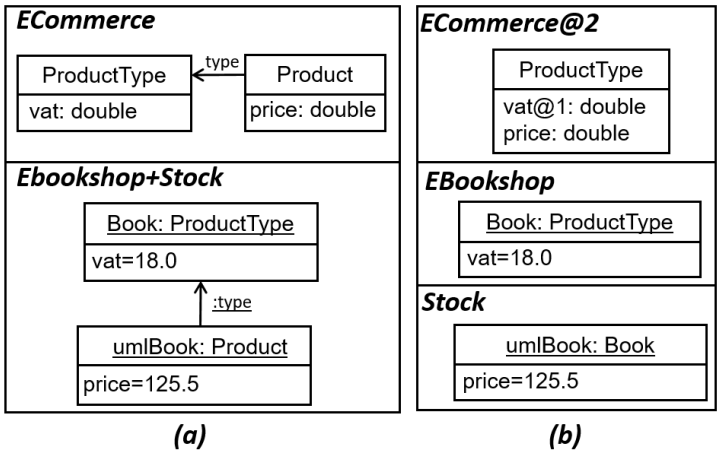


Figura 4.3: Un sistema de comercio electrónico.
(a) Modelado utilizando el patrón tipo-objeto; (b) Modelado multi-nivel.

Alternativamente, se puede usar el modelado multi-nivel para representar el mismo sistema, como lo muestra la Figura 4.3(b). Esta solución usa tres meta-niveles. El modelo superior no requiere productos de modelado explícito, ya que estos serán instancias de las instancias de *ProductType*. En este caso se deben utilizar mecanismos de *caracterización profunda*, como la *potencia* (como se señaló en la sección 2.1.7.4) para describir instancias más allá del meta-nivel inmediatamente inferior. En la Figura 4.3(b), *ProductType* tiene potencia 2, por lo que se puede crear una instancia en los siguientes dos meta-niveles. El atributo *price* también tiene potencia 2 (si un atributo no declara explícitamente una potencia, toma la potencia de su clase contenedora), por lo que se aplica a instancias dos niveles por debajo, mientras que *vat* tiene potencia 1 y solo se aplica a las instancias del siguiente nivel. Como se puede observar, el modelo multi-nivel resultante tiene menos elementos que la solución de dos niveles.

El modelado multi-nivel tiene ventajas en el diseño de la extensibilidad de los meta-modelos, ya que las necesidades de extensibilidad se pueden expresar como instanciación, para lo cual existen mecanismos estándar que describen su funcionamiento. Hay varias herramientas que admiten modelos de varios niveles, como MetaDepth [40] o Melanee [4], analizadas en la sección 3.2.2. Sin embargo, no son directamente interoperables con EMF y su rico ecosistema de modelado.

En los tres escenarios señalados, es notoria la necesidad de expresar cómo ciertos elementos en un meta-modelo base pueden refinarse, en el caso del escenario A (extensión de meta-modelos) y C

(modelado multi-nivel), y expandirse con nuevas características o simplificarse, escenario B (adaptación de DSLs).

4.3 Clasificación de tipos de personalización del meta-modelo

En esta sección, se clasifican los tipos de personalización que se pueden realizar en un meta-modelo y se discute la compatibilidad de los posibles modelos fuente existentes con respecto al meta-modelo personalizado y viceversa. Los tipos de personalización se resumen esquemáticamente a partir de la Figura 4.4 a la 4.11. Los primeros cuatro casos se refieren a clases, mientras que los cuatro casos siguientes involucran atributos (features). Cuando se habla de modificar el meta-modelo, se utilizan varios términos en la literatura, incluida la extensión, personalización, especialización, adaptación, variación o mutación. En el presente trabajo se utiliza el término personalización para referirse a cualquier tipo de modificación, mientras que extensión es un tipo específico de personalización que agrega nuevas subclases, o nuevas clases al meta-modelo, mientras que deja los elementos existentes intactos. Esta distinción es útil, ya que las extensiones preservan la compatibilidad de los modelos existentes con respecto al meta-modelo extendido resultante, mientras que las personalizaciones arbitrarias pueden no preservar dicha compatibilidad.

Se distinguen dos tipos de extensiones: el "*refinamiento*" del meta-modelo y la "*expansión*" del meta-modelo, que se muestran en las Figuras 4.4 y 4.5, respectivamente. Un refinamiento agrega nuevas subclases a las clases existentes en el meta-modelo original (por simplicidad se ha considerado únicamente herencia simple). Las nuevas subclases pueden agregar nuevas características y redefinir las referencias de la clase original.

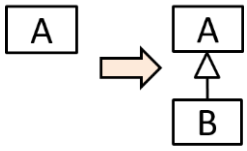


Figura 4.4: Refinamiento del meta-modelo.

Una expansión agrega nuevas clases que no están relacionadas por herencia a las clases existentes del meta-modelo original.

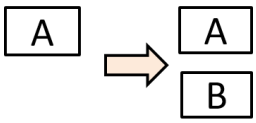


Figura 4.5: Expansión del meta-modelo.

Como muestra la Tabla 4.1, un modelo conforme a un meta-modelo, también se ajusta a un meta-modelo refinado y a uno extendido (columna de compatibilidad del modelo hacia adelante). Sin

embargo, un modelo conforme a un meta-modelo refinado o expandido no necesariamente se ajusta al meta-modelo original (columna de compatibilidad del modelo hacia atrás).

En la tabla se distinguen incompatibilidades que pueden resolverse automáticamente (***R*** -*re*soluble- en la tabla) y que no se pueden resolver automáticamente y requieren intervención humana (***I*** -irresoluble- en la tabla) [34]. Tanto los modelos refinados como los expandidos pueden ajustarse automáticamente al meta-modelo original. En el primer caso, se retipan los objetos de la subclase a la superclase (objetos de tipo B a objetos de tipo A en la Figura 4.4) y se eliminan las instancias de atributo (slots) y enlaces adicionales introducidos por la subclase. De manera similar, para un modelo expandido simplemente se eliminan los objetos de las clases añadidas.

Tabla 4.1: Compatibilidad de los tipos de modificación del meta-modelo.

Tipos: E=Extensión, **P**=Personalización, **PP**=Personalización de Propiedad.
Compatibilidad: ✔ =Compatible, **R**=Resoluble, **I**=Irresoluble.

Tipo	Modificación	Compatibilidad del modelo hacia adelante	Compatibilidad del modelo hacia atrás
E	Refinamiento	✔	R
E	Expansión	✔	R
P	Restricción	R	✔
P	Contracción	R	✔
P	Personalización aditiva conservadora	✔	R
P	Personalización aditiva de ruptura (breaking)	I	R
P	Personalización restrictiva	R	I
P	Personalización restrictiva segura (safe)	R	✔
PP	Abstraer clase	I	✔
PP	Concretar clase	✔	I
PP	Incorporación de herencia	I	I
PP	Supresión de herencia	I	I
PP	Relajación de cardinalidad	✔	I
PP	Restricción de cardinalidad	I	✔
PP	Cambio de cardinalidad	I	I
PP	Relajación del tipo de la feature	✔	I
PP	Restricción del tipo de la feature	I	✔
PP	Cambio del tipo de la feature	I	I
PP	Establecer composición	I	✔
PP	Eliminar composición	✔	I

Por el contrario, para el refinamiento y la expansión, se pueden tener "*restricciones*" (es decir, eliminar una subclase) y "*contracciones*" (eliminar una clase sin superclase), que se muestran en las Figuras 4.6 y 4.7, respectivamente. Estas personalizaciones son inversas de refinamiento y expansión respectivamente, y por lo tanto la compatibilidad hacia adelante y hacia atrás se invierte.

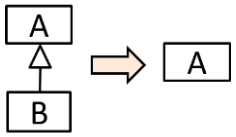


Figura 4.6: Restricción del meta-modelo.

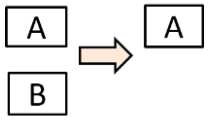


Figura 4.7: Contracción del meta-modelo.

A continuación, se analizan las características de la clase, donde se distinguen dos tipos de personalización: aditiva y restrictiva. El primero tipo, agrega nuevas features (atributos y referencias) a las clases existentes en el meta-modelo original, mientras que el segundo elimina las features de las clases existentes.

Las Figuras 4.8 y 4.9 muestran dos subtipos de cambios aditivos. El primero se llama *conservador*, ya que agrega features opcionales (atributos o referencias) a las clases existentes. Las "*personalizaciones aditivas conservadoras"* preservan la conformidad de los modelos con respecto al meta-modelo actualizado. Los modelos del meta-modelo actualizado pueden ajustarse al original eliminando las instancias de atributo (slots) correspondientes a las nuevas features añadidas.

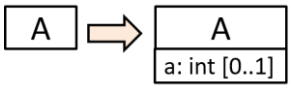


Figura 4.8: Personalización aditiva conservadora.

Las "*personalizaciones aditivas de ruptura"* agregan features obligatorias a las clases existentes (ver Figura 4.9). Estos cambios rompen la conformidad de los modelos existentes, ya que carecen de tales features obligatorias y por lo tanto no son conformes con respecto al meta-modelo actualizado. Tal no conformidad no se puede resolver automáticamente y requeriría la intervención del usuario (ya que puede haber varias formas de arreglar un modelo existente). Por el contrario, los modelos del meta-modelo actualizado no son conformes al meta-modelo original, pero pueden corregirse automáticamente eliminando las características recién agregadas.

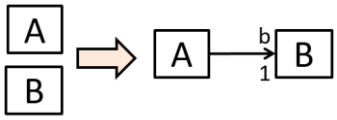


Figura 4.9: Personalización aditiva de ruptura.

Las "*personalizaciones restrictivas"* eliminan features del meta-modelo original. Se hace la distinción entre eliminar features obligatorias (Figura 4.10) y features opcionales (Figura 4.11). La eliminación de features obligatorias hace que los modelos existentes sean incompatibles con el meta-modelo actualizado, pero las incompatibilidades se pueden corregir eliminando las instancias de atributo (slots) de las features obligatorias eliminadas. Por el contrario, los modelos del meta-modelo actualizado no se pueden adaptar de forma automática al meta-modelo original, ya que se podrían elegir muchos valores diferentes para los elementos que se deben agregar.

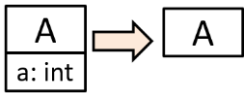


Figura 4.10: Personalización restrictiva.

Las "*personalizaciones restrictivas seguras*" son aquellas que eliminan features opcionales de las clases existentes (véase la Figura 4.11). Estas actualizaciones pueden causar incompatibilidad de los modelos existentes con respecto al meta-modelo actualizado, que se puede resolver automáticamente, eliminando las instancias de atributo y los enlaces correspondientes a las features eliminadas. Por el contrario, los modelos del meta-modelo actualizado siguen siendo conformes al meta-modelo original, ya que este tipo de personalización solo elimina las features opcionales.

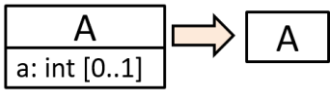


Figura 4.11:personalizaciones restrictivas seguras.

En definitiva, se puede observar que las personalizaciones aditivas conservadoras y las personalizaciones restrictivas seguras son inversas entre sí, mientras que las personalizaciones aditivas y las personalizaciones restrictivas también son inversas

Los cambios presentados son suficientes para los escenarios previstos de extensión y personalización del meta-modelo que se contemplan en esta tesis. Sin embargo, por completitud, a continuación, se discuten otros posibles cambios al meta-modelo. Estos cambios implican cambiar las propiedades de los elementos del meta-modelo existentes, como la abstracción de una clase o la cardinalidad de una referencia. Por esta razón, se los ha llamado *personalizaciones de propiedad* y también se resumen en la Tabla 4.1.

En cuanto a las clases, se puede hacer abstracta una clase concreta. Este cambio rompe la compatibilidad hacia adelante, y generalmente no se puede corregir automáticamente (ya que, por ejemplo, los objetos de la clase abstracta pueden necesitar ser retipados a alguna subclase). Por el contrario, este cambio preserva la compatibilidad con versiones anteriores. Cambiar una clase abstracta a concreta, tiene el efecto inverso: preserva la compatibilidad hacia adelante, mientras que la compatibilidad hacia atrás implica cambios irresolubles.

Con respecto a la herencia, se puede agregar una relación de herencia entre dos clases existentes. Esto puede romper los modelos existentes, ya que las instancias de la subclase pueden recibir características obligatorias de la superclase. En general, dichos cambios no pueden corregirse automáticamente. Además, esto también rompe la conformidad hacia atrás, ya que los objetos de la subclase dejan de recibir los atributos de la clase eliminada. Esto no puede corregirse automáticamente, ya que los objetos de la subclase pueden tener que eliminarse de las colecciones (que eran compatibles con la superclase), y esto puede romper la cardinalidad de la colección. Eliminar una relación de herencia tiene el efecto inverso.

Se puede hacer cambios en las features existentes, con respecto a su cardinalidad, tipo y composición (para referencias). Con respecto a la cardinalidad, se distingue relajar o restringir el intervalo de cardinalidad. En el primer caso, los modelos del meta-modelo original aún se ajustan al meta-modelo actualizado (compatibilidad hacia adelante), pero introduce cambios irresolubles por compatibilidad con versiones anteriores. Esto es así porque para que la feature relajada se ajuste al original, es posible que se tengan varias opciones para eliminar elementos de ella. El caso de

restricción es el inverso de la relajación (ruptura no resuelta para compatibilidad hacia adelante y compatible hacia atrás). También se pueden tener cambios arbitrarios en los intervalos de cardinalidad inferior y superior. Este cambio más general rompe la compatibilidad hacia adelante y hacia atrás, y no puede resolverse automáticamente.

Las features (especialmente las referencias) también se pueden relajar o restringir con respecto al tipo destino. En el primer caso, el destino de una referencia se cambia para apuntar a una superclase del objetivo original. Esto conduce a la compatibilidad hacia adelante, pero produce cambios irresolubles para la compatibilidad con versiones anteriores. Esto es así ya que podría ser imposible eliminar un objeto dado de la referencia sin romper su cardinalidad. La restricción de tipo de referencia cambia el destino de una referencia para apuntar a una subclase, y tiene el efecto inverso que la relajación de referencia. Finalmente, se puede cambiar el tipo de destino de una referencia por otro tipo no relacionado, que es un cambio irresoluble de conformidad tanto hacia adelante como hacia atrás.

Finalmente, una referencia (que no sea composición) puede cambiarse para convertirse en una referencia tipo composición. Esto rompe la compatibilidad, ya que la referencia en el modelo original puede no tener una estructura de árbol. Por el contrario, conserva la compatibilidad con versiones anteriores. Eliminar la composición de una referencia tiene el efecto inverso.

Una vez que se han visto los tipos de personalizaciones posibles para un meta-modelo base, en la siguiente sección se propone un enfoque para especificar el espacio de personalizaciones posibles para un meta-modelo.

4.4 Mecanismo de control de personalización del meta-modelo

Esta sección describe el enfoque que se ha diseñado para controlar cómo se puede personalizar un meta-modelo. El enfoque considera todos los tipos de personalización descritos en la sección 4.3, excluyendo las configuraciones de propiedad (PP), dejadas como trabajo futuro.

En esta tesis se considera las siguientes definiciones relacionadas a la extensión de meta-modelos [16]. Un *meta-modelo base* es un meta-modelo existente que tiene vida propia y que puede encontrarse integrado a herramientas o soluciones. Tiene modelos que se ajustan a éste. Una *extensión de meta-modelo* es la definición de una extensión que se basa parcial o totalmente en conceptos del meta-modelo base. Un *meta-modelo extendido* es el resultado de la aplicación de una o varias extensiones de meta-modelo al meta-modela base.

Como se muestra en la Figura 4.12, el método considera dos fases. En el primero, se definen las reglas de personalización del meta-modelo base. Estas reglas se especifican como un modelo, conforme a un meta-modelo de personalización (que se muestra en la Figura 4.13), que anota los elementos del meta-modelo base. En la segunda fase, el meta-modelo base puede personalizarse de

acuerdo con las reglas establecidas. La Figura 4.12 representa explícitamente el caso de la extensión del meta-modelo. Si bien se admiten otros tipos de personalización, este documento se centra en la extensión del meta-modelo.

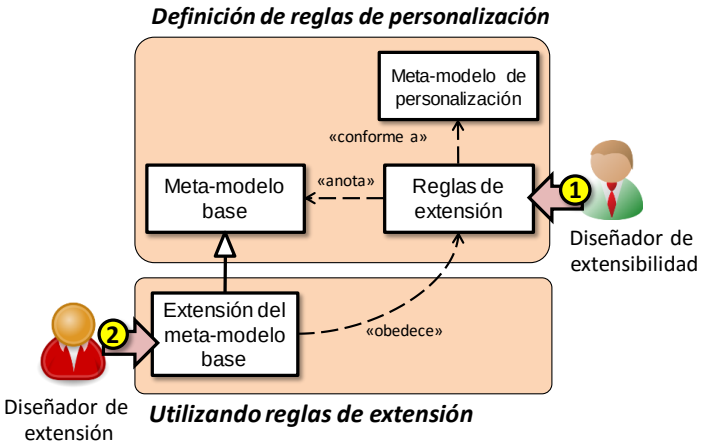


Figura 4.12: Uso de reglas de personalización.

Es común que sean diferentes desarrolladores los que definan las reglas de extensión y las extensiones concretas del meta-modelo. Típicamente, la definición de las reglas de extensión será realizada por el diseñador del meta-modelo base (en general, por un diseñador de extensibilidad), mientras que la reutilización de dicho meta-modelo base implica su extensión de acuerdo con las reglas especificadas (diseñador de extensión en la Figura 4.12). Ambas actividades deben estar soportadas por herramientas que necesitan proporcionar orientación al desarrollador para ampliar el meta-modelo, así como la confianza de que la extensión realizada obedece a las reglas de extensión. La sección 4.5 ofrece una descripción general de la herramienta desarrollada.

Tabla 4.2: Reglas de personalización soportadas.

Regla	Aplicada a	Descripción	Tipo de Personalización
Extend	Clase	La clase etiquetada es considerada extensible: es posible añadir una subclase.	Refinamiento
Delete	Clase, Feature	Permite etiquetar clases y features como opcionales.	Restricción, Contracción, Personalización restrictiva
Update	Clase, Enumeración	Etiqueta una clase o enumeración como "abierta". Para las clases, estas pueden añadir nuevos atributos y referencias. En el caso de enumeraciones, estas pueden añadir nuevos literales.	Personalización aditiva (Clases), Refinamiento (Enumeraciones)
New	Meta-modelo	Etiqueta un meta-modelo como "abierto": este puede añadir nuevas clases y/o tipos de enumeraciones.	Expansión
Redefine	Referencia	Gobierna cómo o si una referencia puede ser redefinida.	Refinamiento

El enfoque admite 5 tipos de reglas de personalización, resumidas en la Tabla 4.2, que cubren la extensión y personalización del meta-modelo. El soporte a la personalización de propiedades se deja para el trabajo futuro. Una especificación de extensión de meta-modelo consiste en un conjunto de operaciones de extensión atómica. Por lo general se aplica en elementos de meta-modelos existentes [29].

- ***Reglas Extend:*** identifican las clases del meta-modelo base que pueden ampliarse con subclases. Conducen a refinamientos del meta-modelo (ver Figura 4.6(a)).
- ***Reglas Delete:*** etiquetan las clases o features opcionales (atributos y referencias). Para el caso de las clases, conducen a restricciones del meta-modelo (en el caso de subclases opcionales) o contracciones (para las clases sin supertipos).
- ***Reglas Update:*** se pueden aplicar a clases o tipos de enumeración. En el primer caso, la clase se considera abierta y se pueden añadir nuevos atributos y referencias, lo que conduce a personalizaciones aditivas (ruptura o conservadoras). En el segundo, se pueden agregar nuevos literales a la enumeración, lo que conduce a un refinamiento. Se considera que añadir nuevos literales de enumeración es un refinamiento, ya que en cierto modo es equivalente a subclasificar una clase base.
- ***Reglas New:*** se aplica al meta-modelo en sí, e indica que puede expandirse con nuevas clases y/o tipos de enumeración, lo que lleva a una expansión del meta-modelo.
- ***Reglas Redefine:*** son aplicables a las referencias del meta-modelo base y regulan cómo se pueden redefinir en las extensiones. Las reglas *redefine* conducen a refinamientos, ya que deberían ser compatibles con una regla *Extend* adecuada de la clase propietaria o una superclase.

En el escenario de extensión de los meta-modelos que se maneja, sólo son relevantes las reglas Extend, Redefine, Enumeration Update, y New.

Las reglas de personalización se han realizado en base a un meta-modelo de personalización, que se muestra en la Figura 4.13. La clase raíz del meta-modelo es *CustomizationModel*, que contiene el nombre del conjunto de reglas de extensión (atributo *name*) y la política para almacenar el meta-modelo personalizado (atributo *save*). Para este último, se han considerado tres opciones (enumeración *TypeSave*): 1) *newMM*: crea un nuevo meta-modelo que contiene tanto el meta-modelo base como la extensión, 2) *theSame*: las modificaciones se realizan en el meta-modelo original y 3) *newPackage*: las extensiones se almacenan en un paquete separado que hace referencia al meta-modelo base. Es posible indicar que si el meta-modelo está abierto, de modo que se pueden agregar nuevas enumeraciones y clases independientes (atributo *expand*).

Un *CustomizationModel* contiene una colección de reglas de personalización. Estas son instancias de alguna subclase concreta de *Customization*, y están organizadas en una jerarquía, donde las clases inferiores (*DeleteFeature*, *DeleteClass*, etc.) tienen una referencia al elemento del meta-modelo que anotan o etiquetan. Los elementos del meta-modelo se colocan en el paquete "Ecore", porque se

utiliza el EMF como plataforma de modelado, pero esta idea también es aplicable a otras plataformas de meta-modelado.

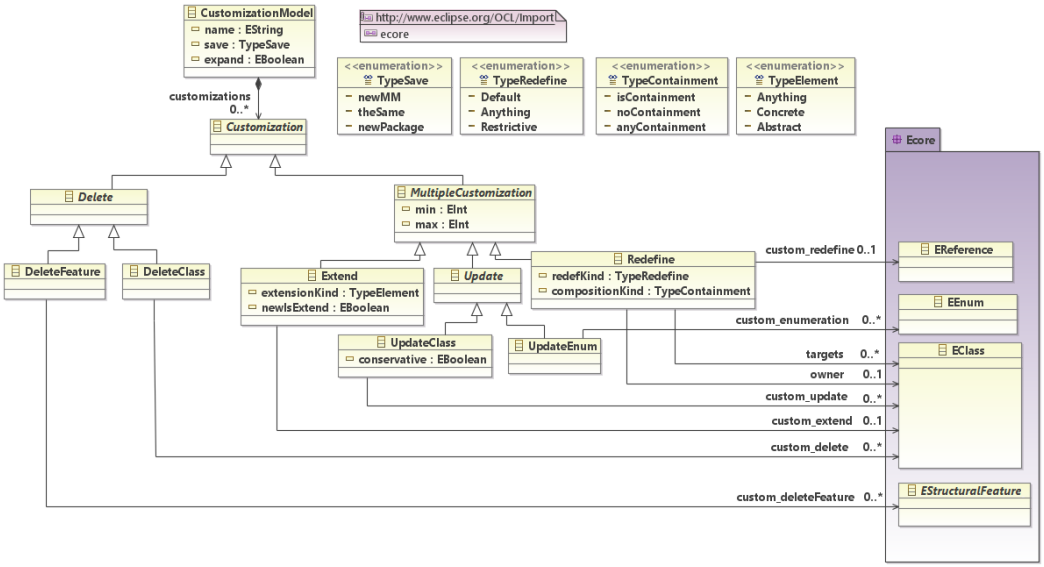


Figura 4.13: Meta-modelo de personalización.

Las reglas *Delete* (subclases de *Delete*) etiquetan como opcionales a clases y features (atributos y referencias). Las reglas *Extend*, *Update* y *Redefine* pueden tener un intervalo de cardinalidad (min..max), y por lo tanto heredan de la clase abstracta *MultipleCustomization*.

Las reglas de extensión permiten extender (crear subclases de) las clases del meta-modelo base seleccionado por la referencia *custom_extend*. El atributo *extensionKind* declara si las posibles subclases deben ser abstractas, concretas o se deja a juicio del ingeniero que extiende el meta-modelo. El atributo *newIsExtend* indica si la subclase creada puede a su vez subclasificarse. La regla permite especificar el número de subclases permitidas, usando el intervalo min..max (donde -1 para max indica ilimitado). De esta forma, es posible especificar si una clase se debe extender exactamente una vez (intervalo 1..1), opcionalmente como máximo una vez (intervalo 0..1), obligatoriamente una o más veces (intervalo 1..- 1), o cero o más veces (intervalo 0..- 1). Se debe tener en cuenta que si una clase del meta-modelo base no tiene una regla de extensión asociada, entonces no se puede extender.

Es posible definir reglas que rigen la redefinición de referencias instanciando la clase *Redefine* y señalando la referencia que se redefinirá usando *custom_redefine*. Por lo tanto, dada una clase *C* que define una referencia *ref* a una clase *D*, se podrá indicar cuántas veces se puede redefinir *ref* cada vez que *C* se extiende (a través del intervalo min..max). En cualquier caso, el destino de las redefiniciones *ref* debe ser compatible con la clase *D*. Además, es posible especificar si las redefiniciones deben ser o no una composición (atributo *compositionKind*). Usando el atributo *redefKind*, se controla la cardinalidad que se puede asignar a cada redefinición, teniendo en cuenta tres posibilidades: 1) *Default:* la cardinalidad de las redefiniciones debe ser la de la referencia *ref*, 2) *Restrictive:* la cardinalidad de las redefiniciones debe ser un intervalo contenido en el intervalo de referencia. Por

ejemplo, si la cardinalidad de *ref* es 0..2, las redefiniciones pueden declarar los intervalos: 0..1, 0..2, 1..2, 1..1 y 2..2; y 3) *Anything:* la cardinalidad de las redefiniciones puede ser cualquiera.

Las referencias pueden conectar dos jerarquías de herencia: las de las clases fuente y destino de la referencia. Por defecto, una regla de redefinición describe cómo (y sí) se redefine una referencia cuando se subclasifica la clase propietaria de la referencia. Por lo tanto, la regla de redefinición se aplica cuando la clase propietaria o una subclase se extiende, y la clase destino de la referencia redefinida debe ser compatible (una subclase) con la clase destino de la referencia original. Sin embargo, es posible que se desee ser más preciso al indicar las clases destino permitidas y cuándo debe realizarse la redefinición. De esta forma, se puede establecer que la regla *Redefine* se aplique sólo cuando se extiende una determinada subclase *A* de la clase propietaria de referencia. Para describir esta posibilidad, la referencia *Redefine.owner* apuntaría a la clase *A*. Del mismo modo, se pueden definir los destinos permitidos de la referencia redefinida. Esto se hace a través de la referencia *Redefine.targets*, que debe apuntar a las subclases de la clase destino de la referencia original.

Cuando una referencia *ref* se redefine por una serie de referencias $ref_1, ..., ref_n$, la referencia *ref* se puede ver como una referencia derivada, resultando de la unión de $ref_1, ..., ref_n$. En términos de UML, las referencias $ref_1, ..., ref_n$ tendrían una relación de subconjuntos con *ref*.

Ejemplo. La Figura 4.14 muestra un ejemplo de cómo funcionan las reglas de redefinición. La parte (a) de la figura muestra un meta-modelo base con algunas reglas de extensión. En particular, indican que la clase *A* puede ampliarse opcionalmente como mucho una vez, mientras que *B* y *C* deberían extenderse obligatoriamente. En los tres casos, las subclases pueden ser abstractas o concretas. La referencia tiene una regla de redefinición, que establece que debe redefinirse una o más veces, siempre que *B* (propietario) se extienda. Los destinos válidos de las redefiniciones son subclases de *C*. Además, las redefiniciones deben ser de contención (*cont*) y deben tener cardinalidad * (*default*).

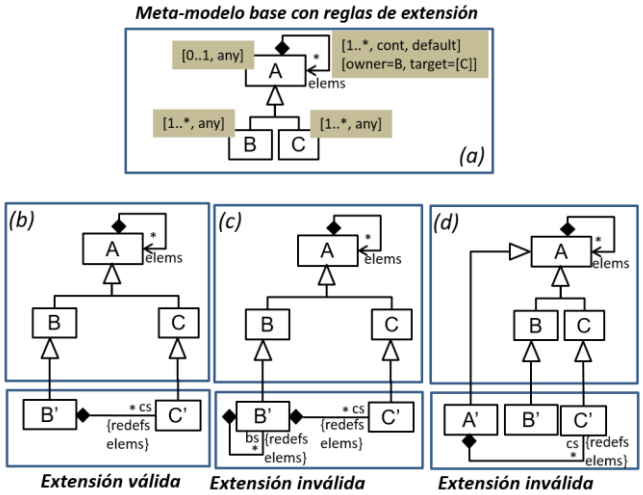


Figura 4.14: Ejemplos de redefinición.
(a) Meta-modelo base con reglas de extensión y redefinición; (b) Extensión válida; (c, d) Extensiones inválidas: (c) el destino de la referencia *bs* es inválido, (d) el origen de la referencia *cs* es inválido.

La Figura 4.14(b) muestra una extensión válida, porque *cs* redefine *elems*, y es propiedad de *B*', que extiende de *B*. Además, el destino de *cs* es correcto, ya que *C*' es una subclase de *C*. En cambio, las Figuras 4.14(c) y (d) muestran extensiones no válidas. En la Figura 4.14(c), *bs* es una redefinición incorrecta de *elems*, porque el destino de *bs* no es una subclase de *C*. En la Figura 4.14(d), *cs* es una redefinición incorrecta de *elems*, porque el propietario de *cs* no es una subclase de *B* (y también porque *B*' no redefine *elems*). Se mostrarán usos realistas y prácticos de cómo controlar el elemento propietario (owner) y el elemento destino (target) de las redefiniciones en la sección 4.6.2.

Finalmente, con respecto a las reglas *Update*, existen dos tipos de reglas: para clases (*UpdateClass*) y para enumeraciones (*UpdateEnum*). Para las clases, las reglas permiten agregar una cantidad de features en el rango (min...max). Si el atributo *conservative* se establece a verdadero, entonces sólo se pueden agregar features opcionales. Esto hace que la regla se configure para una *personalización aditiva conservadora,* conforme a la Figura 4.8. Se debe tomar en cuenta que actualizar una clase significa crear nuevas features y por lo tanto no es necesario especificar reglas *Redefine* adicionales para las features propias. En el caso de las enumeraciones, etiquetarlas con *UpdateEnum* significa que se pueden agregar nuevos literales en el rango (min..max).

El meta-modelo contiene algunas restricciones de integridad en OCL, asegurando una correcta especificación de las reglas de extensión. Estas restricciones de integridad verifican que si una referencia está etiquetada como *Redefine*, entonces la clase propietaria o una subclase debe etiquetarse como *Extend*, y (si no está vacía) la referencia *Redefine.owner* apunta a una subclase del *owner* de la referencia (y de manera similar para *Redefine.targets*). También se prohíbe mezclar intenciones de personalización, ya que pueden dar lugar a posibles conflictos. Por ejemplo, una clase ya etiquetada como *Delete* no puede etiquetarse también como *Update* o *Extend*. Esto es así, ya que si la clase realmente se elimina, entonces no se puede actualizar ni extender. Del mismo modo, una referencia etiquetada como *Redefine* no se puede etiquetar como *Delete*. Esto es así, ya que si se redefine, sería un conflicto eliminarla; mientras que si se elimina primero no se puede redefinir.

Finalmente, se debe tomar en cuenta que se podrían crear consultas en OCL para identificar el tipo de personalización definida, de acuerdo con la clasificación de la Tabla 4.1 Esto permitiría al diseñador de extensibilidad controlar la compatibilidad hacia delante/atrás de los modelos con respecto a posibles personalizaciones del meta-modelo.

Ejemplo: La Figura 4.15 muestra un ejemplo que ilustra algunas reglas de personalización en el área de modelado de procesos. La parte superior de la Figura 4.15 (etiqueta a) presenta un extracto del meta-modelo base, que contiene los conceptos para un lenguaje de modelado de procesos abstracto y neutro (independiente de un dominio concreto). La parte inferior (etiqueta b) muestra su extensión específica de dominio para el modelado de procesos de software. Las clases *ActivityKind* y *ResourceKind* tienen reglas adjuntas que fuerzan su extensión obligatoria con una clase abstracta. Por lo tanto, estas reglas fuerzan la creación de jerarquías específicas de dominio para *Activities* y *Resources*. La referencia *ActivityKind.needs* debe redefinirse una o más veces con cualquier cardinalidad, y las redefiniciones no pueden ser contenciones. En la extensión, esta referencia se ha redefinido tres veces en la clase *SoftwareEngineeringActivity*, modelando que las tareas de ingeniería

de software: 1) tienen cero o más *inputs* que son artefactos; 2) tienen una o más *outputs* que también son artefactos; y 3) son realizados por una o más objetos de tipo *Personnel*.

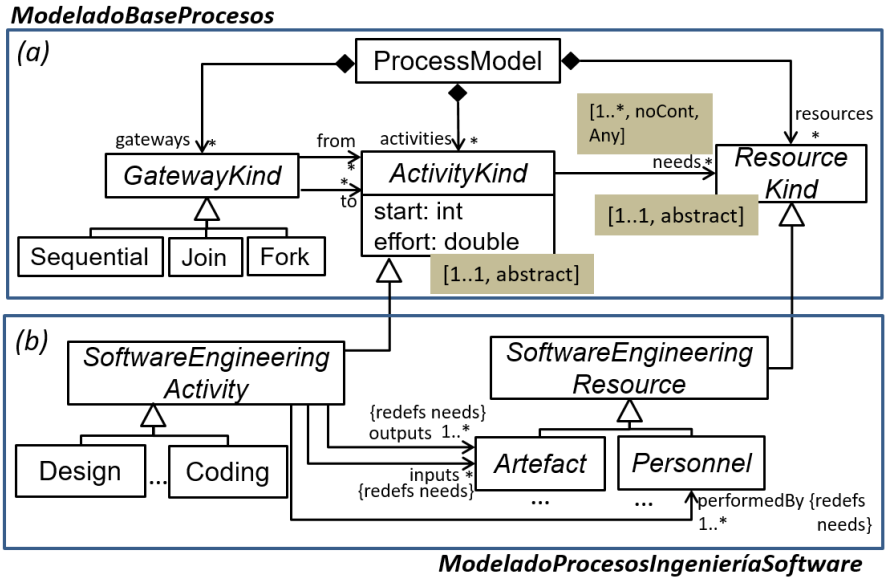


Figura 4.15: Meta-modelo base para modelado de procesos.

El destino de cada una de estas referencias (*Artefact*, *Personnel*) es compatible con el destino de la referencia *needs* (*ResourceKind*). Esta es la semántica predeterminada de la redefinición, cuando no se especifica ningún destino específico en la regla de *Redefine*. Esta redefinición hace que las referencia *needs* se vea como una referencia derivada, que resulta de la unión de referencias *inputs*, *outputs* y *performedBy*.

Por lo tanto, el ejemplo ha demostrado que es posible diseñar un meta-modelo base y controlar: (a) que solo se pueden refinar los tipos de actividad y recurso, (b) las extensiones se deben organizar utilizando una clase abstracta (como *SoftwareEngineernigActivity* y *ResourceKindActivity*), (c) la clase *GatewayKind* y sus subclases no pueden refinarse (p. ej., porque se supone una semántica fija para ellas) y (d) la referencia *needs* debe redefinirse en subclases de *ActivityKind* y *ResourceKind* obligatoriamente, para garantizar que cualquier tipo de actividad específica de dominio requiera algún tipo de recurso de dominio específico). Sin un mecanismo para definir estas reglas de extensibilidad, los patrones de extensión deseables deberían describirse en lenguaje natural, y el diseñador de la extensión tendría la responsabilidad de interpretar correctamente la descripción del lenguaje natural y de verificar que la extensión diseñada se define como se espera.

Una alternativa para la definición de un meta-modelo para expresar personalizaciones de meta-modelos sería la definición de expresiones OCL que especifiquen cómo se puede extender un meta-modelo particular. Sin embargo, esto daría lugar a expresiones complejas y engorrosas, que el diseñador de extensiones necesitaría crear para cada nueva especificación de extensión. Por ejemplo, usando el ejemplo anterior, una expresión OCL que compruebe que la clase *ActivityKind* debe extenderse obligatoriamente con una clase abstracta, se podría dar como:

```
EClass.allInstances()->one(c|c.abstract and c.eSuperTypes -> exists(c2|c2.name='ActivityKind'))
```

La idea es que dicha expresión debería ser verdadera en una extensión concreta válida (como *SoftwareEngineeringProcessModelling* en la Figura 4.15). Si bien la expresión se podría usar para verificar si una extensión es correcta, no se puede usar como guía para crear una extensión adecuada.

4.5 Herramientas de soporte

Para validar el enfoque propuesto, se ha diseñado una arquitectura que consta de un par de herramientas complementarias que funcionan juntas, como se ve en la Figura 4.16. Estas herramientas se denominan colectivamente TACO (**T**ool for met**A**model **C**ust**O**mization) y se han implementado como plug-ins de Eclipse, utilizando el framework EMF como infraestructura de meta-modelado. Las herramientas están disponibles en http://miso.es/tools/taco.html

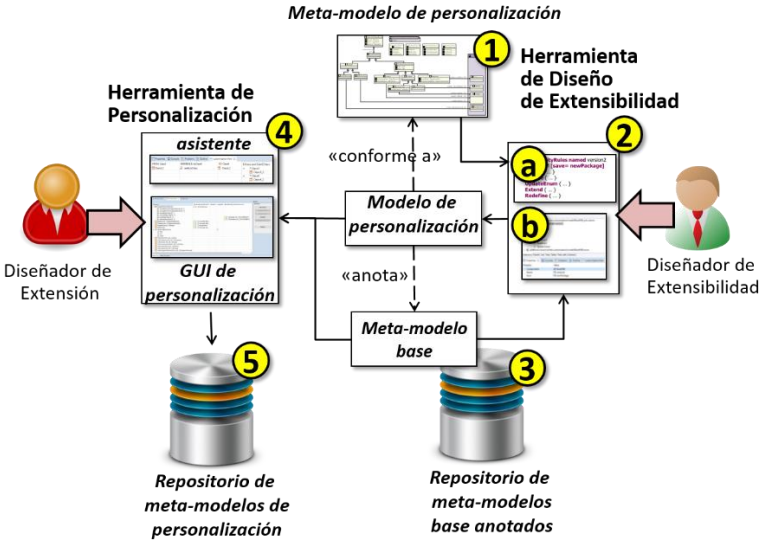


Figura 4.16: Arquitectura de personalización.

La arquitectura de personalización utiliza el meta-modelo de personalización (con etiqueta 1 en la Figura 4.16), que especifica las operaciones que se pueden realizar en el meta-modelo a personalizar. Las reglas de personalización se crean utilizando una *Herramienta de Diseño de Extensibilidad* (etiqueta 2), que permite usar una sintaxis textual (etiqueta a) o un editor basado en árbol (etiqueta b) para crear las reglas. En ambos casos, las reglas se crean como un modelo (instancia del meta-modelo de personalización) que anota el meta-modelo base. El meta-modelo base y las reglas de personalización se almacenan en un repositorio (etiqueta 3).

Una vez que se definen las reglas de extensión, el meta-modelo base puede extenderse de acuerdo con ellas. Para este propósito, se ha creado una herramienta (llamada *Herramienta de Personalización* en la figura, con la etiqueta 4), que permite ejecutar las operaciones del modelo de

extensión. Para esta tarea, el desarrollador es guiado por un asistente de personalización. El meta-modelo personalizado resultante se puede almacenar en un repositorio (etiqueta 5).

A continuación, se ofrece una descripción general de la herramienta de diseño de extensibilidad (sección 4.5.1) y la herramienta de personalización (sección 4.5.2).

4.5.1 Herramienta de diseño de extensibilidad

Las reglas de personalización se pueden especificar usando un editor basado en árbol (Figura 4.17) o usando un DSL textual creado con Xtext (Figura 4.18). La Figura 4.17 muestra un ejemplo de modelo de configuración con valores de ejemplo, donde:

- ***Components*** = baseMM (*.ecore), es el paquete que contiene el meta-modelo que se extenderá.
- ***Expand*** = true, controla si se pueden agregar nuevas clases arbitrarias.
- ***Name*** = baseMM_extend, es el nombre del conjunto de reglas de extensión.
- ***Save*** = theSame, es la opción de almacenamiento del meta-modelo extendido.
- ***Extend***, ***Redefine***, ***UpdateClass***, ***UpdateEnum***, ***DeleteClass*** y ***DeleteFeature***, son las reglas de personalización creadas.

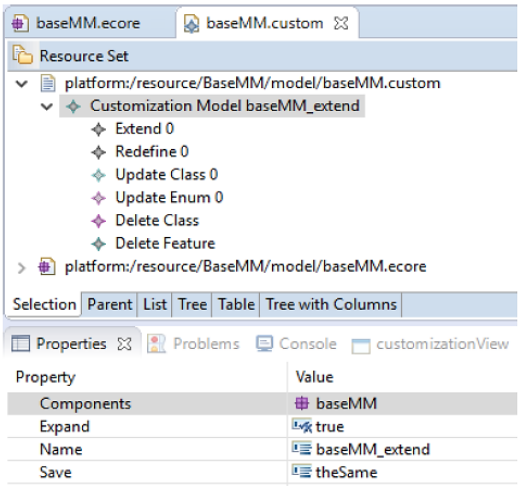


Figura 4.17: Modelo de personalización creado con el editor basado en árbol.

Si la especificación del modelo se realiza a través del DSL de extensibilidad (Listado 4.1), los valores de configuración inicial del modelo se deben asignar como en el caso anterior, así como la definición de las reglas usando la sintaxis del lenguaje.

```
1  ExtensibilityRules named baseMM_extend for baseMM [expandable save=theSame]
2  ExtClass { … }
3  RedReference { … }
4  UpdClass { … }
5  UpdEnum { … }
6  DelClass { … }
7  DelFeature { … }
```

Listado 4.1: Modelo de personalización creado con el DSL textual.

4.5.2 Herramienta de personalización

Una vez que se especifican las reglas de extensibilidad, se pueden usar a través de la herramienta de personalización. Su interfaz principal (Figura 4.18) está organizada en dos secciones. La sección izquierda muestra todos los elementos del meta-modelo base (clases, atributos, referencias y enumeraciones), que se pueden modificar según las reglas de extensión definidas en el modelo de extensión y se muestran en las columnas a la derecha, al lado de cada elemento del meta-modelo base. La sección derecha de la interfaz (panel etiquetado como "*OPTIONS*") ofrece las operaciones de extensión disponibles en forma de botones que se activan o desactivan según las reglas de extensión especificadas para cada elemento del meta-modelo y que se definieron previamente en el modelo de extensión. La herramienta hace disponibles ventanas de diálogo adicionales (Figura 4.19) para la correcta ejecución de cada una de las reglas definidas.

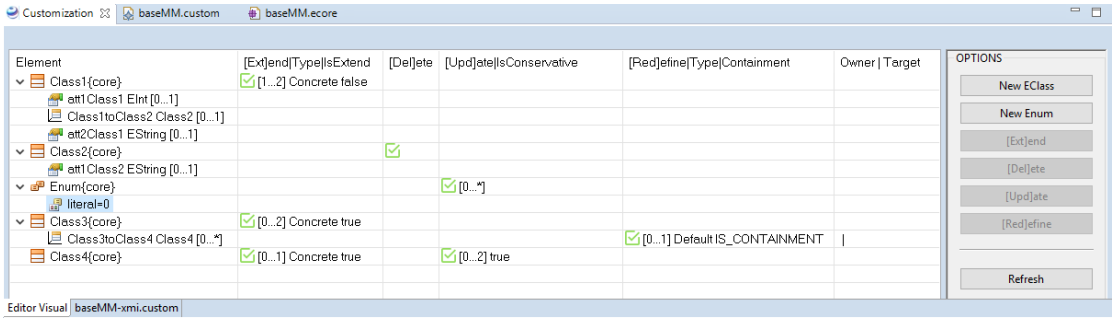


Figura 4.18: Interfaz gráfica de usuario (GUI) principal de la herramienta de personalización.

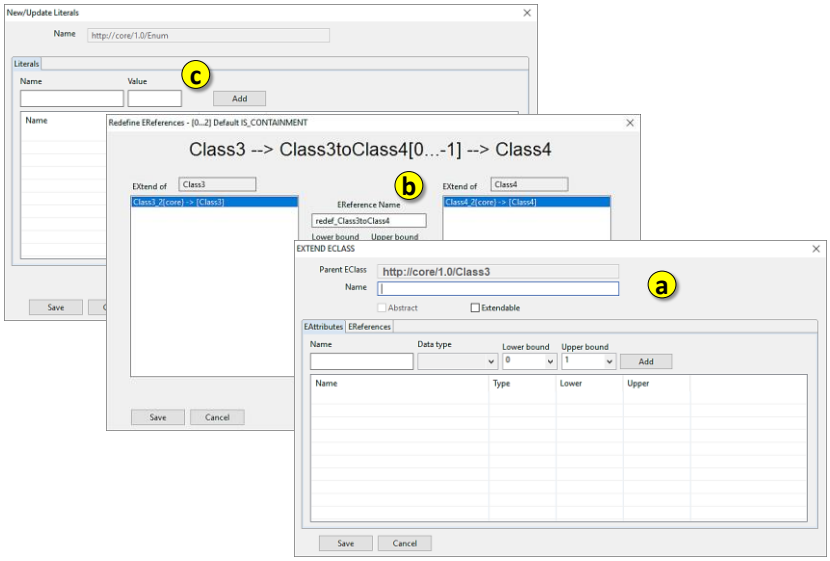


Figura 4.19: Ventanas de diálogo adicionales de la herramienta de personalización. (a) Extendiendo una clase. (b) Redefiniendo una referencia. (c) Añadiendo nuevos literales a un tipo de enumeración extensible.

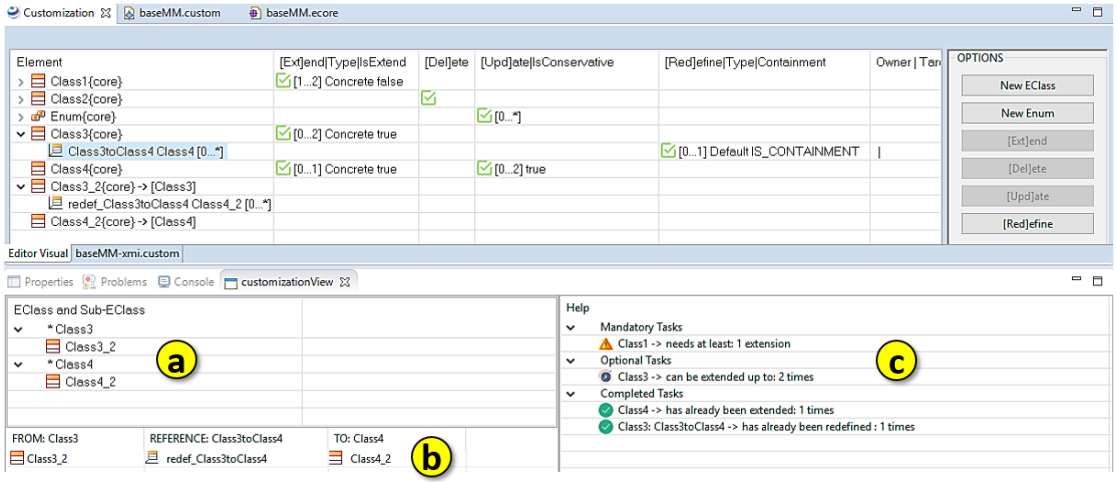


Figura 4.20: Vista general del asistente de personalización.

Para guiar la ejecución de las reglas de extensión, la herramienta incluye un *asistente de personalización* como una vista de Eclipse. El asistente proporciona ayuda para el proceso de extensión de clases y para el proceso de redefinición de referencias (Figura 4.20). El asistente aparece como una vista dividida en tres secciones en la parte inferior de la interfaz principal. En el proceso de extensión de clase, la selección de una clase de la columna *Element* de la interfaz principal muestra todas sus subclases, dando una idea general de la jerarquía de herencia (sección (a) en la Figura 4.20). En el proceso de redefinir las referencias, al seleccionar una referencia de la columna *Element* de la interfaz principal, se muestra la clase de origen y destino de la referencia, y se redefine el nombre de la referencia. También se muestran todas las subclases de las clases de origen y destino de la referencia a redefinir (sección (b) de la figura).

Finalmente, la sección (c) del asistente guía en la creación de la extensión. Esta contiene tres partes: tareas obligatorias (mandatory tasks), tareas opcionales (optional tasks) y tareas completadas (completed task). Las tareas obligatorias son personalizaciones requeridas por algunas de las reglas. Por ejemplo, una regla *Extend* con una cardinalidad de [1..2] es una tarea obligatoria. La Figura 4.20 muestra que *Class1* necesita extenderse al menos una vez.

Las tareas opcionales consisten en la extensión de clases o redefinición de referencias que tienen una cardinalidad con 0 como límite inferior o son tareas obligatorias que ya se han realizado. Por ejemplo, en la figura, si se extiende *Class1* una vez, extenderla otra vez se convierte en una tarea opcional. La extensión de *Class3* (que tiene cardinalidad [0..3]) es una tarea opcional desde el principio. Finalmente, las tareas completadas son acciones de personalización realizadas por el desarrollador.

4.6 Evaluación

A continuación, se evalúa el enfoque propuesto bajo dos perspectivas. En primer lugar, se analiza la aplicabilidad del enfoque de personalización evaluando en qué medida la extensibilidad del meta-

modelo es una necesidad común, para justificar la necesidad de la extensibilidad y se verifica si el enfoque puede usarse para expresar la extensibilidad sobre meta-modelos construidos por terceros. Segundo, se presentan varios casos de estudio relevantes, uno de ellos descritos en esta tesis, y los otros disponibles en la página web de la herramienta en http://miso.es/tools/taco.html. El caso de estudio descrito en la tesis es tomado del estándar DD de la OMG. En estos casos de estudio, se definen diferentes reglas de extensibilidad, que se pueden usar para definir diferentes tipos de extensiones permitidas.

4.6.1 Aplicabilidad del enfoque

Esta sección analiza hasta qué punto el enfoque de personalización es útil para expresar las reglas de extensibilidad que se encuentran en la práctica, y para comprender los escenarios prácticos que se benefician del control de la extensibilidad.

La Tabla 4.3 muestra algunos meta-modelos de diferentes fuentes y cómo se describen sus reglas de extensibilidad usando el enfoque propuesto. Se ha investigado dos repositorios diferentes: el zoo de meta-modelos ATL (disponible en: http://web.emn.fr/x-info/atlanmod/index.php?titleDmain _Page), y los estándares OMG disponibles públicamente [132]. Adicionalmente también se han analizado los meta-modelos publicados en conferencias y revistas en el área de modelado.

Se puede observar que las necesidades de extensibilidad se encuentran en los meta-modelos del zoo de ATL, en los estándares de la OMG y los documentos encontrados en la literatura [46, 47, 73, 131, 125, 135]. En estos trabajos, se observan tres tipos de necesidades, que se describen a continuación.

El primero viene dado por la necesidad de tener mecanismos tipo librería a nivel del meta-modelo, por ejemplo, para definir catálogos extensibles de tipos de actividades (como en Intalio BPMN), diversos tipos de políticas en el modelado de sistemas embebidos (como en MARTE), propiedades del documento (como en SACM [135]) o tipos de servicios (como en PetStore). En este escenario, varias clases o tipos de enumeración podrían etiquetarse como "abiertos" utilizando el enfoque propuesto.

En el segundo escenario, se proporciona un lenguaje base, que debe ampliarse con conceptos específicos de dominio. En algunos casos, la idea es incluir meta-modelos de tecnologías externas, mediante la extensión de algunas clases designadas. Los ejemplos típicos son meta-modelos en el área de ingeniería inversa (como KDM o ASTM) o integración de aplicaciones empresariales (como EAI) donde se espera que los meta-modelos base se amplíen con meta-modelos de lenguajes de programación. En algunos casos, el desarrollador de la extensión necesita seguir patrones complejos para la extensión (un caso típico es KDM [126]), que se describen sólo en lenguaje natural.

Finalmente, en el tercer escenario, existe la necesidad de crear instancias dinámicas de los tipos y sus instancias. Estos casos se modelan utilizando el patrón tipo-objeto [115, 38] o las transformaciones de promoción [51]. El uso del enfoque propuesto en la tesis permite usar subclases,

asegurando una extensión controlada de un meta-modelo base. Se verá en el siguiente capítulo una ampliación de este enfoque de extensión para emular el modelado multi-nivel.

Tabla 4.3: Meta-modelos extensibles.
(**CL=**Extensión de Clase, **EN=**Extensión de Enumeración, **RE=**Redefinición de Referencia).

Meta-modelo	Fuente	Reglas de extensibilidad usadas
EMF-DEVS	Sarjoughian [149, 148]	**CL**: eAtomic, eIOData, eInPort, eOutPort, eCoupled, eEOCoupling, eICoupling, eEICoupling. **RE**: outPorts, inPorts, values.
DSLs4BPM	Heitkötter [76]	**CL**: ProcessBuildingBlock, OrganizationalElement.
Security Policies	Mouelhi [115]	**CL**: PolicyType, RuleType, ElementType, HierarchicalElement. **RE**: ruleTypes, elementTypes, parameters.
TML	Drivalos [51]	**CL**: Trace, TraceLink, Context. **RE**: context, traceLinks.
CloudML	CloudML [38]	**CL**: Artefact, ServerPort, Node. **RE**:provided.
Case representation	ATL zoo	**EN**: ParameterKind.
Intalio BPMN	ATL zoo	**EN**: ActivityKind.
LQN 1.0	ATL zoo	**EN**: LinkType, PhaseType.
XPDL	ATL zoo	**EN**: GatewayTypeKind.
SWRC	ATL zoo	**CL**: Publication, Event, Person, Topic, Organization, Project.
PetStore	ATL zoo	**CL**: Services.
KDM	OMG	**CL**: KDMModel, KDMEntity, KDMRelationship. **RE**: ownedElement, ownedRelationship, outbound, inbound.
ASTM	OMG	**CL**: OtherSyntaxObject, Definition, DataType, Statement, Expression.
DD	OMG	**CL**: Edge, Shape, DiagramElement, Diagram, Style. **RE**: localStype, sharedStyle, source, target. **EN**: ColorKind.
EAI	OMG	**CL**: TDLangClassifier, TDLangComposedType, TDLangElement.
BQS 1.0	OMG	**CL**: BibliographicReference, Provider.
SACM 1.0	OMG	**CL**: DocumentProperty, EvidenceEvent, Provenance, EvidenceAttribute.
MARTE	OMG	**EN**: SynchronizationKind, PoolMgtPolicyKind, PoolMgPolicy, SchedPolicyKind, AccessPolicyKind, ProtectProtocolKind, PLD_Class, QueuePolicyKind, PLD_Technology, MutualExclusionResourceKind, ConcurrentAccessProtocolKind, NotificationResourceKind, ROM_Type, CacheType, LaxityKind, MessageResourceKind, ComponentState, PortType, ConditionType, OptimalityCriterionKind.

Como ejemplo de los beneficios que el enfoque aportaría en el segundo escenario, la Figura 4.21 muestra un extracto del meta-modelo EMF-DEVS (adaptada de [25]) y un pequeño fragmento de una extensión [115]. Este meta-modelo pertenece al área de modelado y simulación basada en componentes. Los autores definen un modelo base (EMF-DEVS) y se espera que el diseñador defina determinados tipos de componentes (con puertos de entrada y salida) y tipos de datos específicos

del dominio para los mensajes intercambiados mediante subclases de *eAtomic*, *eOutputPort*, *eInputPort* y *eIOData*.

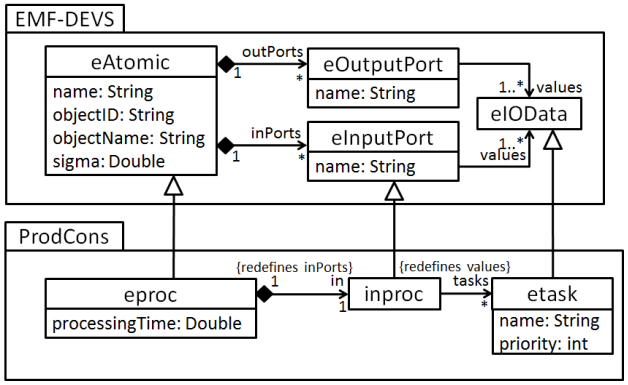


Figura 4.21: Extracto del meta-modelo EMF-DEVS, y una extensión típica.

Por lo tanto, no se espera que el diseñador modifique las clases existentes en EMF-DEVS, mientras que subclasifica *eAtomic* al menos una vez y redefine la asociación entre *eAtomic* y el *port* siempre que dicho *port* esté subclasificado. Sin el enfoque que se propone en esta tesis, tales reglas se expresarían con lenguaje natural, mientras que la TACO permite describirlas, guiando al desarrollador en la extensión y garantizando una extensión correcta de acuerdo con las reglas.

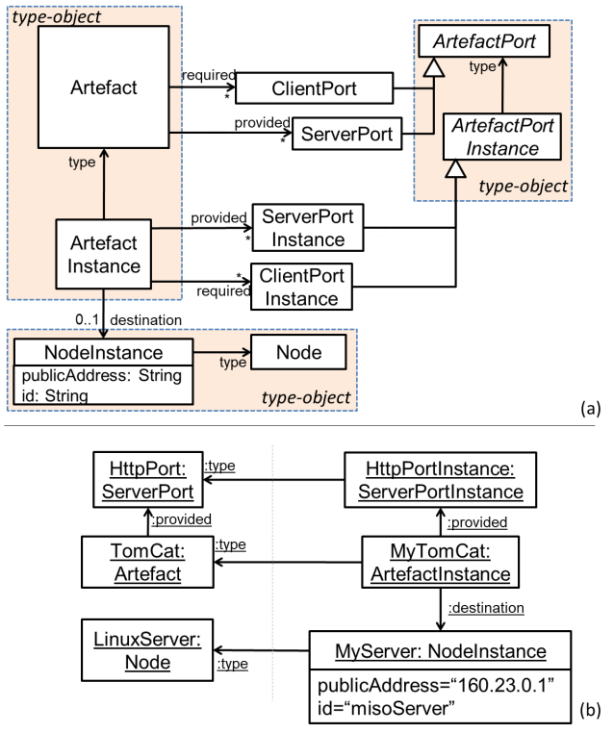


Figura 4.22: Meta-modelo CloudML.
(a) Extracto del meta-modelo; (b) Un modelo (instancia) CloudML.

Como ejemplo del tercer escenario, la Figura 4.22 tomada de [46], muestra una pequeña parte del meta-modelo de *CloudML* (parte a), y una pequeña instancia de ejemplo (parte b). Este meta-modelo se ha diseñado utilizando el patrón tipo-objeto (las instancias de dicho patrón se han señalado en la figura), ya que existe la necesidad de permitir a los usuarios la definición de nuevos tipos (por ejemplo, nuevos tipos de *Artefact* como *TomCat*) y ejemplos de estos (por ejemplo, instancias de *ArtefactInstance* como *MyTomCat*). Para obtener la extensibilidad, la decisión es modelar tipos (como *TomCat*) como objetos. Esto tiene el inconveniente de que, si se necesita que estos tipos definan nuevas propiedades (como ocurre en el caso del meta-modelo), estos atributos deberían emularse con una clase adicional en el meta-modelo.

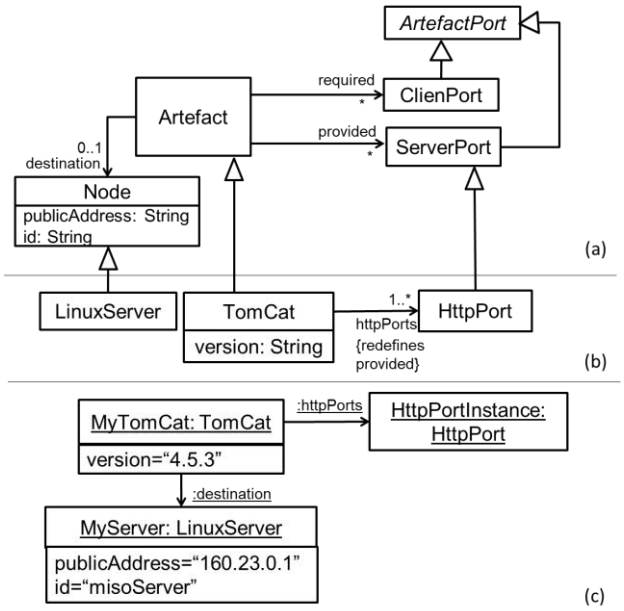


Figura 4.23: Extracto del meta-modelo CloudML
Reconstruido utilizando el enfoque basado en extensión (a), y una instancia (b, c).

En su lugar, se puede reconstruir CloudML utilizando el enfoque de extensibilidad, como se muestra en la Figura 4.23(a). Mediante el uso de reglas de extensibilidad, los usuarios de CloudML pueden extender de manera segura el meta-modelo base de forma modular, sin riesgo de realizar extensiones incorrectas. Como una clara ventaja del enfoque, el meta-modelo básico de CloudML se reduce en tamaño (5 clases y 3 asociaciones frente a 10 clases y 8 asociaciones), ya que cada aparición del patrón de type-objet se representa con una sola clase. Además, al modelar tipos en el nivel del meta-modelo, se puede agregar propiedades a esos tipos (por ejemplo, la versión para *TomCat*) y especificar cardinalidades para asociaciones redefinidas (por ejemplo, para *httpPorts*). Por lo tanto, el enfoque que propone esta tesis permite a los usuario el meta-modelado de dominio específico [44]. En la sección 5 se verá un mecanismo más potente para emular el modelado multi-nivel utilizando extensión.

4.6.2 Caso de Estudio: Definición de reglas de extensión para DI

Para evaluar la viabilidad del enfoque propuesto, en escenarios realistas, se han definido reglas de extensibilidad para varios meta-modelos (como se describe en la sección anterior), incluidos los meta-modelos estándar como DD y KDM de la OMG. Las reglas de extensión están disponibles en el sitio web de la herramienta http://miso.es/tools/taco.html. Esta sección describe con más detalle el proceso de extensión para DD v1.1 [124]. El objetivo de este estudio es definir las reglas de extensión del meta-modelo Diagram Interchange (DI) de la Figura 4.24, para crear una extensión del mismo para UML (similar al propuesto en [59]), adhiriéndose a la arquitectura de la Figura 4.1. El meta-modelo DI permite la especificación formal de la sintaxis concreta de un lenguaje de modelado para que las herramientas puedan intercambiar diagramas [111]. La extensión de DI permite definir varios aspectos gráficos que el usuario puede controlar, como la posición de los nodos y los puntos de enrutamiento de las líneas. Por lo que DI debe ser refinado para que pueda ser utilizado en un lenguaje de modelado particular, como UML.

Entre los elementos principales del meta-modelo DI se destaca la clase *Diagram*, que es un contenedor abstracto de elementos gráficos del diagrama. Los diagramas son elementos con un punto de origen en el sistema de coordenadas x-y. Sus elementos están dispuestos en relación con su punto de origen. La clase *DiagramElement* es el supertipo abstracto de todos los elementos de los diagramas, incluyendo los mismos diagramas. Cuando está contenido en un diagrama, los elementos del diagrama están dispuestos en relación con el origen del diagrama. La clase *Edge* es un elemento de diagrama que se representa como una polilínea, que conecta un elemento de diagrama de origen a un elemento de diagrama de destino, y se coloca en relación con el origen del diagrama. La clase *Shape* es un elemento del diagrama con límites dados que se establece en relación con el origen del diagrama. Finalmente, la clase *Style* contiene propiedades de formato que afectan la apariencia o el estilo de los elementos del diagrama, incluyendo el diagrama en sí.

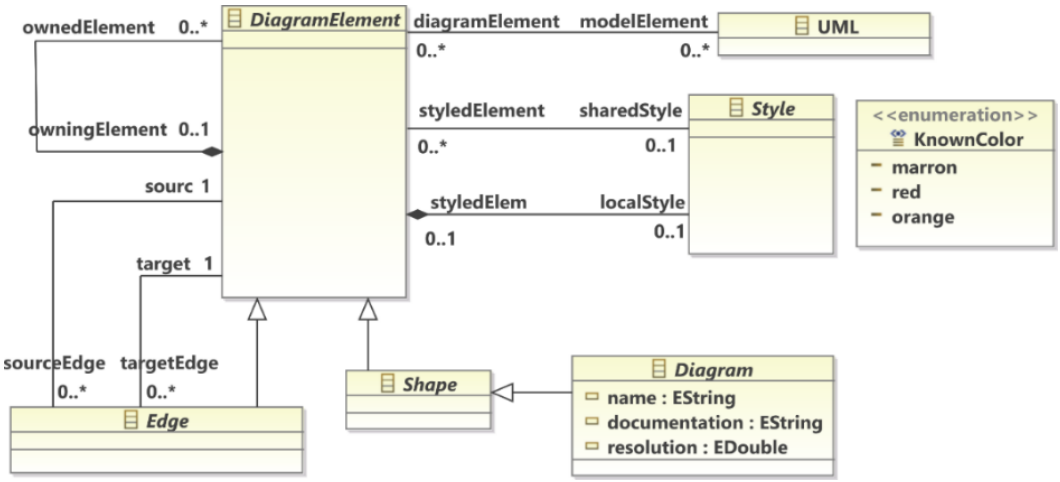


Figura 4.24: Extracto del meta-modelo Diagram Interchange (DI).

De acuerdo con el estándar, las extensiones de este paquete deben hacerse usando un patrón uniforme. El problema es que el estándar DD especifica estas reglas de extensión en lenguaje natural,

que son propensas a errores. Es decir, son convenciones que el desarrollador puede malinterpretar, no entender o simplemente ignorar. Además, los estándares de modelado de facto, como EMF, no son compatibles con la redefinición de las relaciones (que son necesarias en este caso).

La Tabla 4.4 muestra las reglas de extensión, extraídas de las indicaciones en lenguaje natural del estándar. La tabla tiene cuatro columnas, que indican el tipo de regla y el elemento de meta-modelo base al que se aplican, el valor de cada campo de la regla y un ejemplo de uso para crear una extensión para UML (véase Figura 4.26). El Listado 4.2 muestra las mismas reglas en formato de texto.

Tabla 4.4: Reglas de extensión para DI.

Modelo de personalización (valores iniciales)	
Propiedad	**Valor**
Components	DI
Expand	True
Name	DI_extend
Save	newPackage

Reglas de extensión		**Valor**	**Ejemplo de aplicación**
Extend: 1 *(Diagram)*	ExtensionKind	Concrete	-Clase *UMLDiagram* creada.
	Min..Max	1..1	
	NewIsExtend	true	
Extend: 2 *(DiagramElement)*	ExtensionKind	Abstract	-Clase *UMLDiagramElement* creada.
	Min..Max	0.. *	
	NewIsExtend	true	
Extend: 3 *(Edge)*	ExtensionKind	Anything	-Clase *UMLEdge* creada. *UMLEdge* hereda de *UMLDiagramElement*.
	Min..Max	1..*	
	NewIsExtend	false	
Extend: 4 *(Shape)*	ExtensionKind	Anything	-Clases *UMLCompartment, UMLShape* (con atributo *showClassifierShape*) y *UMLLabel* (con atributo *showQualified*) creadas. -*UMLShape* hereda de *UMLDiagramElement*.
	Min, Max	1..*	
	NewIsExtend	false	
Extend: 5 *(Style)*	ExtensionKind	Concrete	-Clase *UMLStyle* (con atributo *fontName* y *fontSize*) creada.
	Min..Max	1..1	
	NewIsExtend	true	
Redefine: 1 *(localStyle)*	RedefKind	Default	-Referencia redefinida *umlLocalStyle* entre *UMLDiagramElement* y *UMLStyle* creada.
	CompositionKind	Containment	
	Min..Max	1..1	
	owner	--	
	targets	--	
Redefine: 2 *(sharedStyle)*	RedefKind	Default	-Referencia redefinida *umlSharedStyle* entre *UMLDiagramElement* y *UMLStyle*.
	CompositionKind	noContainment	
	Min..Max	1..2	
	owner	--	
	targets	--	
Redefine:3 *(ownedElement)*	RedefKind	Default	-Referencia redefinida *umlOwnedElement* entre *UMLDiagram* y *UMLDiagramElement*.
	CompositionKind	Containment	
	Min..Max	1..*	
	owner	Diagram	
	targets	--	
Redefine:4 *(ownedElement)*	RedefKind	Default	-No usada.
	CompositionKind	Containment	
	Min..Max	0..*	

	owner	--	
	targets	--	
Redefine:5, 6 *(source, target)*	RedefKind	Default	-Referencias redefinidas *sourceShape* y *targetShape* entre *UMLEdge* y *UMLShape*.
	CompositionKind	noContainment	
	Min..Max	1..1	
	owner	--	
	targets	--	
UpdateEnum:1 *(KnownColor)*	Min..Max	0..10	-Se agregan colores a la enumeración *KnownColor.*

En total, se han considerado 5 reglas de extensión, 6 reglas de redefinición y 1 regla de actualización de enumeración. La primera regla especifica que la clase *Diagram* debe extenderse obligatoriamente por una clase concreta. Esto es necesario para definir un tipo específico de diagrama para el lenguaje de modelado considerado. Las clases *Shape* y *Edge* también deben refinarse obligatoriamente (por clases concretas) para definir tipos de *Shape* y *Edge* específicos del dominio. *DiagramElement* puede ampliarse opcionalmente, y la razón fundamental es que puede ser útil definir una clase base para los diferentes elementos específicos del dominio. Por lo tanto, se ha definido una extensión opcional por clases abstractas. Finalmente, *Style* también debe extenderse para definir estilos específicos de dominio.

```
1   ExtensibilityRules named DI_extend for DI [expandable save=newPackage]
2   ExtClass { DI.core.Diagram [1..1] concrete extensible}
3   ExtClass { DI.core.DiagramElement [0..*] abstract extensible}
4   ExtClass { DI.core.Edge [1..*] anything }
5   ExtClass { DI.core.Shape [1..*] anything }
6   ExtClass { DI.core.Style [1..1] concrete extensible }
7   RedReference { DI.core.DiagramElement.localStyle [1..1] isContain }
8   RedReference { DI.core.DiagramElement.sharedStyle [1..2] noContain }
9   RedReference {DI.core.DiagramElement.ownedElement [1..*] isContain owner DI.core.Diagram}
10  RedReference { DI.core.Edge.sourc [1..1] noContain targets (DI.core.Shape)}
11  RedReference {DI.core.Edge.target [1..1] noContain targets ( DI.core.Shape )}
12  UpdEnum { DI.core.KnownColor [0..10] }
```

Listado 4.2: Reglas de extensión utilizando el DSL de extensibilidad.

Las reglas de redefinición más interesantes son para *ownedElement* (reglas de redefinición 3 y 4). La regla 3 de redefinición especifica que la redefinición de *ownedElement* es obligatoria para las extensiones de *Diagram*. Esto es necesario para permitir la inclusión de formas y enlaces en el diagrama. La regla de redefinición 4 establece que dicha redefinición es opcional para las extensiones de otras subclases de *Diagram* (como *Node* y *Shape*), ya que eso solo sería necesario para modelar la contención de objetos de tipo *Shape*. Se debe notar que, si bien ambas reglas de redefinición se aplican a *ownedElement*, la regla 3 tiene prioridad cuando se extiende la clase *Diagram* (ya que se aplica específicamente a las subclases de *Diagram*). Finalmente, la regla *UpdateEnum* especifica que la enumeración *KnownColor* se puede actualizar con nuevos colores personalizados.

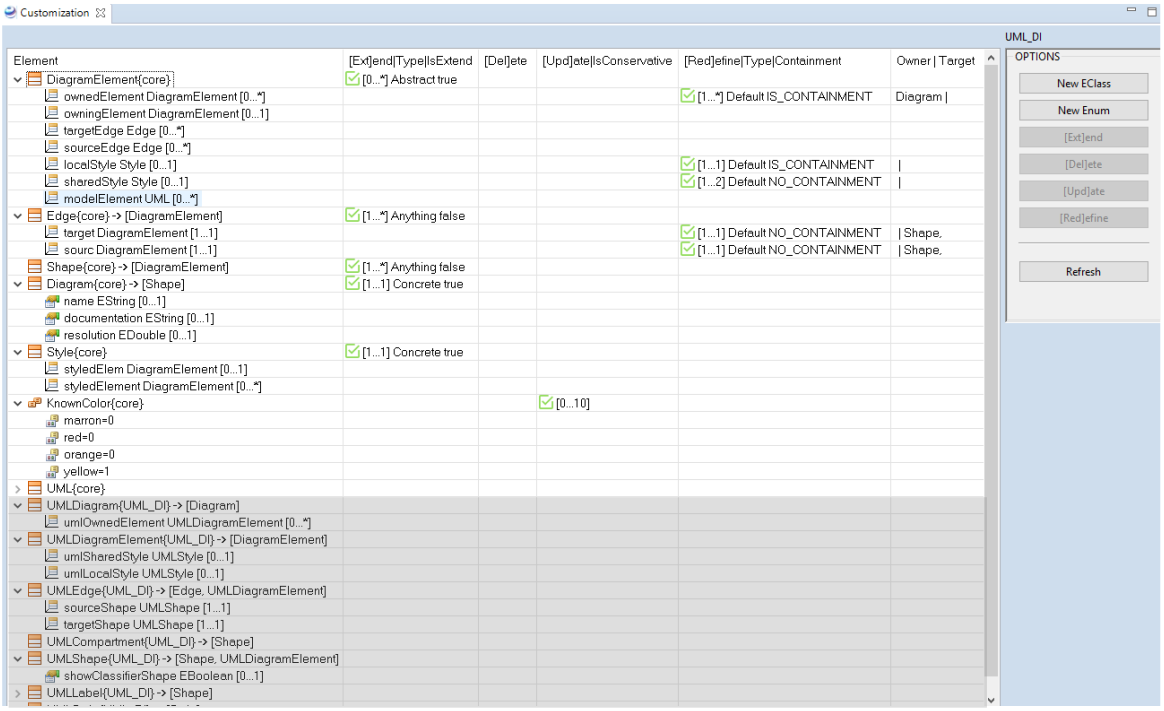


Figura 4.25: Herramienta de personalización.
Ampliación del meta-modelo DI para UML a través de la herramienta de personalización.

Una vez que se especifican las reglas de extensibilidad, se pueden usar a través de la interfaz principal de la herramienta de personalización (Figura 4.25). Como se indicó anteriormente, está organizada en dos secciones, la primera columna de la sección izquierda muestra todos los elementos del meta-modelo base, que se puede extender de acuerdo con las reglas de extensión definidas en el modelo de extensión y se muestran en las columnas de la derecha, al lado de cada elemento del meta-modelo base. Los elementos recién agregados se muestran en filas de colores en la parte inferior de la tabla.

La Figura 4.26 muestra la extensión resultante, construida con la herramienta de extensión (clases y referencias de color obscuro). Esta extensión es para el lenguaje UML, y es un extracto de lo definido en el estándar en sí mismo, y en [59]. Como se puede observar, se ha definido una extensión *UMLDiagram*, que redefinió *ownedElement* para almacenar *UMLDiagramElements*. *UMLDiagram_Element* es la clase base para las formas y enlaces UML.

En general, el uso del enfoque garantiza que cualquier extensión del meta-modelo DI se defina como lo esperan los diseñadores de DD. Además, el enfoque permite definir diferentes tipos de extensibilidad que se los ha llamado *perfiles de extensibilidad*. Por ejemplo, en el caso de DI, se pueden definir reglas de extensibilidad que dan como resultado diagramas en forma de grafos simples [23]. Para este perfil, es posible prohibir el anidamiento de tipos concretos de *Edges* y *Shapes* (permitido por la regla de redefinición 4 en la Tabla 4.4), y prohibir la conexión de *Edges* mediante *Edges* (permitido por las reglas de redefinición 5 y 6 en la Tabla 4.4). De esta forma, el diseñador de

extensiones puede elegir el perfil de extensión más apropiado, mientras que esos perfiles aún dejan espacio para la variabilidad en las extensiones.

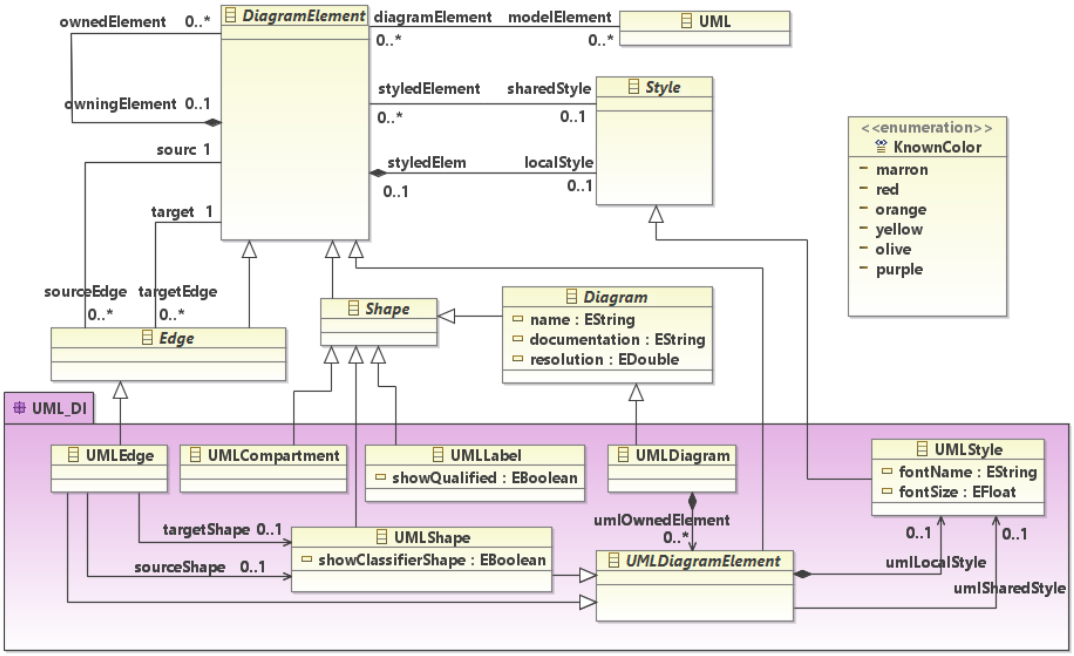


Figura 4.26: Extracto del meta-modelo DI extendido para UML.

La Tabla 4.5 muestra las reglas de redefinición modificadas para este perfil de extensibilidad, que además contendría todas las reglas definidas en la Tabla 4.4, excepto la regla de redefinición 4. La Tabla 4.5 contiene dos reglas, que se aplican a las referencias *source* y *target*. Ambas reglas se muestran juntas, y exigen que las redefiniciones de las dos referencias apunten a extensiones de *Shape* (evitando así apuntar a las extensiones de *Edge*).

Tabla 4.5: Perfil de extensibilidad *Diagrama de gráfico simple* para DI.

Reglas de extensión		Valor	Ejemplo de aplicación
Redefine:5, 6 (source, target)	RedefKind	Default	Referencias redefinidas *sourceShape* y *targetShape* entre *UMLEdge* y *UMLShape*.
	CompositionKind	noContainment	
	Min..Max	1..1	
	owner	--	
	targets	--	

Como ejemplo, la Figura 4.27 muestra una extensión de ejemplo usando el perfil de extensibilidad. Los *shapes* y *edges* anidados están prohibidos al permitir refinamientos de *ownedElements* para únicamente subclases de *Diagram* (se eliminó la regla de redefinición 4). De esta forma, ninguna subclase de *Node* o *SimpleEdge* puede redefinir la referencia *ownedElement*. Además, los tipos de *Shape* específicos del dominio no pueden conectarse con otros *Edges* debido a las nuevas reglas de redefinición 5 y 6 en el perfil. Debe indicarse que estas reglas requieren una multiplicidad de 1..1 para las referencias redefinidas (opción default), lo que deshabilita la posibilidad de crear *hiper-enlaces* (que conectan múltiples nodos origen a múltiples nodos destino).

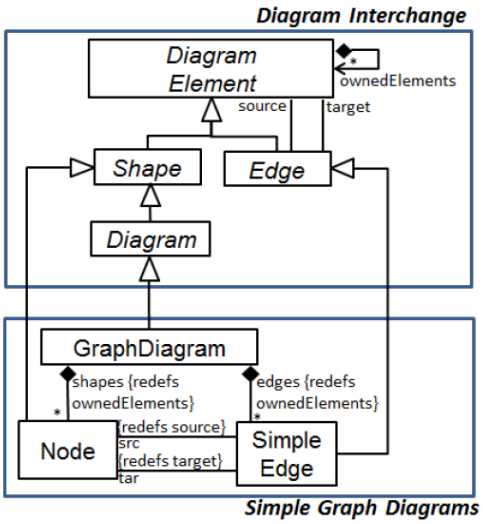


Figura 4.27: Uso del perfil de extensibilidad DI para diagramas tipo grafo.

En definitiva, se puede ver que el mecanismo propuesto para definir las reglas de extensibilidad puede guiar al diseñador de extensiones en la construcción de las extensiones correctas, como espera el estándar DD. Además, el enfoque de personalización propuesto permitiría la creación de diferentes perfiles de extensibilidad, de modo que se puedan elegir diferentes estilos de extensión.

4.7 Resumen y conclusiones

En este capítulo se analizaron tres escenarios que requieren la personalización de meta-modelos. El primer escenario se relaciona con la extensión del meta-modelo, donde se señala que existen meta-modelos que intencionalmente han sido diseñados para ser extendidos para el desarrollo de sistemas más complejos a través de la subclasificación y redefinición. El segundo escenario analizado se refiere a la adaptación de DSLs. Este escenario señala que algunos DSLs no han sido diseñados precisamente desde cero, sino que se han crean extendiendo o adaptando un lenguaje base a un dominio determinado. Por tanto, este escenario admitiría, además de las operaciones señaladas en el primer escenario, operaciones de actualización y simplificación. En el caso de actualización, se podrían actualizar clases añadiendo nuevos atributos y referencias, en el caso de las enumeraciones, añadiendo nuevos literales. Las operaciones de simplificación permitirían eliminar ciertos elementos opcionales del meta-modelo, como por ejemplo clases, atributos y referencias. Finalmente, el tercer escenario se refiere al modelado multi-nivel, una extensión conservadora del enfoque de modelado de dos niveles, pero con ciertas ventajas en el diseño de la extensibilidad de meta-modelos, ya que las necesidades de extensibilidad se pueden expresar como instanciación.

En este capítulo también se ha realizado una clasificación de los tipos de personalización de meta-modelos, donde se discute la compatibilidad de los modelos existentes con respecto al meta-modelo personalizado y viceversa.

Adicionalmente, en este capítulo se describió el mecanismo de control de personalización de meta-modelos propuesto, que ha sido implementado dando lugar a la herramienta llamada TACO. Finalmente, se presentó la evaluación de la propuesta, realizada en base a dos perspectivas. La primera, en base al análisis de la aplicabilidad del mecanismo de personalización propuesto, evaluando la medida en que la extensibilidad de meta-modelos es requerida en meta-modelos construidos por terceros. Adicionalmente, la herramienta ha sido evaluada mediante la realización de varios casos de estudio, donde se verifica el funcionamiento e idoneidad del mecanismo propuesto.

5 Modelado multi-nivel basado en extensión

Como resultado del análisis de los diferentes enfoques y herramientas de modelado multi-nivel de la sección 3.2.1 y 3.2.2, en este capítulo se propone un nuevo enfoque para modelado multi-nivel basado en extensión que incorpora múltiples meta-niveles dentro de un meta-modelo, y una herramienta llamada TOTEM que implementa estos conceptos. A través de varios ejemplos, se ilustran las capacidades del enfoque propuesto y de la herramienta y sus beneficios en términos de interoperabilidad con los frameworks de modelado estándar. Finalmente, se presenta una evaluación empírica y analítica.

5.1 Introducción

Las limitaciones de los diferentes enfoques de modelado de dos niveles han provocado un gran interés en las tecnologías de modelado multi-nivel dentro de la comunidad MDE. Este interés de demuestra por ejemplo por MULTI, una serie de workshops dedicados al modelado multi-nivel [117], un seminario Dagstuhl [2] y la gran cantidad de herramientas de modelado multi-nivel recientemente propuestas, como DeepTelos [83], DeepJava [98], Deep-Ruby [120], DPF Workbench [100], Dual Deep Modeling [121] , FMMLx [37, 64], Kite [71], Melanee [5], MetaDepth [40], ML2 [31, 61], MultEcore [105], OMLM [79] , y SLICER [154], algunas de las cuales han sido analizadas en la sección 3.2.2.

Muchas de estas herramientas ofrecen funcionalidades sofisticadas para modelar en múltiples meta-niveles, como la mutabilidad y durabilidad de atributos [68], referencias profundas (deep

references) [43], potencias duales (dual potencies) [121], o soporte al rediseño (refactorización) [42]. Sin embargo, actualmente existe una falta de interoperabilidad entre las herramientas de modelado multi-nivel y el modelado convencional de dos niveles. Es de considerar que para una adopción más amplia del modelado multi-nivel dentro de la comunidad MDE, la interoperabilidad con los estándares de facto a dos niveles, como EMF, debe mejorarse, para permitir una migración más fácil de soluciones de dos niveles a multi-nivel y viceversa. Esto facilitará la reutilización de los meta-modelos existentes como base para crear modelos multi-nivel y permitirá el uso de herramientas desarrolladas para modelado de dos niveles, para modelos multi-nivel.

Como se ha señalado en la sección 3.3, a nivel práctico, se observa que la interoperabilidad con el meta-modelado a dos niveles es uno de los factores que deben mejorarse. Por lo tanto, con este objetivo en mente, se propone un nuevo enfoque para el modelado multi-nivel, que emula objetos en el nivel de meta-modelo, y donde la creación de instancias se emula con herencia. Mientras que otros enfoques se basan en emular clases con objetos [146], en meta-modelos multi-nivel dedicados [5, 40], o en transformaciones de promoción [105], el enfoque de modelado multi-nivel que esta tesis no se ha explorado hasta ahora. La herramienta implementada llamada TOTEM, que significa "**T**ool f**O**r mul**T**il**E**vel **M**odeling" tiene varios beneficios. Primero, permite una integración más directa con enfoques y frameworks de modelado de dos niveles como EMF. Esto es así ya que los modelos multi-nivel pueden integrarse con pequeñas modificaciones dentro de un meta-modelo, preservando la semántica del modelo multi-nivel. De esta manera, exportar un modelo multi-nivel a dos niveles es más directo que otros enfoques, mientras que el modelado a nivel de instancia se puede hacer usando herramientas estándar de EMF. Además, estos modelos a nivel de instancia pueden manipularse con lenguajes de transformación y herramientas diseñadas para el modelado en dos niveles, como ATL [85], Epsilon [96] o Acceleo, una implementación del estándar MOFM2T OMG [130]. En segundo lugar, el enfoque admite la reutilización de los meta-modelos existentes de EMF para construir un modelo multi-nivel. Esto es así porque el enfoque se basa en meta-modelos estándar anotados con inscripciones de varios niveles.

5.2 Enfoque basado en extensión del meta-modelo para modelado multi-nivel

El enfoque propuesto se basa en la observación de que los modelos multi-nivel pueden integrarse en dos niveles mediante el uso de herencia en lugar de la instanciación, como se señaló en la sección 2.1.7.2 (enfoque de extensión del meta-modelo), así como en el ejemplo de la Figura 4.23. Este enfoque tiene la ventaja de mantener la compatibilidad con estándares, como EMF, pero requiere mejorar el control de cómo se puede realizar la extensión de un meta-modelo, así como emular la faceta de instancia en las clases. Al agregar dichos mecanismos de control, se evita la creación de extensiones incorrectas o modificaciones prohibidas del meta-modelo base. La Figura 5.1 muestra un esquema del enfoque propuesto.

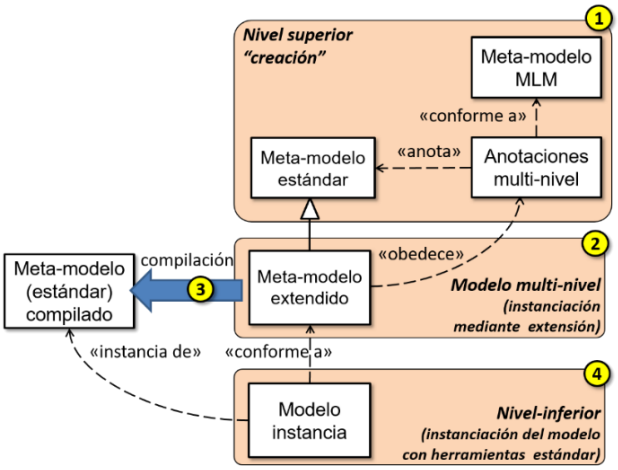


Figura 5.1: Enfoque de extensión del meta-modelo para el modelado multi-nivel.

En la primera fase, a un meta-modelo estándar (etiqueta 1 en la Figura 5.1) se le asigna varias anotaciones multi-nivel. Dichas anotaciones se definen como un modelo, conforme al meta-modelo MLM. Luego (etiqueta 2), la creación de instancias de dicho meta-modelo base se emula a través de la extensión del meta-modelo. Las posibles extensiones están reguladas por las anotaciones multi-nivel definidas. Se debe tener en cuenta que, aunque el nivel inferior también se puede emular con subclases (como lo señalan otros investigadores, como [15]) se ofrece la opción de "compilar" el modelo multi-nivel a un meta-modelo estándar (etiqueta 3). Esta compilación tiene en cuenta la semántica especial de los modelos multi-nivel, como la codificación de las diferencias entre la instanciación y la herencia, y el establecimiento como abstracta de aquellas clases con una potencia mayor que 1. Esto da como resultado un meta-modelo estándar que se puede instanciar utilizando herramientas estándar. Cualquier modelo instancia (etiqueta 4) del meta-modelo compilado también es conforme al modelo multi-nivel. Este enfoque tiene la ventaja de que las herramientas MDE estándar (por ejemplo, para transformaciones de modelos) se pueden usar para manipular el modelo de nivel inferior.

Las siguientes subsecciones describen cada uno de los pasos del enfoque propuesto.

5.2.1 Anotación de meta-modelos estándar con anotaciones multi-nivel

La Figura 5.2 muestra el meta-modelo que se utiliza en el enfoque propuesto (denominado meta-modelo MLM en la Figura 5.1) para aplicar anotaciones de modelado multi-nivel a los elementos del meta-modelo estándar. En general, se pueden aplicar anotaciones a cuatro tipos de elementos: meta-modelos (*EPackage* en EMF), clases (*EClass* en EMF), referencias (*EReference* en EMF) y atributos (*EAttribute* en EMF). De esta modo, el meta-modelo MLM contiene clases que anotan estos elementos. Como la propuesta soporta la caracterización profunda, todos los elementos pueden anotarse con una potencia (todas las clases de anotación heredan de *DeepElement*). Si el valor de potencia es -1, significa que la potencia es ilimitada.

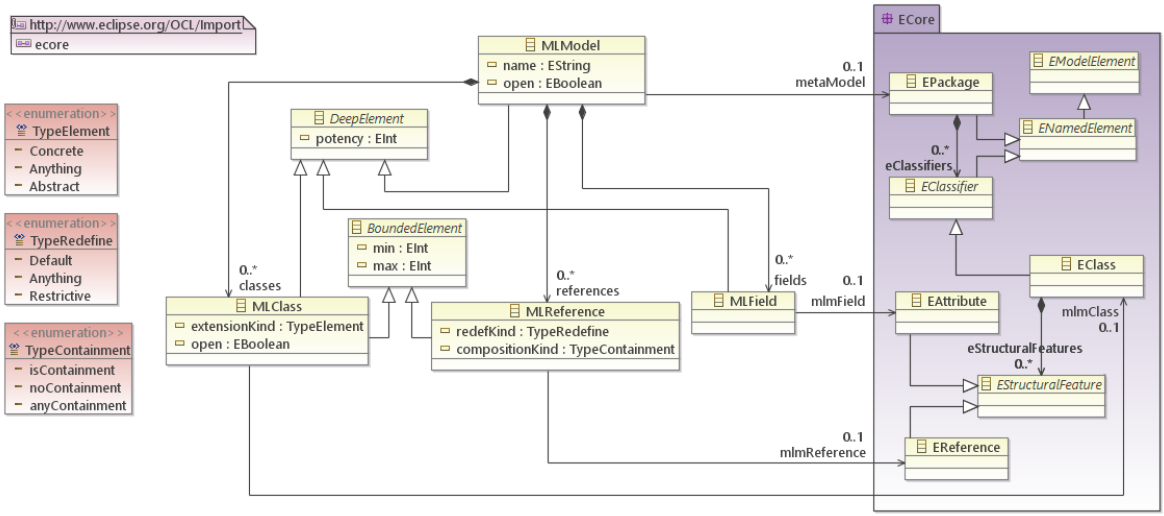


Figura 5.2: Meta-modelo para crear anotaciones multi-nivel.

Los meta-modelos están anotados con objetos *MLModel*. El EPackage anotado se referencia a través de la referencia metaModel. Además, las anotaciones del meta-modelo pueden incluir un nombre opcional para la especificación y un indicador *open* para controlar si es posible agregar extensiones lingüísticas (objetos sin tipo) en las instancias del meta-modelo.

Los objetos *MLClass* anotan *EClasses* a través de la referencia *mlmClass*. La anotación permite controlar la potencia de la clase; el rango de instancias que puede tener el clabject en el siguiente nivel (intervalo [min..max]); si las instancias pueden extenderse lingüísticamente con nuevos campos (si es *true* es open); y si estas instancias deben ser abstractas, concretas o cualquiera de las dos.

Las referencias del meta-modelo pueden ser anotadas con objetos *MLReference*. Estos admiten declarar una potencia para la referencia, un rango de instancias permitidas para la referencia (un intervalo [min..max]), una indicación de las declaraciones de cardinalidad permitidas para las instancias de referencia (*redefKind*), y una indicación de si las instancias de referencia pueden/deben ser etiquetadas como composiciones (*compositionKind*). Se debe considerar que las *EReferences* tienen dos cardinalidades: la definida en la referencia misma y la presente en la anotación (min, max). La última gobierna cuántas instancias de la referencia se pueden crear en el próximo meta-nivel. La cardinalidad definida de la referencia, junto con el flag *redefKind*, restringe la cardinalidad que se puede asignar a las instancias de la referencia. De esta manera, si el valor de *redefKind* es el valor *Default* (predeterminado), la cardinalidad de la instancia debe ser la misma que la cardinalidad del tipo de referencia. Si *redefKind* es *Restrictive* (restrictivo), la cardinalidad de la instancia debe estar contenida en la cardinalidad del tipo. Por ejemplo, si la cardinalidad de la referencia base es 0..2, la referencia de instancia puede declarar intervalos como: 0..1, 0..2, 1..2, 1..1 y 2..2. Finalmente, si *redefKind* es *Anything* (cualquiera), la cardinalidad de la instancia puede ser cualquier valor. Finalmente, los atributos *EAttributes* pueden anotarse con objetos *MLAttribute*, que contienen información sobre la potencia.

El meta-modelo también incluye algunas restricciones de integridad, indicando que la potencia de *MLModel* debe ser mayor que la potencia de sus elementos contenidos, o ilimitada (mientras que la potencia de un elemento no puede ser ilimitada si la potencia de *MLModel* no lo es). De manera similar, la potencia de un objeto *MLClass* debe ser mayor que la potencia de los objetos *MLAttribute* y *MLReference* que anotan las características de la clase. De manera similar a [43], también se exige que la potencia de los *clabjects* sea mayor o igual que la potencia de los *clabjects* de los que estos heredan, directa o indirectamente. Con respecto a la cardinalidad, *BoundedElement.min* debe ser positivo o cero, mientras que *BoundedElement.max* debe ser mayor que *BoundedElement.min*, o negativo (para representar una cardinalidad ilimitada). Por último, se exige que la cardinalidad de un clabject sea más amplia o igual que el intervalo de la cardinalidad mínima de cada subclabject (a través de la relación de herencia), y la suma de las cardinalidades máximas de cada subclase (donde, * más cualquier cosa resulta en *). Por ejemplo, si un clabject *A* tiene dos subclases *B* y *C* con cardinalidades [1..3] y [2..4], entonces la cardinalidad de *A* debería ser más amplia o igual a [1..7] (por ejemplo, puede ser [1..7], [0..8], [0.. *], etc.).

Con la finalidad de mostrar el modo de funcionamiento del enfoque de propuesto, se utilizará el ejemplo que permite definir tipos de componentes, del estilo de los lenguajes *componente-conector*, analizado en la sección 2.1.7. La Figura 5.3 muestra el meta-modelo de un lenguaje con anotaciones multi-nivel (mostradas como cajas de color obscuro adjuntas a cada elemento del meta-modelo). El modelo en sí está anotado con un objeto *MLModel* que indica que tiene potencia 2 y es *open* (por lo que se permiten extensiones lingüísticas en el siguiente nivel). Todos los elementos tienen potencia 2, excepto el atributo *Connector.delay*, que tiene potencia 1. Todos los elementos tienen cardinalidad 0..* (esto se omite, ya que es el valor predeterminado) excepto *Component* y *Connector*, para los cuales se requieren al menos una instancia. El enfoque soporta el control de la naturaleza de la composición de las instancias de referencias. De esta manera, las anotaciones requieren que las instancias de *Component.pTypes* sean composiciones, y que las instancias de cualquier otra referencia no puedan ser composiciones. Las cardinalidades de las instancias de referencia pueden ser cualquiera, excepto las instancias de *Connector.from*, para las que se requiere una cardinalidad igual a *Connector.from*. Todas las clases son cerradas (las instancias no admitirán extensiones lingüísticas), excepto las instancias de *Message*.

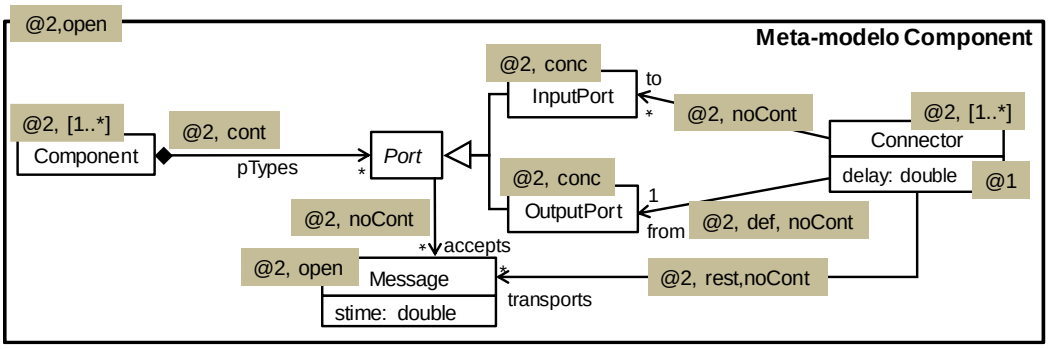


Figura 5.3: Meta-modelo para el ejemplo con anotaciones multi-nivel.

5.2.2 Instanciación de modelos multi-nivel mediante extensión.

El enfoque propuesto reemplaza la creación de instancias por herencia, y por lo tanto emula objetos con clases. Cuando se crea una instancia del meta-modelo base, las subclases creadas también deben anotarse con anotaciones multi-nivel (estas anotaciones se añaden automáticamente). Esto se ilustra en la Figura 5.4, que muestra la instanciación multi-nivel del modelo de la Figura 5.3. Como puede observarse se asignan valores a las instancias de atributo (slot) para las clases que usan anotaciones (por ejemplo, *MChannel.delay*), y se toma en consideración las anotaciones en los objetos padre como una guía para crear las instancias. Por ejemplo, como la referencia *Connector.from* se anota con el valor *default*, significa que sus instancias (*MChannel.from*) deben tener la misma cardinalidad declarada.

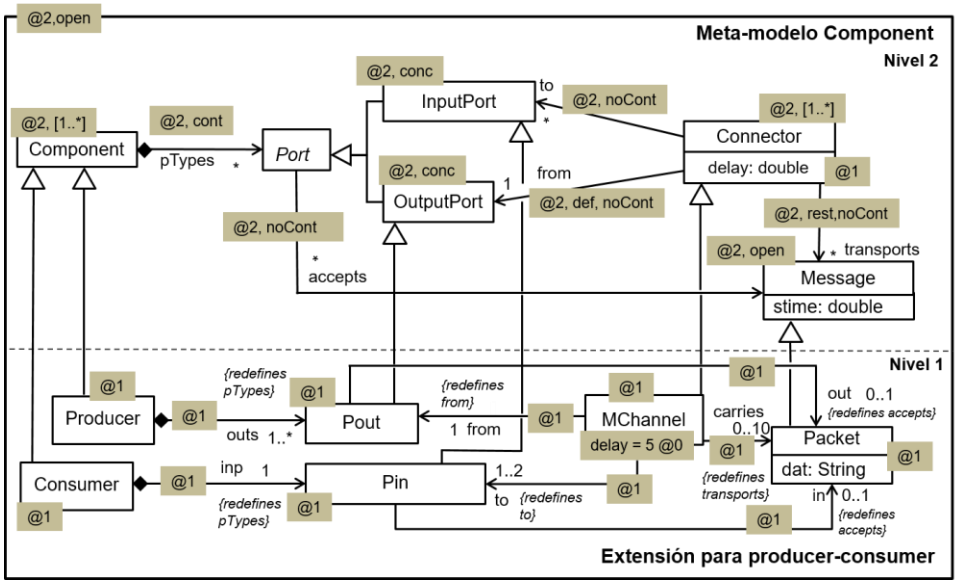


Figura 5.4: Instanciación multi-nivel del modelo.

5.2.3 Compilando el modelo multi-nivel

Opcionalmente, el desarrollador puede compilar el modelo multi-nivel a un meta-modelo estándar. Este proceso elimina las anotaciones multi-nivel, codificando la semántica de las diferentes construcciones multi-nivel en el meta-modelo estándar (incluida la diferencia semántica entre la herencia y la instanciación). Este proceso de compilación toma como referencia el procedimiento descrito en [70]. De esta manera, la compilación tiene cuatro pasos:

- ***Compilación de clabjects:*** cada clase en el nivel 2 o superior se hace abstracta, mientras que las clases con potencia 0 también se vuelven abstractas. Esto es necesario para evitar la creación de instancias de dichos clabjects en el nivel 0. En el primer caso, se estaría saltando niveles al realizar la instanciación. En el segundo, se estrían instanciando clabjects con potencia 0.

Las Figuras 5.5(a) y (b) ilustran estos dos casos. En ambos, el modelo multi-nivel aparece a la izquierda de la flecha y el meta-modelo estándar compilado a la derecha. En la Figura 5.5(a), no se puede establecer la clase *A* como concreta, ya que se podría instanciar en el nivel 0, lo que no está permitido en el modelo multi-nivel. En dicha figura, la clase *B* se vuelve concreta, porque el modelo multi-nivel permite su instanciación. En la Figura 5.5(b), ambas clases *A* y *B* se vuelven abstractas, ya que no se podría instanciar ninguna de ellas en el nivel 0. En el ejemplo de la Figura 5.3, se configuran las clases *Component*, *Port*, *Message*, *InputPort*, *OutputPort* y *Connector* como abstractas.

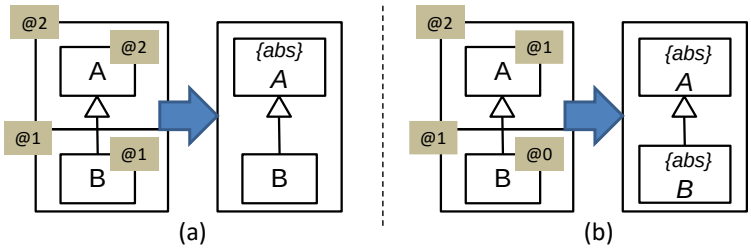


Figura 5.5: Compilación de clases con anotaciones de potencia.
(a) Clases con la misma potencia y nivel; (b) Clases con potencia 0.

- ***Compilación de atributos*:** los atributos (con tipo de dato primitivo) simplemente se copian. Sin embargo, dependiendo de su potencia, es posible que deban establecerse como estáticos y que sus valores se redefinan. En particular, los atributos cuya potencia es inferior al nivel del modelo, deben establecerse como estáticos (atributo de clase). Esto es necesario para emular que dichos valores de atributo se asignan a nivel de clase, y es el mismo para todas las instancias de dicha clase. Debe tenerse en cuenta que, en sistemas de meta-modelado que no admitan atributos de clase (como EMF), el valor se puede emular con un invariante OCL, o como un atributo derivado, pero en tal caso, los atributos serían de solo lectura.

  Por ejemplo, *delay* tiene potencia 1, mientras su propietario el clabject *Connector* tiene potencia 2, y está ubicado en un modelo con nivel 2. De esta manera, el atributo se compila a un atributo estático, donde el valor se redefine en la subclase de *MChannel*, como se muestra en la Figura 5.6(a). En el caso de EMF, el valor del atributo de clase se emularía con un invariante OCL en el contexto de *MChannel* (expresado *self.delay = 5*) o con un atributo derivado. La Figura 5.6(b) muestra otro caso donde el atributo (*stime*) tiene potencia 2, y el nivel del clabject contenedor también es 2. Por lo tanto, se compila como un atributo de instancia.

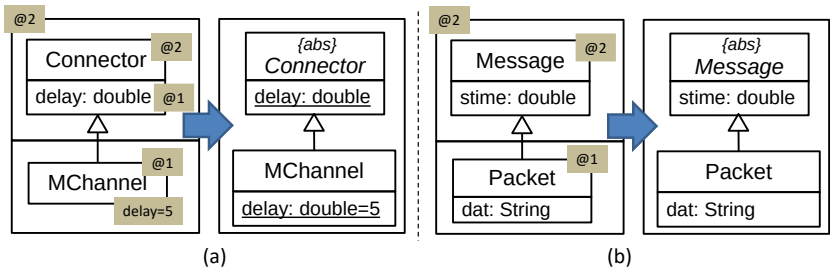


Figura 5.6: Compilación de atributos.
(a) Atributos con una potencia inferior al nivel; (b) Atributos con la misma

potencia que el nivel.

- ***Compilación de referencias:*** similar a los *clabjects*, para evitar instanciaciones ilegales, se eliminan las referencias de nivel 2 o superior, mientras que las referencias de nivel 1 con potencia 0 también se eliminan. Para evitar perder información (y para poder escribir restricciones OCL que consideren esos elementos), las referencias eliminadas se sustituyen por operaciones de consulta que devuelven objetos de las clases especificadas. En sistemas de meta-modelado compatible con OCL, el código de la operación se puede dar a través de una expresión OCL.

  La Figura 5.7 ilustra esta compilación. La referencia *A.r1* se elimina y se sustituye por una operación de consulta abstracta *r1* en el meta-modelo compilado. Entonces, en consecuencia, dicha operación se sobreescribe en *A1* y *A2*. Debe indicarse que se aprovecha el retorno covariante en la sobreescritura de métodos. De esta manera, como todas las instancias de *r1* en *A1* devuelven objetos de tipo *A2*, el método *A1.r1()* devuelve una colección de *A2*. En cambio, hay dos instancias de *r1* en *A2*, es decir, *r3* (con destino *A1*) y *r4* (con destino *A2*). Como no apuntan a la misma clase, se toma la superclase común más específica de todas ellas, que en este caso es *A*. Como hay dos instancias, entonces *A2.r1()* devuelve la unión de tales instancias. Como se puede observar, el objetivo es evitar la creación de instancias de *r1*, al tiempo que se simula la redefinición de referencias mediante la sobreescritura de operaciones. Como las referencias *r2*, *r3* y *r4* tienen potencia 1, se pueden instanciar normalmente en el nivel 0.

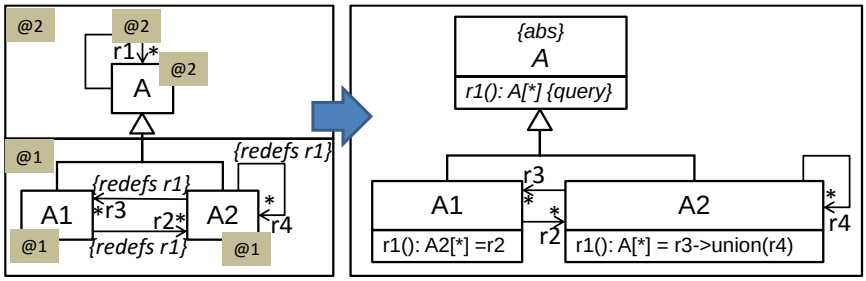


Figura 5.7: Compilación de referencias.

- **Compilación de anotaciones:** los clabjects sin tipo creados por el desarrollador en cualquier meta-nivel pueden contener anotaciones. La compilación sólo necesita abordar las anotaciones de cardinalidad de clase. Estos producen una restricción OCL que restringe el número de instancias de la clase. Dicha restricción OCL se agrega a la clase raíz. Las clases raíz son comunes en meta-modelos EMF. De esta manera, en los modelos EMF, se espera que la clase raíz se instancie una sola vez, y que contenga todas las demás clases directa o indirectamente a través de referencias de contención. Se discutirá más sobre las clases *raíz* en la sección 5.3.

Como ejemplo, la Figura 5.8 muestra la compilación del modelo multi-nivel de la Figura 5.4 utilizando el procedimiento que se acaba de describir. Los clabjects con potencia 2 en el nivel 2 se compilan a clases abstractas, mientras que sus instancias se convierten en sus subclases. Las referencias en el nivel 2 se eliminan y reemplazan por métodos abstractos, devolviendo objetos de la clase y cardinalidad apropiadas (tomadas de las referencias originales). Las instancias de clabjects sobreescriben estos métodos dependiendo de la instanciación de referencia. Por ejemplo, la clase

Component declara que el método *pTypes()* devuelve un conjunto de objetos *Port*, ya que tenía una referencia a *Port*. La clase *Producer* sobreescribe la operación *pTypes* para devolver un conjunto de objetos *Pout* (los de la referencia *outs*). Si *Producer* definiera más referencias del tipo *pTypes*, el retorno del método se sustituiría por la clase más específica común a los destinos de las referencias. Debe tenerse en cuenta que, como *Connector.delay* tiene potencia 1, se convierte en un atributo estático, cuyo valor inicial se establece en *MChannel*.

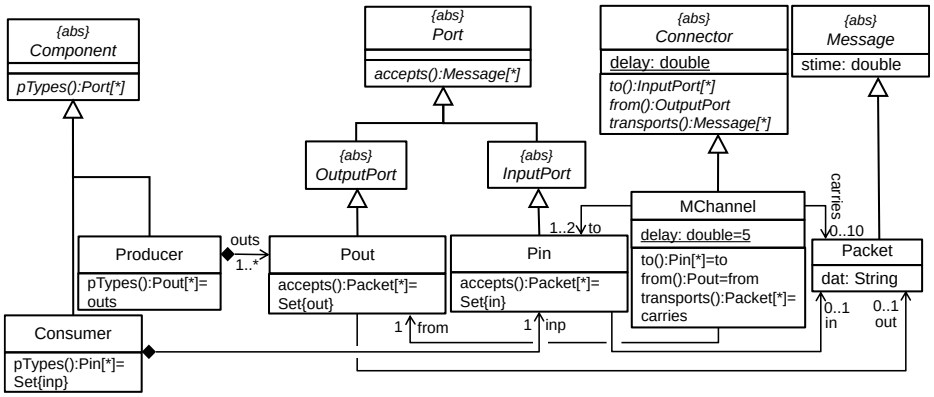


Figura 5.8: Compilación del modelo multi-nivel de la Figura 5.4.

En caso de que la compilación sea para EMF, se agregaría una clase raíz que contenga todas las clases de nivel 2 (*Component, Port, Connector y Message*), las posibles extensiones lingüísticas en el nivel 1 no contenidas en otras clases, e invariantes OCL que se encarguen de las cardinalidades de los clabjects declarados (por ejemplo, cardinalidad [1..*] de *Component* y *Connector*).

5.2.4 Creando el modelo de nivel 0

Como se mencionó, el modelo de nivel 0 se puede crear de dos maneras: (i) mediante la creación de instancias del meta-modelo compilado, y (ii) extendiendo el modelo multi-nivel. Se prefiere la primera opción cuando se desea la compatibilidad con frameworks de dos niveles. Por ejemplo, si el desarrollador necesita definir una transformación para manipular el modelo de nivel 0, dicha transformación se definirá sobre el meta-modelo compilado. Es interesante indicar que, cualquier modelo de nivel 0 construido de esta manera será una instancia válida del modelo multi-nivel original representado usando la extensión del meta-modelo. La segunda forma de crear el modelo emula objetos con clases, y no requiere compilar el modelo multi-nivel. Esta es la opción preferida cuando el desarrollador no quiere usar una herramienta diferente (por ejemplo, un editor de modelo estándar de dos niveles) para crear el modelo de nivel 0.

Como ejemplo, la Figura 5.9(a) muestra un modelo de nivel 0 (crea un objeto *m* de tipo *Packet*), mostrado como una instancia del modelo multi-nivel. En particular, se utiliza "instancia de" para indicar que el modelo ha sido creado utilizando el meta-modelo compilado, y "conforme a" para transmitir que el modelo se ajusta a la representación basada en extensiones de modelos multi-nivel. Debe tenerse en cuenta que lo contrario no es válido: existen modelos de instancia de la representación basada en extensión (en particular, aquellos con un objeto de tipo *Message*) que no

se ajustan a la representación del meta-modelo compilado. Sin embargo, esos modelos no son deseados, ya que en este caso se estarían saltando meta-niveles al instanciar, por ejemplo la clase *Message*. Como puede observarse, la compilación hace explícita la semántica del modelo multi-nivel utilizando facilidades estándar de meta-modelado.

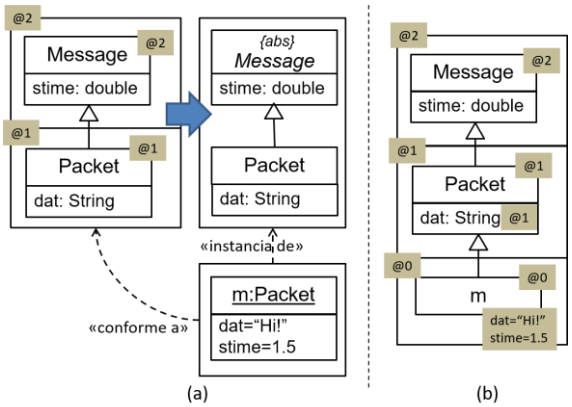


Figura 5.9: Modelo de nivel 0.
(a) Representación basada en objetos de los modelos de nivel 0; (b) Representación basada en clases de los modelos de nivel 0.

La Figura 5.9(b) muestra el mismo modelo expresado usando una representación basada en clases. Por lo tanto, este enfoque representa clabjects en el nivel 0 utilizando clases. Dado que estos clabjects tendrán frecuentemente campos de instancia, se utilizan las anotaciones para almacenar sus valores.

5.3 Arquitectura y herramientas soporte

El enfoque descrito se ha implementado en una herramienta llamada TOTEM. La herramienta es un plug-in de Eclipse, disponible libremente en http://miso.es/tools/totem.html. La página web de la herramienta contiene varios ejemplos y casos de estudio, incluyendo el ejemplo analizado y el "Bicycle challenge", un caso de estudio utilizado en la serie de workshops MULTI (https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/#challenge). La arquitectura de TOTEM se muestra en la Figura 5.10.

TOTEM se compone de dos componentes principales (etiquetas 2 y 3 en la figura). El primer componente es la *herramienta de anotación multi-nivel*. Con esta herramienta, el diseñador del lenguaje anota un meta-modelo EMF existente [156] (etiqueta 1 en la figura) con anotaciones de modelado multi-nivel. Estas anotaciones se crean como un modelo conforme al meta-modelo mostrado en la Figura 5.2. Los elementos de este modelo de anotaciones tienen referencias cruzadas a los elementos del meta-modelo estándar. Para realizar esta tarea, la herramienta de anotación ofrece dos editores al diseñador: un editor tipo árbol y un editor de texto (etiquetas 2 (a) y 2 (b) en la figura). La herramienta de anotación se describe con más detalle en la sección 5.3.1.

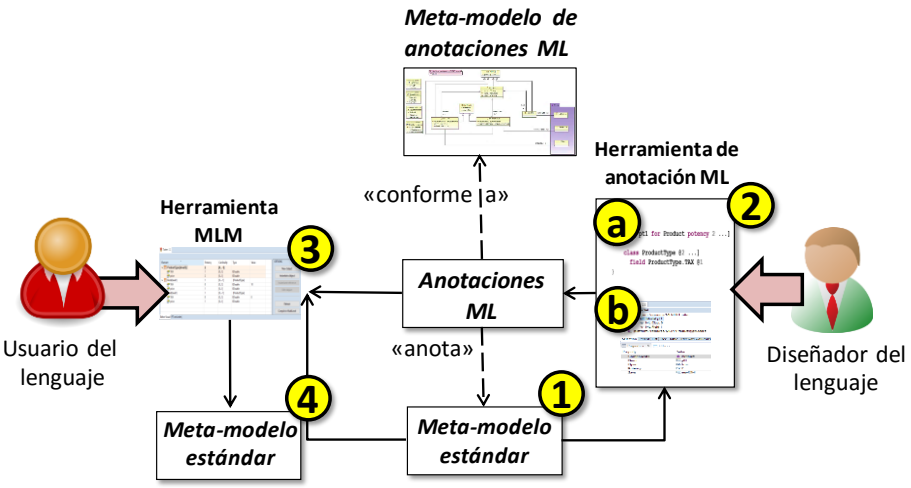


Figura 5.10: Arquitectura de TOTEM.

El segundo componente de TOTEM es la *herramienta de modelado multi-nivel* (etiqueta 3 en la figura). Esta herramienta recibe un meta-modelo estándar con anotaciones multi-nivel y permite su instanciación a cualquier profundidad, incluido el nivel 0. Además, TOTEM puede producir un meta-modelo estándar utilizando el procedimiento de compilación de la sección 5.2.3 (etiqueta 4 en la figura). De esta manera, se pueden utilizar herramientas EMF estándar para instanciar el meta-modelo, o para definir las transformaciones de modelos sobre éste. Este segundo componente se describirá con más detalle en la sección 5.3.2.

TOTEM soporta dos enfoques para crear un modelo multi-nivel completo. En el primero, un meta-modelo EMF estándar existente se anota con anotaciones multi-nivel usando la herramienta de anotación, y luego se instancia usando la herramienta de modelado multi-nivel. En el segundo escenario, el diseñador de lenguaje no parte de un meta-modelo EMF existente, sino que utiliza la herramienta de modelado multi-nivel para crear el modelo de nivel superior del lenguaje. Se discutirá cómo se soportan estos dos escenarios en la sección 5.3.2.

5.3.1 Herramienta de anotación multi-nivel

La herramienta de anotación multi-nivel es útil para el caso en que basemos nuestro modelomulti-nivel en un meta-modelo EMF existente. Las anotaciones pueden ser especificadas usando un editor tipo árbol o usando un DSL textual que se ha creado usando Xtext. Como ejemplo, la Figura 5.11 (a) muestra un meta-modelo estándar simple, mientras que las figuras (b) y (c) muestran las anotaciones usando el editor tipo árbol y el editor de texto. En ambos casos, el meta-modelo en sí mismo se configura con potencia 2, a la clase *ProductType* se le asigna potencia 2, mientras que el atributo *TAX* tiene potencia 1. En el atributo price no se configura una potencia explícita, por lo que obtiene la potencia de su clase contenedora (potencia 2).

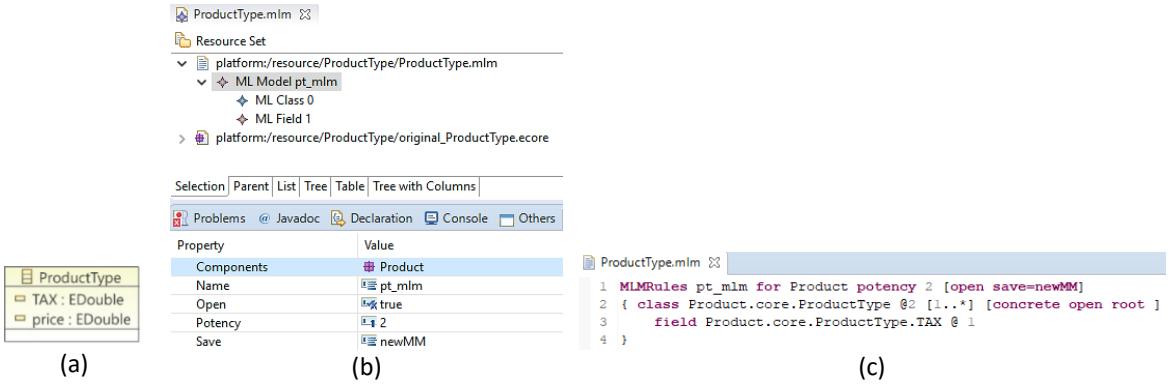


Figura 5.11: Herramienta de anotación multi-nivel.
(a) Ejemplo de meta-modelo estándar; (b) Anotaciones multi-nivel utilizando el editor basado en árboles; (c) DSL textual.

Adicionalmente, la Figura 5.11 (c) muestra algunos detalles que no son visibles en la Figura 5.11 (b). En particular, declara el meta-modelo "open", lo que significa que se pueden incluir clabjects sin tipo en el nivel inferior. También establece un indicador de persistencia (save) que indica que, al instanciar, el modelo resultante debe guardarse en un fichero separado (newMM). La otra alternativa para la persistencia es guardar la extensión en el mismo fichero. A la clase *ProductType* se le asigna un intervalo de instanciabilidad de [1..*] (por lo que su instanciación es obligatoria). También se etiqueta como "concrete" (para que las instancias de esta clase no puedan ser abstractas), "open" (para que las instancias puedan añadir campos y referencias sin tipo), y "root". Este último indicador se utiliza para señalar que ésta será una clase raíz. Tener una clase raíz es común en EMF, por lo que frecuentemente, los modelos contienen sólo un objeto de la clase raíz, que contiene directa o indirectamente todos los demás objetos. Varias clases pueden ser etiquetadas como "root". En la implementación, la compilación añade una clase adicional que actúa como clase raíz, que contiene las instancias de las clases etiquetadas como raíz. Se han omitido los indicadores de persistencia y "root" del meta-modelo de la Figura 5.2, ya que son detalles de implementación.

```
1   MLMRules component_mlm for component potency 2 [save=theSame]
2   {
3      class component.core.Component @2 [1..*] [concrete root]
4      class component.core.Message @2 [0..*] [concrete open root]
5      class component.core.InputPort @2 [0..*] [concrete root]
6      class component.core.OutputPort @2 [0..*] [concrete root]
7      class component.core.Connector @2 [1..*] [concrete root]
8      reference component.core.Component.pTypes @2 [1..*][restrictive container]
9      reference component.core.Port.accepts @2 [0..*][restrictive noContainer]
10     reference component.core.Connector.from @2 [1..1][noContainer]
11     reference component.core.Connector.to @2 [0..*][restrictive noContainer]
12     reference component.core.Connector.transports @2 [0..*][restrictive noContainer]
13     field component.core.Connector.delay @1
14  }
```

Listado 5.1: Anotaciones multi-nivel utilizando el DSL textual.

El Listado 5.1 muestra las anotaciones creadas con el DSL que serán aplicadas a los elementos del meta-modelo *Component* de la Figura 5.3. En total, se consideran 5 anotaciones para que las clases del meta-modelo estándar (clases que posteriormente cumplen la faceta de clabject se incluyen en

TOTEM) puedan ser instanciadas en los siguientes meta-niveles (no se aplica ninguna anotación a la clase *Port* debido a que de ella heredan las clases *InputPort* y *OutputPort*, a las cuales si se les aplican anotaciones), 5 anotaciones para que las referencias puedan también ser instanciadas en los siguientes meta-niveles, y 1 anotación para que al campo *delay* de la clase *Connector* se le puede asignar un valor en el meta-nivel que corresponda. La Tabla 5.1 muestra los valores de las anotaciones del Listado 5.1.

Tabla 5.1: Especificación de los valores de las anotaciones.

Modelo de anotaciones multi-nivel (valores iniciales)	
Propiedad	**Valor**
Components	component
Name	component_mlm
Open	false
Potency	2
Save	theSame

Anotación multi-nivel		Valor	Ejemplo de la aplicación
ML Class: 1 *(Component)*	Potency	2	-Instancias *Producer* y *Consumer* (clabjects de tipo *Component*) creadas en nivel 1.
	Min..Max	1..*	
	ExtensionKind	Concrete	
	Open	False	
	IsRoot	True	
ML Class: 2 *(Message)*	Potency	2	-Instancia *Packet* (clabject de tipo *Message*) creada en nivel 1, campo *dat* es agregado.
	Min..Max	0..*	
	ExtensionKind	concrete	
	Open	true	
	IsRoot	true	
ML Class: 3 *(InputPort)*	Potency	2	-Instancia *Pin* (clabject de tipo *InputPort*) creada en nivel 1.
	Min..Max	0..*	
	ExtensionKind	concrete	
	Open	false	
	IsRoot	true	
ML Class: 4 *(OutputPort)*	Potency	2	-Instancia *Pout* (clabject de tipo *OutputPort*) creada en nivel 1.
	Min..Max	0..*	
	ExtensionKind	concrete	
	Open	false	
	IsRoot	true	
ML Class: 5 *(Connector)*	Potency	2	-Instancia *MChannel* (clabject del tipo *Connector*) creada en nivel 1.
	Min..Max	1..*	
	ExtensionKind	concrete	
	Open	false	
	IsRoot	true	
ML Reference: 1 *(pTypes)*	Composition Kind	isContainment	-Instancia *outs* (referencia de tipo *pTypes*) entre los clabjects *Producer* y *Pout* es creada en el nivel 1. - Instancia *inp* (referencia de tipo *pTypes*) entre los clabjects *Consumer* y *Pin* es creada en el nivel 1.
	Min..Max	1..*	
	Potency	2	
	Redef Kind	Restrictive	
ML Reference: 2 *(accepts)*	Composition Kind	noContainment	- Instancia *out* (referencia de tipo *accepts*) entre los clabjects *Pout* y *Packet* es creada en el nivel 1. - Instancia *in* (referencia de tipo *accepts*) entre los clabjects *Pin* y *Packet* es creada en el nivel 1.
	Min..Max	0..*	
	Potency	2	
	Redef Kind	Restrictive	
ML Reference: 3 *(from)*	Composition Kind	noContainmen	- Instancia *from_redef* (referencia de tipo *from*) entre los clabjects *MChannel* y *Pout* es creada en el nivel 1.
	Min..Max	1..1	
	Potency	2	
	Redef Kind	Default	
ML Reference: 4 *(to)*	Composition Kind	noContainmen	- Instancia *to_redef* (referencia de tipo *to*) entre los clabjects *MChannel* y *Pin* es creada en el nivel 1.
	Min..Max	0..*	
	Potency	2	

	Redef Kind	Restrictive	
ML Reference: 5 *(transports)*	Composition Kind	noContainmen	- Instancia *carries* (referencia de tipo *transports*) entre los clabjects *Pout* y *Pout* es creada en el nivel 1.
	Min..Max	0..*	
	Potency	2	
	Redef Kind	Restrictive	
ML Field: 1 *(delay)*	Potency	1	-Campo *delay* del clabject *Connector* con valor en el nivel 1.

5.3.2 Herramienta de modelado multi-nivel

Una vez que se especifican las anotaciones multi-nivel, el meta-modelo anotado puede ser instanciado con la herramienta de modelado multi-nivel. La Figura 5.12 muestra algunas capturas de pantalla de la herramienta siendo utilizada para instanciar el meta-modelo anotado de la Figura 5.11.

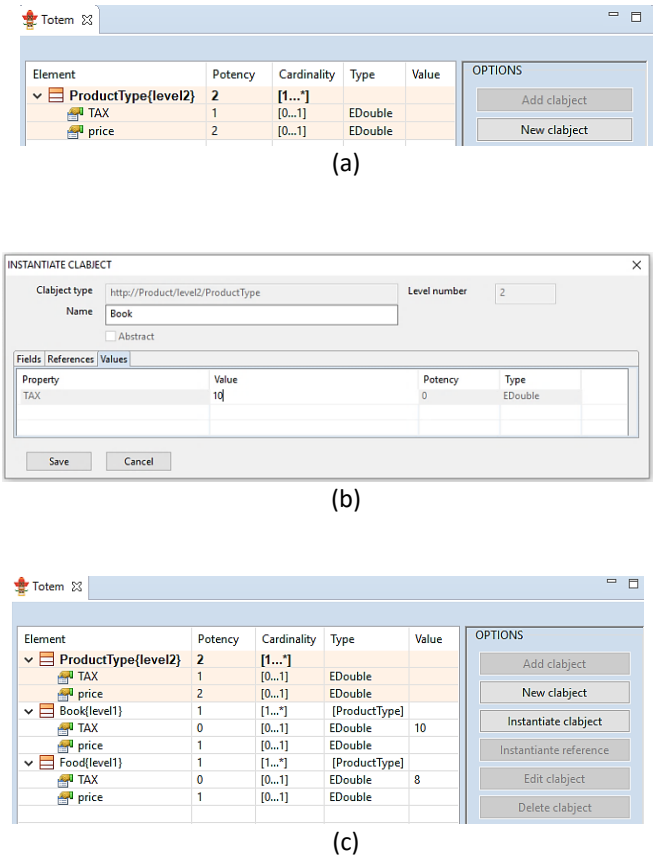


Figura 5.12: Uso de la herramienta de modelado multi-nivel.
(a) GUI principal; (b) Instanciación de un clabject; (c) Instanciación de un modelo.

De manera similar a TACO, su interfaz principal, mostrada en la Figura 5.12(a), está organizada en tres secciones. La sección izquierda muestra todos los elementos del meta-modelo base (clabjects,

campos (fields) y referencias), que pueden ser instanciados de acuerdo a sus anotaciones. En las columnas de la derecha aparecen varias características, como la potencia, la cardinalidad, el tipo (tipo de referencia para asociaciones o tipo de datos para atributos) y el valor (clase de destino para referencia o valor para atributos).

La sección derecha de la interfaz (panel denominado "OPTIONS") contiene botones para instanciar clabjects y referencias, crear clabjects sin tipo o editar clabjects existentes. La Figura 5.12 (b) muestra el cuadro de diálogo usado para instanciar un clabject. Es posible establecer el valor de los atributos, crear nuevos campos si el tipo de clabject es open, e instanciar referencias según corresponda. La Figura 5.12(c) muestra la herramienta después de crear dos instancias de *ProductType*. En ambos casos se puede asignar un valor a *TAX* (ya que tiene potencia 0). Además, el atributo *price* (con potencia 1) también es visible, para permitir asignarle un valor por defecto.

TOTEM soporta un número arbitrario de meta-niveles, y también una potencia ilimitada. Para soportar el escenario donde el diseñador no parte de un meta-modelo EMF existente, se utiliza un meta-modelo genérico anotado sin clases. Este meta-modelo tiene una potencia ilimitada y es abierto (open). Esto significa que podemos instanciarlo para crear un modelo de alguna potencia arbitraria, y clabjects sin tipo (untyped) dentro de él.

5.3.3 Extracción de un meta-modelo estándar

TOTEM permite modelar el nivel 0 dentro de la herramienta, o puede producir un meta-modelo estándar de acuerdo con el procedimiento descrito en la sección 5.2.4.

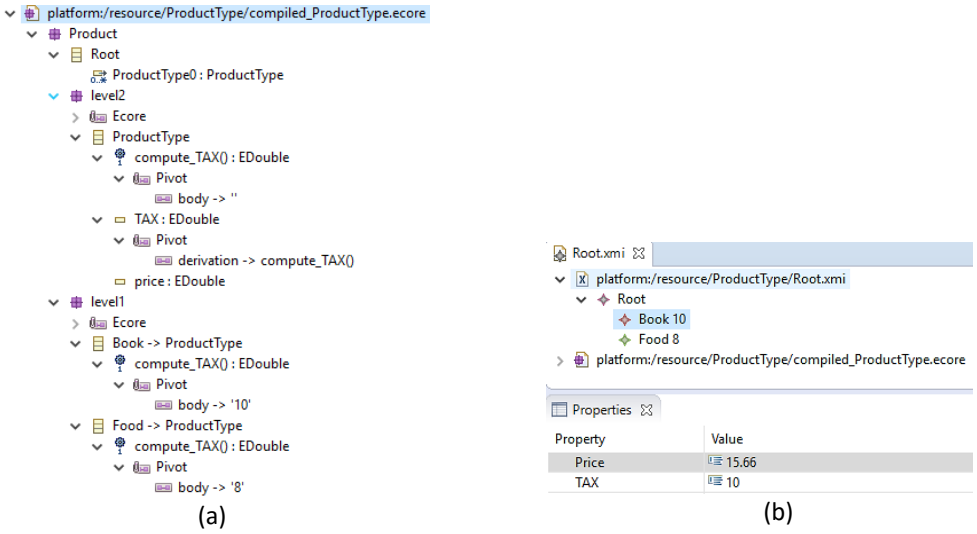


Figura 5.13: Extracción de un meta-modelo estándar.
(a) Meta-modelo estándar compilado; (b) Un modelo en el nivel 0 creado usando el editor tipo árbol de EMF.

La Figura 5.13(a) muestra el meta-modelo generado para el sistema de comercio electrónico analizado en la sección 4.2.3, mientras que la Figura 5.13(b) muestra un modelo (nivel 0) creado con el editor tipo árbol estándar de EMF. Se debe considerar que el atributo *TAX* tiene potencia 1, y por tanto, recibe valores en el nivel 1. Esto se codifica como un *atributo derivado*, que toma el valor utilizando la operación *compute_TAX()*, que se sobreescribe para obtener los valores apropiados para los clabjects *Book* and *Food*. Además, la herramienta genera una clase *raíz* que contiene todas las clases anotadas como *raíz* (*ProductType* en este caso). Esto se usa en el modelo de la Figura 5.13(b) como el contenedor para los objetos del modelo.

5.4 Evaluación

En esta sección, se ofrece tanto una evaluación experimental como una analítica. El objetivo es responder a dos preguntas de investigación (RQ, del inglés research question):

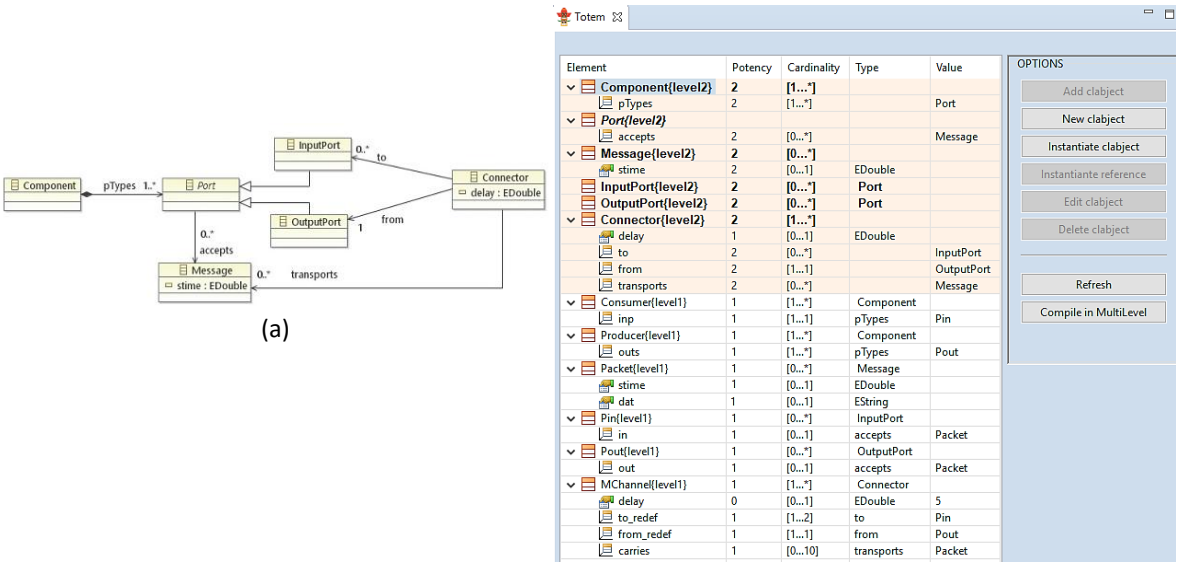
RQ1: ¿es TOTEM capaz de producir soluciones comparables a las herramientas de modelado multi-nivel de última generación?

RQ1.1: ¿cuáles son las ventajas de TOTEM con respecto a las herramientas de modelado multi-nivel de última generación?

Para ello, se ha desarrollado completamente el ejemplo realizado y analizado anteriormente, mostrando su ejecución con TOTEM y luego comparando este mismo modelo con las herramientas de modelado multi-nivel MetaDepth y Melanee (sección 5.4.1). En segundo lugar, se realiza una evaluación analítica del TOTEM en comparación con las herramientas actuales de modelado multi-nivel (sección 5.4.2).

5.4.1 Evaluación empírica

En esta sección, se compara el ejemplo realizado en TOTEM, con su codificación utilizando MetaDepth y posteriormente con Melanee. Estas han sido seleccionadas como representantes de las herramientas de modelado multi-nivel más avanzadas. Nuestro objetivo es medir la complejidad de las tres soluciones, en términos de métricas como el tamaño y en las prestaciones multi-nivel disponibles en cada herramienta. Este experimento tiene como objetivo responder a *RQ1* y *RQ1.1*. La Figura 5.14(a) muestra el meta-modelo EMF con el que se comienza a modelar el nivel 2, mientras que la Figura 5.14(b) muestra el uso de TOTEM para crear el modelo en el nivel 1.



(a) (b)

Figura 5.14: Evaluación empírica.
(a) Modelo de nivel superior para el ejemplo como un meta-modelo Ecore;
(b) Niveles 2 y 1 en TOTEM.

5.4.1.1 Comparación con MetaDepth

El Listado 5.2 muestra el ejemplo usando la sintaxis textual de MetaDepth. El Listado 5.2(a) muestra el nivel 2, que define los clabjects *Component*, *Port*, *InputPort*, *OutputPort, Message* y *Connector*.

```
Model ComponentSystem@2
  Node Component [1..*] {
    pTypes : Port[*];
  }
  abstract Node Port {
    accepts : Message[*];
  }
  Node InputPort : Port{}
  Node OutputPort : Port{}
  Node Message {
    stime: double;
  }
  Node Connector [1..*] {
    delay@1 : double;
    from    : OutputPort;
    to      : InputPort[*];
    transports : Message[*];
  }
}
```

(a)

```
ComponentSystem ProducerConsumer {
   Component Producer{
      outs : Pout[1..*] {pTypes};
   }
   Component Consumer{
      inp : Pin {pTypes};
   }
   OutputPort Pout{
      out : Packet[0..1]{accepts};
   }
   InputPort Pin{
      in : Packet[0..1]{accepts};
   }
   Connector MChannel {
      delay = 5;
      fromp   : Pout {from};
      top     : Pin [1..2] {to};
      carries : Packet[0..10]{transports};
   }
   Message Packet {
      dat : String;
   }
}
```

(b)

```
ProducerConsumer aModel {
   Producer p {
     outs = [po];
   }
   Pout po {
     out = [pk];
   }
   Packet pk {
     dat = "Hi!";
     stime = 1.5;
   }
   MChannel mc {
     fromp = po;
     top = [pi1, pi2];
   }
   Pin pi1;
   Pin pi2;
   Consumer c1 { inp = [pi1]; }
   Consumer c2 { inp = [pi2]; }
}
```

(c)

Listado 5.2: Ejemplo en MetaDepth. (a) Meta-modelo Component (nivel 2); (b) Meta-modelo producer-consumer (nivel 1); (c) Modelo producer-consumer (nivel 0).

Los clabjects sin tipo ontológico se declaran utilizando la palabra clave "Node", y las cardinalidades de clabjects se especifican junto al nombre del clabject (por ejemplo, [1..*] para *Connector*). Si no se especifica ninguna cardinalidad, entonces se asume [0..*]. Por defecto, las instancias de un modelo pueden incluir clabjects sin tipo (es decir, extensiones lingüísticas). De manera similar, a las instancias de un clabject se les pueden agregar atributos sin tipo. Sin embargo, esto puede ser controlado prefijando modelos y clabjects con la palabra clave *strict*. La potencia se indica después del símbolo "@". Si un clabject o campo no declara explícitamente potencia, toma la de su contenedor.

El Listado 5.2(b) muestra una instancia del modelo *ComponentSystem*, con el sistema producer-consumer del ejemplo. La instanciación de un clabject requiere prefijar el nombre del clabject con su tipo. Por ejemplo, "*Component Producer*" crea un clabject con el identificador *Producer*, de tipo *Component*. En este nivel, algunos clabjects asignan valores a los campos con potencia 0 (por ejemplo, *delay = 5*), y como todos los clabjects son extensibles, se pueden declarar nuevos atributos (por ejemplo, *dat: String*). La instanciación de una referencia requiere declarar el tipo del clabject destino (por ejemplo, *out: Packet*), la cardinalidad (por ejemplo, *[0..1]*) y el tipo de la referencia (por ejemplo, *{accepts}*). Finalmente, el Listado 5.2(c) muestra un modelo de nivel 0.

En general, las soluciones de TOTEM y MetaDepth son similares, lo que conduce a métricas de tamaño muy similares. En ambos casos, el nivel 2 tiene 6 clabjects, 5 referencias, 2 campos y 2 relaciones de herencia. El nivel 1 se tiene 6 clabjects en ambos casos, 7 referencias y 2 campos en el caso de MetaDepth, y 7 referencias y 2 campos en el caso de TOTEM. La diferencia se refiere al atributo *stime*, que se muestra en *Packet* en el nivel 1 de TOTEM, para que se le pueda asignar un valor por defecto, si se desea. En MetaDepth, el atributo sólo tendría que declararse explícitamente dentro de *MChannel* si se va a dar un valor inicial. Por lo tanto, el tamaño de ambas especificaciones es similar. Una diferencia crucial es que, cuando los elementos se instancian en TOTEM, los campos también se instancian automáticamente, y se muestran en la GUI. En MetaDepth, y debido a que tiene una sintaxis textual, el desarrollador necesita escribir explícitamente tales instancias de campo (field), lo que requiere más esfuerzo.

Ambos enfoques son similares en el sentido de que ambos permiten expresar las cardinalidades de los clabjects y controlar si se permiten las extensiones lingüísticas. En una comparación más detallada, se observa que MetaDepth carece de relaciones de composición, mientras que las composiciones son nativas en EMF, y por lo tanto TOTEM las soporta. Además, TOTEM soporta la especificación de si las instancias de una referencia deben ser también composiciones. Un modelado fiel del ejemplo en MetaDepth requeriría restricciones OCL que emulen la semántica de la composición. MetaDepth no requiere clases raíz, ya que los modelos son entidades de primera clase en MetaDepth. Por el contrario, al estar basado en EMF, TOTEM requiere etiquetar las clases que deben tomarse como raíz (lo que produce una clase adicional cuando se compila el meta-modelo). Por último, TOTEM tiene flags para controlar los tipos de cardinalidad permitidas en las instancias de una relación dada (como default, restrictive o anything), mientras que esto requeriría restricciones OCL en MetaDepth.

5.4.1.2 Comparación con Melanee

La Figura 5.15 muestra el ejemplo, usando la sintaxis gráfica de Melanee. La figura muestra los dos niveles del meta-modelo. En el nivel 2 se definen los mismos clabjects creados en las soluciones usando TOTEM y MetaDepth (*Component*, *Port*, *InputPort*, *OutputPort, Message* y *Connector).*

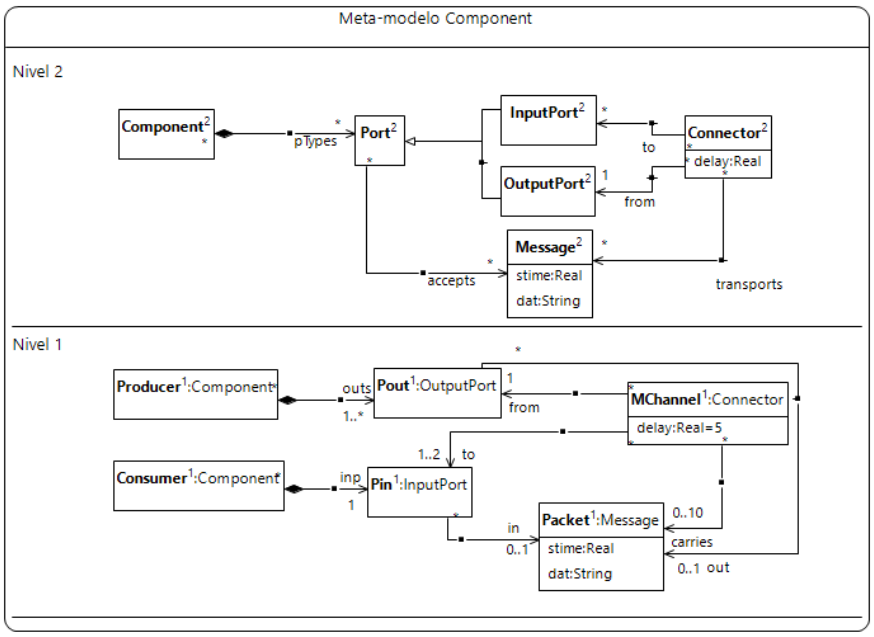


Figura 5.15: Modelo ejemplo usando Melanee.

Melanee proporciona una paleta para la construcción gráfica del modelo multi-nivel. La Figura 5.16 muestra cuadros de diálogo utilizados por Melanee para especificar valores de las propiedades de algunos elementos del modelo multi-nivel.

La Figura 5.16(a) muestra la especificación de las propiedades del clabject *Component* con potencia 2 del nivel 2 (modelo de ejemplo usando Melanee en la Figura 5.15). La propiedad *Level Index=0,* especifica el nivel del modelo en el que se encuentra la entidad (clabject). Cuando se crea una entidad en el primer nivel del modelo, la propiedad tendrá valor 0, el cual es asignado de forma automática. A medida que el clabject es instanciado, el valor de la propiedad se incrementa en 1. *Name=Component*, señala el nombre de la entidad seleccionada en ese momento. *Potency=2, especifica la* potencia de la entidad.

La Figura 5.16(b) muestra la especificación de las propiedades del atributo *stime* del clabject *Message* con potencia 2 del nivel 2 (Figura 5.15). La propiedad *Datatype=Real*, especifica el tipo de dato del atributo. *Durability=2*, define en cuántos niveles posteriores puede existir el atributo. *Mutability=2*, define en cuántos niveles puede cambiar el valor de un atributo. *Name=stime*, señala el nombre del atributo seleccionado en ese momento. *Value* sin valor en la figura, especifica el valor del atributo si lo tuviera.

La Figura 5.16(c) muestra la especificación de las propiedades de la referencia (conexión en Melanee) *pTypes* del clabject *Component* con potencia 2 del nivel 2 (Figura 5.15). La propiedad *Destination=Entity InputPort, señala el* destino (target) de la conexión. *Kind=Basic,* especifica el tipo de la conexión, pudiendo ser básico, composición o agregación. *Lower – Upper= 0..-1,* establece el intervalo de la cardinalidad de la conexión. *Moniker* sin valor en la figura, sirve para poner el rol de la

conexión. *Navigable=true,* especifica la dirección de la conexión. *Type* sin valor, especifica el extremo de la conexión. *Level Index=0,* especifica el nivel del modelo en el que se encuentra la conexión. *Name=pTypes*, es el nombre de la entidad. *Potency=2* señala la potencia de la conexión.

Property	Value
Level Index	0
Name	Component
Potency	2

(a)

Property	Value
Datatype	Real
Durability	2
Mutability	2
Name	stime
Value	

(b)

Property	Value
Destination	Entity InputPort
Kind	Basic
Lower	0
Moniker	
Navigable	true
Type	
Upper	-1
Property	**Value**
Level Index	0
Name	pTypes
Potency	2

(c)

Figura 5.16: Propiedades de los elementos del meta-modelo en Melanee. (a) clabject; (b) atributo; (c) referencia.

Una vez incluidos y configurados todos los elementos del nivel 2, todas sus instancias podrán ser incluidas de forma manual en el nivel 1, a través de la paleta de herramientas. El valor de las potencias de los elementos (clabject, atributos y referencias) es controlada automáticamente por la herramienta. La asignación de valores a los campos con potencia 0 (por ejemplo, *MChannel.delay = 5*) de las clabjects se realiza a través de la ventana de propiedades (Figura 5.16(b)). De igual manera como todos los clabjects son extensibles, se pueden declarar nuevos atributos (por ejemplo, *dat: String*) a través de la paleta de herramientas. La instanciación de una referencia requiere que el clabject origen y destino se hayan instanciado previamente en el nivel inferior, en este caso, es posible redefinir la cardinalidad de las referencias instanciadas, así como asignarles un nuevo nombre.

En general, ambas soluciones son similares, lo que conduce a métricas de tamaño muy parecidas. En ambos casos, el nivel 2 tiene 6 clabjects, 5 referencias, 3 campos (porque el campo *dat* que se agrega en el nivel 1, también es añadido de forma automática en el nivel 2) y 2 relaciones de herencia. En el nivel 1 se tiene el mismo número de elementos en los dos casos: 6 clabjects en ambos casos, 7 referencias y 3 campos.

Otra de las diferencias importantes, es que mientras que en TOTEM se debe especificar explícitamente la cardinalidad de los clabjects, es decir el rango de valores que un clabject puede instanciarse en el nivel inferior, en Melanee los clabjects se pueden instanciar en el siguiente nivel un número indefinido de veces (cardinalidad "*" implícita). Adicionalmente, Melanee por defecto permite crear clabjects sin tipo (extensiones lingüísticas) en cualquier nivel del modelo, mientras que en TOTEM se debe especificar explícitamente el modelo como *open*. Las dos herramientas permiten manejar de forma nativa las relaciones de composición, sin embargo, TOTEM permite especificar si las instancias de una referencia pueden o no ser también composiciones. Melanee no requiere clases raíz, ya que los modelos son entidades de primera clase.

En la Tabla 5.2 se muestra el resumen de los aspectos de comparación de las herramientas utilizadas para resolver el ejemplo (*Component*).

Tabla 5.2: Comparación de las tres herramientas, en base al ejemplo.

Herramienta / Aspecto	MetaDepth	Melanee	TOTEM
Tamaño de elementos de modelado	Similar *Nivel 2:* 6 clabjects, 5 referencias, 2 campos y 2 relaciones de herencia. *Nivel 1:* 6 clabjects, 7 referencias y 2 campos.	Similar *Nivel 2:* 6 clabjects, 5 referencias, 3 campos y 2 relaciones de herencia. *Nivel 1:* 6 clabjects, 7 referencias y 3 campos.	Similar *Nivel 2:* 6 clabjects, 5 referencias, 2 campos y 2 relaciones de herencia. *Nivel 1:* 6 clabjects, 7 referencias y 3 campos.
Entorno de especificación de MLM	Textual	Múltiples formatos y múltiples notaciones	Tabular
Expresión de composición	No	Si	Si
Expresión de cardinalidad	Si	Si	Si, maneja dos tipos de cardinalidad para referencias: 1) la definida en la referencia misma, 2) para especificar cuántas instancias de la referencia se pueden crear en el próximo meta-nivel (intervalo [min..max]).
Admite extensiones lingüísticas	Si	Si	Si
Mecanismo de instanciación profunda	Potencia	Potencia	Potencia
Potencia ilimitada (*)	Si	Si	Si
Requiere definir clase raíz	No	No	Si
Compatibilidad con entornos de dos niveles	No	No	Si

En general, con respecto a la pregunta de investigación *RQ1*, la solución que utiliza TOTEM es igual de compleja que la solución que utiliza una herramienta de modelado multi-nivel de última generación, como MetaDepth y Melanee. En cuanto a *RQ1.1*, TOTEM incluye construcciones avanzadas para controlar composiciones y cardinalidades, que no están presentes en MetaDepth y Melanee. Sin embargo, la principal ventaja de TOTEM es la posibilidad de exportar el modelo multi-nivel a un meta-modelo estándar, y luego usar herramientas EMF estándar para editar el modelo de nivel 0, o para definir manipulaciones de modelos usando lenguajes de transformación estándar, como ATL o Acceleo.

Amenazas a la validez:

En cuanto a la validez interna (factores de confusión que pueden influir en los resultados), sólo se ha utilizado un ejemplo para la comparación. Sin embargo, el ejemplo es sustancial, ejerciendo muchas de las características de las tres herramientas. Para la validez externa (generalización de los resultados), sólo se han comparado con MetaDepth y Melanee. Si bien éstas son herramientas de modelado multi-nivel más consolidadas disponibles en la actualidad, la generalización de las respuestas a las preguntas de la investigación requeriría una comparación con otras herramientas. Una comparación basada en ejemplos con las otras herramientas correría el riesgo de proponer soluciones no óptimas (ya que no se es el autor de esas herramientas). Para ello, se ha realizado una

evaluación analítica en la sección 5.4.2, comparando con las herramientas analizadas en la sección 3.2.2.

5.4.2 Evaluación analítica

En esta sección se presenta una evaluación analítica, comparando las características de TOTEM con las de otras herramientas de modelado multi-nivel. Para ello, el análisis se basa en las características extraídas de la Tabla 3.2, que se resumen en la Tabla 5.3.

Tabla 5.3: Características de TOTEM.

Característica	TOTEM
Mecanismo	Extensión de meta-modelo
Caracterización de instanciación	Profundidad (potencia, potencia ilimitada)
Extensión	Si (objects, fields)
Tipado	No estricto (extensión, sin mezcla) Lingüística, ontológica
Meta-niveles	Extensible
Restricciones	OCL estándar, pero solo afecta al nivel-0 del modelo
Propósito	MDE
Soporte para administración de lenguajes	Si, cualquiera disponible para el estándar EMF
Compatibilidad con modelado de dos niveles y con sus herramientas	Si, ambos como un meta-modelo inicial de primer nivel y para editar modelos de nivel 0

Como muestra la tabla, el enfoque propuesto se basa en la extensión del meta-modelo, un enfoque único entre los enfoques analizados en la Tabla 3.2. Este enfoque incorpora el modelo multi-nivel dentro de un meta-modelo Ecore, presentando ventajas con respecto a la compatibilidad con el estándar EMF, ya que es posible reutilizar directamente los meta-modelos Ecore existentes.

TOTEM soporta caracterización profunda a través de potencia (para clabjects y atributos). Además, también soporta potencia ilimitada, lo que permite una instanciación de profundidad arbitraria y un número de meta-niveles que no necesitan ser fijados a priori. El enfoque admite extensiones lingüísticas, ya que tanto los meta-modelos como los clabjects pueden etiquetarse como abiertos. Como el modelo multi-nivel está incrustado dentro de un meta-modelo Ecore, las restricciones pueden ser añadidas usando OCL estándar. Sin embargo, estas restricciones sólo pueden aplicarse al nivel 0. TOTEM tiene un doble tipado ontológico/lingüístico. Por un lado, se utiliza Ecore (enriquecido con anotaciones multi-nivel) como meta-modelo lingüístico. Por otro lado, el tipado ontológico se codifica dentro del meta-modelo, utilizando relaciones de herencia.

El propósito de TOTEM es su uso dentro de las soluciones MDE, y para ello ha sido construido sobre EMF, como un plug-in de Eclipse. Debido a esta decisión, es compatible con todos los lenguajes de gestión de modelos del ecosistema EMF, incluyendo ATL, Epsilon o Acceleo. Típicamente, las manipulaciones del modelo afectarán a los modelos de nivel 0, pero es posible usar estos lenguajes

estándar para expresar transformaciones también a nivel del meta-modelo. La compatibilidad con el modelado y las herramientas estándar de dos niveles es alta, porque el proceso de modelado puede comenzar con la anotación de un meta-modelo estándar existente, y la herramienta puede producir un meta-modelo estándar al final, que puede ser instanciado usando herramientas EMF estándar.

En general, se puede concluir que el enfoque de TOTEM es único en las herramientas actuales. Soporta las construcciones más avanzadas y los requisitos de herramientas de modelado multi-nivel útiles (mecanismos de caracterización profunda, extensibilidad, meta-niveles extensibles y restricciones). Respondiendo a *RQ1.1*, y en comparación con otros enfoques existentes, aporta una mayor compatibilidad con el modelado y las herramientas estándar de dos niveles, tanto para iniciar un modelo multi-nivel utilizando un meta-modelo Ecore existente, como para instanciar y manipular el nivel 0 utilizando herramientas estándar.

5.5 Resumen y conclusiones

En base a los resultados del análisis de los diferentes enfoques y herramientas de modelado multi-nivel existentes actualmente (revisados en las secciones 3.2.1 y 3.2.2), se ha podido detectar que existe una falta de interoperabilidad entre las herramientas de modelado multi-nivel y el modelado estándar de dos niveles. Por este motivo, se ha propuesto un nuevo enfoque para modelado multi-nivel basado en extensión, que incorpora múltiples meta-niveles dentro de un meta-modelo, y una herramienta llamada TOTEM que implementa estos conceptos.

A través de varios ejemplos, se han puesto a prueba las capacidades del enfoque propuesto y de la herramienta que las implementa, identificándose los beneficios en términos de interoperabilidad con los frameworks de modelado estándar. Finalmente, se presenta una evaluación empírica (experimental) y analítica del mecanismo y de la herramienta. Para la evaluación empírica se ha desarrollado un ejemplo de modelado multi-nivel en TOTEM, y en dos herramientas de modelado multi-nivel relevantes, como son MetaDepth y Melanee, lo que ha permitido identificar las semejanzas, diferencias y beneficios del enfoque propuesto. La evaluación analítica se basa en el modelo de características de la Figura 3.3, y sirve de marco comparativo con el resto de herramienta multi-nivel analizadas en la sección 3.2.2.

6 Discusión y comparativa de enfoques

Considerando que el meta-modelo es el elemento fundamental en el desarrollo de software en MDE y que el software permanentemente se encuentra dentro de un proceso de evolución debido a las nuevas exigencias de su entorno, resulta de gran ayuda contar con frameworks que proporcionen soporte metodológico a la evolución de los meta-modelos. Esta evolución permite desarrollar software que se puede adaptar a nuevos o cambiantes dominios de aplicación, reutilizando los meta-modelos existentes. Se debe señalar que actualmente existen varios esfuerzos independientes que abordan la problemática de la evolución del meta-modelo. Sin embargo, este trabajo se ha centrado en la propuesta de técnicas para la adaptación controlada del meta-modelo para que se ajuste a un dominio más especializado. Por este motivo, se han propuesto dos técnicas para especificar la personalización de meta-modelos, así como el diseño e implementación de los mecanismos para guiar al desarrollador a realizar dicha personalización.

En este capítulo se discuten los resultados de la investigación, analizando cuándo usar uno u otro enfoque de personalización de meta-modelos (basado en reglas de extensión, o basado en modelado multi-nivel), se comparan, y se discuten las ventajas y desventajas de ambos enfoques. Para lo cual, en primer lugar se describen las propiedades del mecanismo de extensibilidad, para posteriormente plantear sus escenarios de uso. La misma descripción se realiza para la personalización de meta-modelos en base al modelado multi-nivel. Adicionalmente, se describe el proceso de personalización de dos meta-modelos estándar, utilizando en cada caso los dos mecanismos de personalización planteados en esta tesis.

6.1 Control de la extensibilidad de meta-modelos

Para cubrir el control de extensibilidad de meta-modelos se ha propuesto un mecanismo, arquitectura y herramientas, que permiten refinar el meta-modelo existente, para que pueda ajuste a un dominio derivado del original. En esta subsección se describen las propiedades del mecanismo, así como sus escenarios de uso.

6.1.1 Propiedades del mecanismo de extensibilidad

A continuación, se señalan las propiedades del mecanismo de extensibilidad propuesto:

- ***Extensibilidad pesada (heavyweight):*** no solo se permite extender los elementos del meta-modelo, sino que también es posible agregar nuevos elementos, y también eliminarlos.
- ***No intrusivo:*** las reglas de extensión son expresadas en un modelo independiente al meta-modelo a modificar. El modelo de reglas se enlaza a los diferentes elementos del meta-modelo a personalizar. Adicionalmente el mecanismo permite modificar el meta-modelo original, o crear una copia con la modificación.
- ***Genérico:*** el enfoque de extensión puede aplicarse a cualquier meta-modelo, a través de los entornos de modelado estándar como EMF.
- ***Persistente:*** las reglas de extensibilidad al estar expresadas en un modelo independiente al meta-modelo a extender, pueden ser especificadas, almacenadas y compartidas si así se requiere.
- ***Incremental:*** al seguir un enfoque basado en modelos, el mecanismo de extensibilidad permite ser ampliado. Por ejemplo, se pueden añadir nuevas operaciones de personalización.
- ***Transparente:*** desde el punto de vista del usuario y de las herramientas, un meta-modelo que ha sido extendido puede ser presentado y manipulado como cualquier meta-modelo estándar.
- ***Dinámico/síncrono:*** las reglas de extensibilidad son aplicables de forma dinámica a los elementos del meta-modelo, y estos cambios son reflejados de forma directa en el meta-modelo.
- ***Preservación de la compatibilidad:*** dependiendo de las reglas de extensibilidad que se apliquen al meta-modelo, los modelos pueden seguir siendo compatibles con el meta-modelo anterior (vease la Tabla 4.1).

6.1.2 Escenarios de uso

El enfoque de extensibilidad de meta-modelos propuesto resulta útil para meta-modelos que han sido diseñados con la intención de ser extendidos a futuro, para ser adaptados a un dominio más específico. Por lo general, las adaptaciones permitidas al meta-modelo base no pueden ser arbitrarias, ya que existiría el riesgo de que los servicios asociados (por ejemplo transformaciones) no puedan funcionar con el meta-modelo extendido. Por lo general, los procesos de extensión de estos meta-modelos se expresan mediante lenguaje natural, lo que puede acarrear errores accidentales. Actualmente no existen mecanismos estándar de extensión de meta-modelos que sean comúnmente aceptados por la comunidad de modelado. No existen mecanismos automatizados que permitan verificar las extensiones definidas por los desarrolladores respecto a la especificación de la extensión, o los medios para guiar al desarrollador en la extensión. Por lo que, el enfoque de extensibilidad propuesto puede ser utilizado cuando las extensiones del meta-modelo requieran ser realizadas de forma automatizada y controlada (por ejemplo, debido a que se ha de preservar la validez de transformaciones asociadas al meta-modelo base).

6.2 Modelado multi-nivel basado en extensión

Tomando como base el mecanismo de extensión, se ha propuesto un enfoque para el modelado multi-nivel, donde se utiliza subtipado en vez de instanciación. A continuación se resumen sus propiedades y se detallan sus escenarios de uso.

6.2.1 Propiedades del mecanismo de modelado multi-nivel

El mecamismo de modelado multi-nivel propuesto ha sido diseñado para permitir emular instanciación mediante los mecanismos de extensión propuestos. Por este motivo, ambos comparten las mismas propiedades, salvo con las diferencias que se detallan a continuación. (véase también la Tabla 5.3).

- ***Extensibilidad pesada:*** la implementación del mecanismo de modelado multi-nivel no admite realizar operaciones de eliminación de elementos del meta-modelo.
- ***No intrusivo:*** en lugar de utilizar reglas de extensibilidad, el mecanismo de modelado multi-nivel emplea anotaciones de modelado multi-nivel, que se enlazan a los elementos del meta-modelo a extender de manera similar a los mecanismos de extensibilidad señalados anteriormente. Esto permite la reutilización de meta-modelos estándar para el modelado multi-nivel.
- ***Compatibilidad:*** el mecanismo se ha diseñado para ser altamente compatible con entornos de modelado a dos niveles.

- ***Caracterización profunda:*** la potencia permite describir propiedades (atributos, referencias) que deben tener las instancias de una clase varios meta-niveles más abajo. El mecanismo basado en extensión no puede controlar esto. La extensión proporciona atributos y referencias, pero no se dice en qué nivel se han de utilizar.
- ***Número arbitrario de meta-niveles:*** la potencia ilimitada permite la descripción de sistemas con un número arbitrario de meta-niveles.
- ***Extensiones lingüísticas:*** se permite la emulación de clabjects, atributos y referencias sin tipo.

6.2.2 Escenarios de uso

El mecanismo de extensibilidad se utiliza cuando se diseña un meta-modelo con necesidad de ampliación. De manera similar, el mecanismo multi-nivel se utiliza cuando un meta-modelo (por ejemplo, de un lenguaje de modelado de procesos) debe poder personalizarse para un dominio específico (por ejemplo, modelado de procesos software). Esto es, con el modelado multi-nivel, se puede definir familias de lenguajes de meta-modelado de dominio específico (por ejemplo, para el modelado de procesos de dominio específico, o el modelado basado en componentes). En este escenario, el mecanismo de extensión no es suficiente, ya que se deben controlar las características de los diferentes elementos (clases, referencias y atributos) del meta-modelo en los distintos meta-niveles. Para ello, el mecanismo multi-nivel ofrece la potencia, pero la asociación de características a meta-niveles no puede expresarse con el mecanismo de extensión. El mecanismo de extensión es "acumulativo", lo que se define en las clases base estarán obligatoriamente en las subclases. Sin embargo, con la instanciación soportada por el modelado multi-nivel se puede decidir si instanciar o no un elemento.

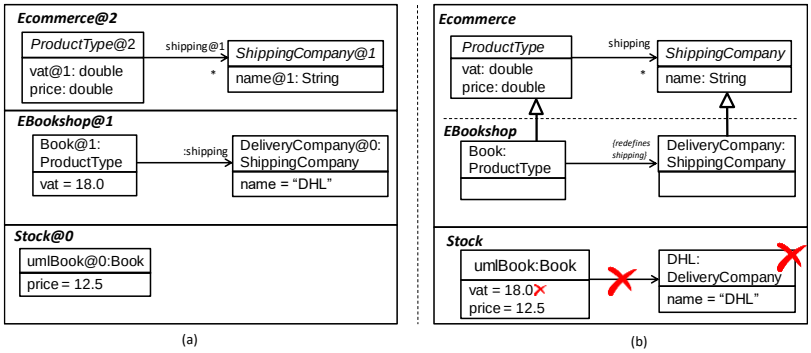


Figura 6.1: Meta-modelo ProductType.
(a) Modelado multi-nivel; (b) Modelado por extensión.

Por ejemplo, la Figura 6.1(a) muestra el meta-modelo *ProductType* expresado con el enfoque multi-nivel. A partir del mecanismo de caracterización profunda a través de la potencia se puede decidir qué y cuándo instanciar un elemento. En la figura se observa que la solución tiene tres meta-niveles. En el primer meta-nivel (*Ecommerce@2*), la clase *ProductType* tiene potencia 2, por lo que se puede crear una instancia en los siguientes dos meta-niveles. Se considera que el atributo *price* también tiene potencia 2, debido a la consideración de que si un atributo no declara explícitamente

una potencia, toma la potencia de su clase contenedora, por lo se aplica a instancias dos niveles por debajo, mientras que *vat* tiene potencia 1, por lo que se convierte en instancia en el siguiente nivel. La clase *ProductType* se encuentra asociada a través de la referencia *shipping* de potencia 1, a 0 o más *ShippingCompany* que tiene potencia 1 y un atributo *name*, también de potencia 1. En el segundo meta-nivel (*EBookshop@1*) de la figura, se muestran que la clase Book (instancia de *ProductType)* está asociada a través de una instancia de la referencia *shipping* (*:shipping*) a *DeliveryCompany* (instancia de *ShippingCompany*) que tiene como *name* "DHL", por lo que en el tercer meta-nivel (*Stock@0*, que se encuentra a nivel de modelo) ya no es posible instanciar DHL porque ya tiene potencia 0.

Con el mecanismo de extensión, señalado en la Figura 6.1(b), la solución tiene dos meta-niveles. En el primer meta-nivel (*Ecommerce*), se encuentra la clase *ProductType* con los atributos *vat* y *price,* asociada a través de la referencia *shipping*, a 0 o más *ShippingCompany* que tiene el atributo *name*. En el mismo meta-nivel, las clases *ProductType* y *DeliveryCompany* son extendidas (a través de una relación de herencia) a las clases *Book* y *DeliveryCompany* respectivamente, las cuales se encuentran relacionadas a través de la redefinición de la referencia *shipping*. En este caso nos podemos dar cuenta que con el mecanismo de extensión no es posible instanciar DHL en este meta-nivel, sino únicamente en el siguiente meta-nivel (*nivel 0*, a nivel de modelo), siempre y cuando *DeliveryCompany* no sea abstracta, pero esto es algo que no se puede controlar desde el nivel más alto de manera fácil. En resumen, el mecanismo de extensión no permite un control fino de las características de los diferentes elementos en meta-niveles inferiores, específicamente cuándo asignar un valor al atributo *vat*, cuándo se debe redefinir la referencia *shipping*, y hasta qué nivel está permitido extender la clase *ShippingCompany*. Con el mecanismo de extensión sólo podemos controlar si la extensión directa de una clase debe ser abstracta (similar a asignarle una potencia 1), pero no hay un control más fino que permita emular la potencia.

El enfoque de modelado multi-nivel es útil cuando existen varios niveles de clasificación, especialmente cuando surge el patrón tipo-objeto. Cuando surge este patrón (o alguna de las variantes identificadas en [46]), un enfoque multi-nivel resulta en modelos más simples. El enfoque propuesto en esta tesis es especialmente útil cuando se requiere mantener la compatibilidad con herramientas de modelado de dos niveles.

El enfoque permite emular modelado multi-nivel dentro de un meta-modelo estándar de dos niveles utilizando el framework de modelado de Eclipse (EMF). De esta manera es posible aprovechar el conjunto de herramientas de MDE que permite crear editores, realizar transformaciones de modelos, y construir generadores de código. El mecanismo propuesto facilita la migración de las soluciones de dos meta-niveles a modelos multi-nivel y viceversa. Resulta viable la reutilización de meta-modelos existentes como base para crear modelos multi-nivel y permitir el uso de herramientas de dos niveles para modelado multi-nivel.

6.3 Comparando la extensión con el modelado multi-nivel

Con la finalidad de analizar más en detalle, y conocer las ventajas y desventajas de los enfoques de personalización de meta-modelos propuestos, se desarrollan dos casos de estudio. En el primer

caso, se utiliza el meta-modelo *Diagram Interchange* (DI) analizado en la sección 4.6.2, que fue desarrollado con la perspectiva de extensibilidad de meta-modelos utilizando TACO. De esta manera, ahora se diseñará una solución utilizando modelado multi-nivel usando TOTEM. En segundo lugar, se utilizará el enfoque basado en extensión para representar el meta-modelo multi-nivel *Component* (componentes) analizado en la sección 5.3.1.

6.3.1 Un enfoque multi-nivel para DI

Como se señaló en la sección 4.6.2, el meta-modelo Diagram Interchage (DI) que se muestra en la Figura 6.2, es parte de la arquitectura del estándar Diagram Definition (DD) de la OMG. Este enfoque soporta la especificación formal de la sintaxis concreta de un lenguaje de modelado para que las herramientas puedan intercambiar diagramas [111].

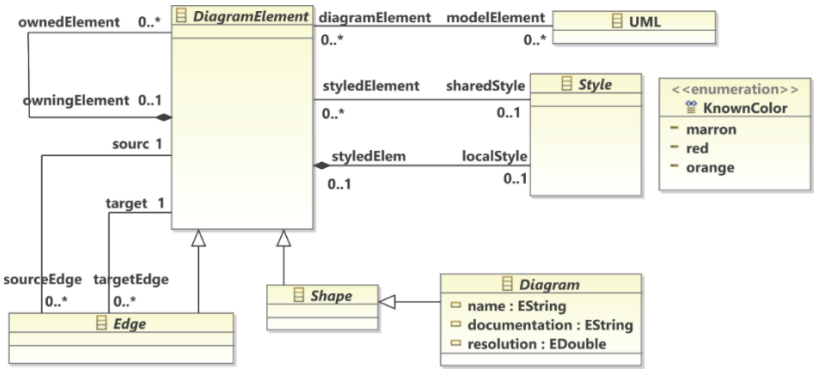


Figura 6.2: Meta-modelo Diagram Interchange (DI).

Para realizar una prueba de extensibilidad al meta-modelo DI con TACO, se utilizarion 11 reglas de extensibilidad, como se muestra en la Figura 6.3 (la primera línea hace referencia a los valores iniciales del modelo de extensibilidad). Se emplearon 5 reglas para controlar la extensibilidad de clases (líneas 2 a 6), 5 reglas para controlar la redefinición de referencias (líneas 7 a 11), y una regla para controlar la actualización de una enumeración (línea 12).

```
DI.custom
1  ExtensibilityRules named DI_extend for DI [expandable save=newPackage]
2  ExtClass { DI.core.Diagram [1..1] concrete extensible}
3  ExtClass { DI.core.DiagramElement [0..*] abstract extensible}
4  ExtClass { DI.core.Edge [1..*] anything }
5  ExtClass { DI.core.Shape [1..*] anything }
6  ExtClass { DI.core.Style [1..1] concrete extensible }
7  RedReference { DI.core.DiagramElement.localStyle [1..1] isContain }
8  RedReference { DI.core.DiagramElement.sharedStyle [1..2] noContain }
9  RedReference {DI.core.DiagramElement.ownedElement [1..*] isContain owner DI.core.Diagram}
10 RedReference { DI.core.Edge.sourc [1..1] noContain targets (DI.core.Shape)}
11 RedReference {DI.core.Edge.target [1..1] noContain targets ( DI.core.Shape )}
12 UpdEnum { DI.core.KnownColor [0..10] }
```

Figura 6.3: Reglas de extensibilidad de TACO aplicadas al meta-modelo DI.

TACO permite a los diseñadores extender meta-modelos conforme a las reglas definidas. En la Figura 6.4 se puede observar que la clase *UMLDiagram* (en las filas más obscuras) del paquete *UML_DI* (creado con la herramienta) es una extensión de la clase *Diagram* del paquete *core* del meta-modelo estándar base. De igual manera se puede observar que la referencia *umlOwnedElement* de la clase *UMLDiagram* del paquete *UML_DI* es una instancia de la referencia *ownedElement* de la clase

DiagramElement del paquete *core*. Adicionalmente, en la figura se observa que la enumeración *KnowColor* tiene el color *yellow* como nuevo literal añadido.

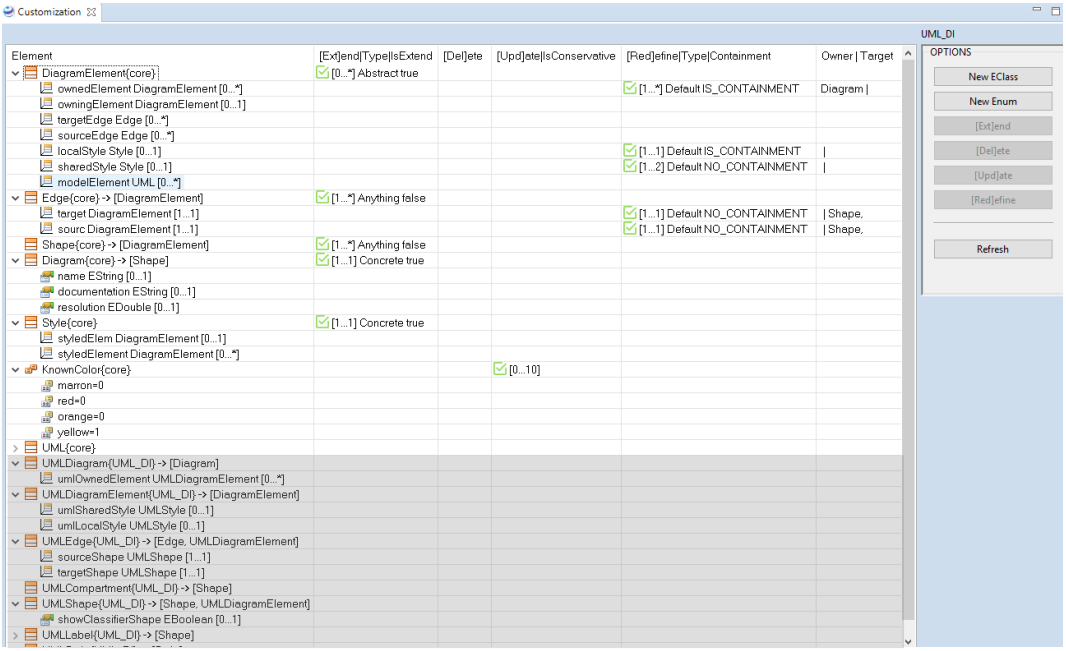


Figura 6.4: Proceso de extensibilidad de DI a través de TACO.

A continuación se expresarán estas reglas de extensión en forma de anotaciones multi-nivel utilizando TOTEM. Dichas anotaciones se muestran en la Figura 6.5. Puede observarse que son necesarias 10 anotaciones para tratar de obtener el mismo resultado. Se han empleado 5 anotaciones para instanciar clases (líneas 3 a 7), y 5 anotaciones para poder instanciar referencias (líneas 8 a 12).

di.mlm

```
1  MLMRules di_mlm for DI potency 2 [save=newMM]
2  {
3      class DI.core.Diagram @2 [1..1] [concrete root]
4      class DI.core.DiagramElement @2 [0..*] [abstract root ]
5      class DI.core.Edge @2 [1..*] [anything open root]
6      class DI.core.Shape @2 [1..*] [anything open root]
7      class DI.core.Style @2 [1..1] [concrete open root]
8      reference DI.core.DiagramElement.localStyle @2 [1..1] [container]
9      reference DI.core.DiagramElement.sharedStyle @2 [1..2] [noContainer]
10     reference DI.core.DiagramElement.ownedElement @2 [1..*] [container ]
11     reference DI.core.Edge.sourc @2 [1..1] [noContainer]
12     reference DI.core.Edge.target @2 [1..1] [ noContainer]
13 }
```

Figura 6.5: Anotaciones multi-nivel de TOTEM aplicadas al meta-modelo DI.

La instanciación del meta-modelo con las anotaciones señaladas se realiza a través de la interfaz principal de la herramienta de modelado multi-nivel (Figura 6.6). En la figura se pueden observar los elementos que han sido instanciados. Por ejemplo, la clase *UMLDiagram* del nivel 1 es una instancia de la clase *Diagram* del nivel 2, de igual manera la referencia *umlOwnedElement* de la clase *UMLDiagram* del nivel 1 es una instancia de la referencia *ownedElement* de la clase *DiagramElement* del nivel 2. Se debe recordar que TOTEM emula la instanciación de clases con herencia (criterio utilizado en la extensibilidad considerada en TACO) y la instanciación de referencias a través de su redefinición. Es interesante señalar que en TOTEM no se puede expresar extensibilidad de enumeraciones para añadir nuevos literales. Tampoco se puede indicar que una instancia de un

clabject debe ser abstracto (sin embargo, con TACO es posible indicar que las extensiones de *DiagramElement* deben ser clases abstractas). El control de la redefinición de referencias es más fino en TACO, ya que permite controlar los destinos válidos para una redefinición. Este control no es posible en TOTEM. Como mostró la Figura 4.27, esta característica sería necesaria si se desea que las instancias de *Edge* no puedan conectarse entre sí (pero si a un *Shape*).

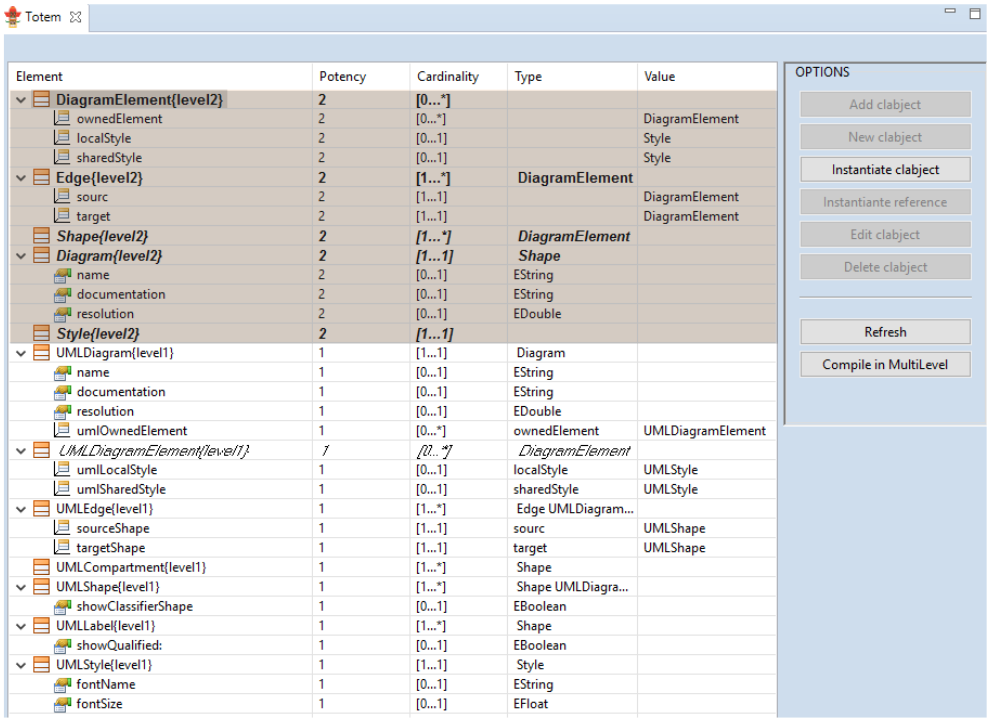


Figura 6.6: Proceso de modelado multi-nivel de DI a través de TOTEM.

Por otra parte, este ejemplo no utiliza toda la potencia del modelado multi-nivel, que permitiría por ejemplo el control de propiedades en clabjects a diferentes niveles. Por ejemplo, utilizando TOTEM sería factible indicar que todas las instancias de *DiagramElemet* deben definir un valor para el *color* mediante un atributo de potencia 1 en la clase *DiagramElement*. Esto no sería posible con TACO, ya que no considera la emulación de valores en el meta-modelo. Adicionalmente, con TOTEM se podría dar una potencia mayor que 2 al meta-modelo *DI* (o potencia ilimitada), que permitiera la definición de familias de sintaxis concretas de manera gradual.

6.3.2 Mecanismo basado en extensión para el meta-modelo de componentes

El meta-modelo *Component* (presentado en la sección 5.2.1), permite crear un lenguaje arquitectónico para que los usuarios del lenguaje puedan definir tipos de componentes, con tipos de puertos de entrada, salida, o ambos. Los tipos de puertos puden definir tipos de mensajes que pueden manejar y conectarse mediante tipos de conectores. En la Figura 6.7 se muestra el meta-modelo *Component* con anotaciones multi-nivel.

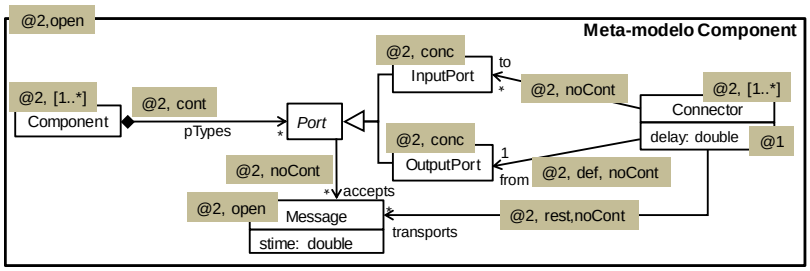


Figura 6.7: Meta-modelo Component con anotaciones multi-nivel.

El proceso de personalización de *Component* a través de la perspectiva de modelado multi-nivel de TOTEM considera la utilización de 11 anotaciones, como se muestran en la Figura 6.8. Se han empleado 5 anotaciones para instanciar clases (líneas 3 a 7), otras 5 para instanciar 5 referencias (líneas 8 a 12), y 1 anotación a nivel de atributo (línea 13).

component.mlm

```
1  MLMRules component_mlm for component potency 2 [save=theSame]
2  {
3      class component.core.Component @2 [1..*] [concrete root]
4      class component.core.Message @2 [0..*] [concrete open root]
5      class component.core.InputPort @2 [0..*] [concrete root]
6      class component.core.OutputPort @2 [0..*] [concrete root]
7      class component.core.Connector @2 [1..*] [concrete root]
8      reference component.core.Component.pTypes @2 [1..*][restrictive container]
9      reference component.core.Port.accepts @2 [0..*][restrictive noContainer]
10     reference component.core.Connector.from @2 [1..1][noContainer]
11     reference component.core.Connector.to @2 [0..*][restrictive noContainer]
12     reference component.core.Connector.transports @2 [0..*][restrictive noContainer]
13     field component.core.Connector.delay @1
14 }
```

Figura 6.8: Anotaciones multi-nivel de TOTEM aplicadas al meta-modelo Component.

La instanciación del meta-modelo se realiza a través de la interfaz principal de la herramienta de modelado multi-nivel (Figura 6.9).

Totem

Element	Potency	Cardinality	Type	Value
Component{level2}	2	[1...*]		
pTypes	2	[1...*]		Port
Port{level2}				
accepts	2	[0...*]		Message
Message{level2}	2	[0...*]		
stime	2	[0...1]	EDouble	
InputPort{level2}	2	[0...*]	Port	
OutputPort{level2}	2	[0...*]	Port	
Connector{level2}	2	[1...*]		
delay	1	[0...1]	EDouble	
to	2	[0...*]		InputPort
from	2	[1...1]		OutputPort
transports	2	[0...*]		Message
Consumer{level1}	1	[1...*]	Component	
inp	1	[1...1]	pTypes	Pin
Producer{level1}	1	[1...*]	Component	
outs	1	[1...*]	pTypes	Pout
Packet{level1}	1	[0...*]	Message	
stime	1	[0...1]	EDouble	
dat	1	[0...1]	EString	
Pin{level1}	1	[0...*]	InputPort	
in	1	[0...1]	accepts	Packet
Pout{level1}	1	[0...*]	OutputPort	
out	1	[0...1]	accepts	Packet
MChannel{level1}	1	[1...*]	Connector	
delay	0	[0...1]	EDouble	5
to_redef	1	[1...2]	to	Pin
from_redef	1	[1...1]	from	Pout
carries	1	[0...10]	transports	Packet

OPTIONS
Add clabject
New clabject
Instantiate clabject
Instantiante reference
Edit clabject
Delete clabject
Refresh
Compile in MultiLevel

Figura 6.9: Instanciación del meta-modelo Component.

Para realizar la prueba de extensibilidad al meta-modelo *Component* con TACO, se han utilizado 10 reglas de extensibilidad como se muestra en la Figura 6.10 (la primera línea de la figura hace referencia a los valores iniciales del modelo de extensibilidad). Se han empleado 5 reglas para extender clases (líneas 2 a 6) y 5 reglas de redefinición de referencias (líneas 7 a 11). Las reglas de extensibilidad de TACO han sido aplicadas a las mismas 5 clases y 5 referencias utilizadas por las anoaciones con TOTEM. Sin embargo, TACO no cuenta con un tipo de regla para referenciar a los atributos para poderles asignar un valor en el modela.

```
componet.custom
1  ExtensibilityRules named component_extend for component [save=theSame]
2  ExtClass { component.core.Component [1..*] concrete }
3  ExtClass { component.core.Message [0..*] concrete }
4  ExtClass { component.core.InputPort [0..*] concrete }
5  ExtClass { component.core.OutputPort [0..*] concrete }
6  ExtClass { component.core.Connector [1..*] concrete }
7  RedReference { component.core.Component.pTypes [1..*] isContain restrictive }
8  RedReference { component.core.Port.accepts [0..*] noContain restrictive }
9  RedReference { component.core.Connector.from [1..1] noContain }
10 RedReference { component.core.Connector.to [0..*] noContain restrictive}
11 RedReference { component.core.Connector.transports [0..*] noContain restrictive}
```

Figura 6.10: Reglas de extensibilidad aplicadas al meta-modelo Component.

El meta-modelo anotado puede extenderse a través de la interfaz principal de TACO, como puede verse en la Figura 6.11.

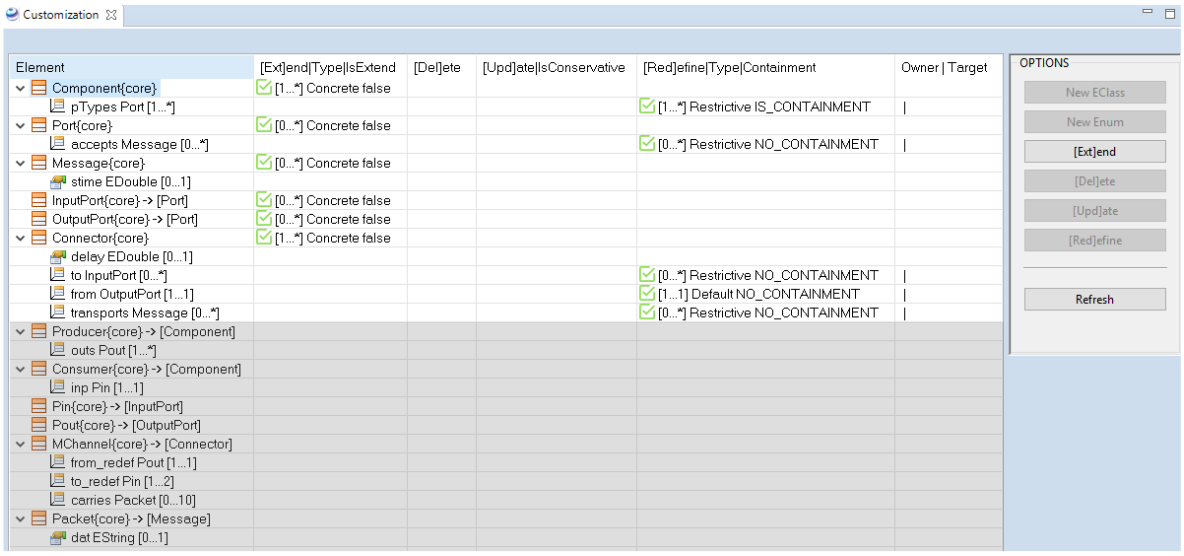


Figura 6.11: Proceso de extensibilidad de Component a través de TACO.

Cabe señalar que en TACO no es posible declarar una regla de extensión que obligue a dar un valor al atributo *delay* de la clase *Connector* cuando se extiende. Esto es debido a que TACO no soporta el concepto de potencia, ni mecanismos de caracterización profunda.

6.3.3 Discusión

Una vez realizados los dos casos de estudio con los dos enfoques de personalización se puede concluir señalando que:

- La realización del proceso de extensión es similar, en el sentido de que primero se deben definir las reglas de extensión (en el caso de TACO) o anotaciones de modelado multi-nivel (en el caso de TOTEM) para señalar los elementos del meta-modelo estándar que serán afectados en el proceso de extensibilidad.
- El proceso de modelado multi-nivel de TOTEM toma como base el proceso de extensibilidad de TACO para poder representar el meta-modelo en varios niveles de clasificación. TOTEM emula de instanciación de clases en base a herencia (extensibilidad) y la instanciación de referencias en base a su redefinición.
- TOTEM puede extender el meta-modelo en varios niveles de clasificación, y controlar las propiedades de instancias en distintos meta-niveles mediante la potencia. TACO sólo puede controlar un meta-nivel (el nivel de meta-modelado).
- Además de las operaciones de extensibilidad, TACO cuenta con operaciones adicionales para la adaptación y simplificación del meta-modelo.
- El meta-modelo extendido con TACO y TOTEM puede ser instanciado (creación de modelos) automáticamente en EMF.
- El diseño basado en modelos de ambos enfoques permite su ampliación y mejora con nuevas primitivas y funcionalidades.

6.4 Resumen y conclusiones

En este capítulo se han discutido los resultados de la investigación realizada. Se han identificado las propiedades y escenarios de aplicación de cada uno de los enfoques de personalización. A lo largo de la redacción de la tesis se ha descrito el desarrollo de un caso de estudio particular aplicando a cada uno de los enfoques, mientras que en la sección 6.3 se ha descrito el desarrollado de cada caso de estudio aplicando los dos enfoques de extensión, con el objetivo de compararlos.

Los casos de estudio y otros disponibles en http://miso.es/tools/totem.html, muestran que los dos enfoques son similares en cuanto a la realización del proceso de extensibilidad, debido a que el mecanismo de modelado multi-nivel de TOTEM se basa en el mecanismo de extensibilidad de TACO. Sin embargo, en términos generales se puede señalar que TACO es aplicable en meta-modelos que han sido diseñados con la intención de ser extendidos a futuro para ser adaptados a dominios más específicos. No obstante, estas extensiones están limitadas a un solo meta-nivel. Por otra parte, TOTEM puede ser aplicado cuando en el meta-modelo existen varios niveles de clasificación,

especialmente cuando surge el patrón tipo-objeto o alguna de sus variantes. Estas diferencias hacen que cada mecanismo de extensibilidad tenga sus propias particularidades.

Finalmente, también se considerará la extensión del mecanismo de extensión para la definición de líneas de producto de meta-modelos [141, 111]. De esta manera, las anotaciones se podrían interpretar como indicaciones de la variabilidad permitida sobre cada uno de los elementos. Estas anotaciones se podrían combinar con las reglas de extensión, lo que permitiría obtener las ventajas de los enfoques abiertos y cerrados para la definición de familias de lenguajes.

7 Conclusiones y Trabajos Futuros

En el Capítulo 1, se identificaron los principales inconvenientes relacionados a la personalización de meta-modelos desde la perspectiva de dos técnicas, la extensibilidad del meta-modelo y el modelado multi-nivel. Este último capítulo vuelve sobre dichos inconvenientes para concluir en qué modo esta tesis ha resuelto dichas limitaciones. Finalmente, se propone el trabajo futuro que se desarrollará tras haber completado esta investigación

7.1 Conclusiones

En esta tesis se han presentado dos mecanismos que permiten personalizar la sintaxis abstracta (meta-modelo) de lenguajes de modelado de dominio específico. Por un lado, se ha diseñado un mecanismo de extensibilidad de meta-modelos a través de un conjunto de reglas que aseguran que la extensibilidad sea realizada correctamente. Por otro lado, este enfoque de extensión se ha utilizado como punto de partida para el diseño de un mecanismo de modelado multi-nivel.

7.1.1 Extensibilidad de meta-modelos

En cuanto a la extensibilidad de meta-modelos, se ha propuesto un mecanismo, una arquitectura y un conjunto de herramientas que soportan la definición de reglas de extensión para meta-modelos y la realización de extensiones específicas de acuerdo con las reglas definidas. Las reglas están

definidas por un modelo de extensión, típicamente construido por el diseñador del meta-modelo a ser extendido. Posteriormente, otros ingenieros pueden usar las reglas de extensión para extender el meta-modelo base. Las herramientas guían en esta extensión asegurando que obedezcan las reglas definidas.

El enfoque propuesto tiene la ventaja de que no es intrusivo, y es genérico, es decir, las reglas de extensión se pueden vincular a cualquier meta-modelo. Adicionalmente, una definición explícita de reglas de extensión evita la introducción de errores accidentales debido al uso del lenguaje natural.

Las ideas de este mecanismo han sido implementadas en la herramienta TACO, mostrándose sus beneficios a través de varios casos de estudio. La herramienta está disponible en http://miso.es/tools/taco.html.

7.1.2 Modelado multi-nivel

En esta tesis, también se han discutido las ventajas del modelado multi-nivel en los escenarios en los que surge el patrón tipo-objeto o algunas de sus variantes. Se han revisado diferentes enfoques de modelado multi-nivel, caracterizando su espacio de diseño a través de un modelo de características. Se ha utilizado este modelo como base para comparar algunas herramientas representativas de modelado multi-nivel. A partir de esta revisión, se observa una falta de integración con las tecnologías estándar de meta-modelado, y se argumenta que una mejor compatibilidad con los estándares podría conducir a una adopción más amplia del modelado multi-nivel, y los beneficios que aporta en términos de reducción de la complejidad accidental [11]. Para mejorar esta situación, se ha propuesto un enfoque novedoso de modelado multi-nivel, que emula los clabjects con clases y la instanciación con herencia. Estas ideas han sido plasmadas en la herramienta TOTEM, y se ha mostrado sus beneficios a través de varios casos de estudio. Disponible en http://miso.es/tools /totem.html

7.2 Trabajos futuros

Los nuevos conceptos propuestos en esta tesis han abierto una serie de líneas de trabajo futuro, que se detallan a continuación.

En cuanto a la extensibilidad de meta-modelos, se abordará el problema de migración de modelos (adelante y hacia atrás) cuando se realizan extensiones concretas. Aunque las reglas actuales permiten expresar las extensiones descritas en estándares como KDM o DD, se analizarán otros sistemas para verificar si es necesario mejorar la expresividad de las reglas de extensión. También se considera implementar las operaciones de tipo "Personalización de Propiedad (PP)" de la tabla 4.1.

Se estudiará el empleo del enfoque basado en extensión para expresar requisitos de reutilización de las transformaciones y generadores de código asociados. De manera similar a [45], dado un artefacto MDE (transformación, generador de código) definido sobre un meta-modelo, generaremos

automáticamente reglas de extensión, que describan las modificaciones que se pueden realizar al meta-modelo original, de tal forma que preserven el comportamiento del artefacto MDE (esto es, la transformación seguiría funcionando sobre cualquier meta-modelo obtenido aplicando las reglas de extensión generadas).

A nivel técnico, se evaluará con usuarios la usabilidad de la herramienta. También se plantea la consideración en la extensión de la sintaxis concreta. De esta manera, no sólo será posible extender meta-modelos, sino DSLs (gráficos o textuales).

Finalmente, se explorará la extensión de los enfoques propuestos para la definición de líneas de productos de meta-modelo [140, 72] .

Respecto al modelado multi-nivel, se planea mejorar diferentes aspectos de la herramienta, por ejemplo, crear un editor gráfico para meta-modelado, integrar el catálogo de refactorings multi-nivel propuesto en [42], y lograr una mejor integración con los lenguajes de gestión de modelos. En particular, se planea hacer posible la evaluación de OCL estándar a niveles diferentes del nivel 0. También se plantea hacer la herramienta TOTEM interoperable con otras herramientas multi-nivel, como MetaDepth o Melanee.

Referencias

[1] S. S. ALHIR. (2006). *"Guide to Applying the UML"*. Springer Science & Business Media. Vol. 1. pp. 411.

[2] J. P. A. ALMEIDA, U. FRANK and T. KÜHNE. (2018). *"Multi-level modelling (Dagstuhl seminar 17492)".* Dagstuhl Reports. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. Vol. 1. pp. 49.

[3] C. ATKINSON. (1997). *"Meta-Modeling for Distributed Object Environments".* 1st International Enterprise Distributed Object Computing (EDOC). Gold Coast, Australia. IEEE. Vol. 1. pp. 90.

[4] C. ATKINSON and R. GERBIG. (2016). *"Flexible deep modeling with Melanee"*. Modellierung 2016 Workshop - Workshop on Modeling 2016 - Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI). Karlsruhe, Germany. Gesellschaft fur Informatik (GI). Vol. P255. pp. 117-121.

[5] C. ATKINSON, R. GERBIG and M. FRITZSCHE. (2015). *"A multi-level approach to modeling language extension in the Enterprise Systems Domain"*. Information Systems.Vol. 54. pp. 289-307.

[6] C. ATKINSON, R. GERBIG and T. KÜHNE. (2014). *"Comparing multi-level modeling approaches".* 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014). Valencia, Spain. Vol. 1286. pp. 53-61.

[7] C. ATKINSON and T. KUHNE. (2003). *"Model-driven development: a metamodeling foundation"*. IEEE Software.Vol. 20. pp. 36-41.

[8] C. ATKINSON and T. KÜHNE. (2001). *"The essence of multilevel metamodeling"*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Toronto,Canada. Springer Verlag. Vol. 1. pp. 19-33.

[9] C. ATKINSON and T. KÜHNE. (2017). *"On Evaluating Multi-level Modeling".* 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017). Austin, TX, USA. ACM/IEEE. Vol. 2019. pp. 274-277.

[10] C. ATKINSON and T. KÜHNE. (2002). *"Rearchitecting the UML infrastructure"*. ACM Transactions on Modeling and Computer Simulation.Vol. 12. pp. 290-321.

[11] C. ATKINSON and T. KÜHNE. (2008). *"Reducing accidental complexity in domain models"*. Software & Systems Modeling.Vol. 7. pp. 345-359.

[12] C. ATKINSON and T. KÜHNE. (2007). *"A tour of language customization concepts"*. Advances in Computers.Vol. 70. pp. 105-161.

[13] ATLANECORE. (2014). *"AtlanEcore metamodel zoo"*. http://www.emn.fr/z-info/atlanmod /index.php/Ecore

[14] ATLANMOD. *"QVT"*. http://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Main_ Page

[15] K. BĄK, Z. DISKIN, M. ANTKIEWICZ, K. CZARNECKI and A. WĄSOWSKI. (2013). *"Partial instances via subclassing"*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 8225 LNCS. pp. 344-364.

[16] R. BENDRAOU, B. COMBEMALE, X. CRÉGUT and M. P. GERVAIS. (2007). *"Definition of an executable SPEM 2.0".* 14th Asia Pacific Software Engineering Conference, ASPCE 2007. Nagoya, Japan. IEEE. Vol. 1. pp. 390-397.

[17] S. BERNER, M. GLINZ and S. JOOS. (1999). *"A Classification of Stereotypes for Object-Oriented Modeling Languages".* International Conference on the Unified Modeling Language. Springer. Vol. 1723. pp. 249-264.

[18] L. BETTINI. (2016). *"Implementing Domain-Specific Languages with Xtext and Xtend"*. Packt Publishing. Vol. 1. pp. 342.

[19] J. BÉZIVIN. (2004). *"In search of a basic principle for model driven engineering"*. Novatica Journal, Special Issue.Vol. 5. pp. 21-24.

[20] J. BÉZIVIN, F. JOUAULT and P. VALDURIEZ. (2004). *"First Experiments with a ModelWeaver".* 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. Vancouver, Canada. Citeseer. Vol. 1. pp. 5.

[21] D. BIRSAN. (2005). *"On plug-ins and extensible architectures"*. ACM Queue.Vol. 3. pp. 40-46.

[22] C. BOCK and M. ELAASAR. (2018). *"Reusing metamodels and notation with Diagram Definition"*. Software & Systems Modeling.Vol. 17. pp. 1079-1103.

[23] P. BOTTONI and A. GRAU. (2004). *"A suite of metamodels as a basis for a classification of visual languages".* IEEE Symposium on Visual Languages and Human Centric Computing 2004. Rome, Italy. IEEE. Vol. 1. pp. 83-90.

[24] M. BRAMBILLA, J. CABOT and M. WIMMER. (2017). *"Model-Driven Software Engineering in Practice"*. Morgan & Claypool. Vol. 2. pp. 182.

[25] M. BRAMBILLA and P. FRATERNALI. (2014). *"Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML"*. Morgan Kaufmann. Vol. 1. pp. 401.

[26] R. BRAUN. (2015). *"Behind the scenes of the BPMN extension mechanism: Principles, problems and options for improvement".* 3rd International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2015. Loire Valley, France. SciTePress. Vol. 1. pp. 403-410.

[27] R. BRAUN and W. ESSWEIN. (2015). *"Extending the MOF for the adaptation of hooks, aspects, plug-ins and add-ons".* 5th International Conference on Model and Data Engineering, MEDI 2015. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Rhodes,Greece. Springer Verlag. Vol. 9344. pp. 28-38.

[28] R. BRAUN and W. ESSWEIN. (2015). *"Towards an integrated method for the extension of MOF-based modeling languages".* 5th International Conference on Model and Data Engineering, MEDI 2015. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Rhodes, Greece. Springer. Vol. 9344. pp. 103-115.

[29] H. BRUNELIERE, J. GARCIA, P. DESFRAY, D. E. KHELLADI, R. HEBIG, R. BENDRAOU and J. CABOT. (2015). *"On lightweight metamodel extension to support modeling tools agility. ".* 11th European Conference on Modelling Foundations and Applications, ECMFA 2015. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). L'Aquila, Italy. Springer. Vol. 9153. pp. 62-74.

[30] J. CABOT and M. GOGOLLA. (2012). *"Object constraint language (OCL): a definitive guide".* International School on Formal Methods for the Design of Computer, Communication and Software Systems. Berlin, Heidelberg. Springer. Vol. 1. pp. 58-90.

[31] V. A. CARVALHO and J. P. A. ALMEIDA. (2018). *"Toward a well-founded theory for multi-level conceptual modeling"*. Software & Systems Modeling.Vol. 17. pp. 205-231.

[32] M. V. CENGARLE, H. GRÖNNIGER and B. RUMPE. (2009). *"Variability within modeling language definitions".* International Conference on Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Denver, CO, United States. Springer. Vol. 5795. pp. 670-684.

[33] H. CERVANTES, P. VELASCO-ELIZONDO and L. CASTRO. (2016). *"Arquitectura de Software: conceptos y ciclo de desarrollo"*. CENGAGE Learning. Vol. 1. pp. 199.

[34] A. CICCHETTI, D. DI RUSCIO, R. ERAMO and A. PIERANTONIO. (2008). *"Automating co-evolution in model-driven engineering".* 12th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2008. Munich, Germany. IEEE. Vol. 1. pp. 222-231.

[35] T. CLARK and U. FRANK. (2018). *"Multi-level constraints".* Proceedings of MODELS 2018 Workshops with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018). Copenhagen, Denmark. Vol. 2245. pp. 103-117.

[36] T. CLARK, P. SAMMUT and J. S. WILLANS. (2015). *"Applied Metamodelling: A Foundation for Language Driven Development (Third Edition)"*. Cornell University. Vol. abs/1505.00149. pp. 228.

[37] T. CLARK, P. SAMMUT and J. S. WILLANS. (2015). *"Super-Languages: Developing Languages and Applications with XMF (Second Edition)"*. CoRR.Vol. abs/1506.03363. pp. 414.

[38] CLOUDAPPLICATIONS. *"CloudML"*. http://cloudml.org/

[39] M. CHINOSI and A. TROMBETTA. (2012). *"BPMN: An introduction to the standard"*. Computer Standards & Interfaces.Vol. 34. pp. 124-134.

[40] J. DE LARA and E. GUERRA. (2010). *"Deep meta-modelling with metadepth".* 48th International Conference on Objects, Models, Components, Patterns, TOOLS 2010. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Malaga, Spain. Springer. Vol. 6141. pp. 1-20.

[41] J. DE LARA and E. GUERRA. (2017). *"A posteriori typing for model-driven engineering: Concepts, analysis, and applications"*. ACM Transactions on Software Engineering and Methodology. Association for Computing Machinery.Vol. 25(4). pp. 60.

[42] J. DE LARA and E. GUERRA. (2018). *"Refactoring Multi-Level Models"*. ACM Transactions on Software Engineering and Methodology. Association for Computing Machinery.Vol. 27(4). pp. 17:1-17.

[43] J. DE LARA, E. GUERRA, R. COBOS and J. MORENO-LLORENA. (2012). *"Extending deep meta-modelling for practical model-driven engineering"*. The Computer Journal.Vol. 57. pp. 36-58.

[44] J. DE LARA, E. GUERRA and J. S. CUADRADO. (2015). *"Model-driven engineering with domain-specific meta-modelling languages"*. Software & Systems Modeling.Vol. 14. pp. 429-459.

[45] J. DE LARA, E. GUERRA, D. DI RUSCIO, J. DI ROCCO, J. SÁNCHEZ, L. IOVINO and A. PIERANTONIO. (2019). *"Automated Reuse of Model Transformations through Typing Requirements Models"*. ACM Transactions on Software Engineering and Methodology. (in press).

[46] J. DE LARA, E. GUERRA and J. SÁNCHEZ. (2014). *"When and how to use multilevel modelling"*. ACM Transactions on Software Engineering and Methodology (TOSEM).Vol. 24. pp. 1-46.

[47] G. DELTOMBE, O. LE GOAER and F. BARBIER. (2012). *"Bridging KDM and ASTM for Model-Driven Software Modernization".* 24th International Conference on Software Engineering & Knowledge Engineering (SEKE'2012). Redwood City, San Francisco Bay, USA. Vol. 1. pp. 517-524.

[48] J. DI ROCCO, D. DI RUSCIO, L. IOVINO and A. PIERANTONIO. (2014). *"Dealing with the Coupled Evolution of Metamodels and Model-to-text Transformations".* 8th ME@MoDELS 2014. Valencia, Spain Vol. 1. pp. 22-31.

[49] D. DI RUSCIO, L. IOVINO and A. PIERANTONIO. (2013). *"Managing the coupled evolution of metamodels and textual concrete syntax specifications".* 39th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2013. Santander; Spain. IEEE. Vol. 1. pp. 114-121.

[50] D. DI RUSCIO, L. IOVINO and A. PIERANTONIO. (2013). *"A methodological approach for the coupled evolution of metamodels and atl transformations".* 6th International Conference on Theory and Practice of Model Transformations, ICMT 2013. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Budapest; Hungary. Springer. Vol. 7909. pp. 60-75.

[51] N. DRIVALOS, D. S. KOLOVOS, R. F. PAIGE and K. J. FERNANDES. (2008). *"Engineering a DSL for software traceability".* 1st International Conference on Software Language Engineering, SLE 2008. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Toulouse, France. Springer. Vol. 5452. pp. 151-167.

[52] ECLIPSE. *"ECORE"*. https://wiki.eclipse.org/Ecore

[53] ECLIPSE. *"Epsilon"*. http://www.eclipse.org/epsilon/

[54] ECLIPSE. *"GMF"*. https://www.eclipse.org/gmf-tooling/

[55] ECLIPSE. *"Graphiti"*. https://www.eclipse.org/modeling/gmp/?project=graphiti#graphiti

[56] ECLIPSE. *"Textual Concrete Syntax (TCS)"*. https://wiki.eclipse.org/TCS

[57] ECLIPSE. *"Xpand"*. https://www.eclipse.org/modeling/m2t/?project=xpand

[58] ECLIPSE. *"Xtext"*. https://www.eclipse.org/Xtext

[59] M. ELAASAR and Y. LABICHE. (2011). *"Diagram definition: a case study with the UML class diagram".* 14th International Conference on Model Driven Engineering Languages and Systems, MODELS 2011. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Wellington, New Zealand. Springer. Vol. 6981. pp. 364-378.

[60] M. FAYAD and D. C. SCHMIDT. (1997). *"Object-oriented application frameworks"*. Communications of the ACM. Association for Computing Machinery (ACM).Vol. 40(10). pp. 32-38.

[61] C. M. FONSECA. (2017). *"ML2: an expressive multi-level conceptual modeling language".* Master's Thesis. University.Universidade Federal do Espírito Santo.

[62] E. FOUNDATION. *"EMF"*. https://www.eclipse.org/modeling/emf/

[63] T. E. FOUNDATION. *"ATL"*. https://www.eclipse.org/atl/

[64] U. FRANK. (2014). *"Multilevel modeling: Toward a new paradigm of conceptual modeling and information systems design"*. Business and Information Systems Engineering.Vol. 6. pp. 319-337.

[65] L. FUENTES-FERNÁNDEZ and A. VALLECILLO-MORENO. (2004). *"An introduction to UML profiles"*. UML and Model Engineering.Vol. 2. pp. 6-13.

[66] J. GARCÍA, F. GARCÍA, V. PELECHANO, A. VALLECILLO, J. M. VARA and C. VICENTE-CHICOTE. (2014). *"Desarrollo de Software Dirigido por Modelos: conceptos, métodos y herramientas"*. RA-MA. Vol. 1. pp. 586.

[67] D. GARLAN and M. SHAW. (1993). *"Advances in Software Engineering and Knowledge Engineering"*. World Scientific Publishing Company. Vol. 2. pp. 180.

[68] R. GERBIG, C. ATKINSON, J. D. LARA and E. GUERRA. (2016). *"A Feature-based Comparison of Melanee and Metadepth".* 3rd International Workshop on Multi-Level Modelling, MULTI 2016. Saint-Malo,France. CEUR-WS. Vol. 1722. pp. 25-34.

[69] C. GONZALEZ-PEREZ and B. HENDERSON-SELLERS. (2006). *"A powertype-based metamodelling framework"*. Software & Systems Modeling.Vol. 5. pp. 72-90.

[70] E. GUERRA and J. DE LARA. (2017). *"Automated analysis of integrity constraints in multi-level models"*. Data & Knowledge Engineering.Vol. 107. pp. 1-23.

[71] E. GUERRA and J. DE LARA. (2018). *"On the quest for flexible modelling".* 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018. Copenhagen; Denmark. Association for Computing Machinery, Inc. Vol. 1. pp. 23-33.

[72] E. GUERRA, J. DE LARA, M. CHECHIK and R. SALAY. (2018). *"Analysing meta-model product lines".* SLE 2018 - Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, co-located with SPLASH 2018. pp. 160-173.

[73] R. HAWKINS, I. HABLI, D. KOLOVOS, R. PAIGE and T. KELLY. (2015). *"Weaving an Assurance Case from Design: A Model-Based Approach".* 16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015. Daytona Beach, United States. IEEE Computer Society. Vol. 2015. pp. 110-117.

[74] R. HEBIG, D. E. KHELLADI and R. BENDRAOU. (2017). *"Approaches to co-evolution of metamodels and models: A survey"*. IEEE Transactions on Software Engineering.Vol. 43. pp. 396-414.

[75] F. HEIDENREICH, J. JOHANNES, S. KAROL, M. SEIFERT and C. WENDE. (2011). *"Model-based language engineering with EMFText".* 4th International Summer School on Generative and Transformational Techniques in Software Engineering IV, GTTSE 2011. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Braga, Portugal. Springer. Vol. 7680. pp. 322-345.

[76] H. HEITKÖTTER. (2012). *"A Framework for Creating Domain-specific Process Modeling Languages".* 7th International Conference on Software Paradigm Trends. ICSOFT 2012 Rome, Italy. Vol. 1. pp. 127-136.

[77] A. HEUER, T. KAUFMANN and T. WEYER. (2013). *"Extending an IEEE 42010-compliant viewpoint-based engineering-framework for embedded systems to support variant management".* International Embedded Systems Symposium. Springer. Vol. pp. 283-292.

[78] IEEE. (1990). *"IEEE 610-1990 - IEEE Standard Computer Dictionary"*.https://standards.ieee.org/standard/610-1990.html.

[79] M. IGAMBERDIEV, G. GROSSMANN, M. SELWAY and M. STUMPTNER. (2018). *"An integrated multi-level modeling approach for industrial-scale data interoperability"*. Software and System Modeling.Vol. 17. pp. 269-294.

[80] ISO. (2012). *"OCL"*. https://www.iso.org/standard/57306.html

[81] J. L. C. IZQUIERDO, J. CABOT, J. J. LÓPEZ-FERNÁNDEZ, J. SÁNCHEZ, E. GUERRA and J. DE LARA. (2013). *"Engaging end-users in the collaborative development of domain-specific modelling languages".* 10th International Conference on Cooperative Design, Visualization, and Engineering, CDVE 2013. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Alcudia, Mallorca. Springer. Vol. 8091 LNCS. pp. 101-110.

[82] M. JARKE, R. GALLERSDÖRFER, M. A. JEUSFELD, M. STAUDT and S. EHERER. (1995). *"ConceptBase—a deductive object base for meta data management"*. Journal of Intelligent Information Systems.Vol. 4. pp. 167-192.

[83] M. A. JEUSFELD and B. NEUMAYR. (2016). *"DeepTelos: Multi-level Modeling with Most General Instances".* The 35th International Conference on Conceptual Modeling. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Gifu, Japan. Springer Verlag. Vol. 9974. pp. 198-211.

[84] Y. JIANG, W. SHAO, L. ZHANG, Z. MA, X. MENG and H. MA. (2004). *"On the classification of UML's meta model extension mechanism".* International Conference on the Unified Modeling Language. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Inner Mongolia University, Hohhot 010021, China. Springer-Verlag. Vol. 3273. pp. 54-68.

[85] F. JOUAULT, F. ALLILAIRE, J. BÉZIVIN and I. KURTEV. (2008). *"ATL: A model transformation tool"*. Science of Computer Programming.Vol. 72. pp. 31-39.

[86] F. JOUAULT, J. BÉZIVIN and I. KURTEV. (2006). *"TCS: a DSL for the specification of textual concrete syntaxes in model engineering".* 5th International Conference on Generative Programming and Component Engineering, GPCE'06. Co-located with the 21st International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006. Portland, OR; United States. ACM. Vol. 1. pp. 249-254.

[87] K. C. KANG, S. G. COHEN, J. A. HESS, W. E. NOVAK and A. S. PETERSON. (1990). *"Feature-oriented domain analysis (FODA) feasibility study"*. Carnegie Mellon University Pittsburgh - Software Engineering Institute.

[88] G.-D. KAPOS, V. DALAKAS, A. TSADIMAS, M. NIKOLAIDOU and D. ANAGNOSTOPOULOS. (2014). *"Model-based system engineering using SysML: Deriving executable simulation models with QVT".* 8th Annual IEEE International Systems Conference, SysCon 2014. Ottawa, ON; Canada. IEEE Computer Society. Vol. 1. pp. 531-538.

[89] S. KELLY and J.-P. TOLVANEN. (2008). *"Domain-Specific Modeling: Enabling Full Code Generation"*. John Wiley and Sons-IEEE Computer Society. Vol. 1. pp. 444.

[90] C. F. KEMERER and S. SLAUGHTER. (1999). *"An empirical approach to studying software evolution"*. IEEE Transactions on Software Engineering.Vol. 25. pp. 493-509.

[91] B. KENNEL. (2012). *"A unified framework for multi-level modeling".* University.Universität Mannheim, pp. 1-267.

[92] S. KENT. (2002). *"Model driven engineering"*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer. Vol. 2335. pp. 286-298.

[93] D. E. KHELLADI, R. BENDRAOU, R. HEBIG and M.-P. GERVAIS. (2017). *"A semi-automatic maintenance and co-evolution of OCL constraints with (meta) model evolution"*. Journal of Systems and Software.Vol. 134. pp. 242-260.

[94] A. KLEPPE. (2008). *"Software Language Engineering: creating domain-specific languages using metamodels"*. Addison-Wesley Educational Vol. 1. pp. 240.

[95] A. G. KLEPPE, J. WARMER, J. B. WARMER and W. BAST. (2003). *"MDA explained: the Model Driven Drchitecture: practice and promise"*. Addison-Wesley Professional. Vol. 1. pp. 192.

[96] D. S. KOLOVOS, R. F. PAIGE and F. A. POLACK. (2006). *"The Epsilon Object Language (EOL)".* 2nd European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA 2006. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Bilbao,Spain. Springer-Verlag. Vol. 4066. pp. 128-142.

[97] T. KÜHNE. (2018). *"Exploring Potency".* 21th International Conference on Model Driven Engineering Languages and Systems (MODELS 2018). Copenhagen, Denmark. ACM/IEEE. Vol. 1. pp. 2-12.

[98] T. KÜHNE and D. SCHREIBER. (2007). *"Can programming be liberated from the two-level style: multi-level programming with deepjava".* ACM SIGPLAN Notices. ACM. Vol. 42. pp. 229-244.

[99] U. KULESZA, V. ALVES, A. GARCIA, C. J. DE LUCENA and P. BORBA. (2006). *"Improving extensibility of object-oriented frameworks with aspect-oriented programming".* 9th International Conference on Software Reuse, ICSR 2006. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Turin; Italy. Springer-Verlag Berlin. Vol. 4039 LNCS. pp. 231-245.

[100] Y. LAMO, X. WANG, F. MANTZ, O. BECH, A. SANDVEN and A. RUTLE. (2013). *"DPF workbench: a multi-level language workbench for MDE/DPF-tooriistakast: mitmetasandilised keeleprotsessorid mudel-orienteeritud projekteerimiseks"*. Proceedings of the Estonian Academy of Sciences.Vol. 62. pp. 3-16.

[101] P. LANGER, K. WIELAND, M. WIMMER and J. CABOT. (2012). *"EMF Profiles: A Lightweight Extension Approach for EMF Models"*. Journal of Object Technology.Vol. 11. pp. 1-29.

[102] M. M. LANKHORST, H. A. PROPER and H. JONKERS. (2009). *"The architecture of the archimate language".* Enterprise, Business-process and Information Systems Modeling. Berlin, Heidelberg. Springer. Vol. 1. pp. 367-380.

[103] LE4MM. (2018). *"XModeler "*. https://www.wi-inf.uni-duisburg-essen.de/LE4MM/

[104] F. MACÍAS, E. GUERRA and J. DE LARA. (2017). *"Towards rearchitecting meta-models into multi-level models"*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 10650 LNCS. pp. 59-68.

[105] F. MACÍAS, A. RUTLE and V. STOLZ. (2016). *"MultEcore: Combining the best of fixed-level and multilevel metamodelling".* CEUR Workshop Proceedings. Saint-Malo,France. CEUR-WS. Vol. 1722. pp. 66-75.

[106] R. C. MARTIN, D. RIEHLE and F. BUSCHMANN. (1997). *"Pattern Languages of Program Design 3"*. Addison-Wesley Professional. Vol. 1. pp. 656.

[107] E. MEDINA and J. TRUJILLO. (2002). *"A Standard for Representing Multidimensional Properties: The Common Warehouse Metamodel (CWM)".* Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 2435. pp. 232-247.

[108] N. MEDVIDOVIC and R. N. TAYLOR. (2010). *"Software architecture: foundations, theory, and practice".* Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. ACM, 471-472. Vol. pp. 471-472.

[109] S. J. MELLOR, T. CLARK and T. FUTAGAMI. (2003). *"Model-driven development: guest editors' introduction"*. IEEE software.Vol. 20. pp. 14-18.

[110] S. J. MELLOR, K. SCOTT, A. UHL and D. WEISE. (2004). *"MDA distilled: Principles of Model-Driven Architecture"*. Addison-Wesley Professional. Vol. 1. pp. 145.

[111] D. MÉNDEZ-ACUÑA, J. A. GALINDO, T. DEGUEULE, B. COMBEMALE and B. BAUDRY. (2016). *"Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review"*. Computer Languages, Systems and Structures.Vol. 46. pp. 206-235.

[112] T. MENS and P. VAN GORP. (2006). *"A taxonomy of model transformation"*. Electronic Notes in Theoretical Computer Science.Vol. 152. pp. 125-142.

[113] M. MERNIK, J. HEERING and A. M. SLOANE. (2005). *"When and how to develop domain-specific languages"*. ACM computing surveys (CSUR).Vol. 37. pp. 316-344.

[114] P. MOHAGHEGHI, V. DEHLEN and T. NEPLE. (2008). *"A Metamodel and Supporting Process and Tool for Specifying Quality Models in Model-Based Software Development"*. Nord. J. Comput.Vol. 14. pp. 301-320.
[115] T. MOUELHIV, F. FLEUREY and B. BAUDRY. (2008). *"A generic metamodel for security policies mutation".* Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on. IEEE. Vol. 1. pp. 278-286.
[116] J. MUKERJI and J. MILLER. (2003). *"The MDA Guide v1. 0.1"*.
[117] MULTI. (2014). *"MULTI series of workshops at the MODELS conference"*. http://miso.es/multi/
[118] J. MYLOPOULOS, A. BORGIDA, M. JARKE and M. KOUBARAKIS. (1990). *"Telos: Representing knowledge about information systems"*. ACM Transactions on Information Systems (TOIS).Vol. 8. pp. 325-362.
[119] B. NEUMAYR, M. A. JEUSFELD, M. SCHREFL and C. SCHÜTZ. (2014). *"Dual deep instantiation and its ConceptBase implementation".* 26th International Conference on Advanced Information Systems Engineering, CAiSE 2014 .Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Thessaloniki, Greece. Springer Verlag. Vol. 8484 LNCS. pp. 503-517.
[120] B. NEUMAYR, C. G. SCHUETZ, C. HORNER and M. SCHREFL. (2017). *"DeepRuby: Extending Ruby with Dual Deep Instantiation".* CEUR Workshop Proceedings. Austin,United State. CEUR-WS. Vol. 2017. pp. 252-260.
[121] B. NEUMAYR, C. G. SCHUETZ, M. A. JEUSFELD and M. SCHREFL. (2018). *"Dual deep modeling: multi-level modeling with dual potencies and its formalization in F-Logic"*. Software and Systems Modeling.Vol. 17. pp. 233-268.
[122] OBEO. *"Sirius"*. https://www.eclipse.org/sirius/
[123] J. ODELL. (1994). *"Power Types"*. Journal of Object-Oriented Programming (JOOP) Vol. 7. pp. 8-12.
[124] OMG. (2015). *"DD"*. https://www.omg.org/spec/DD/1.1/
[125] OMG. (2002). *"ENTERPRISE APPLICATION INTEGRATION (EAI)"*. https://www.omg.org/spec/EAI/1.0/Beta1/
[126] OMG. (2016). *"KDM"*. https://www.omg.org/spec/KDM/1.4
[127] OMG. (2014). *"MDA"*. https://www.omg.org/mda/
[128] OMG. (2014). *"MDA Guide rev. 2.0"*. https://www.omg.org/cgi-bin/doc?ormsc/14-06-01
[129] OMG. (2016). *"MOF"*. https://www.omg.org/mof/
[130] OMG. (2008). *"MOF MODEL TO TEXT TRANSFORMATION LANGUAGE"*. https://www.omg.org/spec/MOFM2T
[131] OMG. (2001). *"NEGOTIATION FACILITY SPECIFICATION (NEG)"*. https://www.omg.org/spec/NEG/1.0/Beta2/
[132] OMG. (2019). *"Object Management Group"*. https://www.omg.org/
[133] OMG. (2014). *"OCL"*. https://www.omg.org/spec/OCL/2.4/
[134] OMG. (2016). *"QVT"*. https://www.omg.org/spec/QVT/About-QVT/
[135] OMG. (2018). *"STRUCTURED ASSURANCE CASE METAMODEL (SACM)"*. https://www.omg.org/spec/SACM/2.0/
[136] OMG. (2014). *"Summary of OMG specifications"*. http://www.omg.org/spec/
[137] OMG. (2017). *"UML"*. https://www.omg.org/spec/UML/2.5.1/
[138] OMG. (2016). *"XMI"*. https://www.omg.org/spec/XMI/2.4.1/
[139] R. F. PAIGE, D. S. KOLOVOS, L. M. ROSE, N. DRIVALOS and F. A. C. POLACK. (2009). *"The design of a conceptual framework and technical infrastructure for model management*

language engineering". 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009. Potsdam; Germany. IEEE. Vol. 1. pp. 162-171.

[140] G. PERROUIN, M. AMRANI, M. ACHER, B. COMBEMALE, A. LEGAY and P.-Y. SCHOBBENS. (2016). *"Featured model types: towards systematic reuse in modelling language engineering".* Modeling in Software Engineering (MiSE), 2016 IEEE/ACM 8th International Workshop on. IEEE. Vol. 1. pp. 1-7.

[141] A. PESCADOR, A. GARMENDIA, E. GUERRA, J. SÁNCHEZ and J. DE LARA. (2015). *"Pattern-based development of Domain-Specific Modelling Languages".* 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015). Ottawa, Canada. IEEE. Vol. 1. pp. 166-175.

[142] M. PIATTINI VELTHUIS and J. GARZÁS PARRA. (2007). *"Fábricas de Software: Experiencias, tecnologías y organización"*. México DF: Alfaomega Grupo Editor, SA de CV. Vol. 1. pp. 550.

[143] K. POHL, G. BÖCKLE and F. VAN DER LINDEN. (2005). *"Software Product Line engineering: Foundations, principles, and techniques"*. Vol. 1. pp. 467.

[144] REMODD. (2014). *"The repository for model-driven development"*. http://www.cs.colostate.edu/remodd

[145] L. M. ROSE, D. S. KOLOVOS and R. F. PAIGE. (2012). *"Eugenia live: a flexible graphical modelling tool".* Extreme Modeling Workshop, XM 2012 - Post-Proceedings, Satellite Event of the IEEE/ACM 15th International Conference on Model Driven Engineering Languages and Systems, MODELS 2012. Innsbruck,Austria. ACM. Vol. 1. pp. 15-20.

[146] A. ROSSINI, J. D. LARA, E. GUERRA and N. NIKOLOV. (2015). *"A comparison of two-level and multi-level modelling for cloud-based applications"*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 9153. pp. 18-32.

[147] J. RUMBAUGH, I. JACOBSON and G. BOOCH. (2004). *"The Unified Modeling Language Reference Manual"*. Pearson Higher Education. Vol. 1. pp. 752.

[148] H. S. SARJOUGHIAN, A. ALSHAREEF and Y. LEI. (2015). *"Behavioral DEVS metamodeling".* Winter Simulation Conference (WSC), 2015. Huntington Beach,United States. IEEE. Vol. 2016. pp. 2788-2799.

[149] H. S. SARJOUGHIAN and A. M. MARKID. (2012). *"EMF-devs modeling".* Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium. Orlando, FL;United States. Society for Computer Simulation International. Vol. 44. pp. 19.

[150] A. SCHLEICHER and B. WESTFECHTEL. (2001). *"Beyond stereotyping: Metamodeling approaches for the UML".* 34th Annual Hawaii International Conference on System Sciences (HICSS-34). Maui,Hawaii,USA. IEEE Computer Society. Vol. 1. pp. 10.

[151] D. C. SCHMIDT. (2006). *"Model-driven engineering"*. COMPUTER-IEEE COMPUTER SOCIETY-.Vol. 39. pp. 25.

[152] B. SELIC. (2003). *"The pragmatics of model-driven development"*. IEEE Software. IEEE.Vol. 20. pp. 19-25.

[153] B. SELIC. (2007). *"A systematic approach to domain-specific language design using UML".* 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC 2007. Santorini Island; Greece. IEEE. Vol. 1. pp. 2-9.

[154] M. SELWAY, M. STUMPTNER, W. MAYER, A. JORDAN, G. GROSSMANN and M. SCHREFL. (2017). *"A conceptual framework for large-scale ecosystem interoperability and industrial product lifecycles"*. Data and Knowledge Engineering. Elsevier B.V.Vol. 109. pp. 85-111.

[155] M. STARON. (2006). *"Adopting model driven software development in industry–a case study at two companies".* 9th International Conference on Model Driven Engineering Languages

and Systems, MoDELS 2006. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Genova, Italy. Springer. Vol. 4199 LNCS pp. 57-72.

[156] D. STEINBERG, F. BUDINSKY, E. MERKS and M. PATERNOSTRO. (2008). *"EMF: Eclipse Modeling Framework"*. Pearson Education. Vol. 1. pp. 744.

[157] J. P. TOLVANEN and S. KELLY. (2016). *"Model-driven development challenges and solutions: Experiences with domain-specific modelling in industry".* 4th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2016. Rome, Italy. SciTePress. Vol. 1. pp. 711-719.

[158] L. TRATT. (2005). *"Model transformations and tool integration"*. Software & Systems Modeling.Vol. 4. pp. 112-122.

[159] M. VÖLTER, T. STAHL, J. BETTIN, A. HAASE and S. HELSEN. (2013). *"Model-driven software development: technology, engineering, management"*. John Wiley & Sons. Vol. 1. pp. 446.

[160] J. B. WARMER and A. G. KLEPPE. (2003). *"The Object Constraint Language: getting your models ready for MDA"*. Addison-Wesley Professional. Vol. 1. pp. 206.

[161] J. WHITTLE, J. HUTCHINSON and M. ROUNCEFIELD. (2014). *"The state of practice in model-driven engineering"*. IEEE Software. IEEE.Vol. 31. pp. 79-85.

[162] R. WOLFINGER. (2008). *"Plug-in architecture and design guidelines for customizable enterprise applications".* 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications,OOPSLA 2008. Nashville, TN; United States. ACM. Vol. 1. pp. 893-894.

[163] G. YANG, M. KIFER and C. ZHAO. (2003). *"FLORA-2: A rule-based knowledge representation and inference infrastructure for the semantic web"*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer-Verlag Berlin.Vol. 2888. pp. 671-688.

[164] Z. ZHU, Y. LEI, A. ALSHAREEF, H. SARJOUGHIAN and Y. ZHU. (2018). *"Domain Specific MetaModeling for Deep Semantic Composability"*. IEEE Access.Vol. 6. pp. 18276-18289.