Lenguajes Específicos del Dominio para la extracción de modelos desde los Espacios Tecnológicos del *grammarware*, *dataware* y *apiware*

Javier Luis Cánovas Izquierdo

Universidad de Murcia

Tesis Doctoral

Dirigida por
Jesús García Molina

Mayo de 2011

Domain-Specific Languages for bridging *modelware* with *grammarware*, relational data and API Technical Spaces

Javier Luis Cánovas Izquierdo

Universidad de Murcia

PhD. Thesis

Advisor
Jesús García Molina

May 2011

D. Jesús García Molina, Catedrático de Escuela Universitaria del Área de Lenguajes y Sistemas Informáticos en el Departamento de Informática y Sistemas, AUTORIZA:

La presentación de la Tesis Doctoral titulada *Lenguajes Específicos del Dominio para la extracción de modelos desde los Espacios Tecnológicos del grammarware, dataware y apiware*, realizada por D. Javier Luis Cánovas Izquierdo, bajo mi inmediata dirección y supervisión para la obtención del grado de Doctor por la Universidad de Murcia.

En Murcia, a 11 de mayo de 2011.

D. Jesús García Molina, Catedrático de Escuela Universitaria del Área de Lenguajes y Sistemas Informáticos en el Departamento de Informática y Sistemas, y Director del Departamento de Informática y Sistemas, INFORMA:

Que la Tesis Doctoral titulada *Lenguajes Específicos del Dominio para la extracción de modelos desde los Espacios Tecnológicos del grammarware, dataware y apiware*, ha sido realizada por D. Javier Luis Cánovas Izquierdo, bajo su inmediata dirección y supervisión, y que el Departamento ha dado su conformidad para que sea presentada ante la Comisión de Doctorado.

En Murcia, a 11 de mayo de 2011.

A mi padre, por regalarme aquel libro de BASIC que le pedí, ahí comenzó todo.
A mi madre, por estar siempre ahí y enseñarme a perseverar.
A ambos, por darme siempre todo lo que he necesitado.
A mí.

Agradecimientos

Después de cuatro años de duro trabajo, atrás quedan muchas experiencias inolvidables y un montón de enseñanzas. Sin embargo, este documento es sólo un hito dentro de mi carrera investigadora. Solo ahora comprendo cuando dicen que la tesis doctoral no es el final sino el comienzo.

Esta tesis no hubiera visto la luz sin la supervisión y guía de Jesús García Molina. Desde un principio apostó por mí y estuvo siempre disponible para cualquier problema que me surgiera. Sus consejos e indicaciones han sido cruciales durante todo el trabajo de esta tesis y estoy seguro que me valdrán en el futuro. Espero haber aprendido a contar las cosas *con maña*.

Han sido muchas las personas que me han acompañado y me han apoyado a lo largo de estos años. Fundamentalmente, los que siempre han estado ahí han sido mis padres y mi hermano, a ellos quiero agradecer su apoyo incondicional. No tengo palabras para agradecerles todo lo que han hecho por mí. Son muchas las horas que, aunque compartía techo con ellos, parecía como si no estuviera en casa. En todo momento han sabido guiarme, aconsejarme y darme fuerzas para seguir adelante.

La mayoría del trabajo descrito en esta tesis ha sido desarrollado dentro de las cuatro paredes de un laboratorio con gente excepcional. Entre ellos me gustaría destacar a Jesús Sánchez, por esas conversaciones trascendentales acerca de transformaciones, la investigación y otras *frikadas*, son muchos los consejos que me ha dado y me han guiado en estos años; a Óscar, porque soportarme los lunes (y no solo los lunes) no tiene precio; a Fernando, por sus sabios consejos a lo largo de estos años; y a Fran y Joaquín (*¡king!*), que son el alma de la fiesta. Sin olvidar a Javi Bermúdez, que aunque sea visitante del laboratorio, también ha estado siempre apoyándome. Son tantos los buenos recuerdos que se han convertido, más que en compañeros de trabajo, en amigos (dentro de la escala *Cuadrado*). Por supuesto, tampoco puedo olvidar agradecer el apoyo de Jesús Perera, Juan Manuel, Astrid, Espinazo y Miguel. Creo que no hay nada mejor como disfrutar en el lugar en el que trabajas, y la verdad es que con todos ellos es imposible no estar a gusto.

También me gustaría agradecer el apoyo recibido por los *cuatro magníficos*: Quique, Javi, Rafa y Pablo. Creo que poca gente sabe tan bien como ellos el proceso de publicar una tesis. Les debo el haberme sacado más de una vez a que me de el fresco para desconectar. Sin olvidar a Mercedes, que siempre tenía palabras de ánimo.

Durante el desarrollo de este trabajo de investigación realicé varias colaboraciones con diferentes grupos de investigación. Me gustaría agradecer el apoyo y la hospitalidad que me brindaron cuando estuve con ellos, realmente fueron experiencias de las que aprendí mucho. En primer lugar, a la gente del grupo de investigación ONEKIN, Óscar Díaz, Gorka, Jokin, Maider, Cristóbal y Sandy, con los que tuve la suerte de trabajar dos veces. En segundo lugar, a los miembros del grupo Atenea, principalmente a Antonio Vallecillo y

José Rivera, que aunque hicimos una visita fugaz, la colaboración fue muy enriquecedora. Finalmente, a la *família* AtlanMod, Jordi Cabot, Fréderic Jouault, Salva, Wolfgang, Cauê, Hugo, Massimo, Guillaume, Hanane y Jean Bézivin, que hicieron de mi estancia una experiencia magnífica e inolvidable, prueba de ello es mi futura postdoc. Durante mi estancia en Nantes también tuve el placer de conocer a Francesca, Marta, María, Pablo Lodeiro, Guilhem, Frederico, Sandra, Ángel y Sirle (y supongo que me dejo a alguien). Entre todos ellos consiguieron que me sintiera como en casa.

Muchas de las técnicas de trabajo y de estudio que he aplicado aquí se las debo a Victoria Sánchez, mi profesora del colegio. Buena parte de mi tesón y ganas de superación en el trabajo es consecuencia de su esfuerzo.

Estoy seguro que me falta mucha gente a la que agradecer su apoyo durante estos años. Sirvan estas líneas para dar las gracias a todos los que en su momento me echaron una mano o me animaron.

Finalmente, creo también importante destacar que durante estos cuatro años he podido disfrutar de una beca predoctoral de la Fundación Séneca, gracias a la cual he podido dedicarme íntegramente al desarrollo del trabajo de esta tesis, así como una beca para realizar la estancia en el grupo de investigación AtlanMod.

Índice general

Índice de figuras

Índice de tablas

Resumen de la tesis

En los últimos años, el paradigma del Desarrollo Dirigido por Modelos (DSDM) se ha consolidado como una de las alternativas más destacables para alcanzar la industrialización del software gracias a su capacidad para elevar el nivel de abstracción al programar y a la mejora en la productividad y calidad del software que conlleva una mayor automatización. Aunque hasta ahora la mayoría de soluciones basadas en el DSDM han estado centradas en la construcción de nuevos sistemas software, las técnicas de este paradigma también han evidenciado su utilidad en otras aplicaciones, entre las que destaca la reingeniería o modernización de software *legacy*, el control y adaptación del funcionamiento de un sistema en tiempo de ejecución y el uso de modelos como una representación intermedia de alto nivel con el propósito de facilitar la integración de diferentes sistemas. Estas otras aplicaciones requieren normalmente la extracción de modelos a partir de artefactos software de diferente naturaleza, como ficheros de código fuente, ficheros de configuración XML, objetos de un API o datos almacenados en una base de datos.

La obtención de modelos a partir de los artefactos origen es un ejemplo de la necesidad de establecer un puente entre dos espacios tecnológicos. Un *espacio tecnológico* define el contexto de trabajo proporcionados por una determinada tecnología, como es el caso del espacio tecnológico del DSDM (*modelware*) o del espacio tecnológico de los lenguajes de programación (*grammarware*). Un puente entre dos espacios tecnológicos tiene como propósito favorecer la interoperabilidad y utilizar las técnicas y herramientas del espacio tecnológico destino sobre artefactos del espacio tecnológico origen. En el caso de la extracción de modelos, una vez obtenidos los modelos es posible aprovechar técnicas del DSDM como transformaciones de modelos y comparación de modelos.

En la actualidad, la extracción de modelos se realiza por medio de herramientas implementadas con lenguajes de programación generales (*General Programming Languages*, GPLs), que es una tarea tediosa y dificulta enormemente su desarrollo. En el trabajo presentado en esta tesis se ha estudiado el problema de cómo facilitar la extracción de modelos desde tres diferentes espacios tecnológicos que cubren la mayor parte de los artefactos que componen un sistema software y que son: *grammarware*, que incluye código fuente expresado en un lenguaje de programación conforme a una gramática; *dataware*, que incluye los datos que conforman con un esquema de base de datos; y el *apiware*, que incluye a los objetos accesibles mediante un API (p. ej., Swing o SWT). Se han definido lenguajes específicos de dominio para la extracción de modelos desde cada uno de estos tres espacios tecnológicos, con el objetivo de facilitar la creación de puentes entre cada espacio tecnológico considerado y el *modelware*. Con ello se consiguen los beneficios que aporta la utilización de DSL frente a soluciones basadas en el empleo de lenguajes GPL, esto es, incrementar la productividad, mejorar la calidad y favorecer el mantenimiento.

Los lenguajes desarrollados son Gra2MoL (*grammarware-modelware*), SchcMoL (*data-ware-modelware*) y API2MoL (*apiware-modelware*). Realmente, Gra2MoL es una primera propuesta de lenguaje de transformación texto-a-modelo que permite transformar cualquier artefacto software conforme a una gramática en un modelo conforme a un metamodelo. Por otro lado, SchcMoL y API2MoL son realmente dos aproximaciones originales que abren la posibilidad de trasladar la filosofía que inspiró Gra2MoL al ámbito de la extracción de modelos de datos y objetos de API, respectivamente. Además, el trabajo con Gra2MoL ha supuesto arrancar una investigación en un área de reciente aparición como la moderniza-ción basada en modelos, en particular, en la experimentación con los metamodelos básicos de la iniciativa ADM (*Architecture-Driven Modernization*) de OMG.

Thesis abstract

Model-Driven Development (MDD) has, in recent years, become one of the most important alternatives to create a real software industry owing to its ability to raise the level of abstraction and automation when developing software. Although MDD techniques have principally been used to create new software systems, this paradigm has also proved to be suitable for other scenarios such as reengineering or the modernization of software applications, managing runtime systems and using models as high-level intermediate representation to integrate several systems. In all of these scenarios it is usually necessary to extract models from the artefacts of which the software system is composed, such as source code files, XML configuration files, API objects o data stored in databases.

Extracting models from system artefacts requires building a bridge between the technical spaces involved. A technical space (TS) is defined as a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities (e.g., the MDD TS is the *modelware* and the language programming TS is the *grammarware*). A bridge between two technical spaces allows the artefacts created in one TS to be transferred to another different TS in order to use the best possibilities of each technology, thus promoting interoperability between applications. When extracting models, once the models have been obtained, *modelware* techniques such as model transformation and model comparison can be applied to scenarios such as modernizing the software system, analizing data or integrating applications.

Existing tools aimed at extracting models are currently implemented by using General Programming languages (GPL), which is a tedious and time-consuming task. In this thesis the candidate has investigated how to facilitate the definition of bridges from three different TSs through the use of DSLs, thus promoting productivity and improving both quality and maintainability. The TSs considered cover the vast majority of software artefacts and are: *grammarware*, which includes source code in a programming language conforming to a grammar; *dataware*, which includes data conforming to a database schema; and *apiware*, which includes those elements which are accesible via API (e.g., Swing or SWT). We have therefore developed a DSL family in order to extract models from three different TSs and to facilitate the creation of bridges between these TSs and the *modelware*.

The languages developed are: Gra2MoL (*grammarware-modelware*), SchcMoL (*dataware-modelware*) y API2MoL (*apiware-modelware*). Gra2MoL is actually a text-to-model transformation language which allows any text-based artefact conforming to a grammar to be transformed into a model conforming to a metamodel. On the other hand, SchcMoL and API2MoL are two novel approaches which have allowed us to use the Gra2MoL design in the context of extracting models from data and API objects, respectively. The work developed in Gra2MoL has also permitted us to experiment on the OMG's ADM (*Architecture-Driven Modernization*) initiative.

1

Extended Abstract

The Yherajk have been watching us here on Earth for a while, and they decided recently that, after several years of observation, it was time to make themselves known to humanity.
Agent to the Starts, John Scalzi.

1.1. Background

The Model Driven Engineering (MDE) paradigm emphasizes the use of models to raise the level of abstraction and automation in the development of software. Abstraction is a primary technique by which human minds cope with complexity, whereas automation is the most effective method for boosting productivity and quality [89]. In MDE, the approach used to increase the level of abstraction is that of defining Domain-Specific Languages (DSLs) whose concepts closely reflect the concepts of the problem domain, thus facilitating understanding and hiding implementation technologies. The use of DSLs is not new in software development but it is used in this paradigm as a way of manipulating models effectively, thus promoting productivity and communication with domain experts [30]. On the other hand, automation allows models expressed in high-level abstraction to be transformed into equivalent computer programs or other models that are suitable for design analyses. The research community has embraced MDE, partly because they see an opportunity to provide significant improvements in the development of software [89].

Software models have shown their potential for automating the tasks of both forward engineering and reengineering or modernization processes, along with managing models at run-time [9]. In MDE-based software processes aimed at the creation of new systems, abstract models are initially built by developers to describe the system, and model transformations are then in charge of generating certain artefacts of the new software system. In model-driven software reengineering, a reverse engineering process is first applied to obtain the initial models, which will later be transformed in order to restructure and generate the system evolved [14, 35]. A model-driven reverse engineering process is composed of two

steps: (1) extracting low-level models from the existing system artefacts and (2) transforming the extracted models into high-level ones, which will be used in the rest of the reengineering process. The model extraction step is therefore an important task in model-driven software reengineering. Finally, models at run-time is a new area of application of MDE in which the system and the model coexist in time and are kept synchronized by means of model transformations. These models allow the maintenance, management and monitoring of highly complex systems made of many parts, which are continuosly running and must be constantly maintained.

Model transformation is therefore a key technology for MDE to succeed, since it allows parts of a system to be automatically generated or reengineered. According to [21], model transformations may be classified into three categories: model-to-model (*m2m*), model-to-text (*m2t*) and text-to-model (*t2m*). *M2m* transformations are applied when both source and target artefacts are models and can in turn be classified into either horizontal or vertical transformations depending on whether the source and target models reside at either the same or different abstraction level, respectively. *M2t* transformations are used to generate the target system artefacts (e.g., source code or XML configuration files) from models. Finally, *t2m* transformations are performed to extract models from existing system artefacts.

The concept of Technical Space (TS) is introduced in [49] as a way to provide guidance with regard to the use of a specific technological context when solving a particular problem. A TS is defined as a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. The artefacts of which a software system is composed conform to the TS used to develope them. For example, the source code of a program conforms to the grammar TS, called *grammarware*, whereas XML documents conform to the XML TS, called *xmlware*. A TS is defined based on a couple of basic concepts which are related to each other by means of a conformance relationship (e.g., program/grammar in *grammarware* or document/schema in *xmlware*). Moreover, TSs are not isolated and bridges can in fact be defined between two different TSs, which may be bidirectional or one-way. Bridging TSs allows the artefacts created in one TS to be transferred to another different TS in order to use the best possibilities of each technology, thus promoting interoperability between applications. For example, the processing of XML data documents from a program is an one-way bridge from *xmlware* to *grammarware* which allows the source code to interpret the XML data, thus promoting data interoperability. When bridging the MDE TS, called *modelware*, with other TSs, two operations should be supported: (1) the extraction of models from software system artefacts conforming to the TS considered and (2) the generation of these artefacts from models. For instance, a bidirectional *grammarware-modelware* bridge should provide both a model extractor from source code and a source generator from models. Thus, *t2m* and *m2t* transformations applied to source code files allow one-way bridges to be implemented between *grammarware* and *modelware*.

There exist specific languages for defining *m2m* and *m2t* transformations, for instance, ATL [40], QVT [96], Epsilon [27] or RubyTL [87] for *m2m* and MOFScript [54], JET [70] and XPand [109] for *m2t*, whereas *t2m* transformations are normally performed by hand-crafted specific solutions. With regard to *m2m* transformations, both the source and target

elements of the transformation are inside *modelware*. However both *m2t* and *t2m* must deal with the different artefacts of which a software system is composed, that is, they must deal with the TS to which the artefact involved in the transformation conforms. In the case of *m2t* transformations, the great majority of existing solutions allow these transformations to be defined through the use of templates, where the text to be generated conforms to a layout which is filled in by model information. On the other hand, solutions aimed at performing *t2m* transformations are normally ad hoc tools which deal with a specific type of artefacts, e.g., the Java source code extractor in MoDisco [11] or the PL/SQL source code extractor in Forms2Net [1]. The main difference between *m2t* and *t2m* is that template-based *m2t* transformations can generate any text-based artefact (i.e., source code, XML files, etc) without regarding the formalism to which such artefact conforms, whereas *t2m* transformations must know such formalism to be able to understand the artefact structure and extract information from it, that is, applying tools and techniques from the corresponding source TS. For example, to generate source code from models we can use code templates (e.g., MOFScript), whereas to obtain models from source code we need to use concepts and techniques of the *grammarware* such as the grammar of the programming language used and parser building. Thus, *t2m* transformations are normally hardcoded in parsers whose implementation is tedious and time-consuming.

1.2. Goals

This thesis aims to facilitate the definition of *t2m* transformations in order to free developers from the task of implementing specific solutions, which is both error-prone and expensive. Similarly, as it has been done in *m2m* and *m2t* transformations, DSL may be used to define *t2m* transformations. Providing DSL for tackling *t2m* tranformations would allow developers to define bridges between TSs effectively since they can work at the same level of abstraction of the problem domain, thus promoting productivity, reliability and maintainability [23, 50].

We have therefore defined a family of DSLs to specify *t2m* transformations for different TSs. The TSs considered cover the vast majority of artefacts of which a software system is composed, and are: the *grammarware*, which includes those software artefacts conforming to a grammar such as the source code; the *dataware*, which deals with data conforming to a relational database schema; and *apiware*, which includes those software artefacts that are accesible via an object-oriented API, that is, the set of runtime objects which are accesible through an API.

Extracting models from both source code and data is an essential task in software reengineering because they are commonly the main assets of an application. On the other hand, software systems usually manage a plethora of API to access different software assets (e.g., databases, middleware, etc). A model-driven software reengineering process therefore aldo normally involves extracting models from legacy artefacts using API.

In addition, while *m2t* transformations can be applied to generate API code and several model-driven reverse engineering techniques for API code currently exist [4, 83], there are no tools that facilitate the definition of *apiware-modelware* bridges. The DSL regarding such

bridge was therefore defined as being bidirectional, thus allowing models to be obtained from a set of objects which are accessible through an API as well as generating API objects from models.

1.3. Methodology

The research work started facing the problem of extracting models from source code. After analyzing several existing solutions, we decided to create a DSL, called Gra2MoL, to define bridges from *grammarware* to *modelware*. In fact, the first objective of this thesis was to deal only with this type of bridges in order to build a model-driven modernization framework integrated into the AGE development environment [85], which would be aimed to modernize source code and Gra2MoL would be used to deal with source code files. However, we later decided to consider other TSs (i.e., *dataware* and *apiware*), thus focusing the scope of this thesis on the extraction of models, which consequently led to our colaboration with the ONEKIN and AtlanMod research groups.

In accordance with the DSL development process described in [30], we first developed the abstract syntax of Gra2MoL and then the concrete syntax, along with the tooling of the DSL. The Gra2MoL language was inspired by *m2m* transformation languages such as ATL [40]. The development of Gra2MoL therefore allowed us to study how techniques used in *m2m* could be applied and adapted to the problem of bridging *grammarware* and *modelware*. Like *m2m* transformation languages, Gra2MoL is a rule-based language and incorporates the concept of binding. However, the way of traversing the source artefacts changes in Gra2MoL, which has to deal with source code files as input. When extracting models from source code, it is necessary to traverse the source code (i.e., the syntax tree) to obtain the scattered information of which the model elements are composed. We therefore incorporated a powerful query language into Gra2MoL in order to perform such operation effectively, rather than using an OCL-based query language as *m2m* transformation languages do.

Gra2MoL was applied in several case studies which allowed us to identify some weak points and to devise solutions. The language features were thus extended during the development of these case studies. For example, when extracting models from Maude source code, the language was improved to support island grammars. On the other hand, dealing with a large number of source code files required the incorporation of an efficient memory management mechanism, and the language was therefore improved to support CDO [62], which is a model repository designed for the scalability, transactionality and distribution of models.

It is also worth noting that Gra2MoL was used to develop one of the first case studies of ADM (Architecture-Driven Modernization). ADM is an OMG initiative whose objective was to develop a set of standard metamodels to represent the information involved in a model-driven modernization process, thus facilitating interoperability among tools. The ADM initiative includes the definition of seven metamodels, although only three are currently available: the Knowledge Discovery Metamodel (KDM), the Abstract Syntax Tree Metamodel (ASTM) and the Software Measurement Metamodel (SMM). The other four

are under development (analysis program, visualization, refactoring, and transformation). In a modernization project, in which Oracle Forms were migrated to a Java platform, we used Gra2MoL to extract ASTM models from PL/SQL source code, which later facilitated the task of obtaining KDM models by means of a *m2m* transformation. These KDM models were eventually used to calculate some metrics in order to assess the modernization effort.

Our collaboration with the ONEKIN research group allowed us to study the problem of extracting models from data stored in relational databases, that is, a bridge from *dataware* to *modelware*. Since existing approaches are based on ad hoc solutions, we decided to create a new DSL, called ScheMoL, which was specially designed to deal with this type of bridges. From our experience in Gra2MoL, we designed ScheMoL by reusing a part of the Gra2MoL core implementation. ScheMoL shares with Gra2MoL the use of rules and the concept of binding, which was adapted to deal with data elements (i.e., tuples). Like Gra2MoL, ScheMoL also incorporates a query language, but was specially built to retrieve information from databases in a transparent manner. It is also important to note that ScheMoL incorporates an extension mechanism that was used to define a set of operations with which exploit the annotated markup embedded in web 2.0 databases.

Finally, the predoctoral internship in the AtlanMod research team allowed the candidate to face the problem of bridging *apiware* with *modelware*. We created a new DSL, called API2MoL, to create bidirectional bridges between the TSs involved. This bridge would therefore allowed us to extract models from run-time application objects conforming to an object-oriented API, and to generate such objects from models. Unlike the DSL described previously, its design was a little different since the problem requirements were not similar. API2MoL also uses rules to define the mappings involved in this bridge but it does not incorporate a query language since it permits the specification of the set of API methods to be called in order to get/set information from/into the objects. The approach also incorporates a discovery process to automatically generate the *apiware-modelware* bridge. This is a novel approach which, to the best of our knowledge, is the first generic approach to build this type of bridges.

1.4. Results

The main result obtained during this thesis was the creation of a family of DSLs for extracting models from various TSs. Before describing each DSL, we will provide a short description of the main concepts of the TSs considered and how the bridges can be defined.

The OMG's four level architecture [97] is commonly used to define the instantiation and conformance relationships among the main concepts involved in a particular TS. This architecture has raised some controversial issues about both the meaning and relationships of each layer, as well as the layers included in each TS [7, 28]. In this document we use the approach presented in [8], which describes an architecture composed of 3+1 layers. The levels of these hierarchies are useful to describe the bridging between TSs. Figure 1.1 shows the concepts and relationships of the TSs considered in this thesis. M1 concepts generally conform to those of M2 (e.g., models conform to metamodels or programs conform to

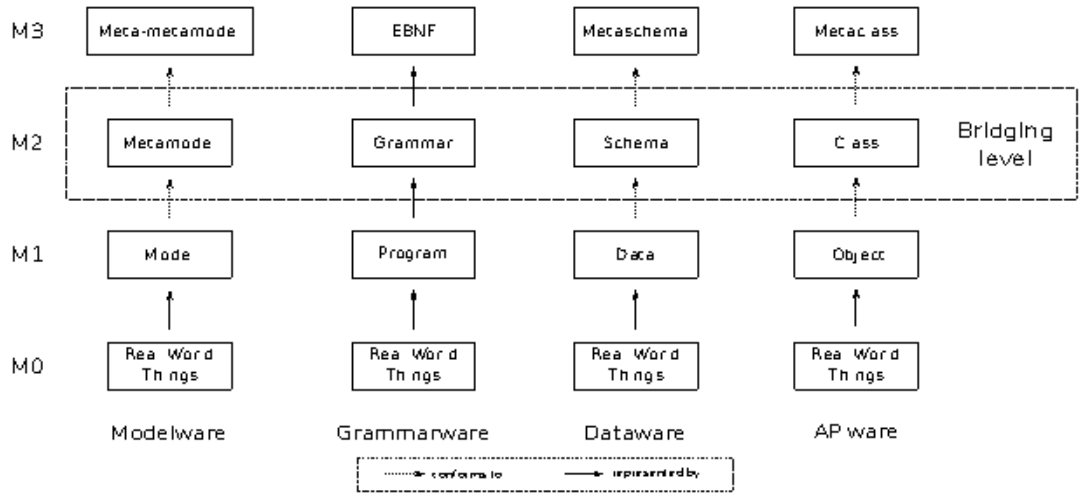


Figura 1.1: Four-level architecture for the TSs considered.

grammars), which in turn conform to M3 concepts (e.g., metamodels conform to meta-metamodel and grammars conform to meta-grammars or EBNF). A bridge between two TSs is defined at M2 level by means of mappings between the elements involved in each TS. On the other hand, the execution of these bridges is carried out at M1 level. For instance, a *grammarware-modelware* bridge defines mappings between grammar elements and metamodel elements, whereas the result of bridging a program conforming to the grammar will be a model conforming to the metamodel. *Dataware-modelware* mappings are similarly defined between schema elements (i.e., tables) and metamodel elements, and *apiware-modelware* mappings are specified between API classes and metamodel elements.

The DSLs created therefore allow mappings to be defined between the elements involved at the M2 level in each TS. Each DSLis briefly presented as follows along with a description of the results obtained. A more detailed explanation of each DSL can be found in the papers indicated.

1.4.1. Gra2MoL

Gra2MoL allows defining bridges from *grammarware* to *modelware* by means of grammar-to-metamodel transformations. Figure 1.2 shows the process involved in this type of transformations. According to Figure 1.2, the input of a Gra2MoL transformation (T) is a program (P) along with the grammar definition (G) it conforms to, a target metamodel (MM_T) and a mapping definition; the output is a model (M_T) which conforms to the target metamodel.

The language has been designed as a rule-based transformation language with rules whose structure is similar to those provided in *m2m* transformation languages such as ATL or RubyTL, with two important differences: i) the source element of a rule is a grammar element rather than a metamodel element, and ii) the navigation through the source code

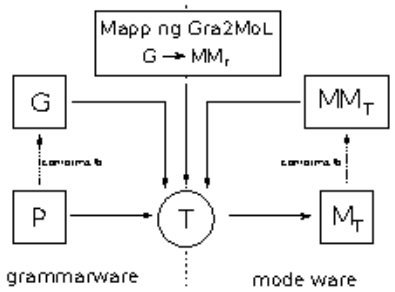


Figura 1.2: The process applied by a Gra2MoL grammar-to-metamodel transformation.

is expressed by a specially tailored query language rather than an OCL-based language.

A Gra2MoL transformation definition therefore consists of a set of transformation rules. Each rule specifies the mappings between a grammar element and a target metamodel element and is composed of four parts:

- The *from* part specifies a grammar non-terminal symbol, and declares a variable that will be bound to a node of the source code syntax tree when the rule is applied. This variable can be used by any expression within the rule. The *from* part can also include query operations (i.e., a filter) to check the structure to be satisfied by the nodes whose type is the non-terminal symbol.
- The *to* part specifies the target element metaclass.
- The *queries* part contains a set of query expressions which allow information to be retrieved from the source code. The result of these queries will be used in the assignments of the mappings part.
- Finally, the *mappings* part contains a set of bindings to assign a value to the properties of the target element.

The execution of a Gra2MoL transformation definition is driven by the bindings of the *mappings* part. Gra2MoL bindings have very similar syntax and semantics to those used in ATL and RubyTL languages but, in this case, the right-hand side can be the variable specified in the *from* part of the rule, a literal value or a query identifier. On the other hand, model extraction from source code requires an intensive use of queries to retrieve scattered information which is necessary to build the target metamodel elements. Gra2MoL therefore incorporates a structure-shy query language inspired by XPath which allows syntax trees to be traversed efficiently in order to obtain this information. A more detailed explanation about the query language and rule evaluation can be found in [B1].

Figure 1.3a shows an example of a Gra2MoL transformation process used in a case study involving PL/SQL code. The elements used in the example have been simplified for the sake of space. The inputs of the Gra2MoL engine are: (1) an excerpt of the PL/SQL grammar including the grammar rules used to define procedures, which can contain a set of statements (i.e., if or while statements); (2) the program conforming to the grammar; (3) an excerpt of the PL/SQL target metamodel, which allows procedures (`Procedure` metaclass) and their statements (`Statement` hierarchy) to be represented; and (4) the

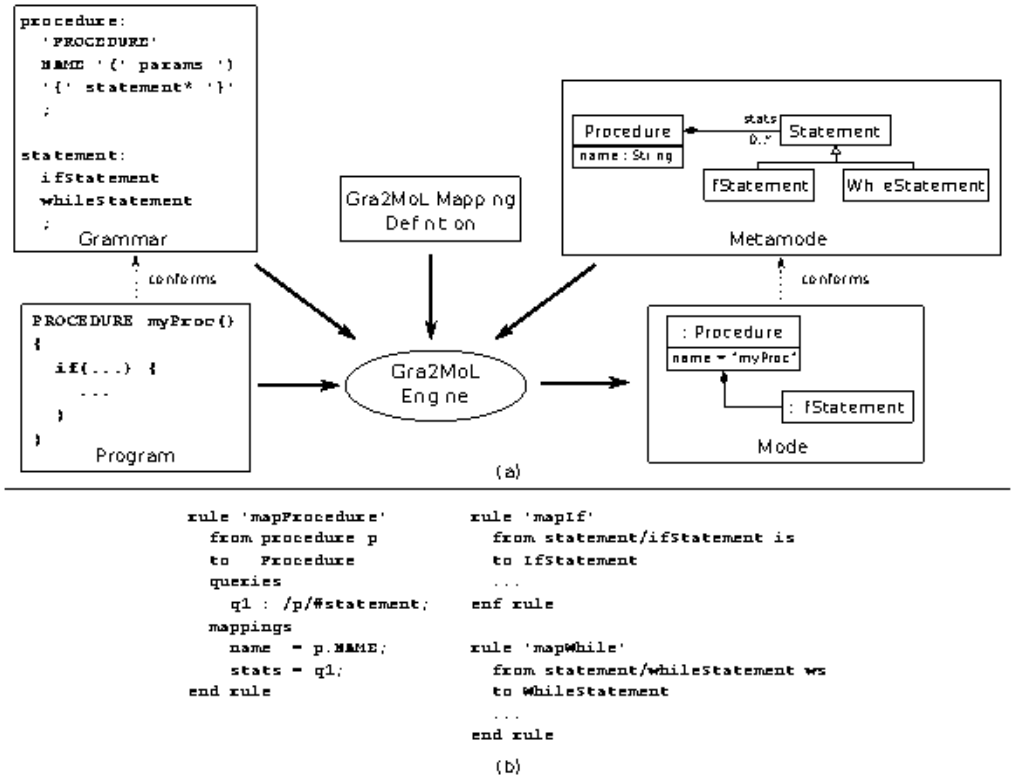


Figura 1.3: Simple example of a Gra2MoL transformation process. (a) The inputs and outputs of the process and (b) the Gra2MoL mapping definition used in this example.

Gra2MoL mapping definition, which is listed in 1.3b and is explained below. The result of the Gra2MoL transformation process is a PL/SQL model conforming to the target metamodel.

The Gra2MoL mapping definition used in this example is composed of three rules. The first rule, called `mapProcedure` starts the transformation process and creates an instance of the `Procedure` metaclass from the `procedure` grammar element. Whereas the first mapping of this rule assigns a value to the `name` attribute accessing the `NAME` terminal element of the grammar, the second mapping is a binding whose right-hand side is a query identifier and whose left-hand side refers to the `stats` reference of the `Procedure` metaclass. The query is therefore executed and rules conforming the binding are then looked up and executed. In this case, both the `mapIf` and `mapWhile` rules conform to the binding since the result of the query are `statement` grammar elements, but only the *from* filter of the `mapIf` rule is satisfied and this rule is therefore executed to create an instance of the `IfStatement` metaclass.

In the past two years, we have successfully applied Gra2MoL to obtain models from the source code of several programming languages. New features have therefore been incorpo-

rated into the Gra2MoL language to improve its efficiency, extensibility and adaptability to the model extraction problem. Some of these features are the following:

- **Skip rules** for dealing with expressions. Grammar rules for describing expression languages have a particular structure and this new type of rule helps developers to define effectively transformation rules for extracting models which represent such expressions.
- **Support for island grammars**. The island grammar mechanism is used when a language contains one or more sublanguages (e.g., the Javadoc language in Java). Island grammars thus allow defining complex languages which are composed of languages which conform to different grammars.
- **Mixin rules** for factorizing the queries and mappings of the rules. In large transformation definitions, there are normally several rules which share some queries or mappings. In these cases, mixin rules allow us to factorize the transformation code in order to minimize the scattering.
- **Extension mechanism**, which allows incorporating new operations to both the query language and mappings section.
- **CDO support** for storing large models in databases.

The approach was first presented at the MoDSE workshop [C1], and the language in ECMDA'09 as a paper in the Foundations track [B1]. Several case studies have been developed in Gra2MoL, which can be downloaded from `http://modelum.es/gra2mol`. It is important to note that we have used Gra2MoL as part of a process for extracting ADM models [A1], which was one of the first case studies of ADM. The language was also used in an SMM engine to calculate metrics, which was presented in the Spanish MDE workshop [D1]. It was also used to extract models from maintainer scripts in the Mancoosi project [15]. We have additionally implemented a Gra2MoL development environment to facilitate the definition and use of gramar-to-metamodel transformations [C2], and have also prepared a new paper including the new features of Gra2MoL, which has been sent to the Model Evolution special issue of the Software and Systems Modeling journal [A2].

1.4.2. ScheMoL

ScheMoL is deeply inspired by Gra2MoL, although it allows defining bridges from *dataware* to *modelware*. The transformation process involved in a ScheMoL transformation is very similar to the process shown in Figure 1.2, but connected to a database in order to retrieve data conforming to the database schema, rather than receiving the grammar and the program. Like Gra2MoL, ScheMoL is a rule-based transformation language and also uses the concept of binding, which was adapted to deal with tuples. Moreover, ScheMoL incorporates a specially adapted query language to collect information from databases.

A ScheMoL transformation is composed of a set of transformation rules and, optionally, a preamble. A ScheMoL rule specifies the mappings between a table of the source database schema and a metaclass of the target metamodel. The structure of ScheMoL rules is very similar to that defined in Gra2MoL but with a few variations. ScheMoL rules are thus composed of four parts:

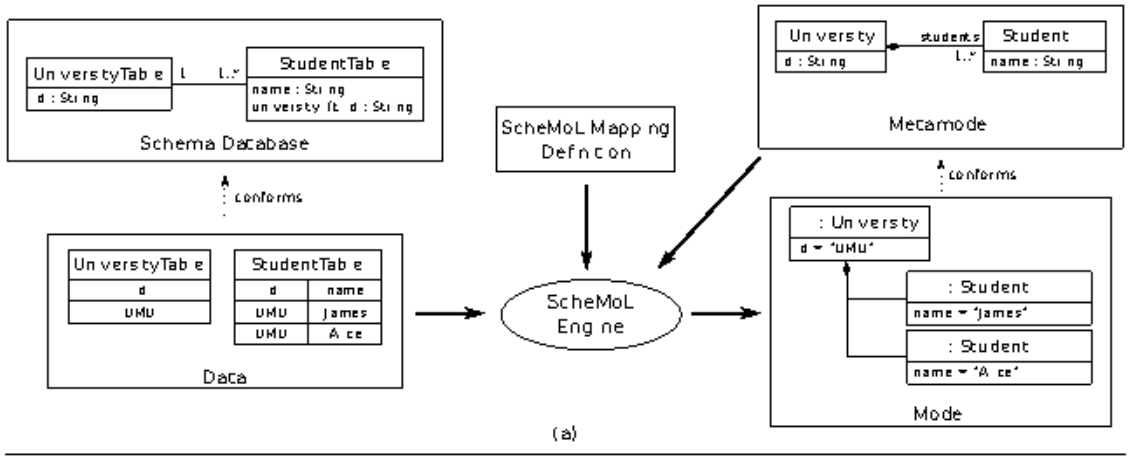
- The *from* part, which specifies the source table together with a variable that will hold the tuple of this table at the time the rule is applied.
- The *to* part, which specifies the target element metaclass along with a variable to hold the instance being generated at enactment time.
- The *filter* part, which is optional and includes a condition expression over the source element, such that the rule will only be triggered if the condition is satisfied.
- The *mapping* part, which contains a set of bindings to set the properties of the target element.

The preamble of a transformation definition allows developers to specify ad hoc foreign keys and views. Since some database systems do not store foreign keys (e.g., MySQL for the sake of efficiency), they can be specified to know the links between database tables, which is required to apply the ScheMoL query language. On the other hand, views facilitate the definition of mapping definitions because they allow the database schema to be described in terms closer to those needed for the transformation without polluting the database schema.

Like Gra2MoL, the execution of a ScheMoL transformation definition is also driven by the bindings of the *mapping* part. However, ScheMoL bindings differ slightly from those used in Gra2MoL. In ScheMoL, the left-hand side of a binding must also be an attribute of the target element metaclass but the right-hand side can be a literal value, a query or an expression, where both queries and expressions deal with tuples. Moreover, ScheMoL incorporates a query language which allows developers to traverse the database tuples transparently without the need of defining complex SQL queries. A more detailed explanation about the query language and rule evaluation can be found in [A3], in which the language is described.

Figure 1.4a shows a simplified example of a ScheMoL transformation process used to extract models from a database storing student data. The inputs of the process are: (1) the database schema defining the `UniversityTable` and `StudentTable` tables, where `StudentTable` has a foreign key to `UniversityTable` (i.e., `university_fk_id` column); (2) data conforming to this database schema; (3) a simple target metamodel to represent universities (`University` metaclass) and their students (`Student` metaclass); (4) and the ScheMoL mapping definition, which is explained below. The result of the ScheMoL transformation process is a model conforming to the target metamodel.

The ScheMoL mapping definition used in this example is composed of two rules, which are listed in Figure 1.4b. The first rule, called `mapUniversity`, starts the transformation process and creates an instance of `University` metaclass from the only tuple of `UniversityTable` table in the example. The first mapping initializes the `id` attribute by accessing the `id` column of the `UniversityTable` tuple. The second mapping is a binding whose right-hand side is a query which collects every tuple in `StudentTable` referring to the current `UniversityTable` tuple and whose left-hand side refers to the `students` reference of the `University` metaclass. Since the `StudentTable` contains two tuples, there are two query results and the binding causes execute the `mapStudent` rule twice, which creates two instances of a `Student` metaclass and initializes the `name` attribute by accessing the `name` column of the `StudentTable` tuple received by the rule.




```
rule 'mapUniversity'
  from UniversityTable uTab
  to   University uni
  mappings
    uni.id = uTab.id;
    uni.students = uTab.@StudentTable;
end rule
```

```
rule 'mapStudent'
  from StudentTable sTab
  to   Student st
  mappings
    st.name = sTab.name;
end rule
```

(b)

Figura 1.4: Simple example of a ScheMoL transformation process. (a) The inputs and outputs of the process and (b) the ScheMoL mapping definition used in this example.

The language was extensively tested by the Onekin research group in several case studies whose objective was to extract models from Web 2.0 data stored in relational databases. The development of these case studies allowed us to devise an extension mechanism in order to permit new operators to be defined in the *mappings* section. Thus, new operators were defined as language extensions in order to support the extraction of models from Web 2.0 data (e.g., support for annotated data).

1.4.3. API2MoL

API2MoL allows defining bridges between *apiware* and *modelware*. The transformation process shown in Figure 1.2 can be adapted to API2MoL by changing the program input element for a set of objects conforming to an object-oriented API. However, this language differs from Gra2MoL and ScheMoL. API2MoL is also a rule-based transformation language but its rules are bidirectional. The API2MoL language is therefore used to define bidirectional mappings between the API classes and metamodel elements. Moreover, it is not necessary to define a query language in order to specify traverses through the API since the rules define which methods have to be called to access and modify the API objects. The current implementation focuses on Java API, although its adaptation to other statically typed object-oriented languages should be straightforward.

An API2MoL transformation definition consists of a set of transformation rules in which

each rule is responsible for mapping a metamodel element (i.e., metaclass) and an API class, including the mappings between the metaclass properties and the API methods to be invoked when reading/writing those properties. API2MoL rules are composed of a header and a set of sections. The header specifies the Java class and the metaclass involved in the mapping, whereas the sections define how the mapping is applied, that is, how the methods from the Java class indicated in the header must be invoked when reading/writing the metaclass properties. There are five types of sections:

- Property section, which specifies a bidirectional mapping between a metaclass feature (i.e., attribute or reference) and an API class feature. Each property section specifies the name of the metamodel feature and a set of statements that describe the kind of access (e.g., get and set access) provided by the API to read/write that specific feature, along with the specific names of the methods that implement that access. It is usually possible to skip specifying the names of the methods since API2MoL can infer them from the statement type.
- Default section, which indicates the name of the metaclass to be used when there is no rule for an object and it is a subclass of the class specified in the rule.
- Multiple section, which is used in the extraction process when the API defines methods that deal with more than one feature at the same time.
- Constructor section, which is used to set the constructor to create an instance of the class specified in the rule. This section is defined when the default constructor is not available.
- Value section, which defines a mapping between a metamodel enumeration type and an enumeration type in the programming language.

A mapping definition is therefore used to both obtain of models from API artifacts and the generate API artifacts from models. Models are obtained by executing an API2MoL definition over the API objects used by the program, whereas in a generation process an API2MoL definition is used to determine the API objects to be generated from the model elements. A more detailed explanation of the transformation execution can be found in [A3], in which the language is described.

Figure 1.5a shows an example of an API2MoL transformation process which is applied to extract models from a very simple Swing Application. The inputs of the process are: (1) the set of runtime objects conforming to the Swing API, which represents a simple application composed of a `JPanel` with two `JButtons`; (2) a simple target metamodel to represent panels (`Panel` metaclass) composed of buttons (`Button` metaclass); and (3) the API2MoL mapping definition, which is shown in Figure 1.5b and is explained below. The result of the API2MoL model extraction process is a model conforming to the metamodel.

The API2MoL mapping definition contains two rules. The API2MoL engine receives an instance of the `JPanel` class and the rule which defines the mapping between the `Panel` metaclass and the `javax.swing.JPanel` Java class is therefore executed. This rule creates an instance of the `Panel` metaclass and the `buttons` reference is then initialized by using the method specified in the `get` statement of the `buttons` property section of the rule (i.e., the `getComponent` method). Since this method returns instances of the

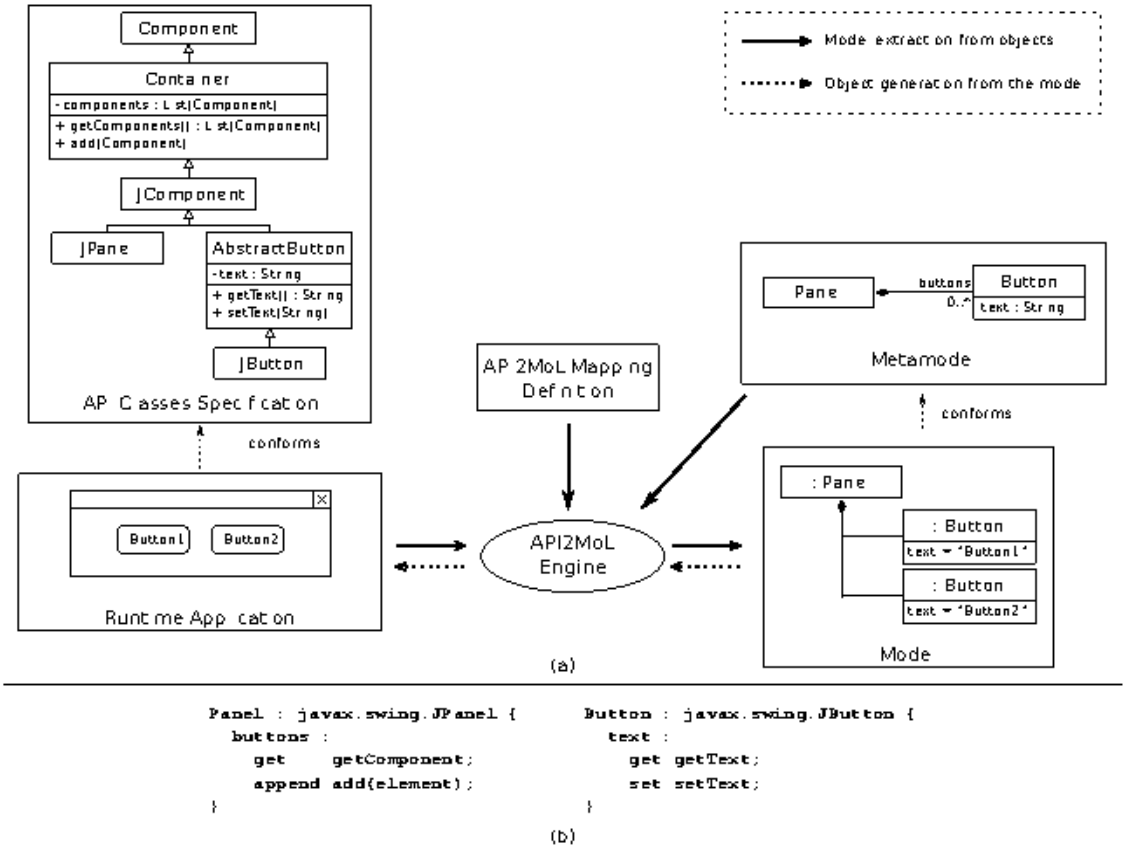


Figura 1.5: Simple example of a API2MoL transformation process. (a) The inputs and outputs of the processes and (b) the API2MoL mapping definition used in this example.

`JButton` Swing class, the rule defining the mapping between the `Button` metaclass and the `javax.swing.JButton` Java class is executed. This rule creates an instance of the `Button` metaclass and the text attribute is then initialized by using the method specified in the `get` statement of the `text` property section (i.e. the `getText` method).

It is important to note that the mapping definition can also be used to apply a generation process and create the objects from a model. In this case, the API2MoL engine would receive an instance of the `Panel` metaclass and would execute the first rule to create an instance of the `JPanel` Java class. The initilization of the `buttons` attribute of `JPanel` would be carried out by calling the method specified in the `append` statement of the `button` property section (i.e., the `add` method). This method would receive the instances of the `JButton` Java class as parameters, which would in turn have to be created by the second rule. The second rule would create an instance of the `JButton` Java class and would initialize the `text` attribute by using the method specified in the `set` statement of the `text` property section (i.e., the `setText` method).

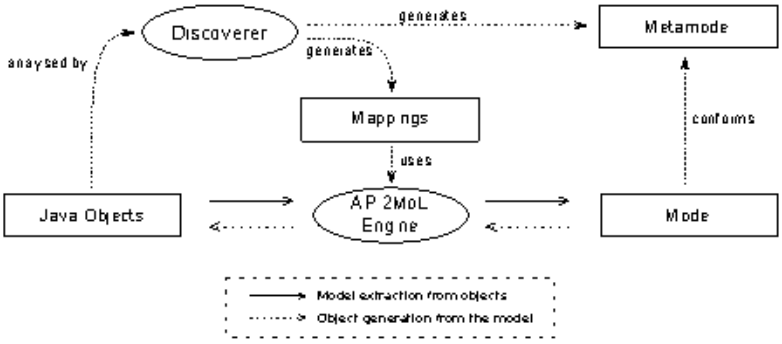


Figura 1.6: The process of discovering both the metamodel and the mapping definition in API2MoL.

Since API have normally a large number of classes, the effort required to define a bridge with API2MoL would not be significantly reduced since the creation of the metamodel and the mapping definition would be tedious and time-consuming. An essential aspect of the API2MoL approach is therefore the application of a discovery process to automatically generate both the target metamodel and the mapping definition from the API classes, as shown in Figure 1.6.

The discovery process deals with API objects at runtime and analyses them by reflection. Since these reflection capabilities are normally provided through a Reflection API, we have created an API2MoL mapping definition that obtains a reflection model describing the API classes. This reflection model can therefore be used to automatically generate both the metamodel and the mapping definition for an API by means of *m2m* transformations. We call this discovery process a "bootstrap process", because we use API2MoL in order to discover the metamodel and mapping definition needed to apply API2MoL itself to an API. A more detailed explanation of this process can be found in [A4].

The API2MoL language is already implemented and we are in the process of publishing the first results in the Information and Software Technology journal [A4]. We have recently sent a second version of the paper since it has undergone a major review.

1.5. Conclusions

The emergence of model driven engineering signifies that the need to represent software artefacts as models is gaining importance. Providing tools and techniques to build bridges towards the *modelware* allows developers to perform the reverse engineering tasks in a model-driven reengineering process. However, these bridges are currently developed by means of creating ad hoc solutions that deal with a specific type of artefacts (e.g., the source code of a concrete programming language), which is an expensive and time-consuming task. Bearing this problem in mind, this thesis presents the Gra2MoL and SchcMoL languages, which allow developers to define bridges from the *grammarware* and the *dataware*, respectively, to the *modelware*. Moreover, the use of *m2t* transformations along with *t2m*

transformations offers a complete solution to the integration of the *modelware* into any TS. Models can thus be used as pivot artefacts in order to integrate several TSs, which allows advantage to be taken of *modelware* techniques such as model transformation and model comparison. In this context, this thesis also presents API2MoL, which allows models to be extracted from API objets and the opposite process to be applied, that is, generating API objects from model elements, thus facilitating the integration between the *apiware* and the *modelware*.

The languages presented in this thesis provide the benefits of using DSLs such as productivity and quality improvements along with a better maintainability of the transformation definitions. Gra2MoL and SchcMoL become an alternative to existing ad hoc solutions whereas API2MoL is a novel approach which is, to the best of our knowledge, the first generic approach to provide complete integration between the *apiware* and the *modelware*.

The TSs considered in this thesis cover the vast majority of the artefacts of which a software system is composed, with the exception of XML documents, which are extensively used when creating software system to represent metadata (e.g, configuration files). The XML TS, called *xmlware*, was not considered because tools to create bridges between the *xmlware* and the *modelware* already exist (i.e., those tools provided by the Eclipse platform [63]). The ontology and web services technologies are examples of TSs which were not considered in this thesis. However, as future work we have identified the creation of a general language to facilitate the definition of bridges between any TS and the *modelware*, as described in Section 1.6.

The initial objective of this thesis was to deal solely with *grammarware-modelware* bridges, and we therefore created Gra2MoL to extract models from source code, as explained in 1.3. We later focused on other TSs (i.e., *dataware* and *apiware*), which led to our the collaboration with the ONEKIN and AtlanMod research groups, and we then decided to create SchcMoL and API2MoL to extract models from relational data and API objets, respectively. Gra2MoL, SchcMoL and API2MoL can thus be considered as a language family for the model extraction domain. A language family has a set of common and variable elements. The languages developed share the use of the four-level OMG architecture to define bridges between TSs. The structure of the languages is thus based on rules which define the mappings between the elements of each TS. A high level of reusability therefore exists, although to a lesser extent in API2MoL since the requirements were not similar, as is explained in 1.3. For example, SchcMoL is deeply inspired by Gra2MoL, in that both design concepts (e.g., rule types and *binding* mechanism) and implementation components (e.g., model manager) are reused.

On the other hand, the variable element of these languages is the use of a query language. Whereas Gra2MoL and SchcMoL provide query languages which have been specially adapted to traverse abstract syntax trees representing source code and database tables, respectively, API2MoL does not incorporate any query language because its rules only have to specify which method must be used to perform the transformation. The Gra2MoL query language is inspired by XPath and provides a *structure-shy* language which facilitates the traversal of syntax trees. The language therefore incorporates the operators // and ///, which allow us to specify what must be found but not how to reach it, thus facilitating

the legibility and concision of queries. On the other hand, the ScheMoL query language facilitates the traversal of database tables by using dot notation and incorporating special operators (i.e., direct and inverse operators).

In both Gra2MoL and ScheMoL, the *binding* mechanism is suited to the problem of extracting models from source code and databases, respectively. From our experience in developing Gra2MoL and ScheMoL transformations, the declarativeness provided by the *bindings* facilitates the definition and comprehension of the transformations.

Since DSL definition tools such as Xtext were not sufficiently mature (i.e., the abstract syntax metamodel generated was of poor quality) when developing Gra2MoL, we decided to use Gra2MoL itself to implement the language. ScheMoL and API2MoL were later implemented by using Gra2MoL as well, which allowed us to implement these languages easily. Gra2MoL therefore facilitated us to extract models conforming to the abstract syntax from the textual transformation definitions conforming to the grammar of the language. It also allowed us to experiment with the definition of external DSLs. However, since DSL definition tools were not used, the supporting tools (e.g., language editor or execution tools) for each language had to be implemented, as occurred with Gra2MoL and is currently taking place in the development of ScheMoL and API2MoL. DSL definition tools have now evolved considerably, and would therefore be used if it were necessary to define a new DSL.

Gra2MoL has been applied to several application scenarios such as Java, Delphi, PL/SQL, Maude, IDL, EBNF specifications, ANTLR grammars and bash scripts. The development of the transformations involved allowed us to improve the language capabilities and to study their suitability for the problem of extracting models from source code.

Gra2MoL was also used in a case study in the context of the ADM initiative. In particular, we defined a process to extract KDM models and calculate some metrics. This experience allowed us to study the ASTM and KDM metamodels of the ADM initiative. These metamodels provide a standard representation of the vast majority of artefacts of which a software system is composed, thus promoting interoperability between modernization tools. However, when representing software artefacts in a precise manner (e.g., each code sentence) it is necessary to use the extension mechanism provided by the metamodels (e.g., SASTM metamodels in ASTM and stereotypes in KDM), which hampers interoperability. Furthermore, there is an important lack of examples, which makes the study and use of these metamodels difficult. It would therefore be more interesting to use models conforming to proprietary metamodels and convert them into models conforming to ADM metamodels in the case of interoperating with other modernization tools. However, it is also worth mentioning that there are not many tools supporting these metamodels.

The structure and operation of API2MoL differ to those of Gra2MoL and ScheMoL. API2MoL supports bidirectional transformations between *apiware* and *modelware*, which allows a complete integration between these TSs. The rule structure was therefore adapted to support bidirectionality. Moreover, the language does not provide a query language, as explained previously. One important feature of API2MoL is the *bootstrap* process, which allows automatically generating a bridge between a concrete pair of <API-metamodel>.

The languages presented in this thesis were validated with several case studies, which

allowed us to identify some weak points and to devise solutions. Moreover, in API2MoL we defined a validation process to verify the language correction and the coverage of both the language and the *bootstrap* process. In this validation process we concluded that, for some APIs, the generation may not be complete since they may present certain particularities that need a special treatment. Nevertheless, the small percentage of these situations makes always worthy the application of our *bootstrap* process to kick-start the process.

In general, the languages presented in this thesis provide developers with the mechanisms needed to integrate different software artefacts with model-based solutions. We thus believe that these languages may become a further step in facilitating the adoption of MDE.

1.6. Future work

We have identified some future work which can be classified according to either the developed language or the scope.

Gra2MoL

The language currently supports the management of syntax trees representing the source code by means of either an in-memory mechanism or using a CDO repository, which allows large syntax trees to be dealt with. Since the query execution relies on the way in which the syntax tree is managed, we are interested in studying the performance and scalability impact of each one. Moreover, with regard to the scalability, the extracted models are usually huge and it would also be interesting to study mechanisms which would allow such models to be managed in an effective manner (e.g., by using a CDO repository for the extracted models).

We are also interested in studying mechanisms to promote the modularity and composability of transformations, such as a phasing mechanism similar to that incorporated into RubyTL. This phasing mechanism would additionally allow us to study a trace query mechanism in order to improve the control of the transformation execution flow.

Finally, we plan to support other parser generators in order to increase the number of existing grammars that can be reused.

ScheMoL

In order to improve the ability to adapt the language to specific data extraction problems, we are interested in incorporating an extension mechanism similar to that used in Gra2MoL, thus allowing developers to improve language capabilities by defining new query operators.

With regard to scalability issues, like Gra2MoL, ScheMoL transformations can extract huge models from databases, and it would also be interesting to incorporate mechanisms to manage such models (e.g., using CDO repositories to store the extracted models).

The language can access MySQL databases and a driver to Firebird is under development. We also plan to support Oracle databases since they are extensively used by industry.

Finally, unlike Gra2MoL, ScheMoL does not include any integrated development environment to manage and execute transformations. We are therefore working on building an Eclipse *plugin* to provide developers with an environment that has been specially adapted to ScheMoL.

API2MoL

The APIMoL engine currently supports *batch* execution, that is, both the extraction and generation processes are performed in one step to API objects and model elements, respectively. Other execution modes could be implemented in order to support incremental extraction/generation. For example, when reengineering GUIs, it would be interesting to use a batch extractor and an incremental generator. A model would thus be initially extracted from every GUI API object and model changes would then be generated incrementally. Another example appears when managing models *at runtime*, which would require both an incremental extractor and a generator. The changes of both the objects representing the system and the model elements would be thus extracted/generated incrementally.

To improve the current implementation strategy (which deals solely with in-memory runtime objects) we plan to support the *Java Debug Interface* (JDI) in order to be able to access in-memory objects which are not directly accessible via reflection because they are not in the same Java virtual machine as API2MoL (e.g., objects of a J2EE deployed application are only accessible by communicating with the application server in which they are deployed). Moreover, since API2MoL only supports Java-based APIs, supporting other programming languages such as C# would also be interesting, and would thus allow us to study how to make the API2MoL engine language-independent.

We are also interested in extending API2MoL to cover non object-oriented APIs, such as web services descriptions or *mashups*. In this respect, we are particularly interested in using API2MoL to facilitate the interaction at the model level between applications and external web services (which could be regarded as a type of external APIs) and to facilitate both the extraction and generation processes from the data provided by these services.

Like ScheMoL, API2MoL requires an IDE to facilitate the definition, management and execution of API2MoL transformations an to configure the *bootstrap* process. Moreover, we are currently studying the need for an extension mechanism to define new statement types, which will allow developers to adapt API2MoL to new API functionalities.

Finally, we also plan to use API2MoL, ScheMoL and Gra2MoL in different scenarios in order to validate the languages and illustrate their applicability.

General language for extracting models

From our experience in developing the three DSLs presented in this thesis, we plan to define a general DSL to create any bridge towards the *modelware*, which would reuse the knowledge of the DSLs created and would allow us to investigate the concept of DSL families. This new language would allow defining mappings between the elements of the technical space considered and the *modelware*.

The development of this new DSL would require us to study the suitability of using rules to define mappings between the elements involved. Certain Gra2MoL and ScheMoL mechanisms such as the use of *bindings* or the need for a query language would also be studied. On the other hand, it would also be interesting to study whether the new language should be bidirectional like API2MoL, which will depend on the domain involved.

The engine of the new DSL would be parameterized by (1) the source element type and (2) the query language, if needed. The engine would therefore provide mechanisms to manage the source elements in order to execute *bindings*. It would also be necessary to study language composition mechanisms in order to incorporate the query language which would be specially adapted to the source domain.

Experimenting on ADM

The ADM initiative is still under development and new metamodels are coming. We plan to continue our research in this area in order to experiment on ADM in real case studies. In this thesis, we have worked with ASTM and KDM, and some future work has been identified. With regard to ASTM, we are interested in studying reusable *m2m* tranformations, since the GASTM metamodel contains elements that are common to any GPL. We also plan to use the *resources* and *abstraction* layers of KDM, which will allow us to define some architectural views such as GUI or business processes.

Finally, the IPMSS metamodel, which allows code patterns to be represented, is currently in the final stage of publication. As further work, we are interested in defining Gra2MoL transformations to extract models conforming to IPMSS from the source, and study whether the use of ASTM models as intermediate representation could facilitate the process.

1.7. Research group and research stays

This thesis has principally been developed in the Modelum Research Group at the University of Murcia. The work has additionally been complemented with two collaborations with research groups at other universities:

- A long-term visiting scholarship to the AtlanMod team (Department of Computer Science, École des Mines de Nantes and INRIA), from April 11th 2010 to July 18th 2010, under the supervision of Dr. Jordi Cabot. The candidate worked on the implementation of API2MoL, a new DSL for bridging *apiware* and *modelware*.

- Two short-term visits to the Onekin research group under the supervision of Dr. Óscar Díaz. The former from September 11th 2009 to September 18th 2009 and the latter from September 19th 2010 to September 24th 2010. The candidate principally worked on the design of ScheMoL, a new DSL for bridging *dataware* and *modelware*.

1.8. Candidate's publications

JCRs indexed journals

[A1] J. L. Cánovas and J. García Molina. An Architecture-Driven Modernization Tool for Calculating Metrics. *IEEE Software*, 27:37–43, 2010.

[A2] J. L. Cánovas and J. García Molina. Extracting Models from Source Code in Software Modernization. In *Software and System Modeling, Model Evolution Issue*, 2011. Under reviewing process.

[A3] O. Díaz, G. Puente, J. L. Cánovas, and J. García Molina. Harvesting Models from Web 2.0 Databases. *Software and System Modeling*, 11, 2011.

[A4] J. L. Cánovas, F. Jouault, J. Cabot, and J. García Molina. API2MoL: Automating the Building of Bridges between APIs and Model-Driven Engineering. In *Journal on Information and Software Technology*, 2011. Under reviewing process, major revision.

LNCS proceedings

[B1] J. L. Cánovas and J. García Molina. A Domain Specific Language for Extracting Models in Software Modernization. In *European Conference on Model Driven Architecture - Foundations and Applications*, volume 5562, pages 82–97, 2009.

International conferences/workshops

[C1] J. L. Cánovas, J. Sánchez Cuadrado, and J. García Molina. Gra2MoL: A Domain Specific Transformation Language for Bridging Grammarware to Modelware in Software Modernization. In *Model Driven Software Evolution workshop*, 2008.

[C2] J. L. Cánovas and J. García Molina. Gra2MoL put into Practice. In *Tools and Consultancy Track in European Conference on Modelling Foundations and Applications*, 2010.

Spanish conferences/workshops

[D1] J. L. Cánovas, B. Cruz, and J. García Molina. Definición y Ejecución de Métricas en el Contexto de ADM. In *Taller sobre Desarrollo de Software Dirigido por Modelos*, pages 136–145, 2010.

[D2] J. L. Cánovas and J. García Molina. Una aplicación práctica de Architecture-Driven Modernization (ADM). In *Tutorial en Jornadas de Ingeniería del Software y Bases de Datos*, 339-339, 2010.

[D3] J. L. Cánovas and J. García Molina. Gra2MoL: Una Herramienta para la Extracción de Modelos en Modernization de Software. In *Demonstración en XIV Jornadas de Ingeniería del Software y Bases de Datos*, pages 162–165, 2009.

[D4] J. L. Cánovas and J. García Molina. Extracción de Modelos en una Modernización Basada en ADM. In *Taller sobre Desarrollo de Software Dirigido por Modelos*, pages 41–50, 2009.

[D5] J. L. Cánovas, O. Sánchez Ramón, J. Sánchez Cuadrado, and J. García Molina. DSLs para la Extracción de Modelos en Modernización. In *Taller sobre Desarrollo de Software Dirigido por Modelos*, pages 1–10, 2008.

[D6] J. L. Cánovas, O. Sánchez Ramón, J. Sánchez Cuadrado, and J. García Molina. Utilidad de las Transformaciones Modelo-Modelo en la Generación de Código. In *Jornadas de Ingeniería del Software y Bases de Datos*, pages 31–40, 2007.

1.9. Tools developed

The languages, case studies and tools developed in this thesis can be downloaded from the following websites::

- **Gra2MoL**. `http://modelum.es/gra2mol`
  The Gra2MoL website includes the resources needed to develop and execute Gra2MoL transformations (i.e., source code, examples, case studies, documentation and Eclipse plugin). The ADM-based case studies can also be downloaded from this site. Moreover, the tool has recently been licensed as EPL.
- **ScheMoL**. `http://modelum.es/schemol`
  The ScheMoL website includes the case studies developed, along with the source code of the DSL.
- **API2MoL**. `http://modelum.es/api2mol`
  The API2MoL website includes the examples used in the validation process, along with the source code of the DSL.

Acknowledgment

Javier Luis Cánovas Izquierdo has obtained a doctoral grant from the Fundación Séneca.

2

Introducción

Para Gaal, aquel viaje era la cúspide de su trayectoria como joven investigador.
La Fundación, Isaac Asimov

Desde comienzos de la pasada década, el interés por el Desarrollo de Software Dirigido por Modelos (DSDM) ha crecido de forma considerable debido a que los modelos han mostrado su valor para dominar la complejidad del software al permitir elevar el nivel de abstracción y automatización en las distintas etapas del ciclo de vida del software. Aunque hasta ahora la mayoría de soluciones basadas en el DSDM han estado centradas en la construcción de nuevos sistemas software, las técnicas de este paradigma también han evidenciado su utilidad en otras aplicaciones, entre las que destaca la reingeniería o modernización de software *legacy*, el control y adaptación del funcionamiento de un sistema en tiempo de ejecución y el uso de modelos como una representación intermedia de alto nivel con el propósito de facilitar la integración de diferentes sistemas. Estas otras aplicaciones requieren normalmente la extracción de modelos a partir de artefactos software de diferente naturaleza, como ficheros de código fuente, ficheros de configuración XML, objetos de un API o datos de una base de datos.

La obtención de modelos a partir de los artefactos origen es un ejemplo de la necesidad de establecer un puente entre dos espacios tecnológicos [49]. Un *espacio tecnológico* define un contexto de trabajo proporcionado por una determinada tecnología, como es el caso del espacio tecnológico del DSDM (*modelware*) o del espacio tecnológico de los lenguajes de programación (*grammarware*). Un puente entre dos espacios tecnológicos tiene como propósito favorecer la interoperabilidad y utilizar las técnicas y herramientas del espacio tecnológico destino sobre artefactos del espacio tecnológico origen. En el caso de la extracción de modelos, una vez obtenidos los modelos es posible aprovechar las técnicas del DSDM como transformaciones de modelos y comparación de modelos.

En el trabajo presentado en esta tesis se ha estudiado el problema de cómo facilitar la extracción de modelos desde tres diferentes espacios tecnológicos que cubren la mayor

parte de los artefactos que componen un sistema software y que son: *grammarware*, que incluye código fuente expresado en un lenguaje de programación conforme a una gramática; *dataware*, que incluye los datos que conforman con un esquema de base de datos; y el *apiware*, que incluye a los objetos accesibles mediante un API. Se han definido lenguajes específicos de dominio para la extracción de modelos desde cada uno de estos tres espacios tecnológicos, con el objetivo de facilitar la creación de puentes entre cada espacio tecnológico considerado y el *modelware*. Estos lenguages son Gra2MoL (*grammarware-modelware*), SchcMoL (*dataware-modelware*) y API2MoL (*apiware-modelware*). Además, el trabajo con Gra2MoL ha supuesto arrancar una investigación en un área de reciente aparición como la modernización basada en modelos, en particular, en la experimentación con los metamodelos básicos de la iniciativa ADM (*Architecture-Driven Modernization*) de OMG.

El resto del capítulo está organizado en cinco apartados. En primer lugar se presentará la motivación del trabajo. A continuación se enumerarán los principales objetivos de esta tesis luego la metodología seguida y los resultados obtenidos. Finalmente se describirá la organización del resto del documento.

2.1. Motivación

En los últimos años, el paradigma del DSDM se ha consolidado como una de las alternativas más destacables para alcanzar la industrialización del software gracias a su capacidad para elevar el nivel de abstracción al programar y a la mejora en la productividad y calidad del software que conlleva una mayor automatización [33, 89]. En DSDM, el desarrollador no necesita escribir todo el código de una aplicación en un lenguaje de programación de alto nivel sino que crea modelos de alto nivel de abstracción mediante *lenguajes específicos del dominio* (*Domain Specific Languages*, DSL) cuyos conceptos y construcciones son cercanos al dominio del problema. A partir de estos modelos es posible generar automáticamente el código de la aplicación mediante la aplicación de *transformaciones de modelos*. De hecho, el impacto que tendrá el DSDM se suele comparar al que tuvieron los lenguajes de programación de propósito general (*General Programming Language*, GPL) y compiladores en los años sesenta [33, 91], en los que se pasó de escribir código ensamblador a programar con lenguajes más abstractos y los programas traductores se encargaban de transformar el programa de alto nivel en código máquina. Las ventajas de crear modelos con DSL son, por tanto, similares a las ofrecidas por los GPL con respecto a los lenguajes ensambladores: mayor productividad, mejora de la calidad y mejor mantenimiento. Cabe destacar que un DSL no es un GPL ya que su ámbito de aplicación se circunscribe al dominio asociado. El uso de los DSL no es algo nuevo sino que se remonta a los primeros años de la programación, pero la creciente aceptación del DSDM, así como la disponibilidad de herramientas que facilitan su creación, ha incrementado el interés por ellos. Los términos lenguajes de modelado y lenguajes de modelado específicos del dominio (*Domain-Specific Modeling Languages*, DSML) son también usados para referenciar a los lenguajes usados para crear modelos, aunque en esta tesis utilizaremos el término lenguajes específicos del dominio (DSL) por considerar que es más extendido.

Las técnicas del DSDM son aplicadas para automatizar determinadas tareas en procesos

de ingeniería directa o reingeniería [9]. En ingeniería directa, los desarrolladores definen inicialmente modelos que luego son transformados para generar el código de los artefactos correspondientes. Por otro lado, en un proceso de reingeniería o modernización, los modelos iniciales son obtenidos a partir de los artefactos del sistema software existente en la fase de ingeniería inversa y, a continuación, son transformados para generar los artefactos del nuevo sistema. Esta fase de ingeniería inversa se organiza en dos etapas: (1) extracción de modelos de bajo nivel a partir de los artefactos del sistema y (2) transformación de los modelos extraídos en otros de alto nivel de abstracción, los cuales son utilizados en el resto del proceso de modernización. Por tanto, el paso de extracción de modelos es crucial en una modernización basada en modelos ya que proporciona los modelos iniciales que representan al sistema existente y son utilizados en las fases siguientes del proceso.

Existe una tercera aplicación del DSDM llamada modelos *runtime* donde los modelos permanecen siempre sincronizados con el artefacto del sistema que representan por medio de la aplicación de transformaciones de modelos. De esta forma, los modelos *runtime* permiten mantener, gestionar y monitorizar sistemas software complejos [12].

Para englobar a todas las aplicaciones anteriores, el término más extendido en la literatura anglosajona es *Model Driven Engineering* (MDE) mientras que los términos *Model Driven Development* (MDD) y *Model Driven Reengineering* (MDR) se utilizan para referirse a la aplicación de este paradigma en tareas de ingeniería directa e inversa, respectivamente. Sin embargo, en la literatura castellana está más extendido el uso del término DSDM para referirse a cualquier aplicación del paradigma. Por este motivo, éste será el término que utilizaremos a lo largo de este documento.

Las transformaciones de modelos juegan un papel fundamental en el DSDM ya que permiten automatizar las tareas del desarrollo de software. Existen tres tipos de transformaciones de modelos [21]: modelo-a-modelo (*m2m*), modelo-a-texto (*m2t*) y texto-a-modelo (*t2m*). Las transformaciones *m2m* permiten transformar unos modelos en otros. Las transformaciones *m2t* se utilizan para generar los artefactos del sistema software a partir de modelos (p. ej., el código fuente o ficheros de configuración en XML). Finalmente, las transformaciones *t2m* se utilizan para extraer modelos de los artefactos del sistema existente.

El concepto de *espacio tecnológico* se presenta en [49] con el objetivo de servir de guía para utilizar una determinada tecnología. Un espacio tecnológico se define como un contexto de trabajo que tiene asociados un conjunto de conceptos, métodos, técnicas y herramientas. Dada esta definición, cada uno de los tipos de artefactos que compone un sistema software podría enmarcarse dentro de un espacio tecnológico, por ejemplo, el código fuente estaría dentro del espacio tecnológico asociado a los lenguajes de programación, llamado normalmente *grammarware*, mientras que los documentos XML estarían dentro del espacio tecnológico asociado a la tecnología XML, que denominaremos *xmlware*. Un espacio tecnológico está basado en dos conceptos básicos que están relacionados por una relación de instanciación (p. ej., programa/gramática en el caso del *grammarware* o documento/esquema en el caso del *xmlware*). Además, los espacios tecnológicos no están aislados, pudiéndose definir puentes entre cada par de espacios tecnológicos, los cuales pueden ser bidireccionales o unidireccionales y tienen como objetivo transformar un artefacto definido en un

determinado espacio tecnológico en una representación propia de otro espacio tecnológico. Esto permite aprovechar las técnicas, métodos y herramientas del espacio tecnológico destino sobre artefactos definidos en un espacio tecnológico diferente, así como facilitar la interoperabilidad entre aplicaciones. Por ejemplo, el procesamiento de documentos XML desde un programa Java requiere un puente unidireccional entre el *xmlware* y el *grammarware*, el cual permite al código del programa Java interpretar los datos XML. Para definir puentes bidireccionales con el espacio tecnológico del DSDM, llamado *modelware*, es necesario ofrecer dos operaciones: (1) extracción de modelos a partir de los artefactos software del espacio tecnológico considerado y (2) generación de dichos artefactos a partir de los modelos. Por ejemplo, un puente bidireccional entre el *grammarware* y el *modelware* debería permitir la extracción de modelos a partir del código fuente así como la generación de dicho código a partir de los modelos. De esta manera, la combinación de transformaciones *t2m* y *m2t*, referidas a ficheros de código fuente, permitirían implementar puentes bidireccionales entre el *grammarware* y el *modelware*.

En la actualidad, existen lenguajes para la definición tanto de transformaciones *m2m* como *m2t*. Por ejemplo, ATL [40], RubyTL [87], QVT [96] o Epsilon [27] son algunos ejemplos conocidos de lenguajes de transformación *m2m* y MOFScript [54], JET [70] o XPand [109] son algunos de los lenguajes de transformación *m2t* más extendidos. Sin embargo, no existen lenguajes especialmente diseñados para la definición de transformaciones *t2m*, las cuales son normalmente implementadas haciendo uso de lenguajes GPL. Mientras que las transformaciones *m2m* trabajan dentro del *modelware* (tanto el origen como el destino de la transformación son modelos), las transformaciones *m2t* y *t2m* tienen que tratar con los diferentes artefactos que componen un sistema, cuya naturaleza es variada y por lo tanto pertenecen a diferentes espacios tecnológicos. En el caso de las transformaciones *m2t*, la mayoría de las soluciones existentes utilizan el mecanismo de plantillas [21] para generar, por ejemplo, el texto correspondiente al código fuente, código XML o sentencias DDL para definir los datos de una base de datos. En el caso de las transformaciones *t2m* no existen lenguajes aplicables a cualquier artefacto de entrada, sino que es preciso implementar soluciones específicas, como por ejemplo el extractor de modelos desde código Java de MoDisco [11] o desde código PL/SQL de Forms2Net [1]. La diferencia principal entre las transformaciones *m2t* y *t2m* es que en el primer caso, el uso de plantillas permite generar cualquier artefacto textual (como código fuente o ficheros XML) sin tener en cuenta el formalismo utilizado para definirlo, mientras que en el segundo caso es necesario conocer dicho formalismo para poder reconocer la estructura e información que contienen, esto es, aplicar herramientas y técnicas del espacio tecnológico al que pertenecen los artefactos software. Por ejemplo, la generación de código fuente a partir de modelos puede realizarse por medio de plantillas como MOFScript, por otro lado, la obtención de modelos desde el código fuente requiere utilizar conceptos y técnicas del *grammarware* tales como la gramática del lenguaje de programación utilizado y el uso de *parsers*. De esta forma, las transformaciones *t2m* son normalmente implementadas utilizando *parsers* cuyo desarrollo es tedioso, propenso a errores y costoso en tiempo.

La existencia de DSL que faciliten la extracción de modelos supondría un beneficio importante en productividad y calidad en comparación a la construcción de *parsers*. Dada

la variabilidad en la naturaleza de los artefactos que componen un sistema software, en principio no parece factible que un sólo lenguaje pudiera cubrir todos los casos.

2.2. Objetivos

El objetivo principal de esta tesis es facilitar a los desarrolladores de soluciones basadas en el DSDM la tarea de extraer de modelos desde tres diferentes espacios tecnológicos, los cuales cubren la mayor parte de los artefactos que componen un sistema software y son:

- *Grammarware*, que incluye el código expresado en un lenguaje software conforme a una gramática, como los ficheros de código fuente de un lenguaje de programación.
- *Dataware*, que incluye los datos que conforman con un esquema de base de datos, como tuplas de una tabla de una base datos relacional.
- *Apiware*, que incluye a todos los objetos creados y accesibles mediante un API, como los objetos de una interfaz de usuario creados con un API como Swing o SWT.

La extracción de modelos a partir de código fuente y datos es una tarea necesaria en cualquier modernización basada en modelos puesto que son los elementos principales de cualquier sistema software. En la actualidad, la obtención de modelos desde el código fuente y los datos se realiza por medio de soluciones basadas en *parsers*, los cuales requieren un gran esfuerzo de implementación. En cuanto a los API, están surgiendo nuevas aplicaciones del DSDM basadas en el manejo de modelos creados a partir de objetos en memoria que representan algún aspecto del sistema y que son creados y accesibles por medio de un API [82, 94]. Sin embargo, actualmente no existen soluciones que automaticen la creación de modelos a partir de objetos de un API en memoria y la generación de estos objetos a partir de modelos, así como el mantenimiento de la sincronización entre objetos del API y sus modelos.

En esta tesis, se ha considerado la extracción de modelos como un proceso de transformación *t2m* que puede ser descrito de forma abstracta mediante un DSL, del mismo modo que se usan lenguajes específicos para las transformaciones *m2m* y *m2t*. Con ello se consiguen los beneficios que aporta la utilización de DSL frente a soluciones basadas en el empleo de lenguajes GPL, esto es, incrementar la productividad, mejorar la calidad y favorecer el mantenimiento [23, 50].

Por tanto, hemos definido una familia de DSL para la extracción de modelos desde los tres espacios tecnológicos mencionados, con el objetivo de facilitar la creación de puentes entre los espacios tecnológicos considerados y el *modelware*. Estos lenguajes son Gra2MoL (*grammarware-modelware*), SchcMoL (*dataware-modelware*) y API2MoL (*apiware-modelware*). Gra2MoL supone una alternativa a la solución actualmente adoptada de implementar *parsers* específicos y su finalidad es mejorar la productividad, así como favorecer el mantenimiento y extensibilidad en la creación de puentes *grammarware-modelware*. Realmente, Gra2MoL es una primera propuesta de lenguaje de transformación *t2m* que permite transformar cualquier artefacto software conforme a una gramática en un modelo conforme a un metamodelo. Por otro lado, SchcMoL y API2MoL son realmente dos aproximaciones

originales que abren la posibilidad de trasladar la filosofía que inspiró Gra2MoL al ámbito de la extracción de modelos de datos y objetos de un API.

Por otra parte, dado que Gra2MoL fue creado principalmente para su aplicación en la modernización, otro de los objetivos del trabajo ha sido experimentar con ADM (*Architecture-Driven Modernization*), la principal iniciativa de modernización dirigida por modelos destinada a ofrecer un conjunto de metamodelos estándares que entre otros beneficios permitan la interoperabilidad entre herramientas de modernización.

2.3. Metodología

Inicialmente, el trabajo de esta tesis se orientó hacia la construcción de un *framework* de modernización basado en modelos para código GPL, el cual sería integrado en AGE [85]. Este *framework* de modernización incluiría, entre otros componentes, un DSL denominado Gra2MoL destinado a la extracción de modelos a partir de código GPL y un conjunto de herramientas destinadas a facilitar el uso de ADM. Sin embargo, después de la creación de la primera versión de Gra2MoL y tras surgir las colaboraciones con los grupos ONEKIN y AtlanMod para crear los lenguajes ScheMoL y API2MoL, la tesis se centró en el problema de la extracción de modelos y en la definición de DSL que soportasen la construcción de los puentes entre los espacios tecnológicos mencionados y el *modelware*. Cabe destacar que el diseño e implementación de Gra2MoL fueron reutilizados para la creación de ScheMoL y API2MoL, sobre todo en el caso de ScheMoL cuyo diseño está basado en Gra2MoL y en cuya implementación se reutilizaron componentes del motor de ejecución de Gra2MoL.

Gra2MoL se ha creado como un DSL externo y de acuerdo a las recomendaciones para la definición de DSL propuestas en [30]. En primer lugar se definió la sintaxis abstracta del lenguaje, luego la sintaxis concreta y finalmente se implementaron las principales herramientas asociadas, esto es, un *parser* y un editor. Para el diseño de este lenguaje nos inspiramos en los lenguajes de transformación *m2m* ATL [40] y RubyTL [87], pero en vez de considerar una correspondencia entre dos metamodelos se contempla una correspondencia entre una gramática y un metamodelo. De esta forma, el desarrollo de Gra2MoL permitió estudiar la aplicación y adecuación de las técnicas utilizadas en los lenguages *m2m* para la construcción de puentes unidireccionales del *grammarware* al *modelware*. De la misma forma que ATL y RubyTL, Gra2MoL es un lenguaje basado en reglas que incorpora el concepto de *binding*. Sin embargo, la forma de recorrer los artefactos origen es diferente, ya que Gra2MoL debe tratar con ficheros de código fuente como entrada. Al extraer modelos desde código fuente, es necesario recorrer el código, esto es, el árbol de sintaxis que lo representa, para obtener la información dispersa que compone a los elementos del modelo. Por ello se definió un lenguaje de consultas especialmente adaptado para obtener información del árbol de sintaxis en vez de utilizar un lenguaje tipo OCL como los utilizados en los lenguajes *m2m*.

Gra2MoL se aplicó a diferentes casos de estudio (Java, Delphi, PL/SQL, Maude, IDL, especificaciones EBNF, gramáticas ANTLR y *scripts* Bash) lo cual nos permitió identificar y solucionar algunas deficiencias así como validar el lenguaje con casos prácticos reales. De esta forma, las características del lenguaje fueron mejoradas y ampliadas durante el

desarrollo de estos casos de estudio. Por ejemplo, al extraer modelos desde código Maude detectamos la necesidad de incorporar el soporte para gramáticas *isla* (*island grammars*). Por otro lado, la necesidad de manejar un gran número de ficheros fuente requirió incorporar mecanismos de gestión de memoria eficientes como el soporte para CDO [62], un repositorio de modelos escalable, transaccional y distribuido.

También es importante destacar que Gra2MoL fue utilizado para desarrollar uno de los primeros casos de estudio de ADM, en concreto la creación de una herramienta para calcular métricas de evaluación del esfuerzo de una migración de aplicaciones Oracle Forms. ADM proporciona siete metamodelos estándar para tareas de modernización, aunque solamente tres de ellos están actualmente disponibles: el metamodelo KDM (*Knowledge Discovery Metamodel*), el metamodelo ASTM (*Abstract Syntax Tree Metamodel*) y el metamodelo SMM (*Software Measurement Metamodel*). Se ideó un proceso para aplicar los metamodelos de ADM en tareas de modernización. Primero se utilizó Gra2MoL para extraer modelos ASTM desde el código (en nuestro caso código PL/SQL de aplicaciones Oracle Forms), lo cual facilitó la obtención de modelos KDM por medio de transformaciones *m2m*. Estos modelos KDM fueron finalmente utilizados para obtener los artefactos deseados mediante una cadena de transformaciones (en nuestro caso un conjunto de métricas).

La colaboración con el grupo de investigación ONEKIN motivó el estudio del problema de la extracción de modelos a partir de datos almacenados en bases de datos relacionales, lo cual supone definir un puente unidireccional del *dataware* al *modelware*. Como una alternativa a las soluciones basadas en el uso de API de acceso a bases de datos (p. ej., JDBC [39]), *frameworks* objeto-relacional (p. ej., Hibernate [37]) o *frameworks* modelo-relacional (p. ej., Teneo [80]), decidimos crear un nuevo DSL, llamado ScheMoL, especialmente diseñado para tratar con el problema de extraer modelos a partir de las tuplas de las tablas de un esquema relacional. Como hemos indicado, ScheMoL reutiliza determinados conceptos y componentes de Gra2MoL. De esta forma, ScheMoL comparte con Gra2MoL el hecho de utilizar reglas así como el concepto de *binding*, el cual fue adaptado para tratar con tuplas. Además, en ScheMoL también se incluyó un lenguaje de consultas, el cual fue adaptado para obtener información de bases de datos de forma transparente.

Finalmente, la estancia predoctoral del candidato en el grupo de investigación AtlanMod permitió abordar el estudio de la extracción de modelos a partir de objetos creados con un API, lo que supone definir un puente entre los espacios tecnológicos *apiware* y *modelware*. En este caso, se definió un DSL denominado API2MoL, para crear puentes bidireccionales entre estos espacios tecnológicos. Este puente permitiría extraer modelos a partir de objetos de en memoria en tiempo de ejecución así como generar dichos objetos a partir de modelos. El diseño de API2MoL fue ligeramente diferente a los DSL anteriores, ya que los requisitos no eran similares. API2MoL también utiliza reglas para definir las correspondencias entre el API y el metamodelo, sin embargo, no requiere incorporar un lenguaje de consultas ya que las reglas solamente deben especificar los métodos que deben ser llamados para obtener o registrar la información necesaria cuando se crea e inicializa el elemento del modelo o el objeto del API, respectivamente. Por otra parte, en la aproximación se incorporó un proceso de descubrimiento para generar automáticamente tanto el metamodelo que representa el API como las correspondencias entre las clases del API y las metaclases del metamodelo

del API.

Al igual que se realizó con Gra2MoL, los lenguajes ScheMoL y API2MoL también fueron validados con diferentes casos prácticos reales. Por ejemplo, ScheMoL sirvió para extraer modelos de aplicaciones web 2.0 como los wikis y los blogs, y API2MoL se utilizó en casos practicos donde se utilizaron diferentes API para la creación de interfaces gráficas de usuario así como en un caso práctico empresarial basado en la extracción de modelos de LinkedIn. Todas estas pruebas también sirvieron para mejorar y adaptar las características de cada uno al problema.

2.4. Organización del Documento

Una vez realizada la introducción al trabajo desarrollado en esta tesis, a continuación se realiza un breve resumen de cada uno de los capítulos de este documento:

- **Capítulo 3**. Se realiza una descripción del contexto en el que se enmarca esta tesis. En primer lugar, definirán los conceptos básicos del paradigma del Desarrollo Dirigido por Modelos. A continuación se presentarán los enfoques DSDM más extendidos y luego se analizarán en detalle las características de los Lenguajes Específicos del Dominio. Finalmente se introducirá la Modernización dirigida por Modelos y la iniciativa ADM de OMG.
- **Capítulo 4**. Presenta el concepto de espacio tecnológico y se describen los cuatro espacios tecnológicos involucrados en esta tesis. También se motiva la necesidad de crear puentes entre espacios tecnológicos.
- **Capítulo 5**. Se describe el lenguaje Gra2MoL. En primer lugar se describe el problema y las aproximaciones existentes. A continuación se describe nuestro enfoque y luego un ejemplo de aplicación de Gra2MoL. Finalmente se comentan algunos escenarios de aplicación y se presentan las características principales del lenguaje y las conclusiones extraídas.
- **Capítulo 6**. Describe cómo llevar a cabo la extracción de modelos KDM en un proceso de modernización basado en ADM. El proceso se ilustra por medio de un caso de estudio basado en el cálculo de métricas en aplicaciones Oracle Forms. En primer lugar se describen los pasos principales del proceso de modernización y luego se describe cómo se han aplicado al caso de estudio. Finalmente se muestran las conclusiones extraídas al utilizar la iniciativa ADM en modernización.
- **Capítulo 7**. Se describe el lenguaje ScheMoL. Inicialmente se describe el problema y luego se presenta nuestra aproximación comparándola con las soluciones existentes. A continuación se muestra un ejemplo de aplicación de ScheMoL y se comentan algunos escenarios de aplicación. Finalmente se presentan las características principales del lenguaje y las conclusiones extraídas.
- **Capítulo 8**. Se describe el lenguaje API2MoL. En primer lugar, se describe el problema y se justifica la necesidad del lenguaje así como la posibilidad de generar automáticamente tanto las definiciones de transformación como los metamodelos de

los API por medio de un proceso de descubrimiento. A continuación se presenta nuestra aproximación y su funcionamiento, describiendo el lenguaje creado con la ayuda de un ejemplo y luego se describe el proceso de descubrimiento. Posteriormente se presenta el proceso de validación aplicado al lenguaje y al proceso de descubrimiento y a continuación el trabajo relacionado. Finalmente se comentan algunos escenarios de aplicación y se presentan las características principales del lenguaje y las conclusiones extraídas al utilizarlo.

- **Capítulo 9**. Finaliza el documento con las conclusiones y trabajo futuro. También se muestra un listado de publicaciones, colaboraciones y trabajo asociado a esta tesis.

3

Fundamentos del desarrollo de software dirigido por modelos

–Los dioses tienen dioses– pensó Tephe.
The God Engines, John Scalzi

El paradigma del Desarrollo de Software Dirigido por Modelos engloba a un conjunto de enfoques que tienen en común el aumento del nivel de abstracción y automatización a través de la utilización de modelos, metamodelos, transformaciones de modelos y lenguajes específicos del dominio. Entre los enfoques existentes destaca MDA, las factorías software y el desarrollo específico del dominio. En este capítulo primero definiremos los conceptos básicos del paradigma y presentaremos los principales enfoques existentes en la actualidad, luego nos centraremos en el desarrollo basado en lenguajes específicos del dominio ya que es el enfoque más relacionado con este trabajo de investigación. El capítulo finaliza presentando la Modernización de Software Dirigida por Modelos ya que el estudio de la aplicación de las técnicas del DSDM a la modernización o reingeniería del software ha sido uno de los objetivos de esta tesis.

3.1. Elementos básicos

Desde sus orígenes, el principal reto de la ingeniería del software ha sido la aparición de una verdadera industria del software capaz de producir software de alta calidad a bajo coste. Los avances en tecnología del software han permitido crear sistemas software cuya complejidad y tamaño los sitúa entre las mayores obras de ingeniería creadas por los seres humanos. Desde la construcción de los primeros compiladores, que son sin duda la innovación que ha originado el mayor incremento de productividad en la creación de software, diferentes innovaciones han permitido mejorar tres aspectos esenciales para conseguir la industrialización del software: nivel de abstracción, automatización y reutilización. Sin embargo, la mejora ha sido limitada y la producción de software todavía sigue siendo una

actividad que demanda una gran cantidad de tareas manuales, lejos de alcanzar el nivel de automatización propio de una verdadera industria [33].

A principios de la pasada década, mientras la tecnología orientada a objetos se extendía por la industria del software de todo el mundo, en el ámbito académico crecía el interés por el Desarrollo de Software Dirigido por Modelos (DSDM), un nuevo paradigma de construcción de software que supera las limitaciones del paradigma orientado a objetos en los niveles de abstracción, automatización y reutilización ofrecidos. A lo largo de estos años ha aumentado continuamente el interés de la comunidad del software por este nuevo paradigma, pero todavía no ha alcanzado el grado de madurez necesario para su aceptación por las empresas de desarrollo de software, siendo necesario avanzar en investigación, construcción de herramientas usables y en la formación de profesionales [89, 90].

Hasta ahora los modelos han servido principalmente para documentar el proceso de creación de software, y en menor medida para razonar sobre la estructura y comportamiento de los sistemas. Sin embargo, el DSDM convierte a los modelos en artefactos de primera categoría que también son parte de una aplicación de igual manera que el código fuente. Esto significa que una aplicación (o partes de ella) puede ser generada automáticamente a partir de modelos expresados con lenguajes de modelado o lenguajes específicos del dominio (*Domain Specific Languages*, DSLs), en vez de usar lenguajes de programación de propósito general, de modo que se pueden producir beneficios en la productividad y calidad similares a lo que supuso el paso de los lenguajes ensambladores a los lenguajes de programación [33, 43, 91]. Este uso de modelos implica, por tanto, un aumento en el nivel de abstracción en que se expresa las soluciones y una mayor automatización en la producción de software.

En realidad, el paradigma DSDM engloba diferentes visiones que comparten cuatro elementos básicos: (1) el uso de modelos para representar diferentes aspectos de un sistema software, (2) los modelos son expresados mediante DSL, (3) la sintaxis abstracta de un DSL es definida mediante un metamodelo sobre el que se pueden definir varias notaciones o sintaxis concretas, y (4) la utilización de transformaciones de modelos para definir cadenas de transformaciones destinadas a generar los artefactos del sistema final (p. ej., código fuente o ficheros de configuración). A continuación vamos a describir cada uno de estos cuatro elementos. Una descripción mas detallada se puede encontrar en [16, 28, 29, 45].

Los *modelos* son esenciales en todas las disciplinas para abordar la complejidad de los sistemas bajo estudio. Un modelo es una simplificación de la realidad que es representada en alguna notación y cuyo objetivo es ayudar a comprender y razonar sobre esa realidad. Los modelos son resultado de un proceso de abstracción que oculta los detalles irrelevantes y muestra sólo aquellos detalles de interés para el fin perseguido. En el caso de la construcción de software, los modelos representan aspectos de un sistema software, tales como los requisitos, la estructura, el comportamiento o el despliegue, y tienen como finalidad (i) ayudar a razonar sobre cómo debe construirse el sistema, (ii) proporcionar documentación que transcienda al proyecto, (iii) facilitar la comunicación entre los desarrolladores y (iv) generar automáticamente código de la aplicación final [31]. Esta última utilidad es la que explota el DSDM y aunque la idea se remonta a la aparición de los primeros lenguajes de modelado del software, antes no se había creado una disciplina en torno a ella. Una cuestión clave cuando se genera código a partir de modelos es la necesidad de que estos se

expresen en un lenguaje formalmente definido.

En DSDM, un modelo es conforme al *metamodelo* del lenguaje (*DSL*) con el que se expresa, de forma similar a un programa que conforma a la gramática de un lenguaje de programación. Un metamodelo describe de forma precisa la sintaxis abstracta de un DSL, esto es, los conceptos del lenguaje y las relaciones entre ellos, junto a las reglas que determinan cuando el modelo está bien formado. Los metamodelos se definen con lenguajes de metamodelado que ofrecen construcciones para crear modelos que especifican metamodelos: un metamodelo es un "modelo de modelos". En la actualidad el lenguaje de metamodelado más extendido es Ecore [99], el cual está basado en MOF [95], y proporciona los conceptos básicos de la orientación a objetos para crear metamodelos como modelos conceptuales que describen lenguajes: los conceptos son representados como clases, las propiedades de un concepto como atributos de la correspondiente clase, las relaciones entre objetos como asociaciones (en el caso de Ecore son referencias entre clases), y la especialización de un concepto como una herencia de clases. Las reglas que indican cuando el modelo está bien formado son normalmente expresadas mediante el lenguaje OCL o un lenguaje basado en él.

Un metamodelo determina un conjunto de posibles grafos, cada uno de los cuales es un posible modelo conforme a dicho metamodelo. Los nodos son instancias de las clases del metamodelo y los arcos son determinados por las relaciones entre dichas clases. De esta forma, los elementos de un modelo son instancias de los elementos (metaclases) de un metamodelo, por lo que se dice que un "modelo es instancia de un metamodelo". El grafo de instancias que representa un modelo conforme a un metamodelo se denomina grafo de sintaxis abstracta, aunque a menudo se usa el término árbol de sintaxis abstracta (*Abstract Syntax Tree*, AST) para referirse a él por su paralelismo con el árbol que representa la sintaxis abstracta de un código conforme a una gramática.

Dado que un lenguaje de metamodelado, como es el caso de Ecore y MOF, también es definido por un metamodelo que se denomina meta-metamodelo, los elementos de un metamodelo son instancias de los elementos de su meta-metamodelo: un metamodelo es conforme a su meta-metamodelo. Por tanto, un metamodelo también es representado por un grafo de instancias de las metaclases de su meta-metamodelo.

Podemos pues establecer las siguientes definiciones:

Modelo. Un modelo representa uno o varios aspectos de un sistema software. Está compuesto por un conjunto de instancias de elementos de un metamodelo y se expresa mediante un DSL. Todo modelo debe conformar a un determinado metamodelo.

Metamodelo. Un metamodelo describe de forma precisa los conceptos, relaciones y restricciones de un DSL, siendo las restricciones un conjunto de reglas que determinan cuando un modelo que es conforme al metamodelo está bien formado.

Conformidad modelo-metamodelo. Relación que denota que cada uno de los elementos de un modelo es instancia de una metaclase del metamodelo.

Como hemos indicado, un DSL ofrece construcciones para especificar soluciones a problemas en un dominio concreto a diferencia de los lenguajes de programación de propósito general. Un principio básico en la definición de un DSL en el contexto del DSDM es la

separación entre sintaxis abstracta y sintaxis concreta. Por una parte se define la sintaxis abstracta como un metamodelo y entonces se pueden definir varias notaciones o sintaxis concretas para este metamodelo. La semántica constituye el tercer elemento de un DSL y normalmente se define mediante la traducción a otro lenguaje cuya semántica esté bien definida, como puede ser el caso de un lenguaje de programación como Java o C#, un lenguaje intermedio como el lenguaje CIL definido para .NET o un lenguaje de especificación formal como Maude [24]. En la sección 3.3 se profundizará en el concepto de DSL cuya definición sería la siguiente.

DSL. Lenguaje software que ofrece construcciones para resolver problemas en un determinado dominio. La especificación de la solución se suele denominar modelo (DSL gráficos) y/o programa (DSL textuales).

Las *transformaciones de modelos* juegan un papel fundamental en el DSDM ya que permiten automatizar tareas en la construcción de software. De acuerdo a [21], existen tres tipos de transformaciones de modelos: modelo-a-modelo (*m2m*), modelo-a-texto (*m2t*) y texto-a-modelo (*t2m*). Una transformación *m2m* es el proceso de generar automáticamente un modelo destino que conforma a un metamodelo destino a partir de un modelo origen que conforma a un metamodelo origen. En realidad, el número de modelos de entrada y salida a la transformación puede ser variable, aunque las transformaciones que convierten un modelo de entrada en uno o varios de salida son las más frecuentes. Cuando el modelo de entrada y el modelo de salida conforman al mismo metamodelo la transformación se denomina endógena, en caso contrario se trata de una transformación exógena. Las transformaciones también se clasifican como horizontales o verticales, según los modelos de entrada y de salida tengan o no el mismo nivel de abstracción, respectivamente. El proceso de transformación es dirigido por una definición de transformación expresada en un lenguaje de transformación *m2m*. Una definición de transformación *m2m* consta de un conjunto de reglas que expresan las correspondencias entre los elementos del modelo de entrada con elementos del modelo de salida, con frecuencia entre un elemento de entrada con uno de salida. Las reglas expresan cómo llevar a cabo la conversión. Algunos ejemplos de lenguajes de transformación *m2m* son ATL [40], RubyTL [87] y Epsilon [27].

Transformación *m2m*. Proceso de generar automáticamente un modelo destino que conforma a un metamodelo destino a partir de un modelo origen que conforma a un metamodelo origen. El proceso es dirigido por una definición de transformación *m2m*.

Definición de transformación *m2m*. Especificación que dirige una transformación *m2m* y que consta de un conjunto de reglas de transformación.

Regla de transformación *m2m*. Establece la correspondencia entre uno o más elementos del metamodelo de entrada con uno o más elementos del metamodelo de salida.

Las transformaciones *m2t* se utilizan para generar artefactos del sistema software (p. ej., código fuente o ficheros de configuración en XML). Una transformación *m2t* es el proceso de generar automáticamente un artefacto software formado por texto a partir de un modelo conforme a un metamodelo. Este proceso de transformación es dirigido por una

definición de transformación que es expresada en algún lenguaje de transformación *m2t* y está formada por un conjunto de reglas. Estas reglas establecen cuál es el texto de salida para un determinado elemento del modelo de entrada. Algunos ejemplos de lenguajes de transformación *m2t* son MOFScript [54] y Xpand [109].

Transformación *m2t*. Proceso de generar automáticamente un artefacto software formado por texto a partir de un modelo conforme a un metamodelo. El proceso es dirigido por una definición de transformación *m2t*.

Definición de transformación *m2t*. Especificación que dirige una transformación *m2t* y consta de un conjunto de reglas de transformación.

Regla de transformación *m2t*. Establece el texto a generar para un determinado elemento del metamodelo de entrada.

Finalmente, las transformaciones *t2m* se utilizan para extraer o inyectar modelos de los artefactos del sistema existente y son principalmente utilizadas en modernización de software, tal y como se comentará en la sección 3.4. Una transformación *t2m* es el proceso de generar automáticamente un modelo a partir de un artefacto software formado por texto. Hasta ahora, estos procesos se han implementado mediante *parsers* construidos cada vez que ha surgido una necesidad de extracción en un proyecto de modernización. Por tanto, no hay disponibles lenguajes basados en reglas diseñados con el fin de expresar transformaciones *t2m*. El principal objetivo de esta tesis ha sido abordar la creación de lenguajes basados en reglas para la tarea de extraer modelos.

Transformación *t2m*. Proceso de generar automáticamente un modelo a partir de los artefactos de un sistema software. El proceso es dirigido por una definición de transformación *m2t*, cuya naturaleza depende de la aproximación tomada para implementar el proceso.

Definición de transformación *m2t*. En el contexto de esta tesis, especificación que dirige una transformación *t2m* y consta de un conjunto de reglas de transformación.

Regla de transformación *m2t*. Establece el elemento del metamodelo a generar a partir del texto que compone el artefacto de entrada.

3.2. Algunos enfoques DSDM

Aunque la idea de generar código a partir de modelos ya había sido planteada con bastante anterioridad, incluso considerando el concepto de metamodelo [57], y el papel de los DSL era conocido desde los primeros años de la programación, es indudable que la iniciativa MDA (*Model-Driven Architecture*) [98], que fue lanzada en noviembre de 2000 por el OMG, ha sido clave en la consolidación del DSDM como un nuevo paradigma de desarrollo. La especificación MDA proporcionó conceptos, lenguajes y estándares que han contribuido a impulsar el DSDM. De hecho, durante la mayor parte de la pasada década fue frecuente confundir MDA con DSDM, en vez de comprender que MDA era sólo una visión del DSDM.

En lo que sigue se presentará brevemente MDA junto a otros dos enfoques DSDM bien conocidos, que son el desarrollo específico del dominio y las factorías del software.

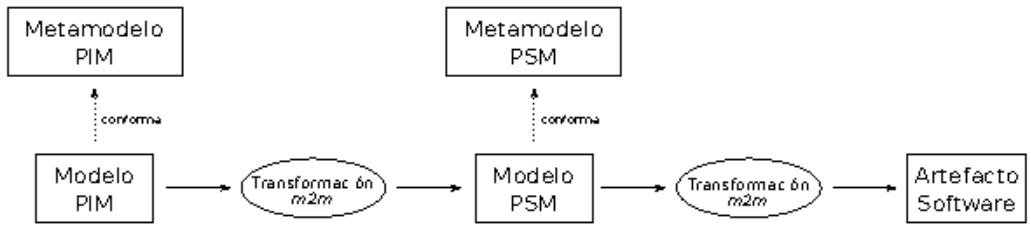


Figura 3.1: Proceso de desarrollo simplificado basado en MDA.

Un concepto clave en **MDA** es el de plataforma. Según el OMG, una plataforma es "un grupo de subsistemas y tecnologías que ofrecen un conjunto coherente de funcionalidades, descritas a través de interfaces y patrones de uso, que pueden ser utilizadas por cualquier otra aplicación soportada por la plataforma sin necesidad de conocer los detalles de implementación" [46]. MDA propone la separación de la especificación de la funcionalidad de su implementación sobre una plataforma concreta. De esta forma, en MDA se definen dos tipos de modelos: los modelos independientes de la plataforma (*Platform Independent Model*, PIM) y los modelos específicos de la plataforma (*Platform Specific Model*, PSM). Los modelos PIM constituyen modelos de alto nivel independientes de cualquier tecnología o plataforma, mientras que los modelos PSM son de más bajo nivel que los PIM y describen el sistema de acuerdo a una tecnología de implementación determinada. El proceso de desarrollo propuesto por MDA arranca con la definición de los modelos PIM, a partir de los cuales se generan los modelos PSM por medio de transformaciones *m2m* y finalmente el código del sistema por medio de transformaciones *m2t*, tal y como se ilustra en la Figura 3.1. La idea clave de esta arquitectura es potenciar la portabilidad, interoperabilidad y mantenimiento. Otro aspecto que también se ve favorecido es la productividad, dado que a partir de los modelos PIM se puede generar automáticamente parte de la aplicación. Por ejemplo, en una aplicación de gestión, el desarrollador solo debería preocuparse de implementar la lógica de negocio no tratada por dicho proceso.

El **desarrollo específico del dominio** propone el uso de DSL para expresar las soluciones mediante conceptos propios del dominio del problema. Las especificaciones expresadas con los DSL son utilizadas por un motor de transformación para generar automáticamente el código final de una aplicación. De esta forma, estas especificaciones pueden ser consideradas como el programa de la aplicación. Normalmente los DSL tienen como finalidad automatizar el desarrollo de aplicaciones para un *framework* o plataforma específicos. Esta aproximación diferencia entre expertos del dominio, que se encargan de definir los DSL y los desarrolladores o usuarios de dichos DSL. Dado que el trabajo de investigación desarrollado en esta tesis se centra en la creación de diferentes DSL para la extracción de modelos en modernización, la siguiente sección tratará en mayor detalle la definición de los DSL. La Figura 3.2 muestra un proceso simplificado basado en este enfoque.

Finalmente, el enfoque basado en **factorías de software** se describe en detalle en [33] y propone combinar el enfoque de las líneas de producto con el DSDM. Una factoría software se define como "una línea de producto software cuyos componentes de producción forman un entorno de desarrollo configurado para permitir el desarrollo rápido de los miembros de

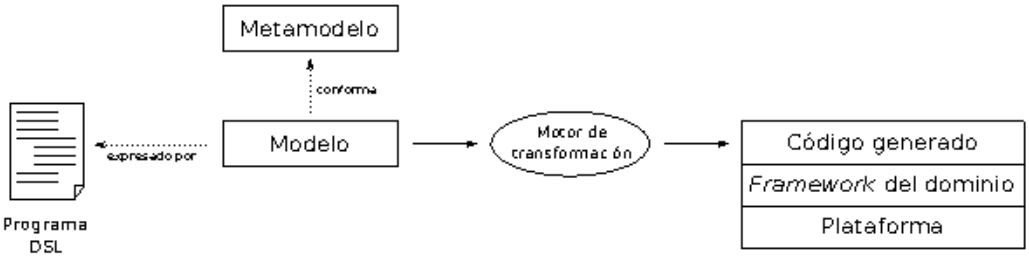


Figura 3.2: Proceso de desarrollo simplificado basado en el enfoque del desarrollo específico del dominio.

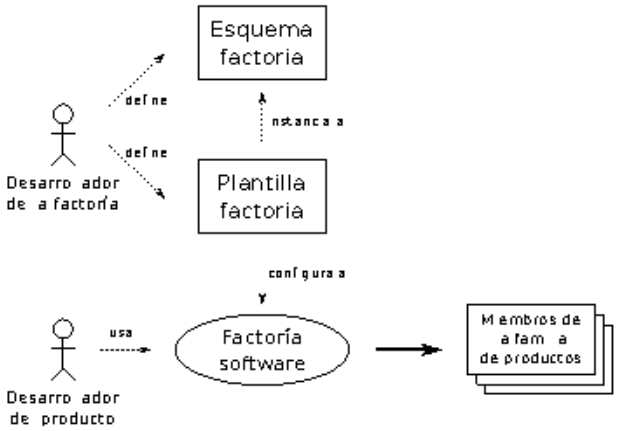


Figura 3.3: Proceso de desarrollo basado en factorías de software.

la familia de productos". De esta forma, el objetivo principal de las factorías del software es ofrecer un conjunto de elementos software (procesos, patrones, DSL, etc) integrados en un IDE para facilitar la creación de artefactos software de calidad para el dominio en el que esté enfocada la factoría. El proceso de desarrollo propuesto por las factorías software se basa en dos elementos principales: *esquema* de la factoría y *plantilla* de la factoría. El esquema de la factoría describe cómo se produce una aplicación a través de elementos software como DSL, transformaciones de modelos, patrones y procesos. Además, el esquema también establece las partes comunes y variables del sistema, para lo que es normal el uso de modelos de características [19]. Por otro lado, la plantilla de la factoría está formada por el código y metadatos que implementan los artefactos del esquema y que son cargados en un IDE para automatizar el desarrollo de los miembros de la familia. La Figura 3.3 ilustra el proceso de desarrollo basado en factorías de software, donde aparecen los roles de desarrollador de la factoría, que define el esquema y la plantilla, y los desarrolladores de un producto concreto, que usan la factoría.

3.3. Lenguajes Específicos del Dominio

Cuando un desarrollador implementa una aplicación debe inicialmente estudiar sus requisitos y pensar la solución en términos del dominio del problema. A continuación diseña e implementa la aplicación utilizando conceptos del dominio de la solución. La labor del desarrollador es ser capaz de resolver este salto semántico existente entre ambos dominios, la cual es realizada haciendo uso de los lenguajes de programación.

Los lenguajes ensambladores requerían un gran esfuerzo para implementar la solución dado que el salto semántico era muy grande. Programar en ensamblador era por lo tanto una tarea tediosa, complicada y poco productiva porque requería expresar los algoritmos teniendo en cuenta la estructura y funcionamiento de los ordenadores. Como respuesta a esta situación, se crearon los lenguajes de programación de propósito general, que ofrecen un nivel de abstracción mucho mayor y permite que los desarrolladores se centren en especificar qué debe hacer el programa sin tener en cuenta los detalles arquitecturales de la máquina que lo ejecuta, ya que un compilador se encarga de generar el código máquina correspondiente, aumentando la productividad y reduciendo así el salto semántico entre el dominio del problema y el dominio de la solución. Sin embargo, el hecho de que un lenguaje de programación sea de propósito general provoca que el desarrollador todavía tenga que adaptar los conceptos y algoritmos ideados en términos del dominio del problema al dominio de la solución, reduciendo considerablemente la productividad. Por este motivo aparecieron los DSL, cuyo nivel de abstracción es mayor y están particularizados a un determinado dominio. Según [21, 30, 91], uno de los principales atractivos que tiene el uso de DSL es el hecho de poder especificar claramente el objetivo o intención de nuestro programa. Esto redunda en un aumento productividad y capacidad de mantenimiento del código del DSL así como de comunicación con los usuarios del dominio. Algunos ejemplos bien conocidos de DSL son SQL, Excel y HTML. Realmente, el uso de DSL no es una característica novedosa del DSDM, aunque actualmente ha crecido significativamente el interés por ellos ya que son un elemento fundamental de dicho paradigma.

El auge de los DSL ha provocado un aumento del número de lenguajes disponibles para construir una aplicación, provocando el problema de la *cacofonía* de lenguajes, aunque en [30] se comenta que este problema suele pasar desapercibido, ya que en la práctica los desarrolladores solo tratan con los DSL asociados a un dominio en concreto. Por otro lado, dado que el desarrollo de una aplicación para un determinado dominio requiere utilizar varios DSL, son necesarias recomendaciones y entornos que faciliten su creación [91]. En este sentido, en [42, 47, 105] se exponen un conjunto de recomendaciones y técnicas para facilitar la definición de nuevos DSL.

Las recomendaciónes propuestas en [105] pueden ser clasificadas en tres tipos, según estén destinadas al diseño, la creación de procesadores o al proceso de creación de un DSL. En cuanto al diseño, se recomienda que los desarrolladores comiencen con la creación de un DSL para un dominio técnico en vez de un dominio del negocio ya que de esta manera pueden aplicar sus conocimientos en un dominio más cercano. Además, también recomienda que el lenguaje permita expresar qué se debe hacer pero no cómo, así como considerar la evolución del lenguaje teniendo en cuenta puntos de extensión o la adaptación a nuevos

requisitos. En cuanto a la creación de procesadores del DSL, destaca la recomendación de utilizar transformaciones *m2m* para simplificar los generadores y evitar la modificación del código generado. Finalmente, las recomendaciones relacionadas con el proceso de creación del lenguaje proponen el uso de un proceso iterativo y la implementación del lenguaje conforme se realiza el análisis del dominio para facilitar su comprensión.

Por otro lado, en [42] se presenta un conjunto de recomendaciones extraídas a partir de la experiencia en el desarrollo de 76 casos reales de definición y uso de DSL gráficos. Entre ellas se puede destacar el uso una notación simple, evitar el uso de perfiles o la importancia de considerar la evolución del lenguaje. En general, el estudio remarca la necesidad de tener un buen conocimiento del dominio del problema y ofrecer una notación y herramientas del lenguage adaptadas a los usuarios.

En [47] se identifican un conjunto de requisitos para la definición de DSL de calidad, los cuales serán comentados posteriormente en la sección 3.3.3.

En la creación de un nuevo DSL normalmente colaboran diferentes tipos de *stakeholders*. Aunque existen varios tipos de clasificaciones [45, 47, 112], en general, siempre se distinguen al menos tres perfiles:

- *Desarrolladores del DSL.* Se encargan de la definición, diseño e implementación del DSL así como de las herramientas necesarias para su utilización.
- *Usuarios de DSL.* Utilizan el DSL y las herramientas asociadas. También realizan una labor de *feedback*, permitiendo la evolución y mejora del lenguaje.
- *Expertos del dominio.* Tienen el conocimiento experto en el dominio del DSL. Normalmente son también los usuarios del DSL, aunque otras veces este perfil coincide con los desarrolladores.

Un proceso de creación de DSL consiste en cuatro etapas fundamentales [50]: estudio de viabilidad, análisis del dominio, diseño e implementación. El análisis del dominio culmina con la identificación de los conceptos del lenguaje y las relaciones entre ellos, y para grandes dominios puede requerir el uso de técnicas como FODA [41]. En cuanto al diseño es preciso distinguir entre DSL gráficos y textuales, que depende principalmente de las construcciones del lenguaje y del grupo de usuarios al que va destinado el DSL, tal y como se comentará a continuación. En la siguiente sección se definen los elementos básicos de un DSL y luego se presentan las principales técnicas de implementación.

3.3.1. Elementos de un DSL

Un DSL está compuesto por, al menos, tres elementos [43, 45, 91]: sintaxis abstracta, sintaxis concreta y semántica. Opcionalmente, una definición de DSL también puede incluir la especificación de los servicios que utiliza y ofrece, los cuales se definen normalmente en forma de interfaces.

La **sintaxis abstracta** de un DSL define sus conceptos así como las relaciones entre ellos, junto con las reglas que determinan cuando el modelo está bien formado. Dentro del DSDM, la sintaxis abstracta es normalmente expresada utilizando un metamodelo.

La **sintaxis concreta** define una notación para la sintaxis abstracta, que puede ser textual, gráfica o híbrida. La notación es un aspecto crucial en la definición de un DSL, ya que

es la parte *visible* y con la que los usuarios deberán lidiar cuando utilicen el DSL. Por ello es muy importante que sea clara, comprensible y adecuada al dominio de aplicación [105]. La notación más adecuada para un DSL es aquella que permite una correcta alineación entre la representación de los conceptos y sus abstracciones en el lenguaje [105]. Debido que normalmente el desarrollo de sintaxis textuales es más fácil y rápido que las gráficas, se recomienda utilizar inicialmente una definición textual del DSL y luego plantearse su conversión a gráfica. Por otro lado, también es muy importante considerar el grupo de usuarios al que va destinado el DSL. Así, las sintaxis textuales suelen ser preferidas por desarrolladores mientras que las sintaxis gráficas lo son para las personas con pocos conocimientos en software [42].

Una notación gráfica es especialmente útil para representar las relaciones entre elementos y aspectos como secuencias de eventos o flujos de datos. Sin embargo, utilizar una notación gráfica para la representación de expresiones puede ser muy complejo. Las notaciones textuales son más adecuadas para la representación de conjuntos de elementos (sentencias o expresiones, por ejemplo) donde existen pocas referencias entre sus elementos o, al menos, no es crucial su representación. En algunos casos puede plantearse el uso de notaciones híbridas. Por ejemplo, en el caso de un DSL para definir máquinas de estados, podría definirse un DSL híbrido cuya parte gráfica permitiera configurar los estados y sus transiciones y luego una parte textual para la definición de las condiciones de guarda.

Por otro lado, también es importante valorar las herramientas disponibles para ambos tipos de notaciones, algunas de las cuales se presentan la sección 3.3.2. La implementación de editores para DSL textuales es, por lo general, más fácil que para DSL gráficos. Además, en el caso de las notaciones textuales, los editores pueden incorporar facilidades como el resaltado de sintaxis o el autocompletado.

La **semántica** de un DSL hace referencia al significado que tienen los constructores del lenguaje y a cómo interpretar las sentencias compuestas por dichos constructores. Al igual que ocurre en los lenguajes de programación, los DSL deben ofrecer mecanismos para permitir al desarrollador comprender el significado y la finalidad de sus elementos.

Existen varias formas para describir la semántica de un DSL [45]: (1) denotacional, (2) operacional, (3) pragmática y (4) traslacional. La semántica denotacional utiliza formalismos matemáticos para describir la semántica de un DSL. Las descripciones denotacionales utilizan conceptos y operadores de estos formalismos que, aunque potentes, son normalmente complejos. La aproximación operacional describe la semántica de un DSL por medio de secuencias de operaciones que describen cómo una máquina abstracta o virtual ejecuta los programas del DSL. Normalmente se utilizan máquinas de estados para ilustrar el significado de las sentencias del lenguaje, mostrando cómo varía el estado del sistema al aplicarlas. La aproximación pragmática es quizás la aproximación más simple, ya que para explicar el significado de las sentencias de un DSL se muestra un conjunto de ejemplos y los resultados ofrecidos al ejecutar el lenguaje. Sin embargo, las descripciones pragmáticas pueden dificultar la comprensión del lenguaje por parte de los desarrolladores, ya que deben deducir la semántica a partir de dichos ejemplos. Finalmente, la aproximación traslacional define la semántica de un DSL por medio de su traducción a otro lenguaje de programación cuya semántica es bien conocida o, al menos, conocida en la comunidad de desarrolladores

del DSL. Es importante destacar que el lenguaje utilizado para la traducción debe ofrecer construcciones con la misma o mayor potencia semántica que las del DSL.

Aunque cualquiera de las aproximaciones anteriores son válidas para especificar la semántica de un DSL, la inmensa mayoría de las veces se utiliza una aproximación traslacional para la definición de DSL en el ámbito de la ingeniería software, traduciendo a un lenguaje de programación general como Java o C# [42, 105]. Esta aproximación puede ser vista como un compilador del DSL a un lenguaje de programación, el cual es a su vez interpretado o compilado a código máquina.

Otros elementos que pueden formar parte de una definición de un DSL son las especificaciones que describen cómo interactuar con otros DSL, esto es, las interfaces de los servicios que ofrece y puede utilizar el DSL. Estas especificaciones son especialmente importantes en el ámbito de la composición de DSL, donde cada DSL-componente ofrece una serie de servicios que son utilizados por el resto, aunque la investigación en esta línea es todavía muy escasa.

3.3.2. Implementación de un DSL

Un DSL puede ser implementado como un DSL interno, DSL externo o haciendo uso de herramientas de definición de DSL. Además, si el DSL es ejecutable, dependiendo de la implementación que se adopte también deberá tenerse en cuenta el modo de ejecución del DSL, la cual puede ser interpretada o compilada a otro lenguaje de programación. A continuación se describirá cada una de estas aproximaciones para implementar un DSL y su uso en el paradigma del DSDM.

Un **DSL interno** o embebido es una forma particular de utilizar un lenguage de programación de propósito general, el cual se denomina lenguaje anfitrión. Este tipo de DSL aprovecha la sintaxis y propiedades del lenguaje anfritrión para crear el lenguaje por medio de una librería de métodos o funciones que proporcionan la sintaxis concreta del DSL. Por este motivo los lenguajes más utilizados son los dinámicos orientados a objetos y funcionales, como Ruby o Lisp, ya que sus sintaxis facilita la personalización de la estructura del lenguaje [85, 86], lo que provoca que este tipo de lenguajes sean textuales. Ejemplos de DSL internos son RubyTL [87], un lenguaje de transformación de modelos embebido en Ruby; Kiama [72], un lenguage de transformación de programas embebido en Scala; y SWUL [22], un lenguaje embebido en Java para definir interfaces gráficas en Swing.

Normalmente, los DSL internos se ejecutan de la misma forma que su lenguaje anfitrión, es decir, si el lenguaje anfitrión es interpretado, el DSL también será interpretado. Existen dos estrategias de implementación de un DSL interno dependiendo de si se requiere utilizar modelos de sintaxis abstracta. Cuando el programa escrito en el DSL puede ser ejecutado directamente conforme se van llamando a los métodos definidos en el lenguaje anfitrión, no es necesario el uso de modelos de sintaxis abstracta. Por otro lado, en aquellos casos en los que sea necesario realizar un análisis previo al programa, la ejecución se divide en dos fases: (1) obtener el modelo de sintaxis abstracta y (2) ejecutar dicho modelo. Es importante destacar que para la obtención del modelo de sintaxis abstracta se utiliza el lenguaje anfitrión mientras que la segunda fase puede estar implementada en el mismo

lenguaje anfitrión o en otro diferente. Por ejemplo, en el caso de RubyTL la ejecución se divide en dos fases, las cuales utilizan el mismo lenguaje de ejecución.

Un **DSL externo** se implementa de forma independiente a cualquier lenguaje de programación, por lo que el desarrollador es libre de utilizar cualquier tipo de sintaxis. Algunos ejemplos de DSL externos son SQL, Awk y HTML, aunque realmente son la gran mayoría.

Para implementar DSL externos normalmente se utiliza el formalismo de las gramáticas para definir la sintaxis concreta, a partir de las cuales se generan los *parsers* encargados de reconocer los programas del DSL. Estos *parsers* construyen un modelo de sintaxis abstracta a partir del cual se obtiene el modelo de sintaxis abstracta, cuya ejecución puede ser interpretada o compilada. Según se comenta en [105], los DSL externos compilados ofrecen normalmente mayor rendimiento y soporte multiplataforma mientras que los interpretados reducen el tiempo de desarrollo y facilitan la evolución del lenguaje.

A diferencia de un DSL interno, la implementación de DSL externos requieren un mayor esfuerzo de implementación debido a que es necesario crear toda la infraestructura del lenguaje. Sin embargo, la incorporación de mecanismos como el manejo de errores, análisis estático o verificación es más fácil en el caso de DSL externos, ya que el desarrollador tiene más libertad para la implementación de dichos mecanismos.

Las **herramientas de definición de DSL** permiten generar automáticamente la infraestructura del lenguaje a partir de su sintaxis abstracta. Este tipo de herramientas permiten crear tanto DSL textuales como gráficos. Las herramientas destinadas a la creación de DSL textuales se clasifican en: (1) orientadas a la gramática, si el proceso de definición parte de la especificación de la sintaxis concreta por medio de una gramática anotada para guiar la generación de la sintaxis abstracta (p. ej., Xtext [111] o EMFText [36]); y (2) orientadas al metamodelo, si el proceso de definición del DSL parte de la especificación de la sintaxis abstracta como un metamodelo (p. ej., TCS [79]). Este tipo de herramientas generan automáticamente un *parser* para obtener modelos conformes al metamodelo de sintaxis abstracta a partir de la definición textual y el generador para aplicar el proceso contrario. Por otro lado, en cuanto a las herramientas para DSL gráficos se pueden destacar GMF [65], las DSLTools [18] o MetaEdit [73], las cuales en vez de un *parser* generan un editor gráfico específico para el DSL.

Finalmente, los entornos de definición de DSL (*language workbenches* en la terminología utilizada en [30]) están emergiendo como una solución completa para facilitar la creación y gestión de DSL. Estos entornos ofrecen herramientas para definir DSL así como un conjunto de utilidades para trabajar los programas del DSL, como por ejemplo generadores de código o vistas. De esta forma, el entorno se convierte en la herramienta principal para trabajar con el DSL. Algunos ejemplos de este tipo de herramientas son MPS [55], Spoofax [75] y OpenArchitectureWare [58].

3.3.3. Requisitos de calidad en un DSL

Con el auge del DSDM, el uso de DSL en los procesos de creación de software se ha incrementado considerablemente y juegan un papel fundamental durante todo el desarrollo de aplicaciones. De esta forma, los atributos de calidad de un DSL tienen un impacto directo

en la calidad del proceso de desarrollo así como del software creado. En [47] se identifican un conjunto de requisitos de calidad deseables en cualquier DSL, los cuales son:

- **Conformidad**. Los conceptos y elementos del lenguaje deben corresponder a conceptos importantes del dominio.
- **Ortogonalidad**. Cada concepto del dominio debe poder representarse por uno y sólo un elemento del lenguaje.
- **Soporte**. Todo DSL debe ofrece un conjunto de herramientas para facilitar su uso (p. ej., para crear, editar y depurar programas del DSL).
- **Integridad**. Cualquier DSL debe poder ser integrado con otras herramientas y/o lenguajes fácilmente.
- **Extensibilidad**. Un DSL debe tener la capacidad de poder incorporar nuevos elementos. Este requisito puede influir en la conformidad y ortogonalidad del lenguaje.
- **Longevidad**. Al igual que a cualquier lenguaje de programación, a un DSL también se le requiere tener una vida útil extensa para justificar los costes de implementación.
- **Simplicidad**. Un DSL debe ser suficientemente simple para expresar los conceptos del dominio fácilmente.
- **Calidad**. Un DSL debe ofrece mecanismos para construir sistemas software de calidad, esto incluye mecanismos para mejorar la fiabilidad o la seguridad.

También se identifican requisitos adicionales para favorecer la calidad de los DSL:

- **Escalabilidad**. En algunos casos, es deseable que un DSL sea escalable, por ejemplo, que ofrezca mecanismos para tratar con definiciones del DSL de tamaño considerable.
- **Usabilidad**. Este requisito hace referencia a conceptos como economía, accesibilidad o comprensión del lenguaje, las cuales facilitan la aplicación del DSL.

3.4. Modernización de Software Dirigida por Modelos

Las técnicas del paradigma del Desarrollo Dirigido por Modelos no son sólo aplicables para crear nuevos sistemas software sino también pueden ser utilizadas en tareas de reingeniería o modernización de software. Así, la Modernización de Software Dirigida por Modelos (*Model-Driven Modernization*, MDM) ha emergido como una nueva disciplina donde los modelos guían todo el proceso de reingeniería [9, 104].

En un proceso basado en DSDM, los desarrolladores definen inicialmente los modelos del sistema que posteriormente son transformados para crear determinadas partes del sistema software. Sin embargo, en un proceso de reingeniería o modernización, se parte de un sistema software ya existente, por lo tanto, es necesario que los modelos iniciales del proceso se obtengan a partir de los artefactos software para entonces aplicar las técnicas del DSDM. La Figura 3.4 muestra las tres etapas fundamentales de un proceso de reingeniería o modernización utilizando el modelo de herradura [88]. Existe una primera fase de *ingeniería inversa* encargada de obtener modelos de alto nivel de abstracción a partir de los artefactos del sistema software origen. A continuación, durante la fase de *reestructuración*, estos

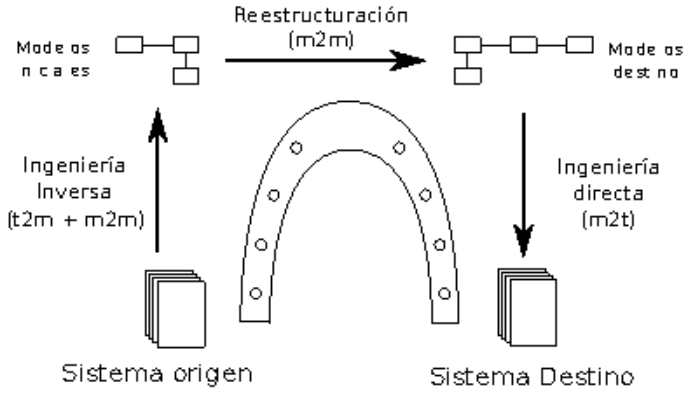


Figura 3.4: Proceso de modernización básico.

modelos extraídos son transformados en otros que conformen a la arquitectura destino, los cuales son finalmente transformados en la fase de *ingeniería directa* para generar los artefactos del nuevo sistema.

La complejidad de las transformaciones involucradas en las tareas de ingeniería inversa dependen del grado de abstracción de los metamodelos a los que deben conformar los modelos a extraer. Por este motivo, la fase de ingeniería inversa suele dividirse a su vez en dos etapas para obtener los modelos a partir de los artefactos del sistema software. El primer paso siempre consiste en la extracción de modelos a partir de los artefactos del sistema por medio de transformaciones *t2m*, las cuales transforman diferentes tipos de artefactos (p. ej., código, datos o xml) en modelos de bajo nivel de abstracción. A continuación, por medio de transformaciones *m2m* se extrae conocimiento de dichos modelos simples para abstraer y componer nuevos elementos de información más abstractos. La fase de reestructuración aplica transformaciones *m2m* para obtener modelos de la nueva arquitectura a partir de los modelos extraídos y finalmente transformaciones *m2t* se encargan de generar los artefactos del sistema destino.

3.4.1. ADM

Tras la propuesta de MDA, el OMG lanzó en 2003 la iniciativa ADM (*Architecture Driven Modernization*) [38] con el objetivo de definir un conjunto de metamodelos estándar para la modernización. En [100] se justifica la necesidad de metamodelos estándar para facilitar la interoperabilidad entre herramientas de modernización. Mientras MDA puede ser considerada la propuesta de OMG para la ingeniería directa dirigida por modelos, ADM es su propuesta para la reingeniería dirigida por modelos (el propio nombre es un juego para connotar lo inverso a MDA).

La iniciativa ADM incluye la definición de siete metamodelos, aunque solamente cuatro de ellos están disponibles actualmente: versión 1.3 del metamodelo KDM (*Knowledge Discovery Metamodel*) [52], versión 1.0 del metamodelo ASTM (*Abstract Syntax Tree Metamodel*) [51] y versiones beta de los metamodelos SMM (*Software Metrics Metamodel*)

[53] e IPMSS (*Implementation Patterns Metamodel for Software Systems*). Los otros tres están todavía en desarrollo (visualización, refactorización y transformación).

Los metamodelos ASTM y KDM se complementan para el modelado de la sintaxis y semántica de los sistemas software, respectivamente. Mientras que ASTM permite representar el código fuente del sistema en forma de árboles de sintaxis abstracta (*Abstract Syntax Tree*, AST), KDM ofrece mecanismos para representar la información semántica de dicho sistema, desde la información relativa al código fuente hasta aquella de mayor nivel de abstracción como los eventos de la interfaz, información de plataforma o reglas de negocio.

Un modelo ASTM representa las sentencias del código fuente y refleja la estructura gramatical de un lenguaje de programación. Sin embargo, los modelos ASTM no son exactamente árboles AST ya que también incluyen información semántica básica (p. ej., información de ámbito de variables y funciones) que produce referencias cruzadas entre elementos del árbol. Estas características los convierten en grafos de sintaxis abstracta.

Para facilitar la reutilización, el metamodelo ASTM está organizado en dos partes: (1) el metamodelo GASTM (*Generic Abstract Syntax Tree Metamodel*), que contiene elementos comunes de la mayoría de lenguajes de programación; y (2) los metamodelos SASTM (*Specific Abstract Syntax Tree Metamodel*), que permiten representar los conceptos propios de cada lenguaje de programación.

Por otro lado, KDM es el principal metamodelo de la iniciativa ADM ya que facilita el intercambio de los metadatos utilizados para representar un sistema software, favoreciendo la interoperabilidad. El metamodelo de KDM es muy grande y está organizado en diferentes dominios o paquetes que permiten definir vistas arquitecturales (p. ej., vista de la plataforma, de la interfaz del usuario o de los datos). Estos paquetes están agrupados en cuatro capas de abstracción para mejorar la modularidad y la separación de aspectos: infraestructura, elementos de programa, recursos y abstracciones. Además, debido a que algunas tareas de modernización (p. ej., análisis estático) requieren hacer uso de información semántica precisa de las sentencias del código, el metamodelo KDM incorpora un paquete adicional denominado micro-KDM, el cual permite representar la semántica de las principales sentencias de los lenguajes de programación.

La capa de *infraestructura* proporciona los elementos básicos para la construcción de los modelos KDM, como las entidades, relaciones entre elementos y trazabilidad. La capa de *elementos de programa* permite representar los artefactos software del sistema a nivel de implementación. La capa de *recursos runtime* representa elementos de mayor nivel de abstracción ya que están ligados a la plataforma de ejecución, como por ejemplo la interfaz de usuario. Finalmente la capa de *abstracciones* trabaja a un nivel de abstracción todavía mayor, representando el conocimiento específico del dominio, como las reglas de negocio o el conocimiento arquitectural. La Figura 3.5b muestra gráficamente la organización de las capas y paquetes de KDM.

En KDM y ASTM se definen diferentes niveles de compatibilidad. Cada nivel especifica el tipo de modelos que una herramienta debe ser capaz de interpretar, exportar e importar para soportar dicho nivel. En KDM, el nivel 0 asegura la compatibilidad con los paquetes de infraestructura y elementos del programa. El nivel 1 extiende al nivel 0 y está definido

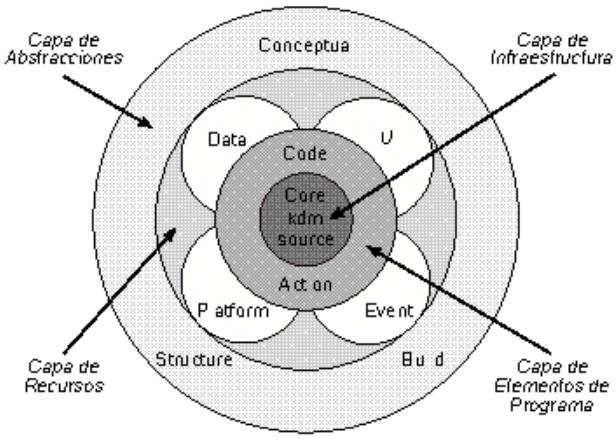


Figura 3.5: Organización de paquetes en KDM.

particularmente para cada paquete de las capas de recursos, abstracciones y micro-KDM. El nivel 2 se define como la compatibilidad a nivel 1 para todos los paquetes de KDM. Por otro lado, en ASTM, se dice que una herramienta es compatible a nivel 0 cuando es capaz de interpretar, exportar e importar modelos conformes a los metamodelos GASTM y SASTM, mientras que será compatible a nivel 1 cuando soporte los elementos semántico de ASTM.

Desde la perspectiva de ADM, los modelos ASTM son la principal fuente para obtener la información necesaria para crear modelos KDM, que a su vez son la base para realizar las actividades ligadas a la mayoría de escenarios de modernización y para obtener otros modelos ADM. Por ejemplo, en el capítulo 6 se mostrará un caso de estudio de extracción de modelos KDM a partir de modelos ASTM.

SMM es un metamodelo que puede representar tanto métricas como mediciones. Incluye un conjunto de elementos para describir las métricas sobre cualquier modelo así como para representar las mediciones de dichas métricas.

Finalmente, el metamodelo IPMSS se encuentra en la fase final de su aprobación y publicación. Este metamodelo está enfocado a modelar patrones en el código fuente, pudiendo ser utilizado en procesos de modernización de código.

En [101] se establecen un conjunto de escenarios de modernización en los que sería posible aplicar ADM, actuando de plantillas para ilustrar y facilitar la aplicación de los metamodelos de la iniciativa. Algunos escenarios son: cambiar el lenguaje de programación de una aplicación, migrar plataformas, integrar aplicaciones, o transformar a una arquitectura SOA. En general, en la mayoría de los escenarios es necesario extraer modelos a partir del código o datos que utiliza una aplicación. También se señala que con frecuencia un proyecto concreto de modernización abarca más de un escenario.

4

Espacios Tecnológicos

Solo eres una compleja serie de dispositivos electrónicos. Yo produzco un sonido,
el sonido entra en tus bases de datos, lo compara con cierta palabra
y tu programa escoge una respuesta automatizada.
Génesis, Bernand Beckett

Para construir aplicaciones software se utilizan normalmente un gran abanico de tecnologías, por ejemplo, una aplicación de compra de artículos por internet es implementada en un determinado lenguaje de programación pero además utilizaría varias tecnologías para implementar las diferentes capas de su arquitectura y almacenar los datos (tecnologías web, XML, bases de datos relaciones, etc.). El éxito del Desarrollo Dirigido por Modelos ha motivado la integración de esta tecnología con otras existentes como las gramáticas de los lenguajes de programación, XML o las bases de datos, lo cual implica convertir en modelos modelos los artefactos de dichas tecnologías. En este capítulo se introducirá el concepto de *espacio tecnológico* como forma de definir una tecnología y, a continuación, se presentarán los espacios tecnológicos que se han tratado durante el trabajo de esta tesis. Finalmente se introducirá el concepto de *puente* entre espacios tecnológicos para definir los procesos de extracción de modelos y se ilustrará con los espacios tecnológicos considerados.

4.1. Definición

En la actualidad, el diseño de una aplicación requiere el estudio de una amplia variedad de alternativas tecnológicas, cada una de las cuales tiene asociadas un conjunto de conceptos, métodos, técnicas y herramientas. Es responsabilidad del grupo de desarrollo evaluar y decicidir cuáles son las más adecuadas para la creación de la aplicación. Sin embargo, tomar esta decisión no siempre es trivial, ya que la mayoría de las veces una tecnología no ofrece todas las ventajas deseables. Muchas veces la solución consiste en utilizar lo mejor de diferentes tecnologías.

El concepto de espacio tecnológico se presenta en [49] para servir de guía para decidir qué tecnologías utilizar en el desarrollo de una aplicación. Un *espacio tecnológico* es un contexto de trabajo que tiene asociado un conjunto de conceptos, métodos, técnicas y herramientas. Además, cada espacio tecnológico está comúnmente asociado a una comunidad de usuarios que comparten conocimientos, *know-how*, apoyo académico, literatura e incluso conferencias. Por ejemplo, uno de los espacios tecnológicos más conocidos es el *grammarware*, que corresponde al contexto de trabajo que surge alrededor de los programas escritos en un determinado lenguaje de programación, el cual está definido por una gramática, tal y como se describirá más adelante. Normalmente, un espacio tecnológico se construye en torno a una pareja de conceptos básicos que se relacionan, por ejemplo, para el caso del *grammarware*, este par de conceptos serían *programa* y *gramática*. Otro ejemplo de espacio tecnológico es el *xmlware*, el cual representa al contexto de trabajo de la tecnología XML, donde los conceptos básicos son *documento* y *esquema XML*.

Otra idea importante de los espacios tecnológicos es que ninguno de ellos está aislado. Así, se pueden construir *puentes* entre diferentes espacios tecnológicos los cuales pueden ser bidireccionales o unidireccionales. Estos puentes permiten representar los elementos de un espacio tecnológico en otro con el objetivo de utilizar la tecnología que mejor se adapte al problema, facilitando además la interoperabilidad.

A continuación se describirán los espacios tecnológicos tratados en este trabajo. En primer lugar se describirá el *modelware*, que es el espacio tecnológico principal sobre el que se articulan las herramientas construidas en esta tesis y cuyos conceptos básicos y técnicas ya han sido descritos en el capítulo anterior. A continuación, se presentarán el resto de espacios tecnológicos tratados en este trabajo, a saber: *grammarware*, *dataware* y *apiware*. Para cada uno de estos espacios tecnológicos se expondrán los conceptos básicos, principales formalismos y técnicas, así como algunas herramientas asociadas.

4.2. *Modelware*

El espacio tecnológico denominado *modelware* hace referencia al contexto de trabajo que existe alrededor del paradigma del DSDM, el cual ya se ha introducido en la sección 3.1. En el *modelware*, la pareja de conceptos básicos son *modelo* y *metamodelo*. Tal y como se ha comentado en la sección 3.1, un modelo representa uno o varios aspectos de un sistema software y se describe por medio de un DSL, el cual se define mediante un metamodelo, como se ha explicado anteriormente. Los elementos de un modelo son instancias de los elementos del metamodelo, de esta forma, decimos que el modelo es conforme a su metamodelo. Por otro lado, los metamodelos son normalmente descritos por medio de un lenguaje de metamodelado, es decir, los elementos del metamodelo son instancias de los elementos del meta-metamodelo, por lo que un metamodelo a su vez es conforme a un meta-metamodelo.

La arquitectura de cuatro capas definida por el OMG [97] se utiliza comúnmente para definir las relaciones de instanciación entre los conceptos básicos de los espacios tecnológicos. Esta arquitectura tiene sus orígenes en propuestas previas y ha suscitado controversia sobre el significado de las capas y de las relaciones entre ellas, así como de las capas precisas

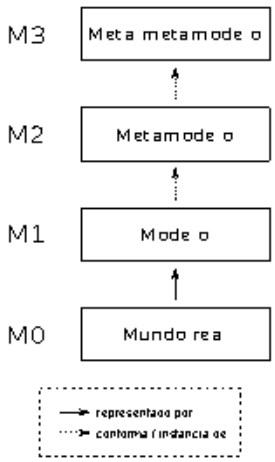


Figura 4.1: Arquitectura de cuatro capas en *modelware*.

para un determinado espacio tecnológico [7, 28]. A lo largo de este documento adoptaremos la aproximación presentada en [8], la cual describe la arquitectura de cuatro capas del OMG como una arquitectura de 3+1 capas. En esta arquitectura, el primer nivel (M0) corresponde a los elementos del sistema o las cosas del mundo real y está relaciona con los elementos del siguiente nivel, el cual los representa, mientras que entre los elementos del resto de niveles (M1, M2 y M3) existe una relación de instanciación. Esta arquitectura se utilizará para caracterizar los espacios tecnológicos considerados con la intención de facilitar la identificación de niveles de abstracción similares de forma parecida a como se realiza en [49], así como la definición de puentes entre ellos, como se mostrará más adelante.

La Figura 4.1 muestra la arquitectura de cuatro capas del *modelware*. Según esta arquitectura, los elementos del sistema se sitúan en el nivel M0 y son representados por modelos, los cuales se sitúan en el nivel M1. Los modelos son conformes a los metamodelos, que se definen en el nivel M2 y que a su vez son conformes a los meta-metamodelos, los cuales se definen a nivel M3. Los meta-metamodelos se definen a sí mismos, por lo que el nivel M3 es el último de la arquitectura.

Algunas de las técnicas y formalismos del *modelware* ya ha sido presentados en la sección 3.1. La aplicación de transformaciones de modelos es una de las técnicas más importantes de este espacio tecnológico, las cuales pueden ser *m2m*, *m2t* y *t2m* mientras que el metamodelado es el principal formalismo del *modelware*.

Un área activa del *modelware* es la construcción de herramientas robustas y de fácil uso que permitan su aplicación industrial. Entre ellas podemos destacar los *toolkits* que reúnen las herramientas básicas (definición de metamodelos y motores de transformación), como por ejemplo OpenArchitectureWare [58] y AGE [85]; herramientas que soportan MDA, como AndroMDA [2]; herramientas de definición de DSL, como EMFText [36] o Xtext [111]; herramientas de visualización de modelos, como la incorporada en el *framework* de modernización dirigida por modelos MoDisco [11]; o repositorios de modelos, como CDO [62].

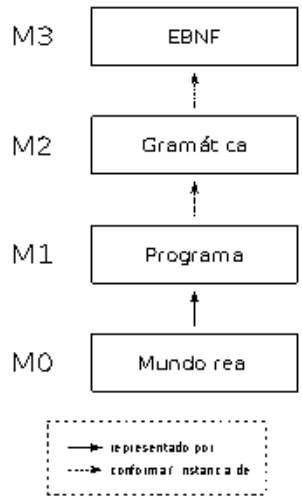


Figura 4.2: Arquitectura de cuatro capas en *grammarware*.

4.3. *Grammarware*

El espacio tecnológico de las gramáticas o de las sintaxis es denominado *grammarware* y es uno de los contextos de trabajo con más antigüedad y fundamentación teórica. Sus inicios se sitúan en el momento de la creación de los primeros lenguajes de programación, los cuales permitieron abstraerse del lenguaje máquina ofreciendo abstracciones más cercanas al dominio del problema.

En *grammarware*, los conceptos básicos son *programa*, el cual es escrito en un lenguaje de programación determinado, y *gramática*, la cual especifica formalmente la estructura de dicho lenguaje. Además, las gramáticas a su vez son conformes a lo que se denomina meta-gramática y que normalmente corresponde con EBNF. La arquitectura de cuatro capas adaptada a los conceptos y relaciones del *grammarware* se muestra en la Figura 4.2. Como se puede apreciar, el nivel M0 corresponde a una determinada ejecución de un programa, el cual es un fenómeno que ocurre en un instante en el mundo real. Por otra parte, las definiciones de los programas están situados en el nivel M1, las gramáticas en el nivel M2 y las meta-gramáticas en el nivel M3.

Las técnicas y formalismos de este espacio tecnológico son muy numerosos. Por ejemplo, en *grammarware* se utilizan técnicas como *parsing* y optimización de gramáticas, o formalismos como gramáticas atribuidas, predicados semánticos o gramáticas libres de contexto, entre otros. En cuanto a las herramientas, existen también un gran número de ellas, la mayoría muy maduras: compiladores de programas para generar código de bajo nivel, como GCC [32]; generadores de *parsers* a partir de una definición gramátical, como ANTLR [61]; o visualizadores de árboles de sintaxis, como la herramienta incorporada en ANTLRWorks [6].

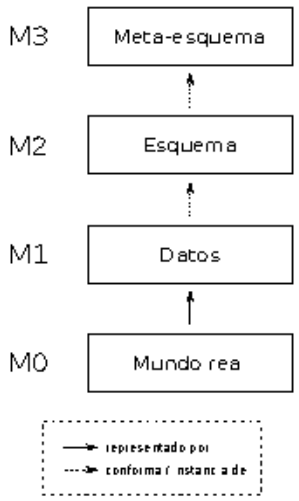


Figura 4.3: Arquitectura de capas en *dataware*.

4.4. *Dataware*

Este espacio tecnológico hace referencia al contexto de trabajo que surge alrededor de las bases de datos. Al igual que el *grammarware*, el *dataware* también ha sido objeto de investigación durante muchos años. Su principal objetivo es la representación, administración y gestión de grandes volúmenes de datos en sistemas gestores de bases de datos. Existen varios tipos de bases de datos, por ejemplo, relacionales, objeto-relacionales u orientadas a objetos. Sin embargo, las bases de datos relacionales son las que más han sido adoptadas por la industria por lo que juegan un papel fundamental en el *dataware*. Por este motivo, en esta tesis utilizamos el término *dataware* para referirnos a la parte del espacio tecnológico dedicada a la tecnología de las bases de datos relacionales.

En *dataware* se gestiona un conjunto de datos que son conformes a un determinado esquema de la base de datos. De esta forma, los conceptos básicos de este espacio tecnológico son *dato*, que es representado por una tupla en una tabla de la base de datos, y *esquema relacional* de la base de datos. Además, los esquemas de bases de datos conforman a su vez al conjunto de reglas específicas para su definición que, en el caso de los sistemas gestores de bases de datos relacionales son determinados por el modelo Entidad/Relacion [13], el cual podría denominarse meta-esquema. La Figura 4.3 muestra la arquitectura de capas para los elementos del *dataware*. En esta arquitectura, los conceptos del mundo real se sitúan a nivel M0, los cuales son representados por los datos a nivel M1. Los datos son conformes al esquema de la base de datos, que se definen a nivel M2 y que a su vez son conformes al meta-esquema definido en el nivel M3.

En el *dataware* existe también un gran número de técnicas, como por ejemplo la normalización de esquemas de base de datos o el diseño utilizando el modelo Entidad/Relación, así como diferentes formalismos, como el álgebra relacional o el cálculo relacional, sobre los que se fundamenta toda la teoría de las bases de datos relacionales. En cuanto a las

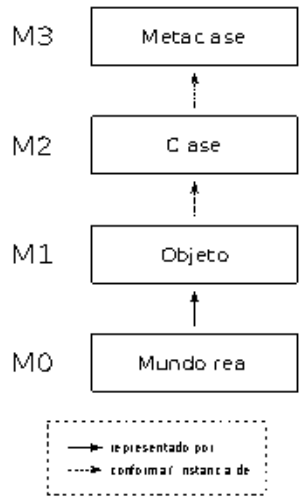


Figura 4.4: Arquitectura de capas en *apiware*.

herramientas, se puede destacar, por ejemplo, las herramientas para el diseño de bases de datos o los gestores de bases de datos.

4.5. *Apiware*

Este espacio tecnológico hace referencia al contexto de trabajo que surge alrededor de la definición y uso de API. El concepto de API es esencial en ingeniería del software como una expresión del principio de ocultación de la información. Los servicios ofrecidos por una aplicación se describen normalmente utilizando varias API, que son una especificación que oculta los detalles de implementación e indica cómo utilizar correctamente los servicios así como el tipo de resultado que ofrecen. Existen diferentes formas de especificar un API, por ejemplo, pueden utilizarse interfaces para especificar librerías en orientación a objetos o descripciones WDSL cuando se utilizan servicios web. Por este motivo, el contexto de trabajo relacionado con los API es quizás uno de los más complicados de delimitar. En esta tesis, al utilizar el término *apiware* haremos referencia a la parte del espacio tecnológico dedicada a los API definidos en lenguajes de programación orientados a objetos.

Un API orientado a objetos está compuesto por un conjunto de clases que definen una determinada interfaz, la cual describe los métodos y propiedades que se ofrecen. Por lo tanto, los conceptos básicos de este espacio tecnológico son *objeto* y *clase*, donde cada objeto es instancia de su correspondiente clase del API. De la misma forma que en *modelware* decimos que un modelo es conforme a su metamodelo, en *apiware* esta relación también se podría considerar entre los objetos sus correspondientes clases: los elementos de un grafo de objetos en memoria son conformes a sus respectivas clases en el API. Por lo tanto, y aunque el concepto de metaclase no es soportado de la misma forma en todos los lenguajes orientados a objetos, es posible abstraer los detalles y definir una arquitectura similar a las presentadas anteriormente, tal y como se ilustra en la Figura 4.4. En esta arquitectura,

los conceptos del mundo real se sitúan a nivel M0 y son representado por los objetos, los cuales se definen en el nivel M1. Los objetos son conformes a las clases, las cuales se definen en el nivel M2 y son a su vez conformes a las metaclases definidas a nivel M3.

En el *apiware* se utilizan técnicas como el uso de patrones de diseño para su implementación o formalismos propios de la orientación a objetos como el uso de interfaces para la especificación del API. Finalmente, existen un conjunto de herramientas como por ejemplo aquellas utilizadas para definir las interfaces de las clases del API (utilizando diagramas de clase o lenguajes como IDL) o para generar automáticamente su documentación.

4.6. Puentes entre Espacios Tecnológicos

Tal y como se ha comentado anteriormente, durante el desarrollo de una aplicación es muy común hacer uso de varias tecnologías o espacios tecnológicos. Por ejemplo, una aplicación de comercio electrónico puede ser desarrollada con un lenguaje de programación como Java (*grammarware*), almacenar datos en una base de datos relacional (*dataware*) y hacer uso de API como JSF para mostrar información gráfica. En este tipo de aplicaciones, la información proveniente de un espacio tecnológico es normalmente utilizada por los elementos de otro espacio tecnológio, por ejemplo, los datos consultados por un programa (uso de *dataware* por parte del *grammarware*) o la visualización de determinada información utilizando un API gráfica (uso de *apiware* por parte del *grammarware*). Así, los puentes entre diferentes espacios tecnológicos son un concepto crucial para permitir la integración de información proveniente de diferentes espacios tecnológicos [28], facilitando la interoperabilidad.

Los puentes entre espacios tecnológicos también juegan un papel fundamental al aprovechar la mayor potencia expresiva de los modelos para representar artefactos de otros espacios tecnológicos. De esta forma, las técnicas y formalismos del *modelware* pueden ser aplicadas para manipular los modelos que representan a los artefactos del sistema. Por ejemplo, una aplicación podría obtener modelos desde bases de datos para su análisis o desde un API gestión y monitorización como JMX [92] para controlar un sistema software.

Otro escenario de aplicación se presenta en la Modernización de Software Dirigida por Modelos, en concreto, en la etapa de ingeniería inversa, donde los artefactos del sistema son analizados para obtener modelos que los representen. Cada uno de estos artefactos tendrá una naturaleza distinta dado que un sistema software integra normalmente información proveniente de diferentes espacios tecnológicos. Por lo tanto, la extracción de modelos puede plantearse como un puente entre el espacio tecnológicos al que pertenece el artefacto y el *modelware*, que es el espacio tecnológico donde se aplica la modernización.

Los puentes pueden ser bidireccionales o unidireccionales, dependiendo de las capacidades de abstracción de cada uno. Por ejemplo, mientras que cualquier gramática podría ser representada como un metamodelo [33, 44], no todo metamodelo puede ser representado como una gramática. Para definir puentes bidireccionales con el *modelware*, es necesario soportar dos operaciones: (1) extracción de modelos a partir de los artefactos software del espacio tecnológico considerado y (2) generación de dichos artefactos a partir de los modelos. Por ejemplo, un puente bidireccional entre el *grammarware* y el *modelware* debería

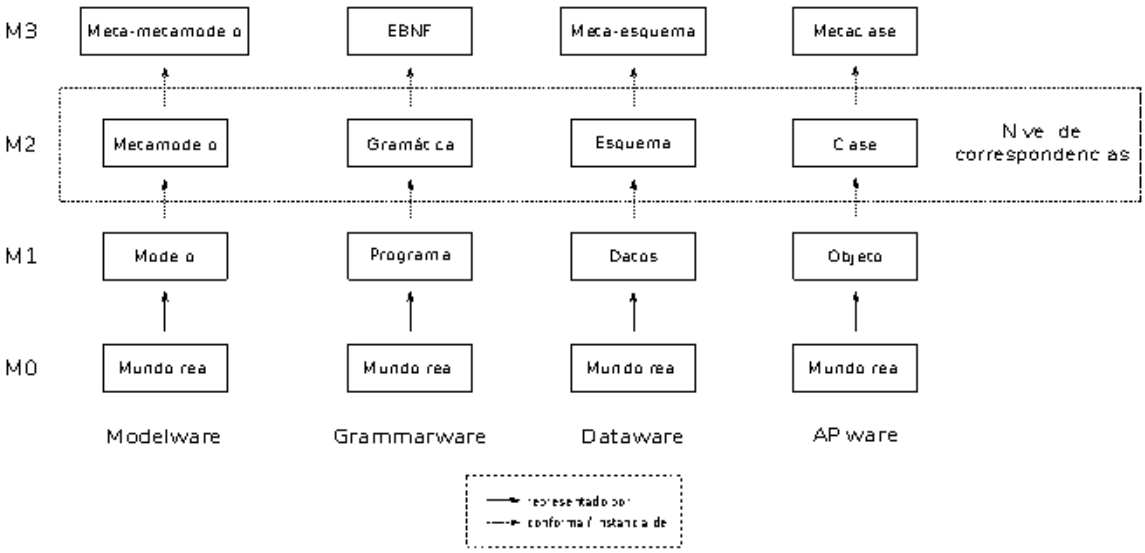


Figura 4.5: Arquitectura de cuatro capas para los espacios tecnológicos considerados en esta tesis y nivel de abstracción utilizado para definir las correspondencias de un puente entre ellos.

permitir la extracción de modelos a partir del código fuente así como la generación de dicho código a partir de los modelos.

Para definir un puente entre espacios tecnológicos deben definirse primero las correspondencias entre sus conceptos principales. Según la arquitectura de 3+1 capas utilizada, definimos un puente entre dos espacios tecnológicos a nivel M2 por medio de la especificación de las correspondencias entre sus conceptos, mientras que su ejecución se realiza a nivel M1. Por ejemplo, un puente entre *grammarware* y *modelware* define las correspondencias entre los elementos de la gramática y los del metamodelo mientras que dicho puente es aplicado a un programa para obtener un modelo conforme a dicho metamodelo. De forma similar, un puente entre el *dataware* y el *modelware* definiría las correspondencias entre los elementos del esquema de la base de datos (las tablas) y las metaclases, mientras que un puente entre el *apiware* y el *modelware* definiría las correspondencias entre las clases del API y las metaclases. La Figura 4.5 muestra las relaciones entre los niveles de los espacios tecnológicos considerados en el trabajo de esta tesis e indica el nivel utilizado para definir las correspondencias.

En esta tesis se presentarán tres DSL para abordar la definición de puentes entre el *modelware* y el *grammarware*, el *dataware* y el *apiware*. Por lo tanto, estos DSL permitirán definir las correspondencias entre los elementos de los espacios tecnológicos implicados.

5

Extracción de modelos desde el código fuente

Llegó al mundo como lo hacen la mayoría de los recién nacidos: gritando.
Las tropas fantasma, John Scalzi

La extracción de modelos a partir del código fuente escrito en un lenguaje de programación de propósito general (*General Purpose Language*, GPL) es una tarea requerida en la mayoría de escenarios de modernización. Por lo tanto, disponer de técnicas y herramientas eficientes para abordar este tipo de extracción de modelos es crucial. En este capítulo presentamos Gra2MoL, un lenguaje específico del dominio concebido para la extracción de modelos a partir del código fuente escrito en un GPL, pero que realmente es un lenguaje que permite definir transformaciones *t2m* para extraer modelos a partir de cualquier texto conforme a una gramática. En primer lugar, se presentarán las alternativas actuales para abordar la extracción de modelos a partir de código, comentando sus principales desventajas y las razones de haber decidido crear un nuevo DSL. A continuación se describirá este DSL, se mostrará un ejemplo de uso y luego un conjunto de escenarios de aplicación. Finalmente, se mostrarán las características principales del lenguaje y el grado de cumplimiento de los requisitos identificados en [47], los cuales pueden servir de guía para la construcción de un DSL de calidad.

5.1. Descripción del problema

En los procesos de extracción de modelos a partir del código GPL, el objetivo es obtener modelos que conforman a un determinado metamodelo a partir de código fuente que conforma a la gramática del GPL, tal y como se muestra en la Figura 5.1. El proceso es una tranformación T que tiene como entradas: un programa P que conforma con una gramática G y el metamodelo MM, al cual debe conformar el modelo resultante M. El proceso

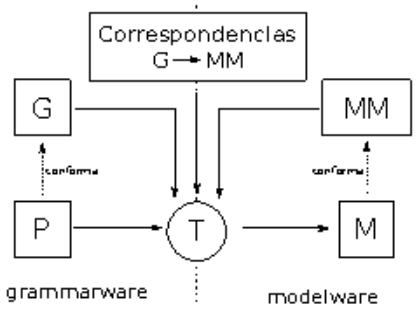


Figura 5.1: Proceso de extracción de modelos a partir del código fuente.

también requiere una definición de las correspondencias entre los elementos de la gramática y los del metamodelo que, como más adelante se expondrá, su naturaleza depende de la aproximación que se considere para aplicar dicho proceso. Nótese que estas correspondencias se definen a nivel M2 dentro de la arquitectura de cuatro capas del OMG comentada en la sección 4.6. El programa de entrada puede ser representado por un árbol de sintaxis abstracta (*Abstract Syntax Tree*, AST) o un árbol de sintaxis concreta (*Concrete Syntax Tree*, CST). A lo largo de este documento, utilizaremos el término *árbol de sintaxis* para referirnos tanto a un AST como a un CST, excepto cuando sea necesaria la distinción.

Tal y como se ha descrito en la sección 4.6, la extracción de modelos desde el código fuente puede ser vista como un puente unidireccional desde el *grammarware* al *modelware*. En este tipo de puentes, además de establecer las correspondencias entre los elementos de ambos espacios tecnológicos, es necesario disponer de mecanismos eficientes para recorrer el árbol de sintaxis que representa el código fuente ya que los elementos del modelo están normalmente compuestos por información dispersa por el código. La causa principal de esta dispersión es la forma de representar las referencias entre elementos del código fuente. Mientras que los modelos tienen estructura de grafo y cualquiera de sus elementos puede referenciar a otro explícitamente, en un árbol de sintaxis que representa código fuente, las referencias entre elementos se hacen implícitamente por medio de identificadores. De esta forma, transformar una referencia basada en identificadores en una referencia explícita en los modelos requiere recorrer el árbol de sintaxis para localizar el elemento referenciado. Por ejemplo, si un elemento del modelo representa una sentencia de llamada a una función que tiene como argumento una variable global, cierta información como el tipo de la variable o la declaración de la función está localizada fuera del ámbito del código donde la función es llamada (en [107] este tipo de transformaciones se denominan global-a-local).

Por lo tanto, la extracción de modelos desde el código fuente requiere principalmente de mecanismos para (1) definir las correspondencias entre los elementos de la gramática y los elementos del metamodelo y (2) recorrer el árbol de sintaxis de forma eficiente para obtener la información dispersa. A continuación se analizarán las principales aproximaciones existentes para llevar a cabo la extracción de modelos desde el código fuente, las cuales son: *parsers* dedicados, herramientas de definición de DSL, lenguajes de transformación de programas y lenguajes de transformación *m2m*.

5.2. Aproximaciones existentes

5.2.1. *Parsers* dedicados

La estrategia más común para abordar la extracción de modelos es la creación de *parsers* dedicados. Dada una gramática y un metamodelo destino, los *parsers* dedicados se encargan de analizar el código conforme a dicha gramática y a continuación generan el modelo destino conforme al metamodelo. Durante el primer paso se obtiene un árbol de sintaxis del código fuente y en el segundo paso se recorre dicha estructura para generar el modelo destino. Por ejemplo, en [35] y [56] se construyen *parsers* dedicados para la extracción de modelos a partir de código PL/SQL y en [1] se construye otro *parser* que trata con código Visual Basic. Sin embargo, el desarrollo de *parsers* dedicados es una tarea que supone un gran esfuerzo de implementación, ya que los recorridos sobre el árbol de sintaxis y la resolución de referencias deben realizarse programáticamente. Además, las correspondencias entre los elementos de la gramática y los del metamodelo también deben ser codificadas en el *parser*. Por este motivo, normalmente se utilizan otras herramientas para la obtención de un AST automáticamente, permitiendo reducir el esfuerzo de desarrollo. Este paso se realiza normalmente utilizando un API que permita trabajar con los árboles de sintaxis de una forma más cómoda. Un ejemplo de tal API es JDT [69], desarrollada en el proyecto Eclipse, que permite trabajar con código Java. Sin embargo, no existen API para todos los lenguajes de programación utilizados en modernización (p. ej., PL/SQL o Cobol). Además, aunque el uso de un API permite abordar la extracción y manejo del AST, todavía es necesaria la implementación de mecanismos para obtener la información dispersa por el código fuente así como codificar las correspondencias entre los elementos, por lo que realmente el uso de API no reduce el tiempo de desarrollo significativamente.

Una posible solución para ayudar a la creación de *parsers* dedicados es proporcionada por MoDisco (*Model Discovery*) [11], un *framework* extensible para aplicar tareas de ingeniería inversa dirigida por modelos el cual está incluido en el proyecto GMT de Eclipse [66]. Este *framework* está actualmente en desarrollo y su principal objetivo es facilitar la implementación de tareas de reingeniería en diferentes escenarios de modernización. MoDisco incorpora un conjunto de metamodelos para describir sistemas software (p. ej., una implementación del metamodelo KDM [52]), herramientas para tratar con sistemas complejos (como un editor de modelos especialmente adaptado para tratar con modelos muy grandes) y *parsers* dedicados (llamados *discoverers*) para obtener modelos de sistemas software *legacy* y utilizarlos en modernización del software. Los *discoverers* disponibles actualmente permiten obtener modelos de árbol de sintaxis a partir de ficheros XML y código Java. Sin embargo, todavía es necesario recorrer dichos modelos para obtener la información dispersa así como crear el modelo conforme al metamodelo destino, lo cual requeriría aplicar transformaciones *m2m*, cuyas desventajas se comentarán más adelante.

5.2.2. Herramientas de definición de DSL

Las herramientas de creación de DSL textuales deben implementar un puente entre el *grammarware* y el *modelware* para poder obtener modelos de la sintaxis abstracta del DSL

a partir de su sintaxis concreta, por lo que podrían ser consideradas para llevar a cabo la extracción de modelos a partir del código. Estas herramientas pueden ser clasificadas en dos grupos dependiendo de si la implementación de este puente parte de la gramática o del metamodelo. En las herramientas que utilizan la gramática como punto de partida, como Xtext [111], el desarrollador utiliza una notación parecida a EBNF para especificar tanto la gramática, que incluye reglas para guiar la generación del metamodelo, como la sintaxis concreta. El metamodelo es entonces generado automáticamente a partir de esta especificación junto con el extractor de modelos conformes a dicho metamodelo, el generador de código desde los modelos y un editor específico del lenguaje. Por otro lado, las herramientas que parten del metamodelo, tales como EMFText [36] y TCS [79], tienen como entrada un metamodelo anotado a partir del cual se genera automáticamente la gramática, el extractor de modelos conformes a dicho metamodelo y el generador de código desde los modelos. De hecho, algunas herramientas como Xtext soportan ambas aproximaciones.

Las herramientas de definición de DSL basadas en el metamodelo no ofrecen una solución eficaz adaptada al problema de la extracción. Tal y como se dice en [79]: "Si el problema a tratar es desarrollar un único lenguaje de propósito general entonces merece la pena dedicar el esfuerzo de programación en desarrollar un *parser*" (en vez de utilizar TCS). Esto es principalmente debido al hecho de que un DSL tiene, normalmente, una estructura mucho más simple que un GPL, por lo que estas herramientas no se adaptan correctamente al problema de la extracción de modelos. Este problema aparece ejemplificado cuando EMFText es utilizado para extraer modelos de código GPL, ya que la herramienta tiene que ser profundamente modificada para adaptarse al GPL específico. De esta forma, el trabajo necesario para implementar un extractor de modelos Java en EMFText requirió tal número de modificaciones que se creó un proyecto nuevo denominado Jamopp [67].

En cuanto a las herramientas de definición de DSL basadas en la gramática, aparecen una serie de limitaciones cuando se utilizan en el ámbito de los GPLs. El metamodelo generado automáticamente es de pobre calidad debido a que incluye elementos superfluos y estructuras heredadas de la gramática, por lo que el gap semántico entre dicho metamodelo y el utilizado como destino de la transformación *t2m* es todavía muy grande. De esta forma, se requiere aplicar una transformación *m2m* para convertir los modelos extraídos por Xtext en modelos conformes al metamodelo destino deseado. Sin embargo, la definición de estas transformaciones es una tarea compleja debido a que las transformaciones *m2m* no ofrecen mecanismos eficientes para resolver el problema de la obtención de información dispersa por el código, como más adelante se expondrá.

Además, otro aspecto negativo de las herramientas basadas en la gramática es que no ofrecen mecanismos para la reutilización de gramáticas, ni de las existentes, como las ofrecidas por herramientas como ANTLR [61]; ni de las definidas en el propio Xtext. Por una parte, adaptar una gramática existente al formalismo utilizado por Xtext es una tarea complicada debido a que no pueden ser especificadas algunas opciones necesarias para reconocer GPLs (p. ej., el uso del mecanismo de *backtracking* en Java o el uso de predicados sintácticos). Por otra parte, las gramáticas definidas en Xtext son difícilmente adaptables porque llevan incluídas un conjunto de reglas orientadas a la generación del metamodelo asociado.

Existen soluciones como las propuestas por Wimmer [108] y Kunert [48] para mejorar la calidad de los metamodelos generados por medio del uso de heurísticas y anotaciones a la gramática. Sin embargo, la calidad del metamodelo generado es todavía baja y sigue siendo necesaria una transformación modelo-a-modelo. Además, todavía no existen herramientas que las implementen.

5.2.3. Lenguajes de transformación de programas

En este grupo encontramos herramientas como Stratego/XL [76] y TXL [81] las cuales permiten expresar la sintaxis abstracta del código fuente por medio de una gramática libre del contexto en vez de un metamodelo. Al utilizar estas aproximaciones en el ámbito de la modernización dirigida por modelos aparecen una serie de limitaciones. En primer lugar, el resultado de una transformación de programas es un programa (código fuente) que conforma a una gramática, por lo que es necesario todavía aplicar un puente desde el *grammarware* al *modelware* para obtener el modelo conforme al metamodelo destino. En segundo lugar, la reutilización de gramáticas no se ve favorecida debido a que cada *toolkit* ofrece su propio lenguaje de definición gramatical. Además, estas aproximaciones ofrecen un limitado número de gramáticas GPL (p. ej., Java en Stratego y TXL).

5.2.4. Lenguajes de transformación *m2m*

Las transformaciones *m2m* pueden ser utilizadas para abordar un proceso de extracción de modelos a partir del código. Sin embargo, esta posibilidad requeriría disponer de modelos en vez de código GPL como entrada, por lo que sería necesario un *parser* que obtuviera modelos conformes a un metamodelo intermedio (p. ej., conformes a un metamodelo para AST) a partir del código fuente. De esta forma, se podría aplicar una transformación *m2m*, aunque todavía existiría un problema: la necesidad de un lenguaje de consultas adecuado al problema. La mayoría de lenguajes de transformación *m2m* como ATL [40] o QVT [96] ofrecen un lenguaje de navegación de modelos basados en OCL [17]. Este lenguaje de navegación resulta eficaz en transformaciones *m2m* pero no se adecúa a las necesidades de las transformaciones *t2m* para código fuente ya que da lugar a la definición de largas cadenas de navegación. La solución sería la integración de un lenguaje de navegación más apropiado a los lenguajes de transformación *m2m*. Sin embargo, ello requería implementar mecanismos en las transformaciones *m2m* para poder soportar diferentes mecanismos de consultas (p. ej., un mecanismo de *plugins*).

5.3. Nuestra aproximación

Dadas las limitaciones de las aproximaciones existentes comentadas anteriormente, decidimos crear un DSL para facilitar el proceso de extracción de modelos a partir del código fuente. Nuestros objetivos fueron reducir el tiempo de desarrollo, facilitar el mantenimiento y favorecer el reuso de gramáticas existentes (p. ej., las gramáticas de ANTLR o JavaCC). Para conseguir estos objetivos identificamos dos decisiones de diseño importantes: (1) cómo definir las correspondencias entre elementos de la gramática y el metamodelo de forma

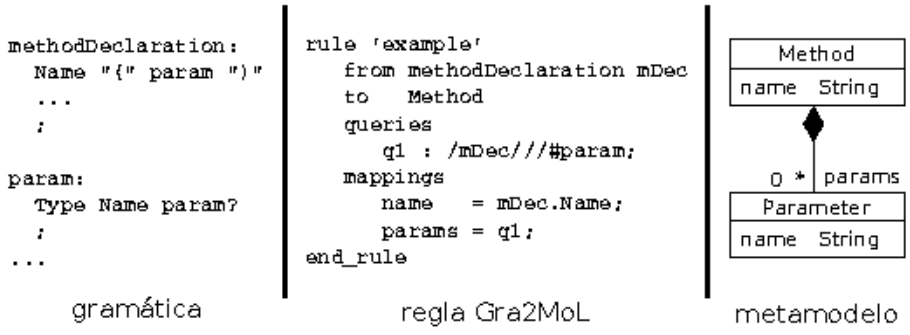


Figura 5.2: Ejemplo simple de una regla Gra2MoL.

fácil y clara y (2) cuál es la notacion más apropiada para obtener la información dispersa en los árboles de sintaxis.

El DSL, denominado Gra2MoL (*Grammar To Model Transformation Language*), ofrece construcciones específicas para definir declarativamente y a alto nivel de abstracción las correspondencias entre los elementos, de forma similar a como se realiza en lenguajes de transformación *m2m* como ATL o RubyTL. En cuanto al soporte para recorrer los árboles de sintaxis, Gra2MoL incorpora un potente lenguaje de consultas inspirado en XPath [110] para facilitar la obtención de información de dichos árboles. La estructura de este lenguaje de consultas se explicará en la siguiente sección.

La Figura 5.2 muestra un ejemplo simple de transformación Gra2MoL. Una definición de transformación Gra2MoL consiste en un conjunto de reglas, donde cada una define una correspondencia entre un elemento de la gramática y un elemento del metamodelo. La definición de transformación utilizada en el ejemplo es muy simple, solamente contiene la regla llamada `example`, la cual transforma un elemento gramatical `methodDeclaration` en el elemento del metamodelo `Method`, de acuerdo a las secciones *from* y *to* de la regla. Por otro lado, la sección *mappings* especifica cómo deben inicializarse las propiedades del elemento del modelo a partir de la información del árbol de sintaxis. En este ejemplo, el atributo `name` del elemento del modelo `Method` es inicializado con el valor del elemento gramatical `Name`, al cual se accede a través del elemento gramatical `methodDeclaration` recibido por la regla (variable `mDec`). A continuación, se inicializa la referencia `params` utilizando la consulta `q1`, la cual obtiene todos los elementos gramaticales `param` que representan a los parámetros del método. De esta forma, una regla Gra2MoL permite establecer las correspondencias entre los elementos de la gramática y del metamodelo explícitamente y la obtención de información del árbol de sintaxis se realiza con el lenguaje de consultas especialmente adaptado.

La Tabla 5.3 muestra una comparativa entre Gra2MoL y las aproximaciones analizadas. Las columnas muestran las propiedades comparadas: la forma de navegar por el árbol de sintaxis, los artefactos que deben crearse, la necesidad de un pre y post-processado, la reutilización de gramáticas existentes o las definidas por la propia aproximación y la finalidad principal de la aproximación. Los artefactos que deben crearse así como el pre

Aprox.	Navegación del árbol de sintaxis	Artefactos necesarios	Pre procesamiento	Post procesamiento	Reuso gramática existentes	Reuso gramática propia	Finalidad
Parser dedicado (+ API)	Código GPL	MM_T P	Ninguno	Ninguno	SI	NA	Extracción de modelos ad hoc
MoDisco	Lenguajes tipo OCL	MM_T D	*Discoverer* (si no se trata con Java o XML)	Trans. *m2m*: $MM_I \rightarrow MM_T$	Si	NA	Extracción de modelos de propósito general
Definicion de DSL	Soporte pobre	G_{xt} MM_T $m2m$	Ninguno	Trans. *m2m*: $MM_I \rightarrow MM_T$	No	No	Creación de DSL
Transf. programas	Stratego incorpora un lenguaje de consultas [106]	MM_T T_{PT} G_{AS} $m2m$	Ninguno	Extraer un programa conforme a G_{AS}	Limitado (pocas gramáticas)	Si	Transformación de programas
Trans. *m2m*	Lenguajes tipo OCL	MM_T P $m2m$	Extractor de modelos intermedios	Ninguno	Si	NA	Transformación de modelos
Gra2MoL	Lenguaje de consultas *Structum-shy*	MM_T T	Ninguno	Ninguno	Si	NA	Extracción de modelos de propósito general

Tabla 5.1: Comparación de Gra2MoL con las aproximaciones analizadas. NA = No aplicable, G = Gramática, MM_D = Metamodelo Destino, MM_I = Metamodelo intermedio, T = Definición de transformación, P = Parser Dedicado, T_{TP} = Definición de transformación de programa, G_{xt} = gramática Xtext, $m2m$ = Transformación *m2m*, G_{AS} = Gramática AST.

y post-procesado determinan el nivel de esfuerzo requerido en cada aproximación. Por ejemplo, las aproximaciones basadas en la gramática y las transformaciones de programas requieren tareas más complejas que las necesarias en Gra2MoL, tales como la definición de transformaciones *m2m* o de gramáticas GPL, mientras que en Gra2MoL solamente es necesaria la definición del metamodelo destino y de la transformación. Por otro lado, tanto las transformaciones *m2m* como MoDisco requieren un gran esfuerzo ya que es necesario definir la transformación *m2m* para obtener el modelo conforme al metamodelo destino. Además, en el caso de MoDisco, también es necesario implementar el correspondiente *discoverer* si el lenguaje de programación no es Java ni XML. En cuanto a la aproximación basada en la creación de un *parser* dedicado, Gra2MoL ofrece un lenguaje especialmente adaptado para la definición de transformaciones *t2m*, por lo que el tiempo de desarrollo se reduce considerablemente.

5.4. Definición de un lenguaje de consultas para árboles de sintaxis

Tal y como se ha comentado anteriormente, las transformaciones *t2m* utilizadas en una extracción de modelos requieren un uso intensivo de consultas para obtener la información

dispersa en el árbol de sintaxis del código fuente. De esta forma, una herramienta para extraer modelos a partir del código fuente debe ofrecer un potente lenguaje de consultas que facilite el acceso a los nodos del árbol de sintaxis que estén fuera del ámbito en el que se esté trabajando. La Figura 5.3 ilustra el problema de la información dispersa en un ejemplo simple de extracción de un elemento del metamodelo a partir de un procedimiento Delphi. Debido a que el problema de la información dispersa aparece tanto en los AST como en los CST, y obtener un CST es más fácil que un AST, Gra2MoL utiliza CST para representar el código fuente, aunque podría ser fácilmente adaptado para tratar con AST. De esta forma, el CST mostrado corresponde a una declaración de un procedimiento que incluye una declaración de variable y una sentencia de asignación que inicializa dicha variable. El elemento del modelo `Assignment` representa expresiones de asignación y tiene dos propiedades para representar la parte derecha e izquierda de la asignación (referencias `receptor` y `expression`, respectivamente). Como se puede observar, mientras que toda la información necesaria para inicializar el atributo de la parte derecha (elemento gramatical `expression`) se encuentra dentro del ámbito actual (representado como un óvalo), la información necesaria para inicializar el atributo de la parte izquierda está fuera de dicho ámbito debido a que la declaración de la variable `msg` está referenciada por un identificador. Por lo tanto es necesario resolver está referencia a la variable `msg` accediendo al nodo que represente la declaración de dicha variable y obtener sus propiedades, concretamente, su nombre (almacenado en el nodo gramatical `ID`).

Con este problema en mente, hemos desarrollado un lenguaje de consultas tipo *structure-shy* inspirado en XPath [110] que permite navegar sobre los CST del código sin tener que especificar cada paso de navegación. El término *structure-shy* se utiliza comúnmente para referirse a descripciones de comportamiento que están poco acopladas a las estructuras de datos sobre las que actúan.

Para navegar un CST, a los nodos del árbol se les asocia un tipo utilizando la definición gramatical donde cada nodo del árbol registra el nombre y el tipo del elemento gramatical. La Figura 5.4 ilustra las reglas de conformidad entre el CST y la definición gramatical, mostrando el CST para varios procedimientos Delphi junto con el correspondiente fragmento de la gramática de dicho lenguaje. Las reglas de conformidad son las utilizadas comúnmente para crear un árbol de este tipo:

- Un elemento no-terminal corresponde a un nodo interno del árbol. Por ejemplo, el elemento no-terminal `declSection` corresponde al nodo del árbol `declSection` en la Figura 5.4.
- Un elemento terminal corresponde a una hoja del árbol. En la Figura 5.4, el terminal `ID` corresponde a la hoja `ID`.
- Una regla de producción es representada por una jerarquía de elementos del árbol cuyo padre corresponde al elemento no-terminal de la parte izquierda de la regla y un hijo por cada elemento gramatical de la parte derecha resultado de aplicar las reglas anteriores. En la Figura 5.4, la regla de producción `declSection` es representada por la jerarquía cuya raíz tiene el nodo del árbol `declSection`.

Una consulta consiste en una secuencia de operaciones donde cada operación incluye un operador, un tipo de nodo y, opcionalmente, expresiones de filtro y de acceso. Además, una

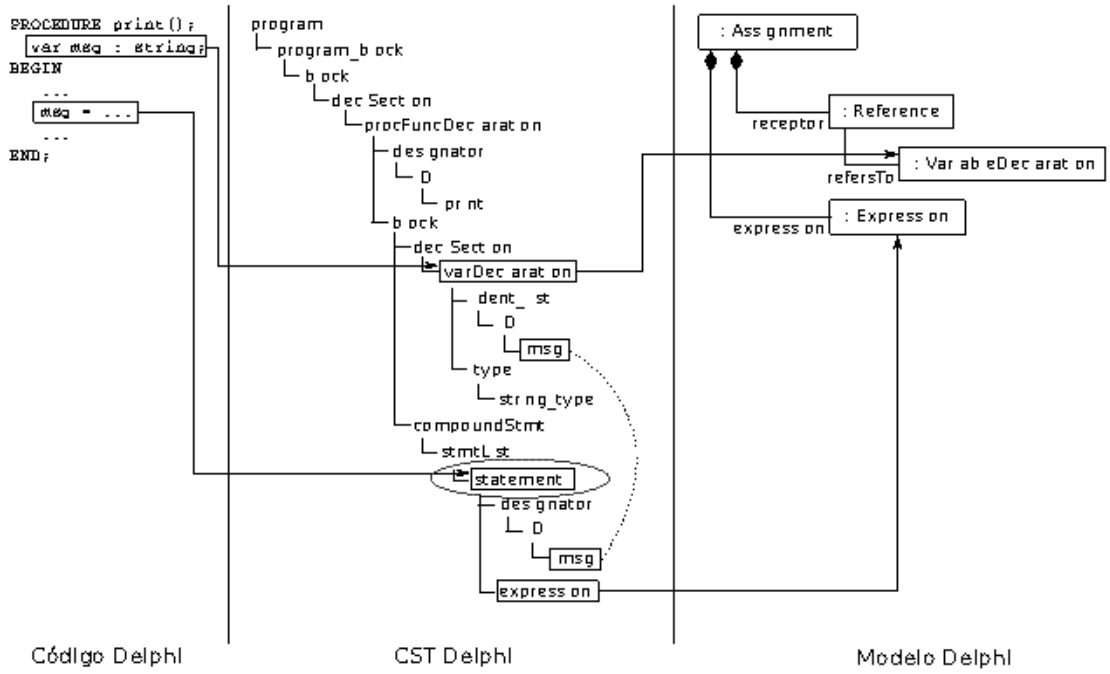


Figura 5.3: Ejemplo del problema de la información dispersa. El óvalo indica el ámbito actual y la línea punteada una referencia basada en identificadores entre dos elementos del árbol.

consulta podría estar prefijada por una sentencia de control. La expresión EBNF para una consulta es la siguiente:

```
[control] { ('/'|'//'|'///') ('#')? tipoNodo [filtro] [acceso] }
```

Existen tres tipos de operadores para consultar y navegar el CST: `/`, `//` y `///`. El operador `/` devuelve los hijos inmediatos de un nodo y es similar a la notación punto (p. ej., en OCL). Los operadores `//` y `///` permiten recorrer todos los hijos (directos e indirectos) de un nodo del árbol. El operador `///` difiere ligeramente del operador `//`. Mientras que el operador `///` realiza la búsqueda de forma recursiva en el CST, el operador `//` limita la búsqueda a aquellos nodos cuya profundidad es menor o igual a la profundidad del primer nodo encontrado. Estos dos operadores permiten ignorar nodos superfluos intermedios, facilitando la definición de consultas, ya que permite especificar lo que se quiere buscar sin tener que describir cómo. En los ejemplos mostrados en las secciones posteriores se ilustrará prácticamente el uso de estos operadores.

Debido a que una consulta puede devolver uno o más subárboles, el operador `#` se utiliza para indicar el nodo raíz que contendrá la información consultada. Este operador debe estar asociado a uno y sólo un operador de entre las operaciones que componen una consulta. Por ejemplo, para extraer todas las declaraciones de variable Delphi definidas en cada procedimiento del CST mostrado en la Figura 5.4, se podría utilizar la siguiente consulta

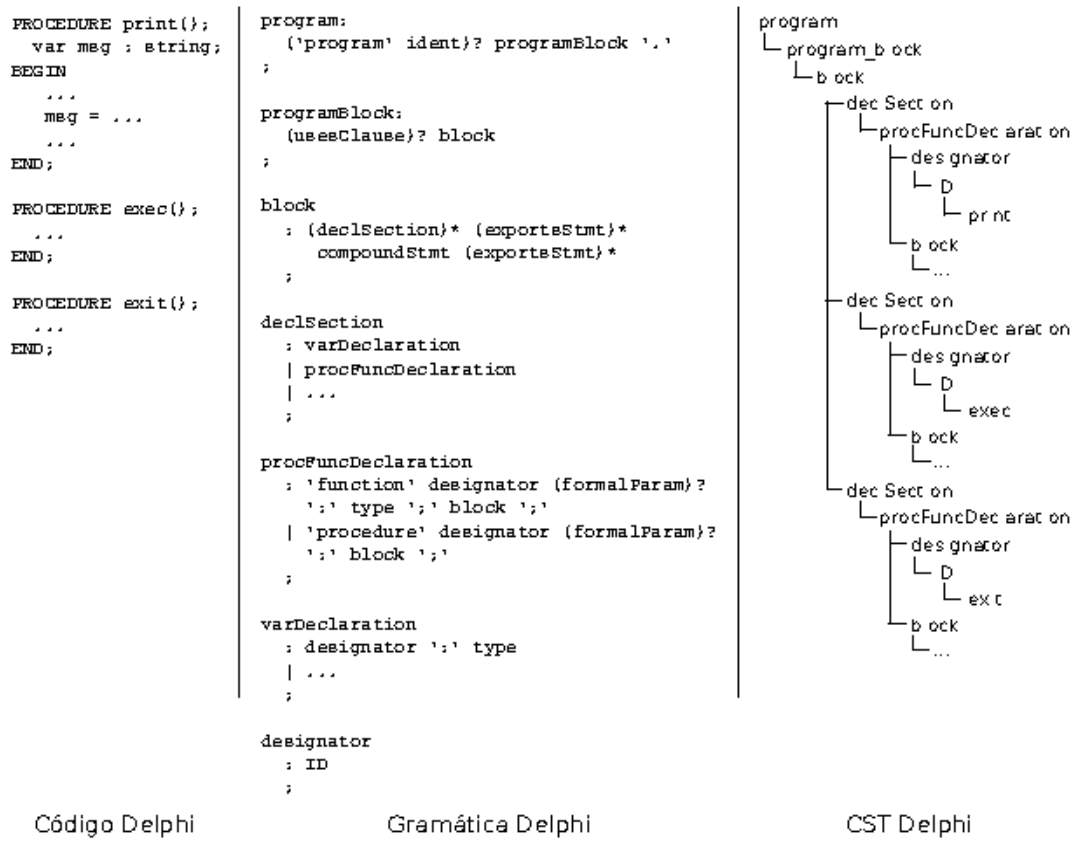


Figura 5.4: Ejemplo de CST para la gramática Delphi.

`/program//#varDeclaration`. Para mostrar la diferencia entre utilizar este lenguaje de consultas y OCL, la Figura 5.5 muestra el ejemplo anterior en OCL. Puede apreciarse la mejora en legibilidad, concisión y naturaleza declarativa.

Las operaciones de consulta pueden incluir una expresión de filtro entre llaves, la cual es una expresión lógica que se aplica a las hojas de nodo especificado en la operación de consulta. Cada operando de una expresión de filtro es una función booleana que comprueba las propiedades de una hoja, como su valor o si dicha hoja existe. Así, solamente aquellos nodos que satisfagan la expresión de filtro serán seleccionados. Por ejemplo, en la Figura 5.4, la consulta `/program//#procFuncDeclaration{ID.exists && ID.eq("print")}` seleccionará todos los elementos gramaticales que representan a un procedimiento, tienen una hoja llamada `ID` y el valor de dicha hoja es `print`.

Finalmente, los operadores de consulta también pueden incluir una expresión de acceso entre corchetes para acceder a los hermanos de un nodo. El acceso se realiza por medio de indexación como si fuera un array. Por ejemplo, en la Figura 5.4, la consulta `/program//#procFuncDeclaration[0]` seleccionará el primer elemento gramatical que representa un procedimiento, que en este ejemplo es el procedimiento `print`.

```
Locals(p : program) : Sequence(varDeclaration)
post result =
  if(p.programBlock.block.declSection = oclIsUndefined())
  then
    Sequence {}
  else
    p.programBlock.block.declSection->
      select (pd | ds.oclIsKindOf(procFuncDeclaration))->
      collect(e | e.block.declSection)->flatten()->
      select (vd | ds.oclIsKindOf(varDeclaration))
  endif
```

Figura 5.5: Consulta OCL para extraer todas las declaraciones de variable de cada procedimiento PL/SQL del CST mostrado en la Figura 5.4.

```
q1 : //#varDeclaration;
q2 : {for each v in q1} //#statement/designator{ID.eq(v.ID)};
                              (a)
_____________________________________________________________
q3 : {greatest varDeclaration.Value} //procFuncDeclaration
                                     //#varDeclaration;
                              (b)
```

Figura 5.6: Ejemplos de sentencias de control para (a) parametrizar las consultas y (b) filtrar el resultado de una consulta.

Adicionalmente, una consulta puede incluir una sentencia de control rodeada de llaves al principio de dicha consulta. Las sentencias de control permiten gestionar la ejecución de una consulta para realizar un pre-procesamiento, es decir, parametrizar un consulta; o un post-procesamiento, es decir, realizar un filtrado a una consulta. Por una parte, la parametrización permite ejecutar consultas utilizando información externa como por ejemplo los nodos resultado de una consulta previa. La Figura 5.6a muestra un ejemplo de parametrización que incluye dos consultas. La consulta `q1` selecciona todos las variables (elementos de tipo `VarDeclaration`) y, a continuación, la consulta `q2` utiliza una sentencia de control de tipo `for each` para obtener todos los elementos de tipo `statement` que utilizan alguna de las variables de `q1`. Nótese que la sentencia de control se encarga de lanzar la consulta `q2` tantas veces como elementos resultado tenga la consulta `q1`, utilizando la variable `p` para almacenar el elemento actual en cada iteración.

Por otra parte, las sentencias de control también puede ser utilizadas para filtrar el resultado de una consulta ya ejecutada. La Figura 5.6b muestra una consulta que selecciona todas las declaraciones de variable (elementos `varDeclaration`) incluídas en un prodecimiento (elemento `procFuncDeclaration`) y, a continuación, la sentencia de control `greatest` selecciona aquellos elementos cuyo valor en la hoja `Value` es el mayor.

Gra2MoL incluye otras sentencias de control como `while` y `least`, las cuales permiten ejecutar una consulta tantas veces como se establezca por la condición del *while* o seleccionar la hoja cuyo valor es el menor, respectivamente. Además éste conjunto puede ser ampliado utilizando el mecanismo de extensiones descrito en la sección 5.7.

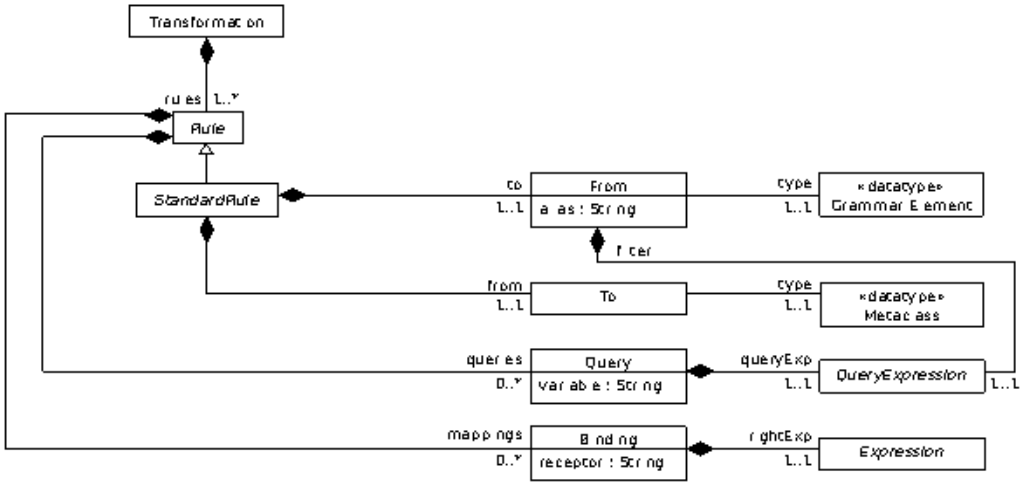
5.5. Definición de transformación en Gra2MoL

Gra2MoL ha sido diseñado como un lenguaje de transformación basado en reglas que tienen una estructura similar a las reglas en los lenguajes de transformación *m2m* como ATL [40] y RubyTL[1] [87], con dos diferencias importantes: i) el elemento origen de una regla es un elemento gramatical en vez de un elemento del metamodelo y ii) la navegación por el código fuente se expresa utilizando el lenguaje de consultas presentado anteriormente, en vez de utilizar OCL.

La sintaxis abstracta de Gra2MoL, expresada como metamodelo, se muestra en la Figura 5.7a y la sintaxis concreta en la Figura 5.7b, la cual muestra el esqueleto de una regla. Como se puede apreciar, una definición de transformación (metaclase `Transformation`) consiste en un conjunto de reglas de transformación (metaclase `Rule`). Gra2MoL define cinco tipos de reglas: *normal*, *toprule*, *copy*, *skip* y *mixin*, tal y como se ilustra en la Figura 5.8. Las reglas de tipo *normal* son utilizadas para especificar las correspondencias entre un elemento gramatical y un elemento del metamodelo, por lo tanto son las reglas más comunes en una definición de transformación. Las reglas de tipo *toprule* y *copy* son una particularización de las reglas de tipo *normal*. Mientras que las reglas de tipo *toprule* se encargan de iniciar la ejecución de la transformación, como se explicará en la sección 5.5.1, las reglas de tipo *copy* permiten transformar un elemento gramatical más de una vez. Por otro lado, las reglas *skip* y *mixin* incorporan comportamiento específico a Gra2MoL y serán explicadas en las secciones 5.5.3 y 5.5.4, respectivamente. Debido a que las reglas de tipo *normal* y *skip* tienen la misma estructura, están categorizadas como reglas estándar, mientras que las reglas *copy* heredan de las reglas de tipo *normal*. Una regla estandar está compuesta de cuatro secciones:

- La sección *from* especifica el símbolo no-terminal de la gramática y declara una variable que se asociará al nodo del árbol cuando la regla es aplicada y que puede ser utilizada por cualquier expresión incluida en la regla. La sección *from* puede también incluir operaciones de consulta, es decir, un filtro, para comprobar la estructura que deben satisfacer los nodos que recibe la regla.
- La sección *to* especifica la metaclase destino.
- La sección *queries* contiene un conjunto de expresiones de consulta para obtener información del CST. El resultado de estas consultas será utilizado en la sección *mappings*.
- Finalmente, la sección *mappings* contiene un conjunto de *bindings* para inicializar las propiedades del elemento del metamodelo destino. El concepto de *binding* será explicado en la siguiente sección. Además, también es posible utilizar sentencias imperativas o de control, como por ejemplo sentencias `if` o `new` para crear instancias de metaclase, tal y como se ilustrará en la sección 5.8.

[1] RubyTL también está inspirado en ATL y fue creado como un DSL embebido a finales del 2005 para disponer de un lenguaje de transformación *m2m* que permitiese experimentar fácilmente con este tipo de transformaciones.



(a)

```
rule '<nombre-regla>'
  from <elemento-gramatical-origen> <alias>
  to   <metaclase-destino>
  queries
    { variable : consulta; }
  mappings
    { propiedad-metaclase = literal | consulta | expression; }
end
```

(b)

Figura 5.7: (a) Un extracto de la sintaxis abstracta de Gra2MoL. (b) Esqueleto de una regla Gra2MoL.

5.5.1. *Bindings* y evaluación de reglas

Un *binding* se utiliza en la sección *mappings* para establecer la relación entre un elemento de la gramática y una propiedad de la metaclase. Este mecanismo tiene una sintaxis y semántica muy similar al utilizado en ATL [40] y RubyTL [87]. Un *binding* se escribe como una asignación utilizando el operador =. La parte izquierda de la asignación es una propiedad de la metaclase destino y la parte derecha puede ser la variable especificada en la sección *from*, un valor literal o un identificador de consulta.

La evaluación de reglas se determina por medio de un mecanismo de planificación basado en *bindings*. Las definiciones de compatibilidad regla-*binding* y transformación bien formada establecidas para RubyTL en [87] son también aplicables en Gra2MoL con algunos pequeños cambios.

Compatibilidad regla-*binding*. Una regla es compatible o conforma con un *binding* si el tipo de su sección *from* es compatible o conforma con el tipo de la parte derecha de la asignación y, el tipo de su sección *to* conforma con el tipo de la parte izquierda de la asignación. Donde la compatibilidad de tipos se define como se detalla a continuación.

Compatibilidad de tipos. Una metaclase A_m es compatible o conforma a otra metaclase

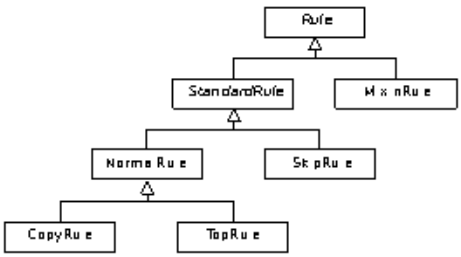


Figura 5.8: Tipos de regla en Gra2MoL.

B_m si es la misma o A_m es subtipo de B_m, mientras que el tipo de un nodo A_n conforma al tipo de otro nodo B_n si es el mismo.

Definición de transformación bien formada. Una transformación está bien formada si para cada *binding* que trata con un tipo no primitivo en la parte derecha de la asignación existe una o más reglas que conforman a dicho *binding* pero solo una de ellas puede ser aplicada. De esta forma, si dos o más reglas conforman a un *binding*, su expresión de filtro en la sección *from* debe permitir que una y sólo una pueda ser aplicada.

5.5.2. Evaluación de reglas

La ejecución de una definición de transformación Gra2MoL está guiada por los *bindings*. Las reglas de tipo *toprule* de una transformación son el punto de entrada y los *bindings* de la sección *mappings* se encargan de comenzar la ejecución de la transformación. Si una transformación no tiene regla de tipo *toprule*, la primera regla de la transformación será considerada como tal. En una transformación Gra2MoL, solamente las reglas de tipo *normal*, *toprule*, *copy* o *skip* son candidatas para ser ejecutadas por un *binding*, mientras que las reglas de tipo *mixin* se aplican cuando son referenciadas por otras reglas, tal y como se explicará en la seccion 5.5.4.

Cuando una regla es aplicada a un nodo, primero se comprueba el filtro de la sección *from*, si existe. Si el filtro se satisface, la regla es ejecutada y se crea una instancia de la metaclase destino. Además, la ejecución de la regla es registrada para evitar ciclos. Finalmente, los *bindings* de la regla son ejecutados, pudiéndose dar tres situaciones según la naturaleza de la parte derecha:

- Si es un valor literal, dicho valor es directamente asignado a la propiedad de la parte izquierda.
- Si es un identificador de consulta, ésta es ejecutada y se busca aquella regla que conforme con el *binding*. Una vez que una regla es encontrada, se ejecuta utilizando el elemento de la parte derecha del *binding* como elemento gramatical.
- Si es una expresión, ésta es evaluada y pueden darse dos situaciones dependiendo de si el resultado es un nodo cuyo tipo es terminal (una hoja del árbol) o no-terminal. Si es una hoja, el resultado es de tipo primitivo y se asigna directamente a la propiedad,

en otro caso, se busca la regla que cumpla el *binding* y se ejecuta, tal y como se ha explicado en el punto anterior.

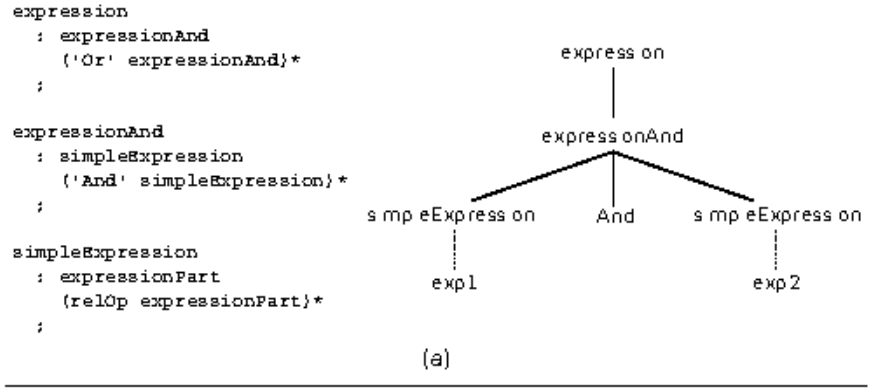
5.5.3. Reglas de tipo *skip*

La transformación de las expresiones aritméticas y lógicas del código fuente requiere que Gra2MoL incorpore un mecanismo especial para tratar con las estructuras gramaticales propias de dichas expresiones. El uso de expresiones en un lenguaje de programación provoca la existencia de un conjunto de reglas gramaticales que prácticamente crean un nuevo sublenguaje con su correspondiente árbol de sintaxis. Normalmente, estas reglas gramaticales se definen de forma encadenada, de modo que cada nueva regla incluye un nuevo operador. Por ejemplo, la Figura 5.9a muestra las reglas gramaticales que reconocen los operadores `AND` y `OR` en una expresión junto con el árbol de sintaxis que se obtiene para la expresión `exp1 And exp2`. Con el uso de reglas de tipo *normal*, las correspondencias entre los elementos gramaticales y los elementos del metamodelo son prácticamente directas, por ejemplo, una expresión `OR` se transforma en un elemento del modelo que representa expresiones `OR`. Sin embargo, las reglas de tipo *normal* no son apropiadas para manejar expresiones debido a que, en algunos casos, reconocer un elemento gramatical de una expresión no supone crear un elemento del modelo. Por ejemplo, reconocer un elemento gramatical de tipo `expression` de la Figura 5.9a no implica la creación del elemento del modelo que representa la expresión binaria `OR`. En general, estos casos son aquellos en los que el elemento de la gramática no contiene el operador correspondiente.

Por tanto, para tratar con expresiones, las reglas Gra2MoL deben disponer de algún mecanismo que permita decidir si se crea el elemento correspondiente del metamodelo según una determinada condición, en el caso del ejemplo, la existencia del operador. Gra2MoL incorpora un tipo especial de regla, denominada *skip*, que está principalmente diseñada para extraer modelos de las expresiones de un lenguaje de programación, aunque también pueden utilizarse para guiar el flujo de ejecución de la transformación, tal y como se ilustrará en la sección 5.8. Las reglas de tipo *skip* permiten retrasar la creación del elemento del metamodelo establecido en la sección *to* hasta realizar comprobaciones en los elementos de la gramática. De esta forma, dependiendo del resultado de dichas comprobaciones, puede transferirse la ejecución a la regla adecuada utilizando el operador `skip` en la sección *mappings*. La Figura 5.9c muestra la regla de tipo *skip* que trata con los elementos gramaticales de tipo `expression` que no contienen el token `OR`, transfiriendo la ejecución a la regla gramatical que trata con el token `expressionAnd`, es decir, la siguiente regla de expresiones del lenguaje. Nótese que las reglas de tipo *skip* se pueden definir de forma encadenada, como se ilustrará en la sección 5.8.

5.5.4. Reglas de tipo *mixin*

De la misma forma que RubyTL, Gra2MoL incluye un tipo especial de regla, denominada *mixin*, especialmente diseñada para ofrecer un mecanismo de reutilización de reglas. De esta forma, los elementos de las secciones *queries* y *mappings* de una regla pueden ser factorizados en una regla de tipo *mixin*, la cual tiene la misma estructura sintáctica que




```
skip rule 'skipExpression'
  from expression{!TOKEN[0].exists} exp
  to   Expression
  queries
    next : /exp/#expressionAnd
  mappings
    skip next;
end rule
```

(b)

Figura 5.9: (a) Reglas gramaticales para analizar expresiones de tipo `AND` y `OR` junto con el árbol de sintaxis correspondiente a la expresión `expr1 And expr2`. (b) La regla de tipo *skip* para el elemento gramatical de tipo `expression`.

una regla de tipo *normal* exceptuando la sección *to*, que no existe. Las reglas de tipo *normal*, *toprule*, *copy* y *skip* pueden importar el uso de reglas de tipo *mixin* haciendo uso de la sección *mixin*, la cual debe incluirse entre las secciones *to* y *queries*. Cuando una regla importa a una regla de tipo *mixin*, el elemento gramatical de la sección *from* debe coincidir.

La Figura 5.10 muestra una regla de tipo *mixin* llamada `varFuncLocator` que es utilizada por una regla de tipo *normal* llamada `mapDesignator` para factorizar las consultas necesarias para localizar las declaraciones de variable (consulta `varLoc` de la regla de tipo *mixin*) y las declaraciones de procedimiento/función en (consulta `metLoc`) en Delphi. La regla `varFuncLocator` será ejecutada justo después de crear el elemento `Declaration` de la regla `myNormalRule` y justo antes de ejecutar la sección *mappings* de la regla `myNormalRule`. En el ejemplo, puede apreciarse que la sección *from* de ambas reglas coincide y la variable `dec` utilizada en la consulta `varLoc` de la regla de tipo *mixin* es asignada al elemento gramatical recibido por la regla de tipo *normal*. Nótese que la regla `mapDesignator` puede hacer uso de las consultas realizadas en la regla de tipo *mixin* (uso de la consulta `varLoc` en la consulta `q1`).

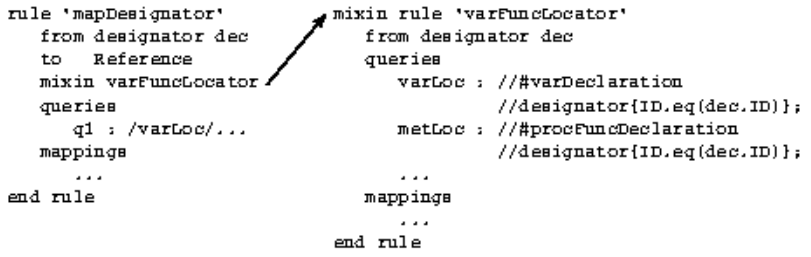


Figura 5.10: Uso de la regla de tipo *mixin* en Gra2MoL.

5.6. Implementación

La ejecución de una transformación Gra2MoL se realiza en dos pasos. En primer lugar, se obtiene el CST del código y el modelo de sintaxis abstracta de la definición de transformación Gra2MoL y, a continuación, se interpreta y ejecuta la transformación. La implementación actual de Gra2MoL soporta definiciones gramaticales de ANTLR, las cuales pueden ser enriquecidas con acciones para crear el CST. Sin embargo, nuestro objetivo es evitar trabajar con definiciones gramaticales que incorporen dichas acciones debido principalmente a dos razones: (1) ahorrar trabajo al desarrollador, evitando que tenga que modificar la gramática manualmente para la creación del CST, y (2) favorecer la reutilización de gramáticas existentes. De esta forma, Gra2MoL incorpora un proceso de enriquecimiento automático que añade las acciones necesarias para crear el CST de forma transparente. El proceso de enriquecimiento también soporta el uso de gramáticas *isla* (*island grammars*), que es un mecanismo aplicado cuando el lenguaje principal está compuesto de uno o mas sublenguajes (p. ej., el lenguaje *JavaDoc* en Java). En este caso, el desarrollador debe hacer que una de las reglas gramaticales del lenguaje principal apunte (comparta nombre) a la regla gramatical del sublenguaje para configurar el proceso de enriquecimiento.

Gra2MoL utiliza internamente un metamodelo para representar CST a partir del código fuente, el cual es construido según las reglas explicadas en la sección 5.4. Este metamodelo se muestra en la Figura 5.11. Existen tres tipos de elementos: `Leaf`, `Node` y `Tree`. Los elementos de tipo `Leaf` representan nodos terminales de la gramática mientras que los de tipo `Node` representan nodos no-terminales y están compuestos de uno o más nodos hijos, que pueden ser de tipo `Leaf` o `Node`. El atributo `type` representa el elemento gramatical asociado al elemento del árbol (utilizado para navegar a través del CST, tal y como se explicó en la sección 5.4). Finalmente, los elementos de tipo `Tree` representan al nodo raíz del árbol.

La figura 5.12a muestra el paso de preprocesamiento para enriquecer la gramática ANTLR (G) por medio del proceso *enriquecedor* previo a la generación del *parser* ($Parser_{G_e}$). Este *parser* genera modelos conformes al metamodelo CST (MM_{CST}) a partir del código fuente. Debido a que los CST pueden llegar a ser de tamaño muy significativo, el *parser* se puede configurar para almacenar dichos CST en un repositorio CDO [62].

Para obtener el modelo de sintaxis abstracta a partir la definición textual de la transformación inicialmente, se estudió la posibilidad de utilizar herramientas de creación de DSL

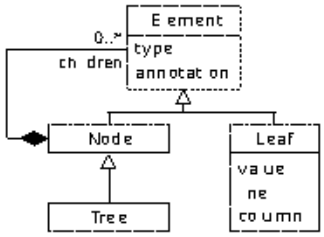


Figura 5.11: Metamodelo CST.

como Xtext. Sin embargo, estas herramientas no estaban todavía maduras. Por ejemplo, la estructura del metamodelo de sintaxis abstracta dependía de la estructura de la gramática que definía la sintaxis concreta. Por otro lado, tampoco se optó por desarrollarlo como un DSL interno debido a que el propio lenguaje puede ser utilizado para definir DSL externos (definiendo la transformación *t2m* encargada de extraer el modelo de sintaxis abstracta a partir de la definición textual). De esta forma, decidimos desarrollarlo como un DSL externo y utilizar un proceso de *bootstrap*, donde utilizamos Gra2MoL para obtener el modelo de sintaxis abstracta que luego será utilizado por el motor de transformación de Gra2MoL. Este proceso ejecuta una transformación Gra2MoL que define las correspondencias entre la gramática que especifica la sintaxis concreta del lenguaje y el metamodelo de su sintaxis abstracta. Para su implementación, primero se desarrolló una versión simplificada del motor de transformación que pudiera tratar con los elementos de la sintaxis de Gra2MoL. Esta primera versión sirvió para desarrollar un motor más elaborado del lenguaje, que fue el utilizado finalmente por el proceso. La Figura 5.12b muestra el proceso de *bootstrap* aplicado, el cual tiene cuatro entradas: (1) la definición textual de la transformación Gra2MoL (*Definición Gra2MoL* $G \rightarrow MM$), (2) la gramática del lenguaje Gra2MoL ($G_{Gra2MoL}$), (3) el metamodelo de sintaxis abstracta de Gra2MoL ($MM_{Gra2MoL}$) y (4) la definición de transformación (*Definición Gra2MoL* $G_{Gra2MoL} \rightarrow MM_{Gra2MoL}$). Como resultado de este proceso, se obtiene el modelo $M_{Gra2MoL}$ que es conforme al metamodelo de sintaxis abstracta del lenguaje.

La arquitectura del motor de ejecución de Gra2MoL incluye los siguientes componentes principales (ver Figura 5.13):

- *Intérprete*, el cual se encarga de ejecutar las reglas, resolver los *bindings* de la sección *mappings* de las reglas y controlar el flujo de ejecución.
- *Motor de consultas*, encargado de ejecutar las consultas sobre el CST.
- *Gestor de modelos*, el cual se encarga de crear y modificar los elementos del modelo destino. Este componente utiliza la infraestructura ofrecida por MoDisco [11] para permitir la gestión de modelos de forma independiente al metamodelo al que deben conformar.

Tal y como se muestra en la Figura 5.13, el *intérprete* de Gra2MoL utiliza el *parser* creado en el paso de preprocesamiento para obtener el CST (M_{CST}) que representa el código. A continuación, utiliza el proceso de *bootstrap* para obtener el correspondiente

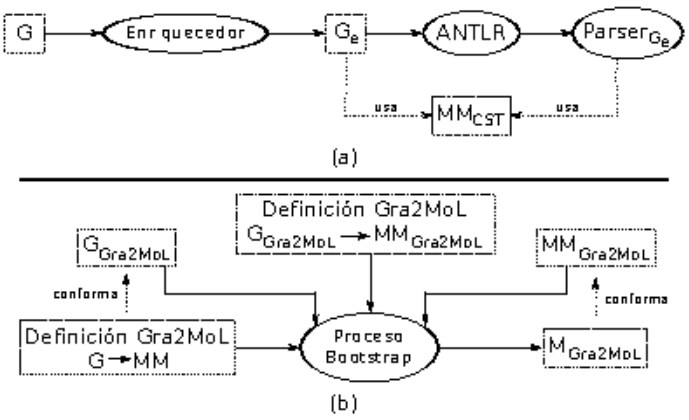


Figura 5.12: (a) Paso de preprocesamiento para enriquecer la gramática. (b) Proceso de *bootstrap*.

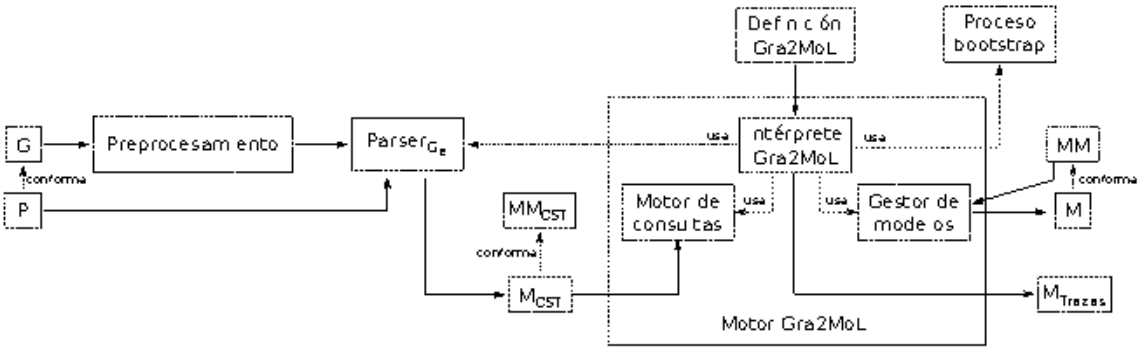


Figura 5.13: Arquitectura del motor Gra2MoL.

modelo que es conforme al metamodelo de sintaxis abstracta del lenguaje a partir de una definición textual de Gra2MoL (*Definición Gra2MoL*). Una vez se dispone del modelo de sintaxis abstracta de la transformación, el intérprete ejecuta las reglas, las cuales dan lugar a la ejecución de las consultas sobre el CST y a la creación de los elementos del modelo. Los artefactos generados por el motor Gra2MoL son el modelo (M) conforme al metamodelo destino (MM) y un modelo de trazas (M_{Trazas}) con información que relaciona los elementos creados con los elementos origen y las reglas aplicadas.

5.7. Mecanismo de extensión

Gra2MoL permite ser extendido para mejorar las capacidades del lenguaje. Existen dos puntos de extensión: (1) los operadores utilizados en la sección *mappings*, es decir, los operadores utilizados en la parte derecha de la asignación del *binding*; y (2) el lenguaje de consultas. Esta extensiones pueden ser implementadas e incorporadas al motor de transformación utilizando el *framework* de extensión ofrecido por el lenguaje, el cual se muestra

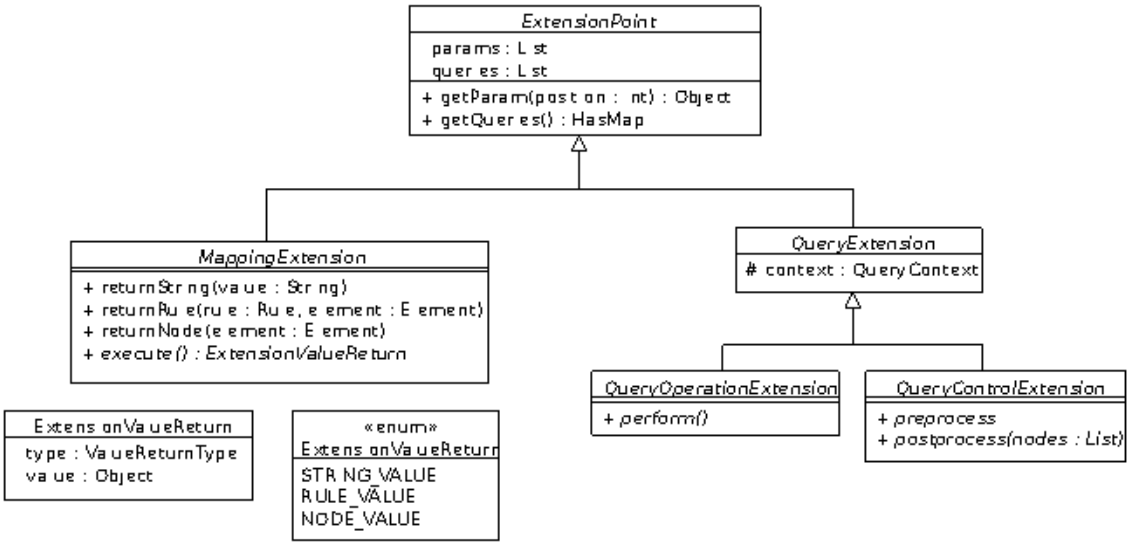


Figura 5.14: *Framework* ofrecido por Gra2MoL para extender el lenguaje.

en la Figura 5.14. Debido a que Gra2MoL ha sido desarrollado en Java, las extensiones se deben implementar extendiendo las clases ofrecidas en este *framework*.

La clase abstracta `MappingExtension` permite incorporar nuevos operadores para la sección *mappings*. Los nuevos operadores deben extender dicha clase e implementar el método `execute`, el cual se encarga de realizar el comportamiento específico del operador, y, opcionalmente, pueden implementar el método `getKeywords` para especificar las palabras clave que identifica al nuevo operador (si dicho método no es implementado, las palabras clave deben ser especificadas por medio de un fichero de propiedades). La clase `MappingExtension` también incluye varios métodos para facilitar al desarrollador la interacción con el motor de ejecución de Gra2MoL. Así, incorpora métodos para facilitar la creación de los valores de retorno del operador, que puede ser un valor de tipo cadena (método `returnString`), una regla (método `returnRule`) o un nodo del árbol (método `returnNode`). Además, la clase `MappingExtension` hereda a su vez de la clase común de la jerarquía `ExtensionPoint`, que incorpora métodos para facilitar el acceso a los parámetros del operador (método `getParam`) o las consultas de la regla (método `getQueries`).

Cuando un *binding* utiliza un nuevo operador debe utilizar la palabra clave `ext` seguido del identificador del nuevo operador y, opcionalmente, una lista de parámetros. Por ejemplo, la Figura 5.15a muestra una regla que utiliza el mecanismo de extensión en la sección *mappings* y llama al nuevo operador `toUpperCase`, que recibe el valor de la hoja `VALUE` del nodo `vd` y la convierte en mayúsculas. La Figura 5.15b muestra la implementación de la clase que hereda de `MappingExtension`.

El lenguaje de consultas también puede ser extendido para incorporar nuevas sentencias de control u operadores para las expresiones de filtro. Al igual que para los nuevos operadores de la sección *mappings*, las extensiones para las consultas deben ser implementadas como clases Java que extiendan a `QueryControlExtension` o `QueryOperationExtension`

Operador de *mapping*

```
rule 'extensionExample'
  from varDeclaration vd
  to ValuedElement
  queries
  mappings
    value = ext toUpperCase(vd.VALUE);
end rule
```

(a)

```
public class UpperCaseExtension extends MappingExtension {
  @Override
  public ExtensionValueReturn execute() {
    String value = (String) getParam(0);
    return value.toUpperCase()
  }
  public String[] getKeywords() {
    return new String[] { "toUpperCase" };
  }
}
```

(b)

Operador de control

```
q1 : { ext removeDuplicates } //#varDeclaration;
```

(c)

```
public class TestControlExtension
    extends QueryControlExtension {
  ...
  public void preprocess() { }
  public List<Element> postprocess(List<Element> nodes) {
    List<Element> resultList = removeDuplicates(nodes);
    return resultList;
  }
  private List<Element> removeDuplicates(List<Element> nodes) {
    ...
  }
  public static String[] keywords() {
    return new String[] { "removeDuplicates" };
  }
}
```

(d)

Operador de filtro

```
q1 : //#varDeclaration{VALUE.isSurroundedBy("<");
```

(e)

```
public class testQueryOperation
    extends QueryOperationExtension {
  ...
  public boolean perform() {
    ExpressionElement element = filter.getElement();
    Leaf leaf = node.getLeaf(element.getName(),
                             element.getPosition());
    return (leaf != null &&
            leaf.isSurroundedBy(getParam(0)))
            ? true : false;
  }
  private boolean isSourrondedBy(String char) {
    ...
  }
  public static String[] keywords() {
    return new String[] { "isSurroundedBy" };
  }
}
```

(f)

Figura 5.15: Ejemplos de extensión del lenguaje Gra2MoL. (a) Uso de la palabra clave `ext` para llamar al nuevo operador de *mapping* y (b) la implementación de dicho operador. (c) Uso de sentencias de control en una consulta y (d) extracto de la clase que implementa dicha sentencia. (e) Uso de operador en una consulta y (f) extracto de la clase que implementa dicho operador.

respectivamente, las cuales se muestran en la parte derecha de la Figura 5.14. Ambas clases heredan a su vez de la clase `QueryExtension`, que permite acceder a la información de la sección *queries* para, por ejemplo, obtener el resultado de cualquier consulta ejecutada previamente. La clase `QueryExtension` hereda a su vez de `ExtensionPoint`. Las nuevas extensiones a las consultas también pueden implementar el método `getKeywords` para especificar las palabras clave que las identifican, tal y como se ha comentado anteriormente.

Cada una de las nuevas sentencias de control debe implementar tanto el método `preprocess` como `postprocess` para especificar el comportamiento que ofrece. El método `preprocess` se llama antes de la ejecución de la consulta mientras que la llamada al método `postprocess` se realiza después y recibe la lista de nodos resultantes de la ejecución de dicha consulta. Estas sentencias de control pueden ser invocadas utilizando la palabra `ext` en la parte de control de una consulta, de forma similar a los operadores de la sección *mappings*. Por ejemplo, la Figura 5.15c muestra una sentencia de control llamada `removeDuplicates` que elimina aquellos elementos de tipo `varDeclaration` cuyo valor de

la hoja `VALUE` está duplicado. La Figura 5.15d muestra un extracto de la implementación de la clase correspondiente para esta nueva sentencia de control. Nótese que solamente es necesario implementar el método `postprocess`, ya que la operación se realiza una vez la consulta ha sido ejecutada.

Por otra parte, los nuevos operadores para las expresiones de filtro deben implementar el método `perform` que heredan de la clase `QueryOperationExtension`. Este método se aplica a cada hoja del nodo al que se aplica el operador y debe devolver un valor booleano que indique si dicha hoja satisface la condición del operador. Por ejemplo, la Figura 5.15c muestra una consulta que utiliza el operador `isSurroundedBy` que comprueba si el valor la hoja `VALUE` del nodo tiene el carácter recibido como parámetro al principio y al final. La Figura 5.15f muestra un extracto de la clase que implementa este nuevo operador.

5.8. Ejemplo

En esta sección se presentará un ejemplo de extracción de modelos utilizado en un caso de estudio para la migración de aplicaciones Delphi a la plataforma Java. Delphi es un lenguaje de programación que es un dialecto de Object Pascal. El lenguaje ha sido utilizado ampliamente en aplicaciones empresariales, especialmente en soluciones RAD. Sin embargo, existe un gran número de aplicaciones desarrolladas en versiones antiguas de Delphi que requieren ser adaptadas o modernizadas (p. ej., para soportar nuevas versiones del lenguaje o migrar a otras plataformas). En nuestro caso de estudio, utilizamos Gra2MoL para obtener modelos conformes a un metamodelo para representar código Delphi a partir de los ficheros de código fuente. Una vez obtenidos estos modelos, pueden aplicarse transformaciones *m2m* para realizar tareas de modernización. A continuación se describirá la transformación Gra2MoL junto con la gramática y el metamodelo utilizados.

Las Figuras 5.16 y 5.17 muestran las partes de la gramática y del metamodelo para Delphi considerados en este ejemplo, respectivamente. Las siguientes secciones expondrán tanto la gramática como el metamodelo así como las reglas de transformación utilizadas en el ejemplo.

5.8.1. La gramática Delphi

La gramática incluye las reglas necesarias para reconocer un subconjunto de sentencias Delphi, algunas de ellas han sido reducidas por simplicidad. La regla gramatical `block` representa bloques Delphi que están compuestos por una sección de declaraciones opcional (regla gramatical `declSection`) y un conjunto de sentencias (regla gramatical `compundStmt`), que puede tener sentencias de exportación delante y/o detrás de su declaración (regla gramatical `exportsStmt`). La regla gramatical `declSection` deriva en las reglas `varDeclaration` y `procFuncDeclaration`, las cuales permiten declarar una variable o un procedimiento/función, respectivamente. Es importante destacar que aunque el ejemplo muestra las reglas gramaticales de declaraciones para ilustrar las referencias entre elementos, éstas han sido profundamente simplificadas por concisión. Las sentencias consideradas en este ejemplo son las de asignación y llamadas a función (las dos alternativas

```
block
  : (declSection)* (exportsStmt)*
    compoundStmt (exportsStmt)*
  ;

declSection
  : varDeclaration
  | procFuncDeclaration
  | ...
  ;

varDeclaration
  : designator ':' type
  | ...
  ;

procFuncDeclaration
  : 'function' designator (formalParam)? ':'
    type ';' block ';'
  | 'procedure' designator (formalParam)? ':'
    block ';'
  ;

compoundStmt
  : 'begin' stmtList 'end'
  ;

stmtList
  : (statement ';')*
  ;

statement
  | designator ':=' expression
  | designator ('(' param ')')?
  | ...
  ;

param
  : expression (',' param)?
  ;

designator
  : ID
  ;

expression
  : expressionAnd ('Or' expressionAnd)*
  ;

expressionAnd
  : simpleExpression ('And' simpleExpression)*
  ;

simpleExpression
  : expressionPart (relOp expressionPart)*
  ;

expressionPart
  : NUMBER
  | STRINGs
  | designator
  | ...
  ;

relOp
  : '=' | '>' | '<' | '<=' | '>=' | '<>'
  | ...
 ;
```

Figura 5.16: Extracto de la gramática Delphi utilizada en el ejemplo.

de la regla `statement` respectivamente).

La gramática también incluye un subconjunto de reglas que son necesarias para reconocer expresiones, las cuales se utilizarán para ilustrar el uso de reglas de tipo *skip*. La regla gramatical `expression` permite definir una operación `OR` lógica donde cada operando se representa por una regla gramatical `expressionAnd`, que a su vez permite definir expresiones lógicas de tipo `AND`. Cada operando de una regla `expressionAnd` se representa por un elemento de tipo `simpleExpression`, que puede utilizar un operador lógico (regla gramatical `relOp`). Los operandos de una regla `simpleExpression` se representan por la regla `expressionPart`, que puede derivar en un número (token `NUMBER`), un valor de tipo cadena (token `STRING`) o una referencia a un elemento (alternativa `designator`).

5.8.2. El metamodelo para Delphi

El metamodelo mostrado en la Figura 5.17 incluye los elementos necesarios para representar las sentencias y expresiones considerados en la gramática del ejemplo. La jerarquía de

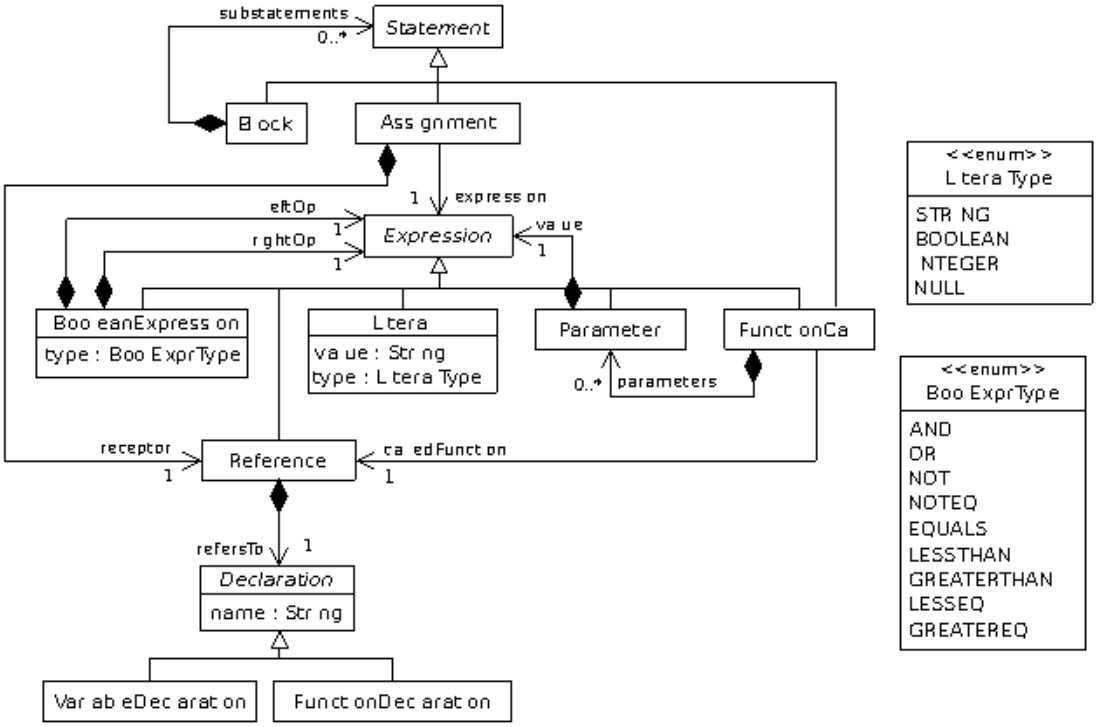


Figura 5.17: Extracto del metamodelo para Delphi utilizado en el ejemplo.

elementos `Statement` representa las sentencias del lenguaje de programación. La Figura incluye la metaclase `Block`, que representa bloques de sentencias (referencia `substatements`); la metaclase `Assignment`, que representa una sentencia de asignación con la parte izquierda (referencia `receptor`) y la parte derecha (referencia `expression`); y la metaclase `FunctionCall`, que representa a una llamada a la función indicando la función llamada (referencia `calledFunction`) y sus parámetros (referencia `parameters`). Es importante destacar que los modelos conformes a este metamodelo permiten representar árboles de sintaxis más ricos que un AST, ya que incorpora metaclases que permiten representar referencias entre elementos del código (como la metaclase `Reference`, explicada más adelante). Así, los modelos son realmente grafos de sintaxis abstracta.

Las expresiones se representan por medio de elementos de la jerarquía `Expression` que incluye las siguientes metaclases: `BooleanExpression` para representar las expresiones lógicas; `Reference`, que permite representar elementos que referencian a otros elementos del árbol de sintaxis (p. ej., el uso de una variable y su declaración o la llamada a una función y su declaración); `Literal` se utiliza para representar valores literales del código; `Parameter`, que representa los parámetros de una función; y, finalmente, `FunctionCall`, que permite representar llamadas a función.

La metaclase `BooleanExpression` tiene referencias a los elementos que constituyen la parte derecha e izquierda de la expresión (referencia `rightOp` y `leftOp`, respectivamen-

te) junto con un atributo que indica el operador de dicha expresión (atributo `type`), que se representa por medio de los valores del enumerado `BoolExprType`. La metaclase `Reference` tiene una referencia al elemento `Declaration` (referencia `refersTo`), que es la raíz de toda declaración en el metamodelo (p. ej., las metaclases `VariableDeclaration` y `FunctionDeclaration`). La metaclase `Literal` incluye los atributos `value` para almacenar el valor literal y `type` para indica el tipo (p. ej., tipo cadena o entero), que se representa por medio de los valores del enumerado `LiteralType`. La metaclase `Parameter` tiene una referencia para indicar el valor de dicho parámetro (referencia `value`). Finalmente, la metaclase `FunctionCall` tiene una referencia a la declaración de la función que es llamada (referencia `calledFunction`) y a los parametros (referencia `parameters`).

5.8.3. Reglas de transformación para las sentencias

En esta sección describiremos las reglas que tratan con las sentencias consideradas en el ejemplo y en la siguiente sección aquellas que tratan con expresiones.

La Figura 5.18 muestra el conjunto de reglas utilizadas para transformar bloques de sentencias. La regla `mapBlock` comienza la ejecución de la transformación. Esta regla tiene solamente un *binding* cuya parte derecha es un identificador de consulta (`stats`) y cuya parte izquierda especifica la referencia `subStatements` de la metaclase `BlockStatement`. Una vez la consulta es ejecutada, se localizan las reglas que conforman con el *binding* y se ejecutan para cada elemento resultado de la consulta. En este caso, tanto la regla `mapCallFunction` como la regla `mapAssignment` conforman al *binding*, sin embargo, el filtro de la sección *from* solamente permite la selección de una de ellas, dependiendo de la existencia del token `:=` en el elemento gramatical `statement`.

La regla `mapAssignment` define las correspondencias entre el elemento gramatical `statement` que contiene el token `:=` y la metaclase `Assignment`. Esta regla crea una instancia de la metaclase `Assignment` y sus consultas obtienen los operandos de la expresión (`lElem` y `rElem`, respectivamente). La regla incluye un conjunto de *bindings* para inicializar las referencias `receptor` y `expression`. Estos *bindings* tienen como parte derecha un identificador de consulta (`lElem` y `rElem`, respectivamente) y como parte izquierda una referencia (`leftOp` y `rightOp`, respectivamente). El *binding* que utiliza la consulta `lElem` provoca la ejecución de la regla `locateFromDesignator` ya que el resultado de la consulta es de tipo `designator` y es la única que conforma con dicho *binding*. Por otro lado, dado que el resultado de la consulta `rElem` es de tipo `expression`, el último *binding* provoca la ejecución de la regla que trata con expresiones, que se tratarán en la siguiente sección.

La regla `locateFromDesignator` define las correspondencias entre el elemento gramatical `designator` y la metaclase `Reference`. La regla crea una instancia de la metaclase `Reference` y su finalidad es localizar el elemento referenciado en el código fuente. De esta forma, las consultas de esta regla se encargan de localizar la declaración de la variable o del procedimiento/función (consultas `varloc` y `metloc`, respectivamente). Estas consultas utilizan el operador `//` para encontrar aquel elemento gramatical `designator` con el mismo identificador que el elemento recibido por la regla, facilitando la definición de la consulta por el árbol de sintaxis. Nótese que dicha referencia es una referencia cruzada entre elemento del árbol de sintaxis y la facilidad para definir la consulta para

```
toprule 'mapBlock'
  from block b
  to   Block
  queries
    stats : /b/compoundStmt//#statement;
  mappings
    subStatements = stats;
end_rule

rule 'mapAssignment'
  from statement{TOKEN[0].eq(":=")} st
  to   Assignment
  queries
    lElem : /st/#designator;
    rElem : /st/#expression;
  mappings
    receptor   = lElem;
    expression = rElem;
end_rule

rule 'mapFunctionCall'
  from statement{!TOKEN[0].eq(":=")} st
  to   FunctionCall
  queries
    dElem : /st/#designator;
    eElem : /st///#expression;
  mappings
    calledFunction = dElem;
    parameters     = eElem;
end_rule

rule 'mapParameter'
  from  expression exp
  to    Parameter
  queries
  mappings
    value = exp;
end_rule
```

```
rule 'locateFromDesignator'
  from designator d
  to   Reference
  queries
    varloc : //#varDeclaration
             //designator{ID.eq(d.ID)};
    metloc : //#procFuncDeclaration
             //designator{ID.eq(d.ID)};
  mappings
    if(metloc.hasResults) then
      refersTo = metloc;
    else
      refersTo = varloc;
    end_if
end_rules

rule 'mapProcFuncDeclaration'
  from  procFuncDeclaration pfDecl
  to    astm::gastm::FunctionDefinition
  queries
    ...
  mappings
    ...
end_rule

rule 'mapVariableDeclaration'
  from  varDeclaration varDecl
  to    astm::gastm::VariableDefinition
  queries
    ...
  mappings
    ...
end_rule
```

Figura 5.18: Reglas utilizadas en el ejemplo.

resolverla. Por otro lado, en la sección *mappings*, una sentencia condicional se encarga de comprobar si se ha encontrado una variable o un procedimiento/función, es decir, cuál de las consultas pertinentes ha dado resultado y posteriormente inicializa la referencia `refersTo`. Si se encuentra un procedimiento/función, se evalúa el *binding* que utiliza la consulta `metloc` la cual ejecuta la regla `mapProcFuncDeclaration`, ya que es la única que conforma al *binding*. Si se encuentra una variable, se evalúa el *binding* que utiliza la consulta `varloc`, la cual ejecuta la regla `mapVariableDeclaration`. En ambos casos, la ejecución de las reglas `mapProcFuncDeclaration` y `mapVariableDeclaration` permite inicializar la referencia `refersTo` para que apunte a la instancia de `FunctionDeclaration` o `VariableDeclaration`, respectivamente. Debido a que el ejemplo solamente trata con las reglas que consideran las sentencias de asignación y llamada a función, estas últimas dos

reglas no se describen completamente.

La regla `mapFunctionCall` define las correspondencias entre el elemento gramatical `statement` que no contiene el token := y la metaclase `FunctionCall`. Esta regla también crea una instancia de `FunctionCall` y sus consultas obtienen el nombre del procedimiento/-función (consulta `dElem`) y el conjunto de parámetros (consulta `eElem`). La última consulta ilustra el significado del operador `///`. Debido a que la regla gramatical `param` es definida recursivamente, el operador `///` permite recorrer el CST recursivamente para obtener todos los elementos de tipo `param`. Esta regla incluye un conjunto de *bindings* para inicializar las referencias `calledFunction` y `parameteres` de la instancia de metaclase creada. El primer *binding* tiene como parte derecha un identificador de consulta (consulta `dElem`) y como parte izquierda la referencia `calledFunction`. Una vez que la consulta ha sido ejecutada, la regla que conforma al *binding* es localizada y ejecutada. En este caso, la única regla que puede ser ejecutada es `locatefromDesignator`, que ya ha sido explicada anteriormente. El segundo *binding* de la regla `mapCallFunction` tiene como parte derecha un identificador de consulta (consulta `eElem`) y como parte izquierda la referencia `parameters`. En este caso, una vez es ejecutada la consulta, se ejecuta la regla `mapParameter` para cada elemento resultado, ya que es la única que conforma al *binding*.

La regla `mapParameter` define las correspondencias entre el elemento gramatical `expression` y la metaclase `Parameter`. La regla crea una instancia de la metaclase `Parameter` y contiene solamente un *binding*, cuya parte derecha es el elemento gramatical recibido en la sección *from* de la regla y cuya parte izquierda es la referencia `value`. Las reglas que conforman este *binding* son las que tratan con expresiones, que se explicarán en la siguiente sección.

5.8.4. Reglas de transformación para las expresiones

La Figura 5.19 muestra la lista de reglas que tratan con las expresiones utilizadas en el ejemplo. Cuando se definen las reglas de transformación que tratan con las expresiones, el patrón a seguir es el siguiente. Para cada regla gramatical que trate con expresiones, deben añadirse dos reglas. La primera de ellas es una regla de tipo *skip* que trata con el elemento gramatical que no contiene el token que representa al operador y la segunda es una regla *normal* que trata con el elemento gramática que sí contiene al token. La nueva regla de tipo *skip* debe transferir la ejecución a la siguiente regla gramatical que trata con expresiones, mientras que la segunda debe transformar el elemento gramatical en la correspondiente instancia de metaclase que represente a dicha expresión. Por ejemplo, las reglas `skipOr` y `mapOr` tratan con el elemento gramatical `expression`. La regla `skipOr` es una regla de tipo *skip* que transfiere la ejecución de la transformación al elemento gramatical `expressionAnd` si el elemento `expression` no tiene el token `OR`. Por otro lado, la regla `mapOr` es una regla normal que trata con el elemento gramatical `expression` que contiene el token `OR`.

Así, cuando un elemento gramatical `expression` está siendo evaluado (p. ej., el último *binding* de la regla `mapParameter`), pueden ejecutarse las reglas `skipOr` o `mapOr` dependiendo de la existencia del token `OR`. Si el token no existe, la regla `skipOr` es ejecutada y la consulta `next` localiza el siguiente elemento gramatical que trata con expresiones, es decir, el elemento `expressionAnd`. A continuación se ejecuta la sentencia `skip` en la sección

Operador Or

```
skip rule 'skipOr'
 from expression{!TOKEN[0].exists} exp
 to   Expression
 queries
  next : /exp/#expressionAnd
 mappings
  skip next;
end rule
```

```
rule 'mapOr'
 from expression{TOKEN[0].exists} exp
 to   BooleanExpression
 queries
  lElem : /exp/#expressionAnd[0];
  rElem : /exp/#expressionAnd[1];
 mappings
  leftOp  = lElem;
  type    = "OR";
  rightOp = rElem;
end rule
```

Operador And

```
skip rule 'skipAnd'
 from expressionAnd{!TOKEN[0].exists} exp
 to   Expression
 queries
  next : /exp/#simpleExpression
 mappings
  skip next;
end rule
```

```
rule 'mapAnd'
 from expressionAnd{TOKEN[0].exists} exp
 to   BooleanExpression
 queries
  lElem : /exp/#simpleExpression[0];
  rElem : /exp/#simpleExpression[1];
 mappings
  leftOp  = lElem;
  type    = "AND";
  rightOp = rElem;
end rule
```

Operador =

```
rule 'skipRelOp'
 from simpleExpression exp
 to   Expression
 queries
  next : /exp/expressionPart
 mappings
  skip next;
end rule
```

```
rule 'mapRelOp'
 from simpleExpression/relOp{TOKEN[0].eq("=")} exp
 to   BooleanExpression
 queries
  lElem : /exp/#expressionPart[0];
  rElem : /exp/#expressionPart[1];
 mappings
  leftOp  = lElem;
  type    = "EQUALS";
  rightOp = rElem;
end rule
```

```
rule 'mapNumber'
 from  expressionPart
       {NUMBER.exists} exp
 to    Literal
 queries
 mappings
  type  = "INTEGER";
  value = exp.NUMBER;
end rule
```

```
rule 'mapString'
 from  expressionPart
       {STRING.exists} exp
 to    Literal
 queries
 mappings
  type  = "STRING";
  value = exp.STRING;
end rule
```

```
skip rule 'skipDesignator'
 from  expressionPart/designator exp
 to    Expression
 queries
  next : /exp/#designator;
 mappings
  skip next;
end rule
```

Figura 5.19: Reglas de tipo *skip* utilizadas en el ejemplo.

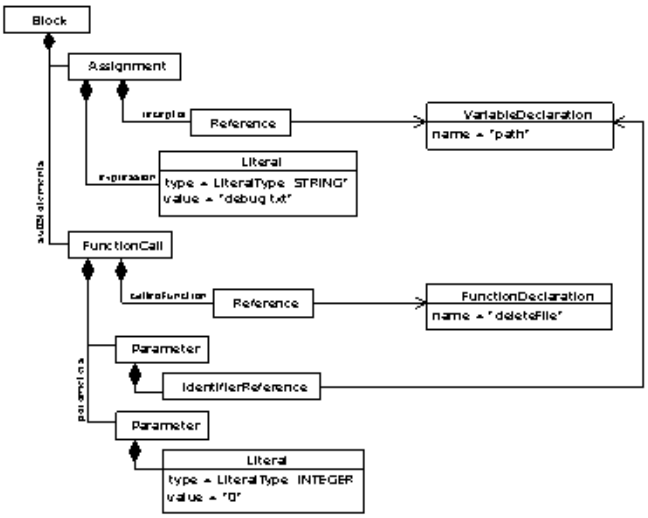
mappings de la regla, que transfiere la ejecución a aquella regla cuya sección *from* conforma con el elemento `expressionAnd` y la sección *to* lo hace con la metaclase `Expression`. En este caso, las reglas candidatas son `skipAnd` y `mapAnd`. Por otro lado, si el token `OR` existe, se ejecuta la regla `mapOr`, se crea una instancia de la metaclase `BooleanExpression` y se inicializa tanto el operador (atributo `type`) como sus operandos (referencias `leftOp` y `rightOp`), que causan la ejecución de aquellas reglas que conforman a estos bindings. En este caso, las reglas candidatas son también `skipAnd` y `mapAnd`. Por cuestión se simplicidad, las reglas mostradas solamente tratan con expresiones con dos operandos.

Nótese que el patrón es repetido en las reglas `skipAnd` y `mapAnd` para el elemento gramatical `expressionAnd` así como en las reglas `skipRelOp` y `mapRelOp` para el elemento gramatical `simpleExpression`. Por simplicidad, la regla `mapRelOp` solo trata con el ope-

```
path : string;
procedure deleteFile (path : string, mode : integer);
begin
  ...
end

begin
  path := 'debug.txt';
  deleteFile(path, 0);
end
```

(a)



(b)

Figura 5.20: (a) Código Delphi utilizado en el ejemplo. (b) Modelo resultante al aplicar la transformación al código Delphi de ejemplo.

rador =. Las reglas `mapNumber`, `mapString` y `skipDesignator` son las últimas que tratan con las expresiones. Las reglas `mapNumber` y `mapString` crean una instancia de la metaclase `Literal`, inicializan el valor del atributo `type` a `INTEGER` o `STRING`, respectivamente, y almacenan el valor del elemento literal en el atributo `value`. Por otro lado, la regla `skipDesignator` transfiere la ejecución a aquellas reglas que tratan con el elemento gramatical `designator`, es decir, la regla `locateFromDesignator` descrita anteriormente. Es importante destacar que en este caso el uso de la regla de tipo *skip* difiere ligeramente del uso para expresiones. En este caso, la regla de tipo *skip* es utilizada para transferir la ejecución a aquella regla que se encarga de transformar un elemento concreto (el elemento gramatical `designator`), permitiendo al desarrollador controlar el flujo de ejecución de la transformación Gra2MoL.

La Figura 5.20a muestra un fragmento de código Delphi y la Figura 5.20b el modelo creado como resultado de ejecutar las reglas de transformación anteriores. Nótese que la transformación se inicia con el código enmarcado de la Figura 5.20a.

5.9. Escenarios de aplicación

Los puentes entre el *grammarware* y el *modelware* están principalmente motivados para aplicar tareas de ingeniería inversa, en concreto, de extracción de modelos. Tal y como se ha comentado anteriormente, un proceso de modernización o reingeniería software utiliza los modelos extraídos para generar los artefactos del sistema destino. Gra2MoL está principalmente diseñado para cubrir esta necesidad en la modernización de sistemas software, permitiendo definir las transformaciones que llevan a cabo el proceso de extracción.

Sin embargo, es importante destacar que realmente el lenguaje puede ser utilizado para definir transformaciones *t2m* de cualquier definición textual conforme a una gramática libre de contexto, lo que permite ser aplicado en otros escenarios de aplicación. En primer lugar, Gra2MoL puede utilizarse para definir DSL externos textuales. Tal y como se ha comentado en la sección 3.3.2, la sintaxis concreta de un DSL externo textual está definida normalmente por su gramática, mientras que las sintaxis abstracta se representa mediante un metamodelo. Por lo tanto, una vez se disponen de la gramática y del metamodelo del DSL, Gra2MoL permite definir una transformación entre ambos para obtener modelos conformes al metamodelo de sintaxis abstracta a partir de definiciones textuales conformes a la gramática. De hecho, los tres DSL presentados en esta tesis utilizan transformaciones Gra2MoL para obtener el modelo de sintaxis abstracta (en el caso de Gra2MoL utilizando el proceso de *bootstrap*, tal y como se ha comentado en la sección 5.6).

En segundo lugar, Gra2MoL puede utilizarse para implementar metamodelos descritos por una gramática. En estos casos, la transformación Gra2MoL trata con la gramática que describe el metamodelo (p. ej., la utilizada en la especificación del metamodelo ASTM [51]) y con el meta-metamodelo, el cual permite definir metamodelos (p. ej., el meta-metamodelo Ecore). Este procedimiento es el que se ha aplicado para obtener el metamodelo GASTM a partir de la especificación ASTM, tal y como comentaremos en la sección 6.1, permitiendo obtener fácilmente la última versión del metamodelo conforme se publican nuevas especificaciones.

Finalmente, Gra2MoL también puede utilizarse en el ámbito de la modernización de bases de datos, en particular, en aquellos escenarios de modernización que traten con la definición del esquema de la base de datos. De esta forma, Gra2MoL podría ser aplicado para obtener modelos que representen el esquema de la base de datos relacional a partir de la definición textual de dicho esquema y aplicar tareas de reingeniería o modernización como la optimización de la definición de las tablas o su adaptación a nuevos requisitos. En este sentido, Gra2MoL podría ser utilizado en escenarios de contextualización de datos de forma parecida a como se describe en [60]. En este tipo de escenario, los modelos extraídos tanto del código como del esquema se utilizan para detectar las relaciones entre ellos, esto es, qué partes de los datos son utilizadas por el código.

5.10. Características del lenguaje

La Figura 5.21 muestra las características principales de Gra2MoL de acuerdo al diagrama de características presentado en [20], el cual describe un *framework* basado en modelos

de características para la clasificación de lenguajes de transformación. Gra2MoL es un lenguaje unidireccional cuyo dominio en el origen es el *grammarware* y el dominio en el destino es el *modelware*. Una transformación Gra2MoL está compuesta por un conjunto de reglas que transforman elementos de la gramática en elementos del modelo extrayendo la información necesaria del CST del código fuente por medio de un potente lenguaje de consultas. Cada vez que se ejecuta una transformación Gra2MoL se crea un nuevo modelo, es decir, el proceso de transformación no permite crear modelos incrementalmente. Las reglas se resuelven implícitamente y de forma determinista utilizando el mecanismo de *bindings*, aunque el desarrollador puede alterar la planificación de reglas utilizando reglas de tipo *skip*. Ademas, la condición de aplicación de las reglas depende del filtro de la sección *from*. El lenguaje también incorpora reglas de tipo *mixin* como mecanismo de reuso así como reglas de tipo *copy* para permitir transformar un elemento gramatical más de una vez. En cuanto a la información de trazabilidad, el motor de ejecución de Gra2MoL crea un modelo de trazas automáticamente que relaciona los elementos creados con los elementos origen y las reglas aplicadas, aunque el lenguaje no tiene mecanismos para consultar dicha información. Además, Gra2MoL no incorpora ningún mecanismo de modularidad.

5.11. Conclusiones

En este capítulo se ha descrito el lenguaje Gra2MoL, un DSL la extracción de modelos a partir de código fuente, el cual está basado en reglas de tipo ATL y utiliza un lenguaje de consultas especialmente adaptado para extraer información del árbol de sintaxis que representa el código fuente. Desde nuestro conocimiento, Gra2MoL es el primer lenguaje disponible para definir este tipo de transformaciones. En el sitio web `http://modelum.es/gra2mol` pueden encontrarse todos los recursos que componen la herramienta.

Con el objetivo de analizar las características básicas ofrecidas por Gra2MoL, se discutirá cómo el lenguaje cumple con los requisitos identificados en [47], los cuales pueden servir de guía para la construcción de DSL de calidad. Cada requisito se evalúa de menor a mayor cumplimiento utilizando los valores 1 a 5.

Conformidad (5). Todos los elementos utilizados en Gra2MoL corresponden o hacen referencia a conceptos del dominio del problema de establecer el puente unidireccional entre el *grammarware* y el *modelware* necesario para extraer modelos de texto que es conforme a un gramática. Las reglas permiten definir las correspondencias entre los elementos de la gramática y los del metamodelo. Así, La sección *from* de una regla hace referencia al elemento gramatical mientras que la sección *to* lo hace al elemento del metamodelo. El lenguaje utiliza el concepto de *binding* para establecer las correspondencias entre las propiedades del elemento del metamodelo y la información del elemento gramatical, el cual ha sido especialmente adaptado para trabajar en este dominio. Además, el lenguaje de consultas ofrece operadores para facilitar la navegación del árbol de sintaxis que representa el código fuente conforme a la gramática utilizada.

Ortogonalidad (5). En el lenguaje no existen elementos que representen más de un concepto del dominio. Cada elemento del lenguaje tiene una finalidad que no es utilizada

por ningún otro elemento.

Soporte (5). Gra2MoL ofrece un conjunto de herramientas para facilitar la definición y ejecución de transformaciones. El lenguaje se distribuye como un *plugin* de la plataforma Eclipse el cual incorpora el motor de ejecución así como un editor específico con ayudas al desarrollador (p. ej., resaltado de sintaxis, vista *outline* o autocompletado). Además, el lenguaje también puede ser ejecutado independientemente por medio de tareas ANT.

Integración (4). El lenguaje, al haber sido desarrollado en Java como *plugin* de Eclipse, es fácilmente integrable con otras herramientas o lenguajes, aunque deben estar incluidas en esta plataforma.

Extensibilidad (5). Gra2MoL ofrece un mecanismo de extensiones que permite incorporar nuevos operadores en la sección *mappings* así como al lenguaje de consultas. Esta característica le permite adaptarse a problemas específicos.

Longevidad (4). El hecho de que la extracción de modelos a partir del código fuente sea una actividad fundamental en todo proceso de modernización de software que utilice el paradigma DSDM, aporta a Gra2MoL un grado elevado de utilidad en un largo período de tiempo.

Simplicidad (4). El proceso de definición de una transformación en Gra2MoL consiste en: (1) definir las reglas que establecen las correspondencias entre los elementos del *grammarware* y del *modelware* y (2) definir las consultas para extraer la información del código, utilizando un lenguaje de consultas que incorpora operadores para facilitar la extracción de información desde el código fuente. Estas tareas no tienen un grado de complejidad elevado y el esfuerzo necesario para realizarlas solamente depende del tamaño y complejidad de la gramática y del metamodelo involucrados en la transformación, los cuales deben ser conocidos exhaustivamente.

Calidad (4). Este requisito hace referencia a la calidad de los artefactos generados por el DSL. Al ser Gra2MoL un lenguaje interpretado, no se genera código a partir de una definición de transformación, por lo que solamente pueden ser considerados los modelos resultantes de un proceso de transformación. Por otro lado, la calidad de dichos modelos no depende intrínsecamente del proceso de transformación sino de la definición del metamodelo y las reglas realizado por el desarrollador.

Escalabilidad (3). La extracción de modelos desde el código fuente trata normalmente con un gran número de ficheros de los que está compuesto un sistema software. Dado que las consultas utilizadas en un proceso de transformación se aplican sobre el árbol de sintaxis (CST), que realmente se representa como un modelo, es de vital importancia asegurar la escalabilidad en el acceso a dicho árbol. Por este motivo, Gra2MoL utiliza CDO como repositorio de modelos para almacenar el árbol de sintaxis y facilitar el acceso al CST, favoreciendo la escalabilidad. Por otro lado, también es importante considerar el tamaño de los modelos resultantes de una transformación. En este caso, también sería interesante considerar el uso de repositorios de modelos para mejorar la escalabilidad, lo cual está identificado como trabajo futuro de la herramienta.

Usabilidad (4). Gra2MoL está inspirado en lenguajes de transformación *m2m*, facilitando su adopción por los desarrolladores de la comunidad del DSDM acostumbrado a la definición de este tipo de transformaciones, especialmente utilizando ATL.

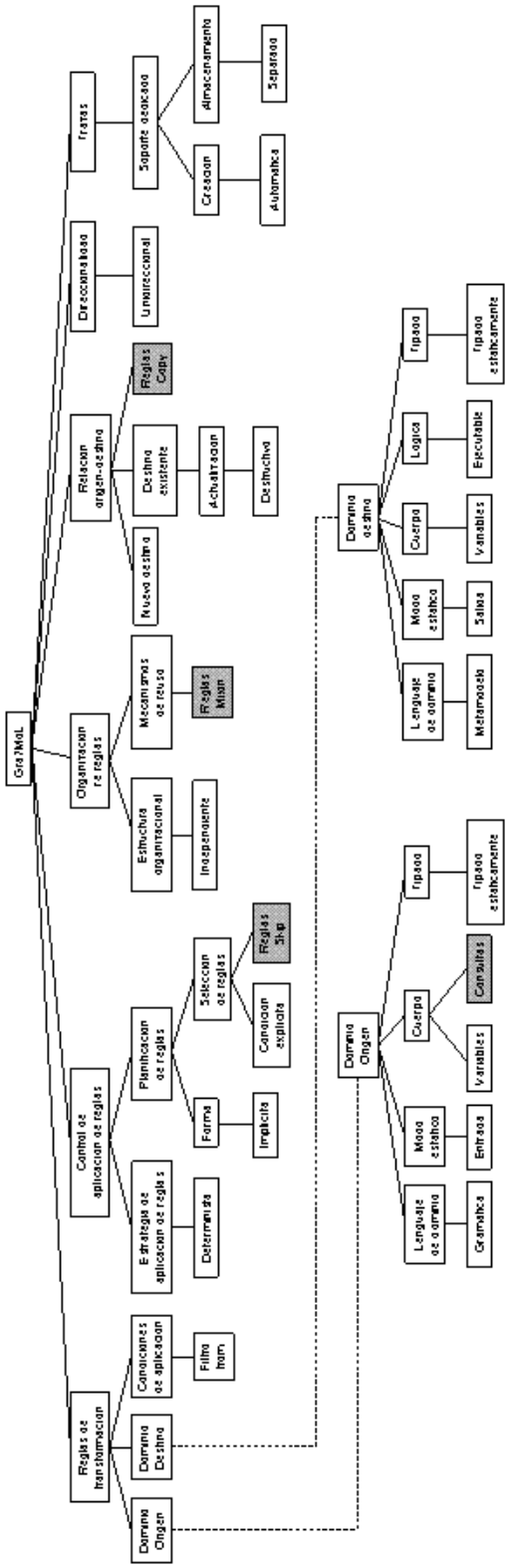


Figura 5.21: Diagrama de características de Gra2MoL de acuerdo a [20]. Se marcan en gris las características propias del lenguaje.

6

Extracción de modelos KDM utilizando transformaciones de modelos

Haciendo caso omiso de los proyectiles de artillería y las ondas expansivas de los explosivos, los exterminadores se abrieron paso por la fuerza.
Horus, señor de la guerra. Volumen I, Dan Abnett

Como ya se comentó en el capítulo 3, ADM es considerada la principal iniciativa para la modernización de software basada en modelos. Sin embargo, en la actualidad el número de experiencias prácticas que utilicen los metamodelos propuestos por ADM es muy limitado [59, 60, 104]. Esta falta de casos prácticos puede estar principalmente motivada por el hecho de que el metamodelo principal de ADM, denominado KDM, fue publicado tardíamente en el 2008. Por este motivo, decidimos aplicar Gra2MoL en el contexto de ADM, permitiéndonos probar el lenguaje con un caso de estudio real y valorar la iniciativa ADM, en particular, el uso de los metamodelos ASTM y KDM. Aplicamos ADM a un caso de estudio de construcción de una herramienta de cálculo de métricas para aplicaciones Oracle Forms, destinadas a cuantificar el esfuerzo de migración de este tipo de aplicaciones. En este caso de estudio ilustramos las dos tareas principales de todo proceso basado en ADM: extracción de modelos KDM y su uso para generar los artefactos implicados en el proceso de modernización. En este capítulo describiremos cómo las transformaciones de modelos (*t2m*, *m2m* y *m2t*) pueden automatizar estas tareas.

6.1. Una herramienta y proceso de modernización con ADM

La falta de herramientas y procesos que utilicen ADM y sirvan de ejemplo a los desarrolladores que trabajen en tareas de modernización dificulta considerablemente la adopción de esta iniciativa. Por este motivo, decidimos experimentar con los metamodelos propuestos por ADM en un caso de estudio real donde se extrajesen modelos KDM a partir del

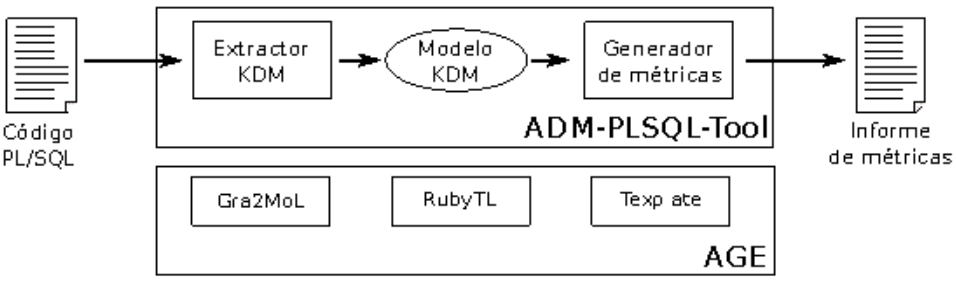


Figura 6.1: Visión general de la herramienta desarrollada. La herramienta utiliza los metamodelos ADM para generar informes de métricas automáticamente a partir de código PL/SQL.

sistema software y posteriormente una herramienta aplicase una tarea de modernización con dichos modelos. La herramienta de modernización que presentamos es fruto de nuestra experiencia en un proyecto de migración de aplicaciones Oracle Forms a la plataforma Java, donde una parte importante del proceso de migración está relacionada con los *triggers* escritos en PL/SQL del sistema heredado. Desde nuestra experiencia, un aspecto crucial que determina el tiempo y el esfuerzo requeridos para migrar estos *triggers* es su nivel de acoplamiento con la interfaz de usuario, esto es, el número y tipo de sentencias que acceden a la interfaz gráfica. Por lo tanto, disponer de una herramienta capaz de analizar dicho acoplamiento sería de gran utilidad para estimar los costes de migración del sistema. Esta herramienta servirá de ejemplo práctico para ilustrar el proceso basado en ADM, la cual puede ser descargada desde `http://modelum.es/gra2mol/metrics-adm`.

La extracción de modelos KDM a partir del código fuente es una tarea crucial en la aplicación de ADM debido a que es el primer paso para representar el sistema software. Estos modelos constituyen el punto de partida para la automatización del proceso de modernización, ya que son utilizados para aplicar las principales tareas de modernización. Por ejemplo, los modelos KDM pueden utilizarse para crear vistas arquitecturales, generar código fuente o crear informes para tomar decisiones en el proceso de modernización, como es nuestro caso, donde generaremos informes de métricas para conocer el esfuerzo requerido para migrar una aplicación Oracle Forms.

Nuestra herramienta por lo tanto está compuesta por dos elementos principales (ver Figura 6.1): un *extractor*, que genera modelos KDM a partir del código PL/SQL; y un *generador de informes de métricas* para modelos KDM . Esta herramienta está construida sobre la plataforma AGE (*Agile Generative Environment*) [85], que ofrece un conjunto de DSL para realizar tareas básicas de DSDM (p. ej., RubyTL como lenguaje de transformación *m2m* y Texplate como lenguaje de transformación *m2t*). Por otro lado, para la extracción de modelos a partir del código fuente hemos utilizado Gra2MoL.

Antes de describir cómo se ha implementado cada componente de la herramienta, la Figura 6.2 muestra una visión general del proceso basado en ADM que se ha aplicado. Como puede observarse, el proceso de extracción de modelos KDM se ha separado en dos pasos debido a la gran diferencia semántica que existe entre los conceptos del lenguaje de

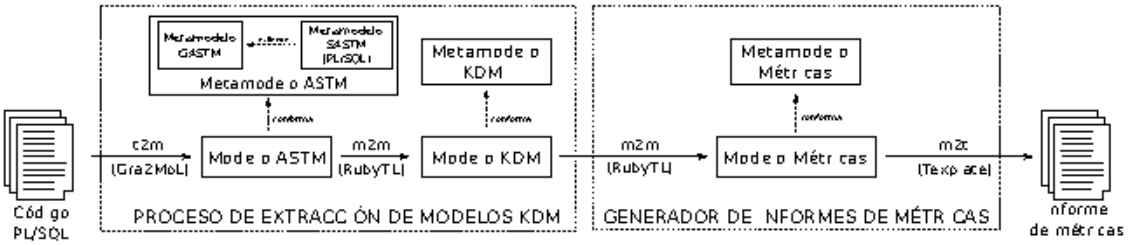


Figura 6.2: Proceso ADM compuesto de dos pasos: extracción de modelos KDM y generación de informes de métricas. Los elementos indicados entre paréntesis particularizan el proceso a nuestro ejemplo de herramienta.

programación y los del metamodelo KDM. En primer lugar, se aplica una transformación *t2m* para extraer modelos ASTM a partir del código fuente y, a continuación, una transformación *m2m* se encarga de generar los modelos KDM a partir de dichos modelos ASTM. Para realizar el paso de código PL/SQL (concretamente, el código PL/SQL de los *triggers*) a ASTM aplicamos una transformación Gra2MoL, mientras que para el paso de ASTM a KDM utilizamos el lenguaje de transformación *m2m* RubyTL.

Una vez se dispone de los modelos KDM, se aplican transformaciones de modelos para obtener los artefactos deseados en modernización. En nuestro caso, se definió una transformación *m2m* en RubyTL para obtener modelos de métricas a partir de los modelos KDM extraídos. Tal y como se comentó en la sección 3.4.1, ADM proporciona el metamodelo SMM para la representación de métricas y mediciones en modernización. Sin embargo, debido a que durante el desarrollo de nuestra herramienta este metamodelo todavía no se había publicado, decidimos definir un metamodelo propio para la representación de las métricas identificadas, en concreto, los niveles de acoplamiento de los *triggers*. De esta forma, se aplica una transformación *m2m* en RubyTL del metamodelo KDM al métamodelo de métricas definido y, finalmente, se generan los informes a partir de los modelos de métricas por medio de una tranformación *m2t* en Texplate.

Es importante destacar que mientras la extracción de modelos ASTM y su transformación a modelos KDM es un paso común en todo proceso de modernización basado en ADM, lo que se hace a continuación con los modelos KDM depende de la tarea a automatizar en el proceso. Nuestra herramienta aplica un proceso de transformación para obtener modelos de métricas, pero otro ejemplo sería la generación de código fuente a partir de los modelos KDM del sistema origen en un proyecto de migración, el cual se realizaría en dos pasos: (1) aplicación de transformaciones *m2m* para transformar los modelos KDM del sistema original en modelos KDM que representen la arquitectura del nuevo sistema y (2) aplicación de transformaciones *m2t* para generar el código del nuevo sistema. A continuación se describirán cada uno de los pasos del proceso desarrollados en nuestra herramienta.

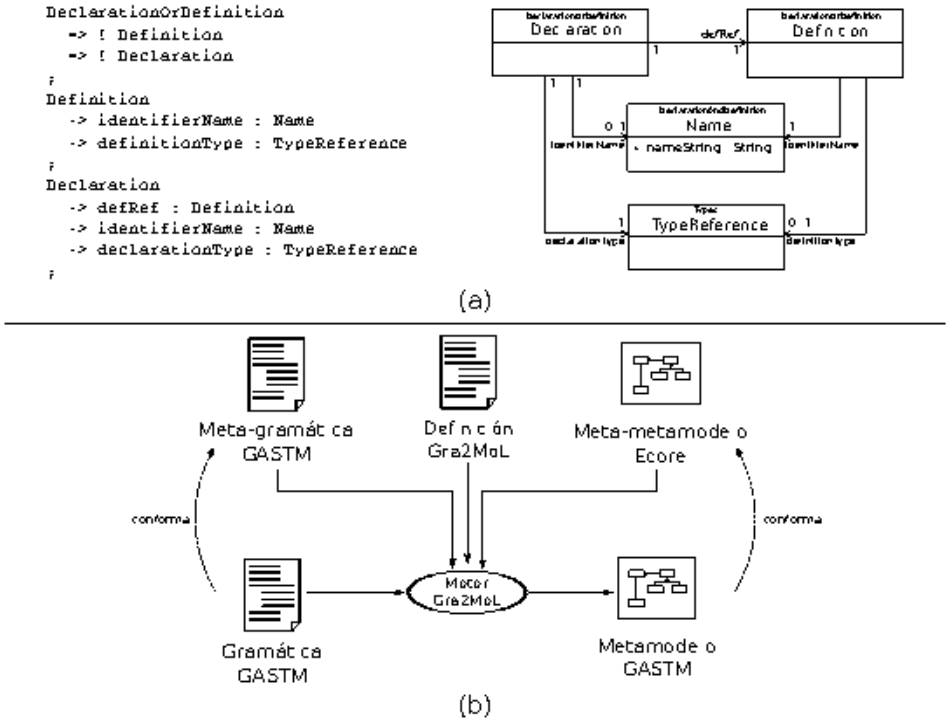


Figura 6.3: (a) Descripción del metamodelo GASTM de forma textual y gráfica según el documento de especificación. (b) Proceso Gra2MoL aplicado para extraer el metamodelo GASTM.

6.2. Extracción de modelos ASTM

Para extraer modelos ASTM a partir del código fuente se aplican transformaciones *t2m*, las cuales establecen las correspondencias entre los elementos del código y los elementos del metamodelo ASTM. El metamodelo ASTM está formado por el metamodelo GASTM y el metamodelo SASTM del lenguaje de programación que se está utilizando, el cual debe crearse si no existe para dicho lenguaje.

Dado que en el momento de crear la herramienta el metamodelo GASTM no estaba actualizado a la última versión descrita en la especificación de ASTM, un paso previo a la extracción de modelos ASTM consistió en la actualización de dicho metamodelo. Para realizar este paso utilizamos el lenguaje Gra2MoL, el cual puede utilizarse para generar automáticamente un metamodelo a partir de una gramática que lo describe, tal y como se comenta en la sección 5.9. La especificación ASTM contiene la descripción de la sintaxis abstracta de GASTM en dos formatos principales: de forma textual, mediante una gramática expresada en una variante de la notación BNF; y de forma gráfica, utilizando diagramas UML para expresar el metamodelo. La Figura 6.3a muestra ambas representaciones de una parte del metamodelo GASTM.

Para aplicar Gra2MoL se definió una meta-gramática que reconociera la variación de

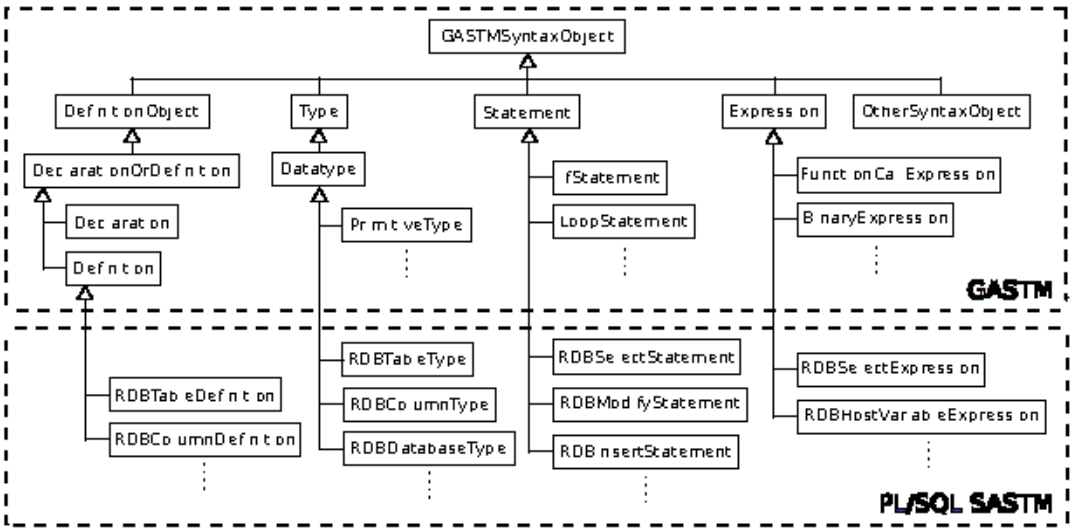


Figura 6.4: Una parte de los metamodelos GASTM y SASTM para el lenguaje PL/SQL. Solamente se muestran los elementos sintácticos.

BNF que describe el metamodelo ASTM y se establecieron las correspondencias entre los elementos de dicha meta-gramática y los elementos del meta-metamodelo Ecore. Tal y como se ilustra en la Figura 6.3b, el motor de transformación Gra2MoL toma la especificación textual del metamodelo GASTM, que es conforme a la meta-gramática definida, y genera un metamodelo que es conforme al meta-metamodelo Ecore. La ventaja de utilizar este proceso es la posibilidad de obtener automáticamente la última versión del metamodelo a partir de la especificación.

Por otro lado, el metamodelo SASTM para PL/SQL tuvo que ser definido para poder representar los elementos específicos del lenguaje. La Figura 6.4 muestra una parte de los metamodelos GASTM y SASTM utilizados, donde los elementos del metamodelo SASTM heredan de los elementos del GASTM. Por ejemplo, las sentencias de definición de datos (p. ej., `RDBTableDefinition`, que representa la sentencia de creación de tablas; o `RDBColumnDefinition` para crear columnas en una tabla) heredan del elemento `Definition` del GASTM, que representa cualquier elemento de definición en el lenguaje. Por otro lado, las sentencias de manipulación de datos (p. ej., `RDBSelectStatement` para las sentencias `select`; o `RDBInsertStatement` para las sentencias `insert`) heredan del elemento `Statement` del GASTM, que representa a cualquier sentencia del lenguaje.

Tal y como se ha comentado anteriormente, definimos una transformación Gra2MoL para extraer modelos ASTM a partir del código PL/SQL. Las reglas de transformación Gra2MoL utilizadas se clasifican en cuatro tipos según el tipo de información con la que traten: declaraciones, tipos de datos, sentencias y expresiones. Las reglas que tratan con las declaraciones son normalmente muy simples ya que deben crear el elemento del modelo que corresponde al elemento gramatical (p. ej., la declaración de una variable) y asignarle el tipo de datos. Cuando el tipo de datos es primitivo, las reglas simplemente deben realizar una

asignación con los elementos del metamodelo ASTM que representan dichos elementos. Sin embargo, si se utilizan tipos definidos por el usuario, es necesario definir reglas que tratan con este tipo de elementos para localizar su declaración en el código y crear el elemento del modelo que los representen. Por otro lado, las reglas que tratan con las sentencias también requieren normalmente localizar diferentes elementos de información del código para inicializar las propiedades del elemento del modelo. Finalmente, las reglas que tratan con las expresiones utilizan reglas de tipo *skip* para llevar a cabo la extracción de modelos.

Como puede observarse, en general, la tarea principal a llevar a cabo durante el proceso de extracción de modelos a partir del código fuente consiste en agrupar información dispersa en el código para crear los elementos del modelos. Tal y como se ha comentado en la sección 5.1, esta dispersión de información está principalmente provocada por el uso de referencias implícitas en el código que deben ser transformadas en referencias explícitas en el modelo. El lenguaje de consultas de Gra2MoL facilitó la tarea de localizar esta información e inicializar las propiedades de los elementos del modelo así como establecer las referencias entre dichos elementos, permitiendo de esta forma que los modelos ASTM resultantes sean compatibles a nivel 1.

6.3. Generación de modelos KDM

La generación de los modelos KDM se lleva a cabo por medio de la aplicación de una serie de transformaciones *m2m*. En primer lugar, una transformación *m2m* genera modelos KDM de nivel 0 a partir de los modelos ASTM. Los modelos KDM de nivel 0 están compuestos a su vez por dos modelos: el modelo de código y de inventario. El modelo de código representa el código fuente del sistema mientras que el modelo de inventario es un catálogo de los artefactos que componen el sistema (p. ej., ficheros y directorios). A partir de estos modelos KDM de nivel 0, se aplican de nuevo transformaciones *m2m* para generar uno o más modelos KDM de nivel 1, dependiendo de la vista arquitectural que se desee. En nuestro caso, fue necesario definir la transformación *m2m* para obtener modelos KDM de nivel 1 para el paquete micro-KDM a partir de los modelos de nivel 0, ya que dicho nivel 1 ofrece la información necesaria para calcular las métricas identificadas.

Es importante destacar que el metamodelo KDM no contiene elementos para representar sentencias o expresiones específicas de un lenguaje de programación particular, sino que utiliza elementos `ActionElement` para esta finalidad. La Figura 6.5a muestra un extracto del metamodelo KDM que incluye la metaclase `ActionElement` y algunos de los elementos utilizados para representar la semántica de las sentencias. Un elemento `ActionElement` indica en su atributo `kind` el tipo de sentencia que está representando (p. ej., sentencia de asignación o de control). Por otro lado, la referencia `codeElement` representa los elementos del código utilizados en la sentencia (p. ej., los valores inmediatos se representan con elementos de tipo `Value` o las variables con elementos de tipo `StorableUnit`), mientras que la referencia `actionRelation` representa la semántica de la sentencia (p. ej., la llamada a una función se representa con elementos de tipo `Calls`, la escritura en una variable con elementos de tipo `Writes` o su lectura con elementos de tipo `Reads`). La Figura 6.5b muestra la representación en KDM de la sentencia `:WCGA := NULL`, la cual se compone de

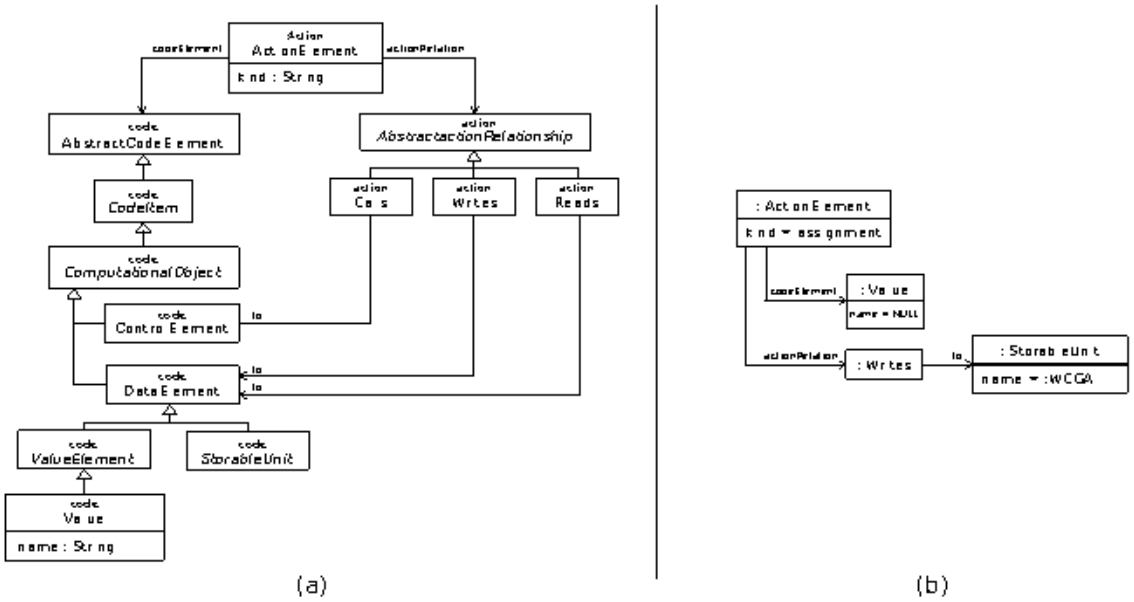


Figura 6.5: Ejemplo de modelo KDM. (a) Extracto del metamodelo KDM. (b) El modelo KDM para la sentencia `:WCGA := NULL`.

un elemento `ActionElement` cuyo valor `kind` es `assignment`, la referencia `codeElement` apunta a una instancia de la metaclase `Value` que representa al valor inmediato `NULL` y la referencia `ActionRelation` apunta a una instancia de la metaclase `Writes`, que indica que la sentencia escribe en la variable `:WCGA` (representada por una instancia de la metaclase `StorableUnit`).

El valor del atributo `kind` de la metaclase `ActionElement` puede establecerse a criterio del desarrollador. Sin embargo, el paquete micro-KDM establece un conjunto de valores estándar para favorecer la interoperabilidad (p. ej., el valor utilizado en el atributo `kind` de la Figura 6.5b es el valor establecido por el paquete micro-KDM para representar una asignación). Así, los modelos KDM de nivel 1 para el paquete micro-KDM se obtienen incluyendo el valor correcto en el atributo `kind` de los elementos `ActionElement` de los modelos KDM de nivel 0.

En nuestro generador de modelos KDM, utilizamos el lenguaje RubyTL para obtener los modelos de nivel 1 para el paquete micro-KDM a partir de modelos ASTM. Para hacer la transformación más modular y favorecer la separación de conceptos, utilizamos el mecanismo de fases ofrecido por este lenguaje [87] para independizar la construcción de los modelos de código e inventario en dos fases. La primera fase genera el modelo KDM de nivel 0 y, a continuación, asigna los valores de Micro-KDM al atributo `kind` de los elementos `ActionElement`. Así, la principal tarea de esta fase es la creación correcta de cada uno de los elementos `ActionElement` para las sentencias y expresiones del código. La segunda fase genera el modelo de inventario, que está compuesto por los *triggers* del código. De esta forma, cada elemento del modelo de inventario que representa a un *trigger*

también hace referencia a su código asociado.

6.4. Uso de modelos KDM para calcular métricas

Una vez se han obtenido los modelos KDM, se dispone de una representación apropiada del sistema existente para obtener los artefatos necesarios en el proceso de modernización. En nuestro caso, utilizaremos los modelos KDM de nivel 1 para el paquete micro-KDM para obtener métricas que midan el nivel de acoplamiento entre el código y la interfaz gráfica en el código PL/SQL de los *triggers* de aplicaciones Oracle Forms.

Hemos definido diferentes métricas para calcular el nivel de acoplamiento, el cual es directamente proporcional al esfuerzo para migrar los *triggers* (a mayor nivel de acoplamiento, mayor es el esfuerzo). Estas métricas están basadas en el conteo de sentencias que hacen uso de la interfaz, su localización así como su tipo (lectura o escritura). El nivel de acoplamiento de un elemento PL/SQL que accede a la interfaz gráfica se clasifica en tres categorías: reflexivo (p. ej., el uso de funciones `NAME_IN` o `COPY`) e imperativo, cual puede ser de lectura (p. ej., uso de una variable en la parte `where` de una consulta `select`) o de escritura (p. ej., una sentencia de asignación). El acoplamiento reflexivo es el más difícil de migrar ya que implica estudiar el contexto del código en tiempo de ejecución.

De esta forma, deben localizarse y contarse el número de sentencias reflexivas así como el número de lecturas y escrituras en las sentencias imperativas en cada *trigger* para determinar el nivel de acoplamiento. Los modelos KDM extraídos permiten realizar estos cálculos fácilmente ya que el metamodelo KDM incorpora elementos para representar las operaciones de lectura y escritura (elementos `Read` y `Write` del metamodelo). Por otro lado, debe analizarse el atributo `kind` de los elementos `ActionElement` para determinar si una sentencia es reflexiva o imperativa. Las sentencias reflexivas serán aquellas que representen llamadas a determinados métodos desde el código fuente (el atributo `kind` deberá tener valor `Call`). El cálculo de estas métricas se realizó por medio de una transformacion *m2m* en RubyTL que convierte el modelo KDM extraído en modelos conformes al metamodelo de métricas construido, tal y como se muestra en las Figuras 6.6a y 6.6c, las cuales muestran el metamodelo de métricas y el modelo conforme a dicho metamodelo que representa las métricas para el modelo KDM de la Figura 6.6b, que coincide con el mostrado en la Figura 6.5b pero se incluye por claridad. Finalmente, para visualizar los datos, se aplicó una transformación *m2t* en Textplate para generar un fichero con los datos separados por comas (fichero CSV).

Estas métricas fueron aplicadas a los *triggers* PL/SQL de una aplicación Oracle Forms utilizada en la Universidad de Murcia para la gestión de los estudiantes. La Figura 6.7 muestra el nivel de acoplamiento para los *triggers* de tres Forms de dicha aplicación. La información visualizada ayuda a entender el esfuerzo necesario que debería aplicarse para cada uno de ellos. En este caso, el Form *Europrojects* sería más difícil de migrar que los otros dos si se considera el acoplamiento de interfaz, aunque también es importante destacar que deberían tenerse en cuenta otros aspectos como el tamaño o complejidad.

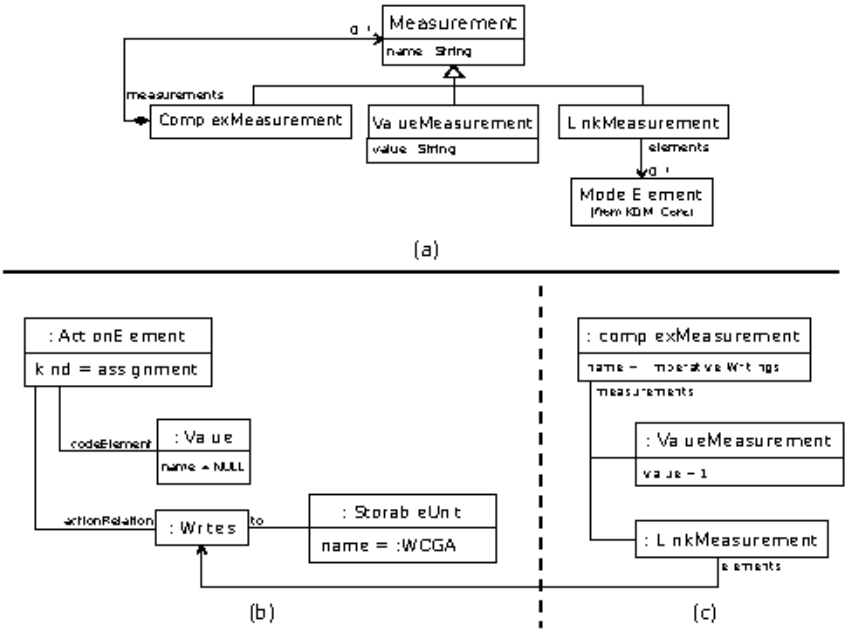


Figura 6.6: Ejemplo de modelos utilizados para calcular métricas. (a) El metamodelo de métricas. (b) El modelo KDM para la sentencia `:WCGA := NULL`. (b) El modelo de métricas para el modelo KDM mostrado.

6.5. Conclusiones

El desarrollo de la herramienta basada en ADM presentada en este capítulo nos permitió estudiar y valorar los metamodelos de esta iniciativa para un caso práctico real de migración de aplicaciones. Este trabajo nos permitió extraer una serie de conclusiones que presentamos a continuación.

En primer lugar, es necesario tener un buen conocimiento de la gramática del lenguaje de programación así como de los metamodelos ASTM y KDM para poder definir las transformaciones *t2m* y *m2m* utilizadas para la extracción de modelos KDM. Aunque el metamodelo KDM es de tamaño muy significativo, el desarrollador solo trabaja con los conceptos de los paquetes con los que esté trabajando. Por otro lado, la falta de ejemplos de modelos KDM dificulta el proceso de aprendizaje. En cuanto a ASTM, es un metamodelo más simple que KDM pero es necesario definir un metamodelo SASTM para el lenguaje de programación con el que se está tratando. Este paso requiere conocimientos profundos del lenguaje de programación así como del metamodelo GASTM para identificar correctamente los puntos que deben ser extendidos.

Es importante remarcar que el desarrollador tiene la responsabilidad de elegir el nivel de compatibilidad de KDM con el que se va a trabajar, el cual está determinado por los aspectos del sistema (vistas arquitecturales) que sea necesario considerar durante el proceso de modernización. Por ejemplo, el extractor de modelos KDM creado en nuestra herramienta solamente genera modelos KDM de nivel 1 para el paquete Micro-KDM porque

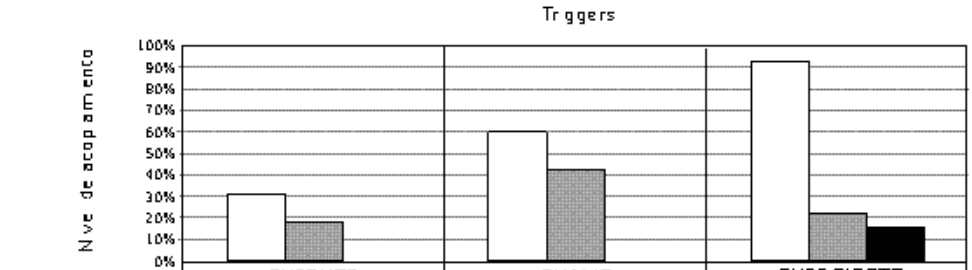


Figura 6.7: Nivel de acoplamiento de tres Forms de Oracle de un sistema de gestión de alumnos. Cada barra indica la proporción de *triggers* que contienen un tipo particular de acoplamiento. Un *trigger* puede contener más de un tipo de acoplamiento.

es el nivel suficiente para calcular las métricas identificadas. Sin embargo, para otras tareas de modernización será necesario utilizar modelos KDM de nivel 1 para otros paquetes del metamodelo (p. ej., uso del paquete de interfaz gráfica para procesos de modernización que necesiten representar esta parte del sistema software).

El esfuerzo para implementar un proceso de extracción de modelos a partir de código PL/SQL se vio drásticamente reducido al utilizar Gra2MoL, el cual permitió definir la transformación *t2m* de forma declarativa y su lenguaje de consultas facilitó la obtención de información dispersa en el código. Además, también permitió obtener modelos ASTM de nivel 1, lo que facilitó la definición de la transformación de ASTM a KDM, gracias a la información semántica extra introducida a este nivel de compatibilidad de ASTM.

La organización de ASTM en GASTM y SASTM permite definir transformaciones *m2m* reutilizables, ya que las reglas que tratan con los elementos del GASTM podrían ser reutilizables para cualquier lenguaje de programación. De esta forma, el uso de RubyTL y su mecanismo de fases podría facilitar la definición de transformaciones componibles, utilizando este mecanismo para ejecutar la transformación de ASTM a KDM en dos fases y para componer la transformaciones GASTM a KDM y SASTM a KDM, tal y como se comentará en la sección 9.2.

Desde una visión más general, las soluciones basadas en ADM se componen de elementos que intercambian modelos KDM para facilitar el reuso. Nosotros hemos creado un productor de modelos (el extractor de modelos KDM a partir del código) así como un consumidor de estos modelos (el generador de métricas), trabajando ambos con modelos KDM de nivel 1 para el paquete micro-KDM. De esta forma, estos componentes podrían ser incorporados en otras soluciones basadas en ADM que necesitaran trabajar con modelos KDM de PL/SQL.

Desde nuestra experiencia, los metamodelos propuestos por la iniciativa ADM favorecen el intercambio de metadatos gracias a la existencia de una serie de metamodelos estándar.

Sin embargo, todavía existen algunos aspectos que impiden un nivel completo de interoperabilidad. Por un lado, sería deseable que los valores posibles para el atributo `kind` de los elementos `ActionElement` siempre sean conformes al paquete micro-KDM. Actualmente, un modelo KDM que no sea nivel 1 para este paquete puede establecer los valores para el atributo `kind` a criterio del desarrollador, lo que impediría compartir dichos modelos entre herramientas que no utilizan el mismo conjunto de valores. Además, los paquetes de la capa de *recursos runtime* y de *abstracciones* son, en la mayoría de los casos, excesivamente simples, lo que obliga al desarrollador a extenderlos para poder representar el sistema software y de nuevo limita la interoperabilidad entre herramientas que no conozcan dichas extensiones. Por otro lado, la falta de metamodelos SASTM para la mayoría de lenguajes de programación obliga a los desarrolladores a definir metamodelos propios, lo que reduce considerablemente la interoperabilidad.

7

Extracción de modelos a partir de datos relacionales

En la antigüedad el conocimiento se había perdido porque ya no existían copias.
En la actualidad se perdía porque existían copias de todo.
Impávido, Jack Campbell

Los datos, junto con el código fuente, son uno de los principales elementos que componen un sistema software. Por este motivo, la extracción de modelos a partir del esquema de una base de datos así como de los propios datos es una tarea necesaria en la mayoría de los escenarios de modernización. Mientras que Gra2MoL puede utilizarse para la extracción de modelos a partir del esquema de la base de datos, tal y como se ha comentado en la sección 5.9, el lenguaje ScheMoL está especialmente diseñado para extraer modelos a partir de datos almacenados en una base de datos relacional. En este capítulo presentamos el lenguaje ScheMoL, el cual está profundamente inspirado en Gra2MoL, del que reutiliza algunos componentes y decisiones de diseño. De hecho, ScheMoL puede considerarse una adaptación de Gra2MoL al problema de la extracción de modelos desde bases de datos relacionales. La estructura del capítulo es similar a la del capítulo anterior. En primer lugar se describirá el problema y las principales alternativas actuales para abordar este tipo de extracción de modelos. A continuación, se describirá el DSL y se mostrará un ejemplo de uso. Finalmente, se describirán un conjunto de escenarios de aplicación, las características del lenguaje y las conclusiones. Al igual que se hizo para Gra2MoL, en las conclusiones se describirá el grado de cumplimiento de los requisitos identificados en [47].

7.1. Descripción del problema

En los procesos de extracción de modelos a partir de datos almacenados en una base de datos relacional (de aquí en adelante, extracción de modelos de datos), el objetivo es

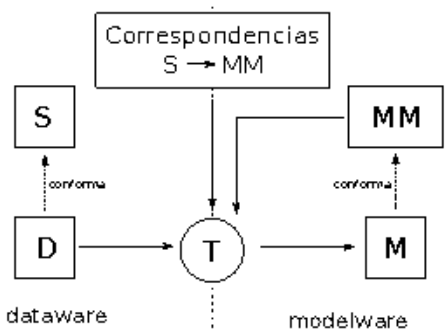


Figura 7.1: Proceso de extracción de modelos a partir de datos almacenados en una base de datos relacional.

obtener modelos que conforman a un determinado metamodelo a partir de los datos que conforman al esquema de la base de datos relacional, tal y como se muestra en la Figura 7.1. El proceso es similar al mostrado en la Figura 5.1 pero cambiando los artefactos de entrada. En este caso, el proceso es una transformación T datos-a-modelo (*d2m*) que tiene como entradas los datos D que conforman al esquema S, y el metamodelo MM al cual debe conformar el modelo resultante M. Al igual que en el caso de Gra2MoL, el proceso utiliza una definición de correspondencias entre los elementos del esquema de la base de datos y los del metamodelo, cuya naturaleza depende de la aproximación que se aplique para ejecutar el proceso. Además, nótese que las correspondencias también se definen a nivel M2 dentro de la arquitectura de cuatro capas del OMG comentada en la sección 4.6.

Tal y como se ha descrito en la sección 4.6, la extracción de modelos de datos puede ser vista como un puente unidireccional desde el *dataware* al *modelware* que debe soportar las siguientes tareas: (1) acceder a la base de datos para obtener los datos necesarios y (2) utilizar los datos extraídos para crear e inicializar las instancias de los elementos del metamodelo destino. Para abordar el primer paso es necesario utilizar un mecanismo que permita obtener los datos eficientemente. De forma parecida al problema de la información dispersa en el código fuente en Gra2MoL, cuando existen metaclases que representan información agregada de la base de datos, también es necesario disponer de un lenguaje de consultas eficaz que permita obtener dicha información. Por otro lado, dado que puede existir un salto semántico muy significativo entre el esquema de la base de datos y el metamodelo al ser definidos independientemente, el segundo paso requiere mecanismos suficientemente expresivos para definir las correspondencias existentes entre los elementos de cada uno de ellos. Estas correspondencias pueden clasificarse en cinco escenarios principales, los cuales están íntimamente relacionados con los escenarios identificados en las correspondencias objeto relacional:

- *Correspondencias tupla-modelo.* Este caso se da cuando existe una correspondencia directa entre una tupla y una instancia de metaclase, donde las columnas de la tupla son utilizadas para inicializar las propiedades del elemento del modelo. Por ejemplo, en la Figura 7.2a, cada una de las tuplas de la tabla `TablaA` corresponde a una

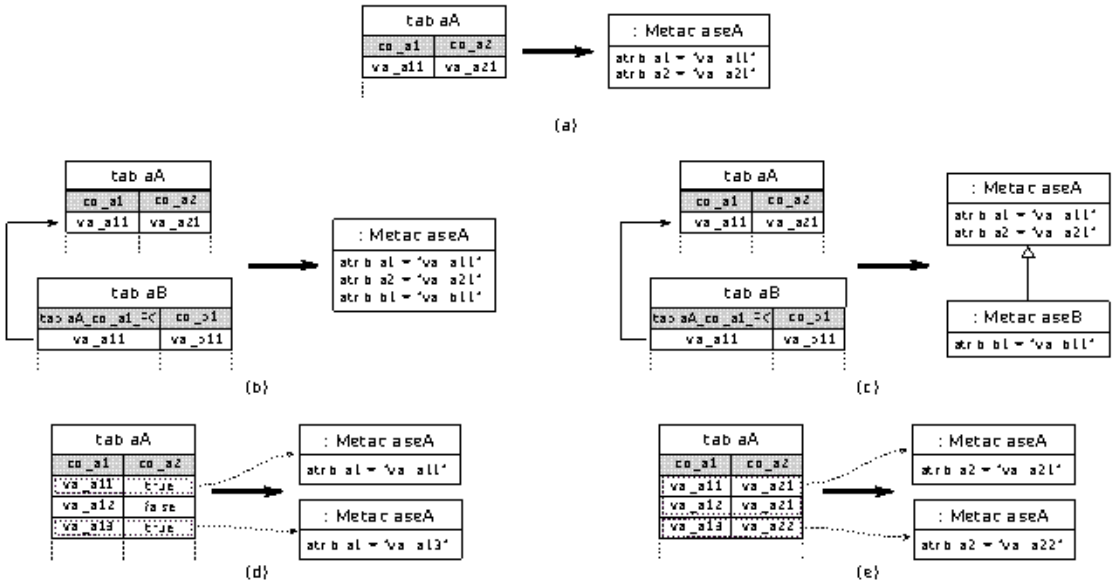


Figura 7.2: Escenarios básicos en una extracción de modelos de datos.

instancia de la metaclase `MetaclaseA` y los valores de sus columnas son utilizados para inicializar los atributos de la instancia de la metaclase. De esta forma, los valores de los atributos `atrib_a1` y `atrib_a2` de la instancia de la metaclase `MetaclaseA` se obtienen de los valores de las columnas `col_a1` y `col_a2` de la tupla de la tabla `tablaA`, respectivamente.

- *Correspondencias multitupla-modelo.* La normalización de la base de datos puede provocar que los datos de una entidad se dispersen entre diferentes tablas. Sin embargo, este proceso puede tener que ser invertido durante la extracción de elementos del modelo ya que una metaclase puede representar información que es almacenada en diferentes tablas. Este es el caso de las instancias de la metaclase `MetaclaseA` en la Figura 7.2b, la cual representa información que es almacenada en diferentes tablas (`tablaA` y `tablaB` en la Figura, donde la columna `tablaA_col_a1_FK` de la tabla `tablaB` es una clave ajena a la columna `col_a1` de la tabla `tablaA`). De esta forma, los valores de los atributos `atrib_a1` y `atrib_a2` se extraen de las columnas `col_a1` y `col_a2` de las tuplas de la tabla `tablaA` mientras que el atributo `atrib_b1` se extrae de la columna `col_b1` de la tupla de la tabla `tablaB` cuya clave ajena referencia a la tupla de la tabla `tablaA` considerada. Nótese que los valores de los atributos se obtienen a partir de los valores de las columnas de tuplas localizadas en diferentes tablas.
- *Correspondencias de especialización.* Este caso es similar al anterior, ya que la estructura de las tablas en la base de datos es la misma y sólo cambia la estructura de las metaclases del metamodelo, que representan a una jerarquía. Por ejemplo, en la Figura 7.2c, la metaclase `MetaclaseB`, que es subclase de `MetaclaseA`, representa

información que es almacenada en las tablas `tablaB` y `tablaA`. De esta forma, los valores de los atributos de las instancias de la metaclase `MetaclaseB` se obtienen de los valores de tuplas localizadas en diferentes tablas de la misma forma que en el caso anterior.

- *Correspondencias de columna.* Este escenario se da cuando una metaclase del metamodelo se deriva a partir de la información de una columna de una tabla de la base de datos. Por ejemplo, en la Figura 7.2d, la metaclase `MetaclaseA` representa a las tuplas de la tabla `TablaA` cuyo valor en la columna `col_a2` sea `true`. Así, en el ejemplo existen dos instancias de `MetaclaseA`, una para cada valor diferente de la columna `col_a1` con el valor de la columna `col_a2` a `true`.
- *Correspondencias de colección.* En este caso, una metaclase representa a una colección agrupada de elementos de una tabla. Por ejemplo, en la Figura 7.2c, la metaclase `MetaclaseA` se obtiene a partir de la agrupación en base a la columna `col_a2` de las tuplas de la tabla `tablaA`. Así, en el ejemplo existen dos instancias de `MetaclaseA`, una para cada grupo de valores de la columna `col_a2`.

7.2. Aproximaciones existentes

Actualmente, las transformaciones *d2m* requeridas en un proceso de extracción de modelos de datos se pueden implementar mediante el uso de API y *frameworks* de acceso a bases de datos. Existen tres aproximaciones principales para la implementación de este tipo de transformaciones: (1) implementación de extractores específicos que utilicen un API de acceso a la base de datos (p. ej., implementación del extractor en Java y uso del API JDBC [39]); (2) uso de *mappers* objeto-relacionales (p. ej.,Hibernate [37]); y (3) uso de *mappers* modelo-relacionales combinados con transformaciones *m2m* (p. ej., Teneo [80] combinado con ATL [40]). En todas ellas las correspondencias entre los elementos se definen programáticamente mientras que para realizar las consultas a la base de datos recurren al uso de lenguajes como SQL o JPQL [26].

Estas aproximaciones pueden ser comparadas teniendo en cuenta dos dimensiones principales: naturaleza declarativa y expresividad. En primer lugar, las aproximaciones pueden ser comparadas en función de la potencia declarativa que ofrezcan para realizar las tareas principales en las que se descompone un proceso de extracción de modelos de datos, que son: (i) acceso a la base de datos; (ii) creación y modificación de modelos; (iii) control del flujo, es decir, en qué orden deben ser creados los elementos del modelo; (iv) resolución de referencias, esto es, cómo transformar las referencias entre los elementos de una base de datos (p. ej., claves primarias y ajenas) en referencias entre los elementos de un metamodelo; y (v) consulta de datos. Por otro lado, el nivel de expresividad de una aproximación depende de los mecanismos que ofrezca para abordar los cinco escenarios básicos en una extracción de modelos de los datos que se han presentado anteriormente.

Las secciones siguientes describirán cada una de las aproximaciones y sus niveles de expresividad y naturaleza declarativa. A continuación, la sección 7.3 mostrará nuestra aproximación y la tabla 7.1 un resumen de todas las aproximaciones tratadas.

7.2.1. Uso de API de acceso a la base de datos

La implementación de extractores específicos es una solución donde el desarrollador debe implementar toda la lógica del proceso de extracción por medio de un GPL, pudiéndose ayudar de un API que facilite el acceso a la base de datos así como de un *framework* para la creación del modelo destino (p. ej., JDBC o ODBC para acceso a la base de datos y el *framework* EMF para la gestión y creación de modelos).

En este tipo de extractores, el uso de un GPL ofrece suficiente libertad para implementar el proceso de extracción y tratar con todos los escenarios involucrados en el proceso, aunque realmente no existen mecanismos específicos. Por otro lado, la naturaleza declarativa de esta solución es muy reducida ya que el desarrollador debe hacer explícitos todos los pasos requeridos en el proceso de extracción: *joining* de tablas, iteraciones en los resultados, correspondencias entre elementos de la base de datos y metaclases, etc. Esta tarea, además de ser costosa su implementación, afecta negativamente al mantenimiento del código.

7.2.2. Uso de *mappers* objeto-relacionales

El esfuerzo requerido para implementar el acceso a la base de datos y obtener el conjunto de datos a utilizar en la extracción puede ser reducido utilizando *mappers* objeto-relacionales como Hibernate, los cuales soportan la persistencia en bases de datos relacionales de los objetos creados por una aplicación. La persistencia de estos objetos es realizada automáticamente de modo que cualquier cambio en los datos de la base de datos se ve reflejado en los objetos y viceversa, dado que los objetos están sincronizados con la base de datos. Esto permite al desarrollador interactuar directamente con los objetos en vez de acceder y consultar la base de datos.

En un proceso de extracción de modelos que utiliza *mappers* objeto-relacionales existen dos fases: (1) extracción de objetos de la base de datos y (2) transformación de dichos objetos en elementos del modelo conformes al metamodelo destino. La primera fase es gestionada automáticamente por medio del *mapper* objeto-relacional y requiere definir las correspondencias entre los elementos de la base de datos y los objetos. Por ejemplo, en el caso de Hibernate se utiliza el estándar JPA (*Java Persistence API*) [26], que es el API de persistencia creada en la plataforma Java EE, para definir estas correspondencias. En JPA estas correspondencias son de tipo tupla-modelo de forma predeterminada aunque pueden enriquecerse para definir correspondencias multitupla-modelo y de especialización, ofreciendo un mayor nivel de expresividad y facilitando la siguiente fase del proceso. Sin embargo, las correspondencias de columna y de colección no son soportadas, por lo que deben tratarse en la siguiente fase.

Durante la segunda fase, se crean los elementos del modelo a partir de los objetos persistentes. Además, también deben establecerse las correspondencias no tratadas en la primera fase. Por lo tanto, aunque este tipo de aproximaciones se aprovechan de la expresividad para el acceso a la base de datos ofrecida por los *mappers* objeto-relacionales, el desarrollador todavía tiene que definir una parte del proceso de extracción imperativamente por lo que el esfuerzo para implementar esta solución también es considerable.

7.2.3. Uso de *mappers* modelo-relacionales

Finalmente, los *mappers* modelo-relacionales como Teneo facilitan la persistencia de modelos EMF en bases de datos relacionales. De forma parecida a los *mappers* objeto-relacionales, el desarrollador debe definir las correspondencias entre los elementos de la base de datos y los del metamodelo, liberando al desarrollador de tener que tratar con objetos intermedios, tal y como ocurría en la aproximación anterior. En Teneo, estas correspondencias se incluyen por medio de anotaciones JPA en el metamodelo.

Aunque esta aproximación libera al desarrollador de implementar la transformación de los objetos persistentes a los elementos del modelo requerida en la segunda fase del caso anterior, adolece de las mismas limitaciones expresivas, ya que internamente también utiliza JPA. Así, las correspondencias de tipo columna y colección tampoco son soportadas.

Mientras que en el caso anterior era necesario implementar aquellas correspondencias que no trataba el *mapper* a nivel de objetos, en este caso dicha implementación debe realizarse a nivel de modelos. De esta forma, el modelo que es obtenido a partir del *mapper* debe ser transformado por medio de transformaciones *m2m* para alcanzar el nivel de abstracción del metamodelo destino. Sin embargo, incluso al utilizar transformaciones *m2m*, las correspondencias de columna y de colección deben resolverse mediante el uso de mecanismos imperativos como las *called rules* de ATL o los *helpers* de RubyTL, ya que estos lenguajes no ofrecen construcciones suficientemente declarativas para resolver el problema, aumentando considerablemente el esfuerzo de implementación.

7.3. Nuestra aproximación

Dadas las limitaciones identificadas en las aproximaciones anteriores, decidimos crear un nuevo DSL denominado ScheMoL para facilitar la extracción de modelos a partir de datos de una bases de datos relacional. Para su diseño se tuvieron en cuenta los requisitos de un puente entre *dataware-modelware* expuestos en la sección 7.1. También se tuvieron en cuenta los niveles de expresividad y naturaleza declarativa comentados anteriormente con el objetivo de ofrecer una solución que cubriera las necesidades en estos aspectos.

ScheMoL está profundamente inspirado en el lenguaje Gra2MoL y comparte con éste la forma de definir las correspondencias entre los elementos de cada espacio tecnológico. Al igual que Gra2MoL, ScheMoL utiliza reglas para definir las correspondencias entre los elementos del *dataware* y del *modelware*, es decir, una regla ScheMoL define la correspondencia entre una tabla y una metaclase. Estas reglas especifican un conjunto de *bindings* que establecen las correspondencias entre las propiedades de la metaclase y la información de la base de datos. Por otro lado, mientras que en una transformación *m2m* se utilizan lenguajes tipo OCL para recorrer el modelo origen, o en una transformación Gra2MoL se utiliza un lenguaje especialmente adaptado para recorrer el código fuente, en una transformación ScheMoL es necesario disponer de un lenguaje de consultas para recorrer la base de datos y especificar los *bindings*. En el *dataware* existe el lenguaje SQL para consultar bases de datos, cuyo principal mecanismo para relacionar tablas es el *join*, donde determinadas columnas son utilizadas para relacionar tuplas de tablas diferentes. Sin embargo,

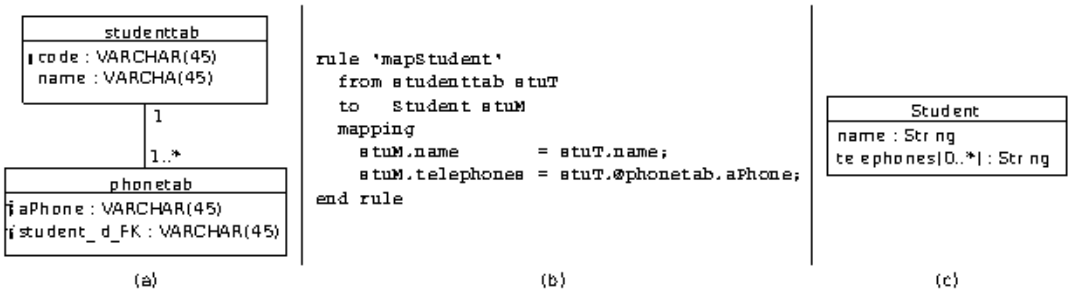


Figura 7.3: Ejemplo simple de una regla ScheMoL.

nuestro objetivo es ofrecer un lenguaje de consultas que permita abstraerse de la necesidad de indicar cuales son las columnas que relacionan las tablas, facilitando la obtención de colecciones de datos. Por este motivo, ScheMoL también incorpora un lenguaje consultas especialmente diseñado para acceder a las bases de datos de forma transparente, el cual puede ser utilizado en las reglas para obtener información de otras tuplas.

La Figura 7.3b muestra un ejemplo simple de regla ScheMoL para extraer elementos del modelo que son instancias de la metaclase `Student` (ver Figura 7.3c) a partir de la tabla `studenttab` (ver Figura 7.3a). Las secciones *from* y *to* de la regla indican la tabla y la metaclase implicadas en la correspondencia que define la regla, respectivamente. Por otro lado, en la sección *mappings* se especifica cómo deben inicializarse las propiedades del elemento del modelo a partir de la información de la base de datos. En este ejemplo, el atributo `name` es inicializado con el valor de la columna `name` de la tupla con la que está tratando la regla (variable `stuT`), la cual es una tupla contenida en la tabla `studenttab`. A continuación, se inicializa el atributo multivaluado `telephones` con todos los valores que corresponden a los teléfonos del elemento utilizando un lenguaje de consultas especialmente adaptado, el cual se presentará en la siguiente sección.

Las reglas de ScheMoL se ejecutan utilizando el concepto de *binding* de forma parecida a como se realiza en Gra2MoL, evitando tener que definir explícitamente el proceso de extracción, tal y como era necesario en la mayoría de las soluciones descritas anteriormente. De esta forma, el lenguaje permite definir declarativamente las tareas de creación del modelo, control del flujo y resolución de referencias. Por otro lado, la potencia declarativa requerida para realizar la tarea de consultar la base de datos es ofrecida por el lenguaje de consultas incorporado en ScheMoL.

Para llevar a cabo la inicialización de las propiedades de la metaclase se utiliza el lenguaje de consultas incorporado, el cual dota a ScheMoL de un nivel de expresividad suficiente para definir las correspondencias identificadas en los cinco escenarios básicos. Por un lado, es posible definir correspondencias tupla-modelo, multitupla-modelo y de especialización, ya que el lenguaje de consultas permite acceder a las columnas de la tuplas de diferentes tablas de la base de datos. Por otro lado, las correspondencias de columna y de colección se definen por medio del uso de expresiones de filtro y agrupación en las consultas. La siguiente sección se dedicará a explicar detalladamente este lenguaje de consultas y la

Naturaleza declarativa				
Tarea	**JDBC**	**Hibernate**	**Teneo + m2m**	**ScheMoL**
Acceso a la base de datos	No	Si	Si	Si
Creación del modelo	No (EMF)	No (EMF)	Si	Si
Control de flujo	No (Java)	No (Java)	Si	Si
Resolución de referencias	No (Java)	No (Java)	Si	Si
Lenguaje de consultas	SQL	JPQL	JPQL + OCL	Específico/SQL
Tecnologías requeridas	JDBC, EMF, Java, SQL	JPA, Java, JPQL	JPA, JPQL, lenguaje *m2m*	ScheMoL, SQL
Expresividad				
Correspondencia	**JDBC**	**Hibernate**	**Teneo + m2m**	**ScheMoL**
Tupla-modelo	No	Si	Si	Si
Multitupla-modelo	No	Si	Si	Si
Especialización	No	Si	Si	Si
Columna	No	No	Limitada	Si
Colleción	No	No	Limitada	Si
Finalidad principal	Acceso a base de datos	Persistencia de objectos	Persistencia de modelos	Extracción de modelos

Tabla 7.1: Comparación de la naturaleza declarativa y expresividad de ScheMoL con el resto de soluciones.

sección 7.7 ilustrará el lenguaje con un ejemplo.

La tabla 7.1 contrasta el nivel de expresividad y naturaleza declarativa de ScheMoL con las soluciones anteriores, respectivamente. En cuanto a la naturaleza declarativa, puede observarse que solamente las aproximaciones basadas en *mappers* modelo-relacionales como Teneo son capaces de ofrecer la potencia declarativa suficiente para especificar el proceso de extracción completo. Sin embargo, su aplicación requiere utilizar un buen número de tecnologías, lo que dificulta su desarrollo. Por otro lado, en cuanto al nivel de expresividad, solamente ScheMoL ofrece mecanismos para cubrir todos los escenarios identificados. En el resto de soluciones, es necesario hacer uso de soluciones específicas (como la programación usando un GPL en el caso de JDBC o el uso de transformaciones *m2m* en el caso de Teneo) para resolver las correspondencias.

7.4. Un lenguaje de acceso a datos relacionales en transformaciones *d2m*

Tal y como se ha comentado anteriormente, ScheMoL incorpora un lenguaje para acceder a los datos relacionales desde las reglas de transformación. Este lenguaje permite definir consultas, las cuales son traducidas a SQL por el motor de consultas de ScheMoL, facilitando el acceso transparente a cualquier base de datos. Una consulta consiste en una secuencia de operaciones que son aplicadas a una tupla de una tabla de la base de datos. Cada operación incluye un operador de modo de navegación, el nombre de una columna o de una tabla y, opcionalmente, una expresión de filtro o de agrupamiento. La expresión EBNF para una consulta es la siguiente:

```
tupla {.[@](NombreColumna | NombreTabla) [filtro] [grupo]}
```

Una consulta facilita la navegación entre las tablas relacionadas por sus claves ajenas por medio de la notación punto, de forma parecida a como se realiza en los lenguajes OCL. Cada operación de consulta especifica el modo de navegación por medio del operador, que puede ser directo o inverso, y el nombre de la columna o de la tabla a utilizar, dependiendo del modo especificado. El operador `.` permite aplicar una navegación directa mientras que el operador `.@` lo hace a la inversa. Así, la consulta `var.nombreColumna1.nombreColumna2` es una navegación directa que obtiene el valor de la columna llamada `nombreColumna2` de la tabla referenciada por la columna `nombreColumna1`, que es una clave ajena de la tabla que contiene la tupla `var`. Por ejemplo, la consulta `tuplaB.tablaA_col_a1_FK.col_a2` obtendría el valor de la columna `col_a2` de la tupla referenciada por la clave ajena `tablaA_col_a1_FK` de la tupla `tuplaB` de la tabla `tablaB` en la Figura 7.2b.

Por otro lado, la consulta `var.@nombreTabla` es una navegación inversa que obtiene aquellas tuplas de la tabla `nombreTabla` que hacen referencia a la tupla `var`, es decir, tienen una clave ajena a la tabla que contiene la tupla `var`. Por ejemplo, la consulta `tuplaA.@tablaB` obtendría todas aquellas tuplas de la tabla `tablaB` que hacen referencia a la tupla `tuplaA` de la tabla `tablaA` en la Figura 7.2b.

La navegación inversa también puede utilizarse para obtener todas las tuplas de una determinada tabla de la base de datos. En este caso, el operador inverso se aplica sobre una tupla especial de tipo `Database`, que representa a todas las tablas de la base de datos. Por ejemplo, la consulta `db.@tablaA` obtiene todas las tupla de la tabla `tablaA` de la base de datos (representada por `db`, que es de tipo `Database`).

Las operaciones de consulta directas e inversas pueden combinarse en una misma expresión. Por ejemplo, la consulta `tuplaA.@tablaB.col_b1` obtiene el valor de la columna `col_b1` de todas las tuplas de la tabla `tablaB` que referencian a la tupla `tuplaA` de la tabla `tablaA` en la Figura 7.2b.

Las consultas en ScheMoL también pueden incorporar opcionalmente expresiones de filtro y de agrupamiento. Las expresiones de filtro se indican entre llaves y especifican la condición que deben cumplir las tuplas, de tal forma que solamente aquellas que satisfagan dicha condición serán seleccionadas. Por ejemplo, la consulta `db.@tablaA{col_a2 = true}` seleccionará todas las tuplas de la tabla `tablaA` que tienen el valor `true` en la columna `col_a2` en la Figura 7.2d.

Por otro lado, las expresiones de agrupación se indican entre corchetes y permiten agrupar los resultados de una consulta. Por ejemplo, la consulta `db.@tablaA[col_a2]` agrupa las tuplas de la tabla `tablaA` según el valor de la columna `col_a2` en la Figura 7.2c.

Finalmente, el lenguaje soporta el uso de funciones para realizar cálculos sobre el resultado de una consulta. Por ejemplo, la función `count()` cuenta el número de tuplas devueltas por una consulta o la función `sum(nombreColumna)` suma los valores de la columna `nombreColumna` en todas las tuplas devueltas por una consulta. Así, la consulta `tuplaA.@tablaB.count()` contaría todas las tuplas devueltas por la consulta `tuplaA.@tablaB` en la Figura 7.2b mientras que la consulta `tuplaA.@tablaB.sum(col_b2)` sumaría los valores de la columna `col_b2` de las tuplas devueltas por dicha consulta.

7.5. Definición de transformación en ScheMoL

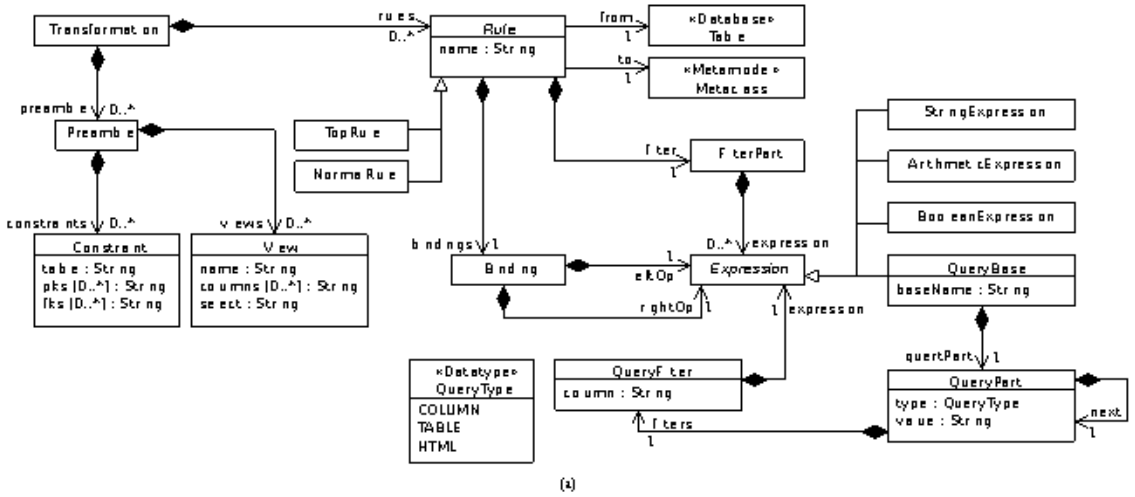
ScheMoL es un lenguaje de transformación basado en reglas muy similares a las utilizadas en Gra2MoL, pero con dos diferencias importantes: (1) el elemento origen de una regla es un elemento del esquema de base de datos, es decir, una tabla en vez de un elemento gramatical; y (2) la navegación por la base de datos se realiza por medio del lenguaje de consultas presentado en la sección anterior.

La sintaxis abstracta de ScheMoL, expresada como metamodelo, se muestra en la Figura 7.4a mientras la estructura de la sintaxis concreta se muestra en la Figura 7.4b, la cual muestra el esqueleto de una regla ScheMoL. Una definición de transformación ScheMoL (metaclase `Transformation`) está compuesta por un conjunto de reglas (metaclase `Rule`) y, opcionalmente, un preámbulo (metaclase `Preamble`). De forma parecida a Gra2MoL, en ScheMoL existen varios tipos de regla: *toprule* y *normal*. Ambas comparten estructura pero las reglas de tipo *toprule* se encargan de iniciar la ejecución de la transformación, como se explicará en la sección 7.5.1. La estructura de una regla en ScheMoL es muy similar a la utilizada en Gra2MoL, pero en ScheMoL el filtro de la regla tiene una sección explítica y las consultas no se definen en su propia sección sino junto a los *bindings*. Así, en ScheMoL una regla está compuesta por cuatro secciones:

- La sección *from*, que especifica la tabla origen junto con una variable que se asociará a la tupla cuando la regla sea ejecutada.
- La sección *to*, que especifica la metaclase destino junto con una variable que se asociará a la instancia de dicha metaclase cuando la regla sea ejecutada.
- La sección *filter* es opcional e incluye una expresión condicional que se aplica a la tupla origen de forma que solamente aquellas tuplas que satisfagan dicha condición ejecutarán la regla.
- La sección *mapping*, que contiene un conjunto de *bindings* para establecer las propiedades del elemento del metamodelo destino.

El preámbulo de una transformación ScheMoL permite especificar las claves ajenas/primarias de las tablas así como vistas de la base de datos. En primer lugar, el lenguaje de consultas se basa en la existencia de claves ajenas para permitir consultar la base de datos de forma transparente. Sin embargo, puede que una base de datos no incluya la definición de estas claves, por ejemplo, éste es el caso de MySQL, donde es común no incluirlas por motivos de eficiencia. Por este motivo, las claves ajenas pueden ser especificadas explícitamente en la sección del preámbulo de una transformación. Además, opcionalmente también pueden especificarse las claves primarias.

Por otro lado, las vistas son el mecanismo utilizado cuando no es posible obtener la información necesaria para inicializar las propiedades del elemento del modelo por medio de las consultas (p. ej., necesidad de utilizar filtros propios de SQL o condiciones de agrupamiento complejas). De esta forma, las vistas permiten adaptar la base de datos de forma transparente para facilitar la definición de la transformación. Una definición de vista tiene el mismo formato que una vista en SQL y se utiliza de forma transparente desde el lenguaje de consultas como una tabla más.




```
rule <nombre-regla>
  from <tabla-origen> <Id-elemento-origen>
  to   <metaclase-destino> <Id-elemento-destino>
  [ filter <expresión|Id-elemento-origen)> ]
  mapping
    { Id-elemento-destino.propiedad-metaclase = [ literal | consulta | expresión ] }
end rule
```

(b)

Figura 7.4: (a) Un extracto de la sintaxis abstracta de ScheMoL. (b) Esqueleto de una regla ScheMoL.

7.5.1. *Bindings* y evaluación de reglas

ScheMoL comparte el concepto de *binding* de ATL, también utilizado en Gra2MoL, pero adaptándolo al problema de la extracción de modelos de datos. Los *bindings* en ScheMoL se utilizan para establecer la relación entre una tupla de la base de datos y una propiedad de la metaclase. También se escriben como una asignación donde la parte izquierda es una propiedad de la metaclase destino y la parte derecha puede ser un valor literal, una consulta o una expresión.

Las definiciones de compatibilidad regla-*binding* y transformación bien formada definidas en la sección 5.5.1 también son aplicables en ScheMoL con un ligero cambio. Una tabla A_t conforma o es compatible a otra tabla B_t si son la misma.

La ejecución de una transformación ScheMoL está guiada por los *bindings*. Las reglas de tipo *toprule* de una transformación son el punto de entrada y sus *bindings* se encargan de comenzar la ejecución de la transformación. Si una transformación no tiene regla de tipo *toprule*, la primera regla de la trasnformación será considerada como tal. La sección *from* de una regla de tipo *toprule* especifica cómo y cuántas veces debe ejecutarse dicha regla. Si especifica una tabla, la regla de tipo *toprule* es ejecutada con cada una de las tuplas de dicha tabla. Por otro lado, si utiliza la palabra clave `Database`, denota a la colección de todas las tablas de la base de datos y la regla será ejecutada solamente una vez, permitiendo generar un elemento del modelo que actúa como raíz del modelo destino.

Una regla en ScheMoL recibe como entrada una tupla de la tabla origen (sección *from*) y devuelve un elemento del modelo que es instancia de la metaclase destino (sección *to*). Al ejecutar una regla, primero se comprueba la expresión condicional indicada en la sección *filter*, si existe. Si el filtro se satisface, se crea una instancia de la metaclase destino y se registra la ejecución de la regla para evitar ciclos. Finalmente, los *bindings* de la sección *mappings* son ejecutados para inicializar los atributos de la instancia de la metaclase, pudiéndose dar tres situaciones dependiendo de la naturaleza del elemento de la parte derecha:

- Si es un valor literal, dicho valor es directamente asignado al atributo especificado en la parte izquierda.
- Si es una expresión, la expresión es evaluada y su resultado asignado al atributo de la parte izquierda. ScheMoL soporta expresiones de tipo booleano, cadena y enteros junto con sus operaciones básicas.
- Si es una consulta, ésta es ejecutada, pudiéndose dar dos situaciones dependiendo del resultado de dicha consulta y del tipo de la propiedad de la parte izquierda. Si ambos son de tipo primitivo, el resultado de la consulta es asignado a la propiedad directamente. Por otra parte, si la propiedad es una referencia y la consulta devuelve un conjunto de tuplas entonces, para cada tupla de dicho conjunto, se busca y ejecuta aquella regla que conforme con el *binding*, es decir, aquella regla cuyos tipos en las secciones *from* y *to* conforman con los tipos de la parte derecha e izquierda del *binding*, respectivamente. De esta forma, los *bindings* de una regla provocan el *disparo* de otras reglas conforme se ejecuta la transformación.

7.6. Implementación

La arquitectura del motor de ejecución de ScheMoL incluye los siguientes componentes principales (mostrados en la Figura 7.5):

- *Intérprete*, el cual se encarga de ejecutar las reglas, resolver los *bindings* de la sección *mappings* de las reglas y controlar el flujo de ejecución.
- *Generador SQL*, encargado de transformar el lenguaje de consultas de ScheMoL en consultas SQL compatibles con el sistema gestor de base de datos utilizado.
- *Gestor de modelos*, el cual gestiona la creación y modificación de elementos en el modelo destino.

El proceso de implementación de ScheMoL se vio facilitado por el hecho de haber desarrollado anteriormente el lenguaje Gra2MoL. En primer lugar, determinados componentes fueron reutilizados o adaptados, como por ejemplo el gestor de modelos o el motor de resolución de *bindings*. En segundo lugar, Gra2MoL fue utilizado para implementar ScheMoL como DSL textual, definiendo una transformación Gra2MoL para obtener modelos conformes al metamodelo de la sintaxis abstracta de ScheMoL a partir de definiciones textuales conformes a la gramática del lenguaje.

La Figura 7.5 ilustra el proceso de ejecución de una transformación ScheMoL. El *intérprete* obtiene el modelo de sintaxis abstracta a partir de la definición textual por medio

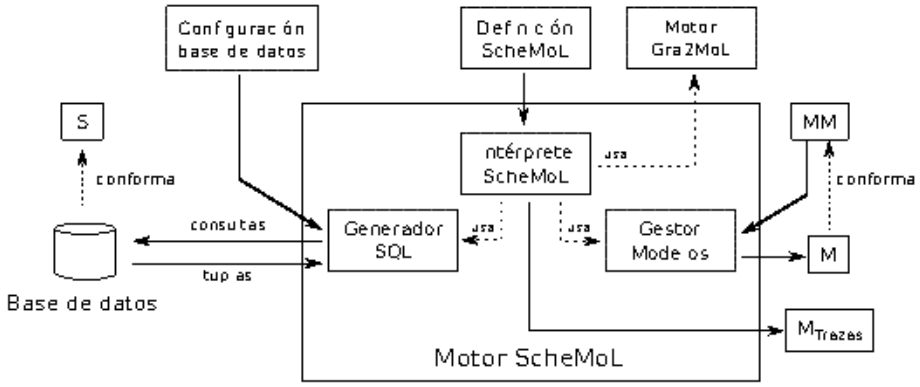


Figura 7.5: Arquitectura del motor ScheMoL.

de una transformación Gra2MoL. Una vez se dispone del modelo de sintaxis abstracta de la transformación, el intérprete ejecuta las reglas, las cuales dan lugar a la ejecución de consultas sobre la base de datos y a la creación de los elementos del modelo. El resultado del proceso de transformación es el modelo conforme al metamodelo destino y un modelo de trazas con información que relaciona los elementos creados con los elementos origen y las reglas aplicadas.

7.7. Ejemplo

El lenguaje ScheMoL se ilustrará con un ejemplo de extracción de modelos a partir de una base de datos de una universidad. La Figura 7.6 muestra el esquema de base de datos y el metamodelo destino de este ejemplo. En el caso de la base de datos, su diseño está especialmente enfocado a modelar las asignaturas y personas, que pueden ser profesores o alumnos. Debido a la normalización, los datos de las personas están dispersos en las tablas: `studenttab`, que almacena la información propia de los alumnos; `lecturertab`, que almacena la información propia de los profesores; y `personTab`, que almacena la información común a ambos. La base de datos también almacena información complementaria de los alumnos en las tablas `phoneTab`, que reúne el conjunto de teléfonos asociados a un alumno y `registrationTab`, que almacena la relaciones M:N entre los estudiantes y las asignaturas, representadas por la tabla `subjecttab`. Además, las asignaturas también están relacionadas con los profesores que las imparten.

Por otro lado, el metamodelo está enfocado a modelar el personal de la universidad y las titulaciones. La universidad está representada por la metaclase `University`, de modo que una instancia de `University` contiene tanto al personal de la universidad (referencia `itsPeople`), que es representada por una jerarquía cuya raíz es la metaclase `Person`; como las titulaciones (referencia `itsDegrees`), que son representadas por la metaclase `Degree`. La jerarquía de `Person` contiene a las subclases `Student`, que representa a un estudiante, y `Lecturer`, que representa a un profesor y además hace referencia a los proyectos en los que participa (representados por la metaclase `Project`). Como puede observarse, la

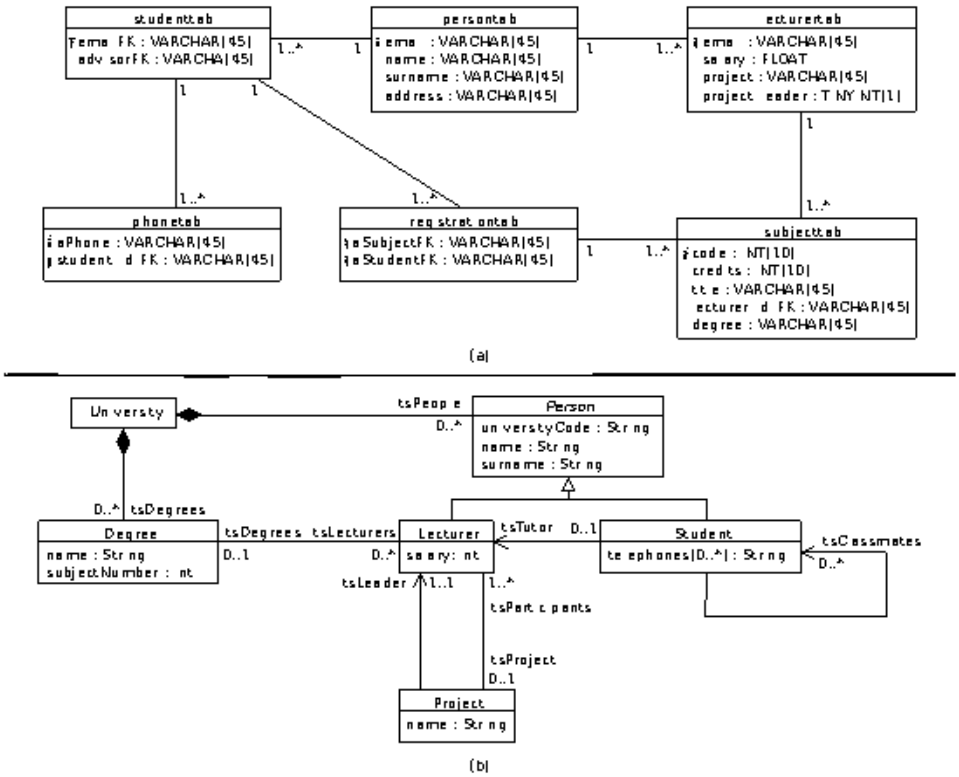


Figura 7.6: Ejemplo de extracción de modelos desde una base de datos que representa el personal de una universidad. (a) Esquema de la base de datos. (b) Metamodelo al que deben conformar los modelos extraídos.

representación del personal es similar a la utilizada en la base de datos, aunque en la base de datos las tablas han sido normalizadas. Además, la metaclase `Student` tiene una referencia para representar a los compañeros de un estudiante (referencia `itsClassmates`), esto es, aquellos estudiantes que comparten asignaturas. Por otro lado, los conceptos de `Degree` y `Project` son representados en el metamodelo como metaclases, mientras que en la base de datos no existen explícitamente. Así, la metaclase `Degree` se obtiene a partir de la agrupación de la columna `degree` de la tabla `subjecttab`, por lo que es una correspondencia de colección, mientras que la metaclase `Project` se obtiene a partir de las tuplas de la tabla `persontab` cuya columna `project_leader` es `true`, por lo que es una correspondencia de columna.

La transformación utilizada para este ejemplo se muestra en la Figura 7.7, la cual consta de un preámbulo y cinco reglas de trasnformación (una para cada clase concreta del metamodelo).

La transformación define un preámbulo para establecer las claves primarias/ajenas no incluidas en la base de datos así como un ejemplo de vista. En cuanto a las claves primarias/ajenas, se indican para la tabla `studenttab`, para la cual se especifica que la columna

```
preamble
  constraints
    table studenttab |
      primaryKey |emailFK|
        foreignKey |advisorFK|
          references lecturertab|emailFK|
    )
  end constraints
  views
    create view busyStudent |emailFK, advisorFK| as
    select * from studenttab stu
    where not exists |
      select * from subjecttab sub where not exists |
        select * from registrationtab
          where aSubjectFK = sub.code
          and aStudentFK = stu.emailFK)|;
  end views
end preamble

toprule 'mapUniversity'
  from Database db
  to   University uni
  mapping
   uni.itsPeople  = db.@lecturertab;
   uni.itsPeople  = db.@studenttab;
   uni.itsDegrees = db.@subjecttab[degree];
end rule

rule 'mapLecturer'
  from lecturertab lect
  to   Lecturer ulect
  mapping
    ulect.universityCode = "Lecturer " + lect.emailFK;
    ulect.name           = lect.emailFK.name;
    ulect.surname        = lect.emailFK.surname;
    ulect.salary         = lect.salary;
    ulect.itsProject     = db.@lecturertab
                           {project leader = 1 &&
                            project = lect.project};
end rule
```

```
rule 'mapProject'
  from lecturertab lec
  to   Project pro
  filter
   lec.project leader == 1
  mapping
    pro.name            = lec.project;
    pro.itsLeader       = lec.emailFK;
    pro.itsParticipants = db.@lecturertab
                          {project = lec.project};
end rule


rule 'mapStudent'
  from studenttab stu
  to   Student ustu
  mapping
    ustu.universityCode = "Student " + stu.emailFK;
    ustu.name           = stu.emailFK.name;
    ustu.surname        = stu.emailFK.surname;
    ustu.telephones     = stu.@phonetab.aPhone;
    ustu.itsClassmates  = stu.@registrationtab.aSubjectFK.
                          @registrationtab.aStudentFK;
    ustu.itsTutor       = stu.advisorFK;
end rule


rule 'mapDegree'
  from subjecttab sub
  to   Degree deg
  mapping
   deg.name           = sub.degree;
   deg.itsLecturers   = db.@subjecttab
                        {degree = sub.degree}.lecturer idFK;
   deg.subjectNumber = db.@subjecttab
                        {degree = sub.degree}.count|);
end rule
```

Figura 7.7: Reglas de transformación ScheMoL utilizadas en el ejemplo.

`emailFK` actúa de clave primaria y la columna `advisorFK` es una clave ajena a la columna `emailFK` de la tabla `lecturertab`. Por otro lado, la vista se ha incluido para ilustrar su uso en el preámbulo, la cual calcula aquellos alumnos que están inscritos en todas las asignaturas.

La ejecución de la transformación comienza con la regla de tipo *toprule* llamada `mapUniversity`, la cual establece la correspondencia entre la base de datos (uso de `Database` en la sección *from*) y la metaclase `University`. La ejecución de esta regla crea el elemento `University` que es la raíz del modelo resultante. La sección *mappings* contiene tres *bindings*, encargados de inicializar las referencias de la instancia de `University` creada, las cuales son `itsPeople` y `itsDegrees`. Los primeros dos *bindings* inicializan, en primer lugar, la referencia `itsPeople`. El primero de ellos ejecuta la consulta `db.@lecturertab`, la cual devuelve todas las tuplas de la tabla `lecturertab` de la base de datos, y ejecuta aquella regla que conforme con el *binding*, que es la regla `mapLecturer`. El segundo de ellos ejecuta la consulta `db.@studenttab`, la cual devuelve todas las tuplas de la tabla `studenttab` de la base de datos, y ejecuta aquella regla que conforme con el *binding*, que en este caso es la regla `mapStudent`. El último *binding* de la regla ejecuta la consulta

`db.@subjecttab[degree]`, la cual devuelve las tuplas de la tabla `subjecttab` agrupadas por la columna `degree`, y ejecuta la regla que conforme con dicho *binding*, que en este caso es `mapDegree`. Es importante destacar que, mientras que los dos primeros *bindings* de la regla han permitido establecer correspondencias tupla-modelo, el último de ellos ha permitido definir una correspondencia de colección, donde una metaclase representa una colección agrupada de elementos de una tabla, tal y como se ha comentado anteriormente.

La regla `mapLecturer` define la correspondencia entre la tabla `lecturertab` y la metaclase `Lecturer`. Por lo tanto, su ejecución crea una instancia de la metaclase `Lecturer`. Esta regla está compuesta por seis *bindings*, los cuales inicializan cada uno de los atributos y referencias de una instancia de `Lecturer`. Los cuatro primeros son *bindings* cuya resolución es inmediata, ya que las consultas o expresiones de la parte derecha dan como resultado valores cuyo tipo es primitivo al igual que el tipo del atributo de la parte izquierda. Así, el primer *binding* de la regla es una expresión que construye el código identificativo del elemento `Lecturer` a partir de la expressión `"Lecturer_" + lect.emailFK`, donde se utiliza la consulta `lect.emailFK` que devuelve el valor de la columna `emailFK` de la tupla recibida por la regla. Los *bindings* segundo y tercero utilizan las consultas `lect.emailFK.name` y `lect.emailFK.surname` para obtener el valor de las columnas `name` y `surname` de la tupla referenciada por la columna `emailFK`, que es una clave ajena a la tabla `persontab`. Estos valores se asignan a los atributos `name` y `surname`, respectivamente. El cuarto *binding* utiliza una consulta similar a la utilizada en el primer *binding* para establecer el valor del atributo `salary`. Finalmente, el último *binding* de la regla utiliza la consulta `db.@lecturertab{project_leader = 1 && project = lect.project}` para obtener todos aquellos proyectos para los que el elemento `lect`, que es la tupla recibida por la regla, es líder (su columna `project_leader` tiene valor `1`). Este *binding* ejecuta la consulta y dispara aquella regla que conforme con el *binding*, que en este caso es `mapProject`. Nótese que este último binding permite expresar correspondencias de tipo columna. Además, también es importante destacar que un elemento `Project` es representado en la base de datos como las tuplas de la tabla `lecturertab` cuya columna `project_leader` tiene valor `1`, dado que la base de datos no contempla dicho concepto.

La regla `mapProject` define la correspondencia entre la tabla `lecturertab` y la metaclase `Project`, por lo que su ejecución crea una instancia de dicha metaclase. El primer *binding* de esta regla asigna el valor devuelto por la consulta `lec.project` al atributo `name`. El segundo *binding* utiliza una consulta similar al anterior pero inicializa el atributo `itsLeader`, que representa al líder del proyecto. Finalmente, el último *binding* inicializa la referencia `itsParticipants`, que representa todos los participantes del proyecto. Para ello, ejecuta la consulta `db.@lecturertab{project = lec.project}` para inicializar la referencia `itsParticipants`, que obtiene todas las tuplas de la tabla `lecturertab` cuya columna `project` coincide con el valor de la colulmna `project` de la tupla recibida por la regla, y ejecuta la regla `mapLecturer`.

La regla `mapStudent` define la correspondencia entre la tabla `studenttab` y la metaclase `Student`. Su ejecución crea una instancia de la metaclase `Student` e inicializa todos sus atributos y referencias. Los tres primeros *bindings* siguen un formato similar a los tres primeros de la regla `mapLecturer`, explicados anteriormente. El cuarto *binding*

```
select distinct(aStudentFK)
from registrationtab
where aSubjectFK in (select aSubjectFK
   from registrationtab
   where aStudentFK=primaryKeyOfTheTupleAtHand)
```

Figura 7.8: Consulta `stu.@registrationtab.aSubjectFK.@registrationtab.aStudentFK` definida en SQL.

utiliza la consulta `stu.@phonetab.aPhone`, que obtiene la columna `aPhone` de todas las tuplas de la tabla `phonetab` que hacen referencia a la tupla actual (variable `stu`), y asigna dichos valores al atributo multivaluado `telephone`. El quinto *binding* utiliza la consulta `stu.@registrationtab.aSubjectFK.@registrationtab.aStudentFK` para obtener los compañeros del estudiante recibido como tupla de la regla y ejecuta aquella regla que conforme con el *binding*, que es la propia regla `mapStudent`. Esta consulta es un ejemplo de la potencia expresiva y la concisión que se puede alcanzar al utilizar los operadores de consulta de forma encadenada. La Figura 7.8 muestra la misma consulta definida en SQL, nótese la mejora en concisión y facilidad de definición al utilizar la notación punto. Finalmente, el último *binding* ejecuta la consulta `stu.advisorFK`, para obtener el profesor asociado al elemento `stu` tratado por la regla, y ejecuta la regla `mapLecturer` para inicializar la referencia `itsTutor`.

Finalmente, la regla `mapDegree` establece la correspondencia entre la tabla `subjecttab` y la metaclase `Degree`, por lo que su ejecución creará una instancia de dicha metaclase. El primer *binding* de esta regla utiliza la consulta `sub.degree` para establecer el atributo `name` de la instancia de metaclase. Los últimos dos *bindings* utilizan un filtro en la consulta para obtener las tuplas de la tabla `subjecttab` que representan al elemento `Degree` en la base de datos. Así, el segundo binding ejecuta la consulta `db.subjecttab{degree = sub.degree}.lecturer_idFK` para obtener los profesores asociados con el elemento `Degree` tratado por la regla, los cuales están representados por los valores de la columna `lecturer_idFK` de todas las tuplas para dicho `Degree`, y ejecuta la regla `mapLecturer` para inicializar la referencia `itsLecturers`, la cual representa a los profesores que imparten alguna asignatura en la titulación representada por `Degree`. Finalmente, el tercer *binding* utiliza la consulta `db.subjecttab{degree = sub.degree}.count()`, la cual calcula el número de tuplas del `Degree` actual e inicializa el atributo `subjectNumber`, que representa el número de asignaturas de la titulación representada por `Degree`.

7.8. Escenarios de aplicación

ScheMoL es un lenguaje para la extracción de modelos a partir de datos de bases de datos relaciones, tal y como se ha descrito a lo largo de este capítulo. Sin embargo, en el ámbito de las bases de datos pueden encontrarse diferentes tipos así como diferentes usos. En este sentido, el lenguaje fue utilizado por el grupo de investigación ONEKIN para abordar el problema de la extracción de modelos a partir de bases de datos utilizadas por aplicaciones Web 2.0. La particularidad principal de este tipo de bases de datos es que los datos de

los campos de la base de datos se encuentran anotados, es decir, utilizan etiquetas HTML para agregar semántica al contenido (p. ej., utilizando RDFa [84] o microformatos [34]). Estas anotaciones incorporan información semántica que debe ser extraída para inicializar los elementos del modelo. De esta forma, el proceso de extracción también debe tener en cuenta el contenido de los campos de una tupla.

Por este motivo, el lenguaje SchcMoL fue extendido para tratar con información anotada en los valores de las tuplas de las tablas de la base de datos. Así, el lenguaje de consultas fue enriquecido con dos funciones aplicables al resultado de una consulta: `class(htmlClass)`, que aplicada a un valor de tipo cadena, recupera el contenido del elemento cuya clase HTML es `htmlClass`; y `property(htmlPropertyName)`, que aplicada también a un valor de tipo cadena, obtiene el contenido de la propiedad `htmlPropertyName`. El lenguaje, junto con estas funciones para tratar con contenido anotado, fueron utilizados en casos de estudio desarrollados para extraer modelos desde sistemas de gestión de wikis y de blogs, los cuales pueden consultarse en `http://onekin.org/schemol`.

Otro escenario de aplicación de SchcMoL es en el ámbito del análisis de datos. En este contexto, los modelos extraídos por SchcMoL pueden ser utilizados para comprobar criterios de calidad o inconsistencia en los datos. Por ejemplo, los modelos extraídos de una base de datos que almacena la información de un sistema de gestión de wikis pueden ser utilizados para comprobar que todas las páginas estén asignadas a una categoría.

7.9. Características del lenguaje

La Figura 7.9 muestra las características principales de SchcMoL de acuerdo al diagrama de características presentado en [20]. SchcMoL es un lenguaje unidireccional cuyo dominio en el origen es el *dataware* y el dominio en el destino es el *modelware*. Una transformación SchcMoL está compuesta por un conjunto de reglas que transforman elementos de la base de datos, es decir, tuplas, en elementos del modelo extrayendo la información necesaria de la base de datos por medio de un potente lenguaje de consultas. Al igual que Gra2MoL, cada ejecución de la transformación crea un nuevo modelo destino. Las reglas se resuelven implícitamente y de forma determinista utilizando el mecanismo de *bindings*. Además, la condición de aplicación de las reglas depende de la sección *filter*. En cuanto a la información de trazabilidad, el motor de ejecución de SchcMoL crea un modelo de trazas automáticamente que relaciona los elementos creados con los elementos origen y las reglas aplicadas. Finalmente, SchcMoL no dispone de mecanismos de modularidad ni de reutilización de reglas así como tampoco es posible controlar explícitamente el flujo de ejecución de la transformación.

7.10. Conclusiones

En este capítulo se ha descrito el lenguaje SchcMoL, un DSL para la extracción de modelos a partir de datos de una base de datos relacional. El lenguaje está fuertemente inspirado en Gra2MoL, del cual reutiliza su diseño así como el uso de algunos elementos de su arquitectura. Además, SchcMoL incorpora un lenguaje de consultas especialmente adaptado para la

extracción de información de la base de datos. En el sitio web `http://modelum.es/schemol` pueden encontrarse todos los recursos que componen la herramienta.

Al igual que se realizó para Gra2MoL en la sección 5.11, a continuación se discute la adecuación del lenguaje con los requisitos para DSL presentados en [47]. Debido a que ScheMoL está basado en Gra2MoL, algunos de ellos son igualmente aplicables adaptando simplemente su descripción al dominio del *dataware*. De la misma que forma que se ha realizado en el capítulo 5, cada requisito se evalúa de menor a mayor cumplimiento utilizando los valores 1 a 5.

Conformidad (5). Todos los elementos utilizados en ScheMoL usan conceptos del dominio del problema de establecer un puente unidireccional entre el *dataware* y el *modelware* necesario para extraer modelos de datos almacenados en una base de datos relacional. Al igual que en Gra2MoL, las reglas de ScheMoL permiten definir las correspondencias entre los elementos de la base de datos y los del metamodelo. Así, la sección *from* de una regla hace referencia a la tabla del esquema mientras que la sección *to* lo hace a la metaclase. El lenguaje también utiliza el concepto de *binding* para establecer las correspondencias entre las propiedades del elemento del metamodelo y la información de la base de datos, el cual ha sido adaptado para trabajar en este dominio. Además, el lenguaje de consultas ofrece operadores para facilitar la navegación en la base de datos.

Ortogonalidad (5). En el lenguaje no existen elementos que representen más de un concepto del dominio. Cada elemento del lenguaje tiene una finalidad que no es utilizada por ningún otro elemento.

Soporte (2). ScheMoL actualmente sólo ofrece las herramientas necesarias para su ejecución a través de una tarea ANT. A diferencia de Gra2MoL, no existe un entorno especialmente adaptado en Eclipse para trabajar con ScheMoL, aunque hemos identificado esta necesidad como trabajo futuro.

Integración (1). El lenguaje ha sido desarrollado en Java pero actualmente no ofrece mecanismos específicos para facilitar la integración con otras herramientas. Por este motivo, se ha identificado la creación de un *plugin* para la plataforma Eclipse como trabajo futuro con el objetivo de mejorar la integración del lenguaje.

Extensibilidad (1). Tal y como se ha comentado en la sección 7.8, el lenguaje ScheMoL fue ampliado para contemplar operadores que faciliten el manejo de datos anotados en las bases de datos, los cuales fueron incluidos directamente en la sintaxis del lenguaje. Sin embargo, a diferencia de Gra2MoL, ScheMoL no ofrece ningún mecanismo para extender el lenguaje, el cual se ha considerado como trabajo futuro.

Longevidad (4). Se puede considerar un alto grado de longevidad dado que la extracción de modelos a partir de datos de una base de datos es una actividad importante en los procesos modernización de software que utilice el paradigma DSDM.

Simplicidad (4). Al igual que en Gra2MoL, el proceso de definición de una transformación en ScheMoL consiste en: (1) definir las reglas y (2) utilizar el lenguaje de consultas

para extraer información desde la base de datos. Estas tareas no tienen un grado de complejidad elevado y el tiempo necesario para realizarlas depende principalmente del tamaño y complejidad de los elementos (metamodelo y esquema de base de datos) involucrados en la transformación.

Calidad (4). ScheMoL es un lenguaje interpretado por lo que este requisito no es directamente aplicable a este DSL. Solamente puede considerarse la calidad de los modelos resultantes de un proceso de transformación, el cual realmente depende del metamodelo utilizado y las reglas definidas.

Escalabilidad (2). A diferencia de Gra2MoL, ScheMoL ejecuta las consultas en la base de datos directamente sin necesitar ninguna estructura de datos intermedia que pueda comprometer la escalabilidad. Sin embargo, al igual que Gra2MoL, los modelos resultado de una transformación ScheMoL pueden tener un tamaño considerable, dado que las bases de datos contienen normalmente una gran cantidad de información. En este caso, también se ha identificado como trabajo futuro el estudio de mecanismos para almacenar modelos grandes, como un repositorio de modelos.

Usabilidad (4). ScheMoL está profundamente inspirado en Gra2MoL, el cual está a su vez inspirado en lenguajes de transformación *m2m* como ATL. Por estos motivos, creemos que su adopción por los desarrolladores de la comunidad del DSDM no supondría un gran esfuerzo.

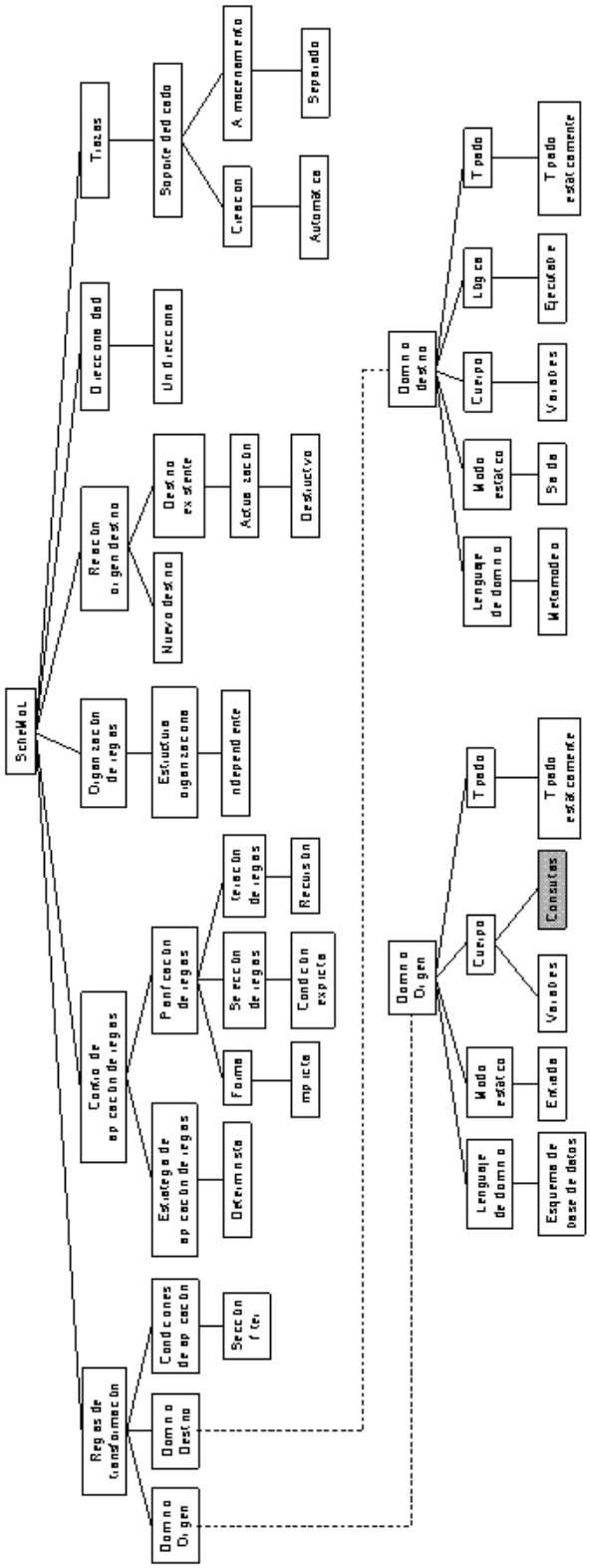


Figura 7.9: Diagrama de características de SchcMoL de acuerdo a [20]. Se marcan en gris las características propias del lenguaje.

8

Integración de API con DSDM

::En tu entrenamiento, no. En tu integración. Empezamos la instrucción mañana. Pero tu integración es igual de importante. No puedes entrenarte si no estás integrado – dijo Brahe
Las tropas fantasma, John Scalzi.

Los escenarios de aplicación del DSDM que requieren la integración de API con modelos, como el uso de modelos *runtime* o el acceso a datos Web 2.0 desde modelos, aprovechan la conversión de los objetos de un API en modelos para aplicar las técnicas propias del DSDM. En este capítulo se presenta el lenguaje API2MoL, que facilita la creación de puentes bidireccionales entre API y metamodelos con el propósito de automatizar la extracción de modelos desde los objetos de un API y la generación de dichos objetos a partir de los modelos. Dadas las características propias de API2MoL que lo diferencian de Gra2MoL y ScheMoL, este capítulo tiene una organización diferente a los capítulos 5 y 7. En primer lugar describiremos los procesos de extracción y generación, así como la necesidad del lenguaje y la posibilidad de generar automáticamente los artefactos necesarios para crear un puente entre un API concreto y su metamodelo por medio de un proceso de descubrimiento. A continuación presentaremos el lenguaje y funcionamiento con la ayuda de un ejemplo y luego describiremos el proceso de descubrimiento desarrollado. Finalmente, de la misma forma que se hizo para los lenguajes anteriores, se comentan algunos escenarios de aplicación, se presentan las características del lenguaje y las conclusiones, donde también se describirá el grado de cumplimiento de los requisitos identificados en [47].

8.1. Descripción del problema

La manipulación de API es una necesidad que se presenta en la mayoría de las aplicaciones de las técnicas del DSDM. Así ocurre en los procesos de ingeniería directa y reingeniería, donde es necesario generar el código del API a partir de los modelos y extraer modelos a partir de los artefactos software, respectivamente. Por otro lado, los API son también

utilizados para acceder y modificar los datos de una aplicación en tiempo de ejecución (p. ej., los datos de aplicaciones Web 2.0) en aplicaciones basadas en DSDM. Esta interacción en tiempo de ejecución requiere que los objetos del API puedan interoperar con las aplicaciones basadas en DSDM, es decir, que los objetos del API puedan ser extraídos como modelos y viceversa. Mientras la generación de código del API a partir de modelos puede ser automatizada mediante los lenguajes de plantillas (p. ej., MOFScript [54] y Xpand [109]) y existen propuestas para aplicar técnicas de ingeniería inversa a partir del código del API, actualmente no existen herramientas que faciliten la interoperabilidad entre los API y el DSDM.

La interoperabilidad entre los API y el DSDM se puede lograr por medio de la construcción de un puente bidireccional entre ambos espacios tecnológicos, el cual, tal y como se comentó en la sección 4.6, debe soportar dos operaciones: (1) extraer modelos a partir de los objetos de un API, y (2) generar objetos de un API a partir de modelos. Nuestra propuesta es un DSL llamado API2MoL para definir los puentes entre el *apiware* y el *modelware*. De esta forma, a diferencia de Gra2MoL y SchcMoL, API2MoL no solo permite aplicar un proceso de extracción de modelos sino también un proceso de generación, es decir, permite obtener modelos a partir de los objetos de un API así como generar dichos objetos a partir de los modelos.

A continuación definiremos genéricamente los procesos de extracción y generación para API orientadas a objetos ya que son quizás las más utilizadas en el desarrollo de software. En este tipo de API, los artefactos son clases que se instancian en tiempo de ejecución. En el resto de este capítulo nos centraremos en los API Java pero la adaptación a otros API orientados a objetos sería directa. Una vez presentados los procesos de extracción y generación, motivaremos la necesidad de un lenguaje para expresar las correspondencias entre la especificación del API y el metamodelo del API, las cuales son los artefactos principales para guiar dichos procesos. Finalmente, analizaremos cómo construir automáticamente el puente para simplificar el trabajo de los desarrolladores.

8.1.1. El proceso de extracción de modelos a partir de los objetos de un API

La extracción de modelos a partir de los objetos de un API requiere la definición de un metamodelo que represente los elementos del API. Con el objetivo de simplificar el proceso y no perder la información que es representada por el API, este metamodelo contendrá una metaclase para cada clase del API. Esta aproximación es similar a la aplicada en los procesos de extracción de modelos a partir del código fuente donde se obtienen modelos AST cuyo nivel de abstracción es relativamente cercano al código. De esta forma, el grafo de objetos del API, es decir, el resultado de una interacción específica entre el programa y el API, sería expresado como un modelo que conforma a dicho metamodelo. Una vez se disponen de estos modelos, se pueden aplicar transformaciones *m2m* para alcanzar el nivel de abstracción deseado.

De este modo, el modelo extraído contendría un elemento para cada objeto del API en el programa. Estos elementos del modelo conforman a las metaclases que representan

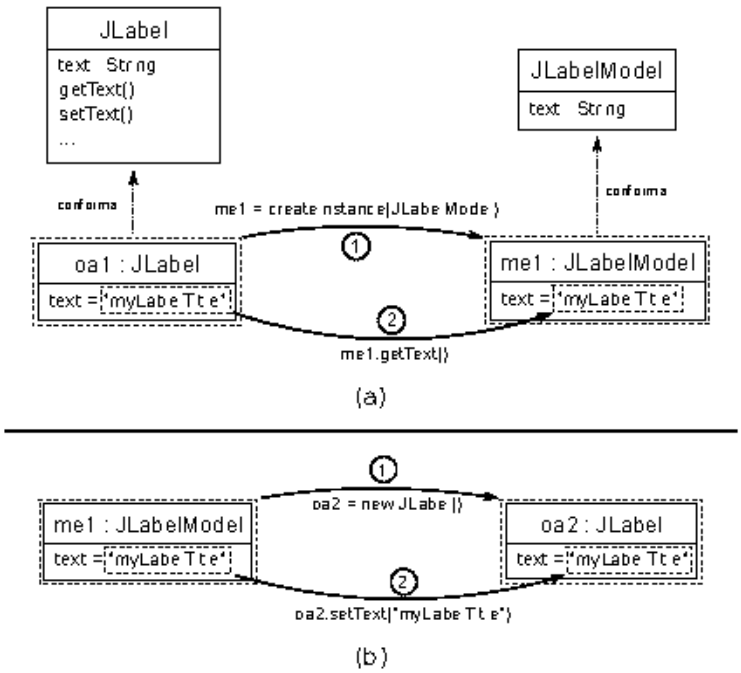


Figura 8.1: (a) Proceso de extracción de modelos a partir de objetos Java en API2MoL. (b) Proceso de generación de modelos en objetos Java en API2MoL.

las correspondientes clases del API a las cuales el objeto conforma, y son inicializados por medio de los valores que devuelven los métodos del API (p. ej., los métodos de tipo *get*). La Figura 8.1a ilustra el proceso de extracción de modelos del API por medio de un ejemplo simple que solamente utiliza un objeto `JLabel` del API Swing. La metaclase `JLabelModel` corresponde a la clase `JLabel` de la clase del API y el elemento del modelo `em1`, que es una instancia de `JLabelModel`, es creada para representar el objeto del API `oa1`. Por otro lado, las propiedades de `m1` (p. ej., `text`) son inicializadas utilizando los métodos de tipo *get* de `JLabel` (p. ej., `getText`). Como puede observarse, el proceso de extracción debe conocer las correspondencias entre las clases del API y las metaclases del metamodelo, las cuales se definen a nivel M2 dentro de la arquitectura de cuatro capas del OMG tal y como se ha comentado en la sección 4.6.

8.1.2. El proceso de generación de objetos del API a partir de los modelos

El proceso de generación es aplicado de forma parecida al de extracción, pero en dirección contraria. Un proceso de generación tiene un modelo como entrada y genera un objeto del API por cada elemento del modelo. Estos objetos generados son inicializados invocando a los métodos ofrecidos por el API para su manejo (p. ej., los métodos de tipo *set*). De la misma forma que antes, las correspondencias entre las clases del API y las metaclases del metamodelo también son utilizadas para dirigir el proceso e identificar los métodos que inicializan los objetos. La Figura 8.1b ilustra el proceso de generación de objetos para

el ejemplo mostrado en la Figura 8.1a. El objeto del API llamado `oa2` se crea a partir del elemento del modelo `me1` de tipo `JLabelModel` y, a continuación, el atributo `text` se inicializa llamando al correspondiente método de tipo *set* de `JLabel` (p. ej., `setText`). Es importante destacar que el proceso de generación propuesto crea los objetos API en memoria en tiempo de ejecución, facilitando su alteración y mantenimiento. Otra posibilidad podría ser generar el código fuente que incluyera el conjunto de llamadas para crear los artefactos del API e inicializar su atributos.

8.1.3. Un lenguaje de correspondencias para definir los procesos de extracción y generación

Como se ha comentado anteriormente, una herramienta que implemente un puente entre el *apiware* y el *modelware* debe manejar las correspondencias entre las clases del API y las metaclases del metamodelo para ejecutar los procesos de extracción y generación. El conocimiento de cómo realizar estos procesos puede implementarse directamente en la herramienta, pero ésta sería una tarea complicada y específica para un API. Una alternativa sería utilizar algún formalismo (p. ej., XML o un DSL) para expresar las correspondencias entre un API y un metamodelo, de modo que la herramienta fuera parametrizable por dichas correspondencias. De esta forma, la herramienta no sería específica para un par concreto <API, metamodelo>, sino que sería genérica. Con esta finalidad, decidimos crear un nuevo DSL denominado API2MoL para especificar estas correspondencias.

API2MoL es un lenguaje basado en reglas que permite definir las correspondencias entre las clases del API y las metaclases del metamodelo declarativamente. Así, una definición en dicho DSL consiste en un conjunto de reglas definidas para un par concreto <API, metamodelo>. Además, las reglas son bidireccionales, de forma que una misma definición puede ser utilizada para aplicar tanto el proceso de extracción como el de generación. La Figura 8.2a ilustra nuestra aproximación para el ejemplo de extracción/generación descrito anteriormente, mientras que la Figura 8.2b muestra la definición de correspondencias en API2MoL. Esta definición incluye solamente una regla para especificar la correspondencia entre la clase del API `JLabel` y la metaclase `JLabelModel` (`JLabel : JLabelModel`). La regla especifica cómo extraer/generar el atributo `text` por medio de una sección de la regla que indica los métodos ofrecidos por el API. Esta sección está dividida en sentencias que especifican los métodos a utilizar en el proceso de extracción y generación. Así, para extraer el valor del atributo `text` se utilizaría el método `getText` (indicado por la sentencia `get getText()`), mientras que para realizar el proceso de generación se utilizaría el método `setText` (indicado por la sentencia `set setText()`). Esta definición API2MoL sería interpretada por el puente genérico para extraer elementos del modelo conformes a la metaclase `JLabelModel` a partir de objetos conformes a la clase del API `JLabel` y generar dichos objetos a partir de elementos del modelo conformes a la metaclase `JLabelModel`. En la sección 8.2 se describirán todos los elementos ofrecidos por el DSL para poder adaptar las correspondencias a la funcionalidad ofrecida por el API.

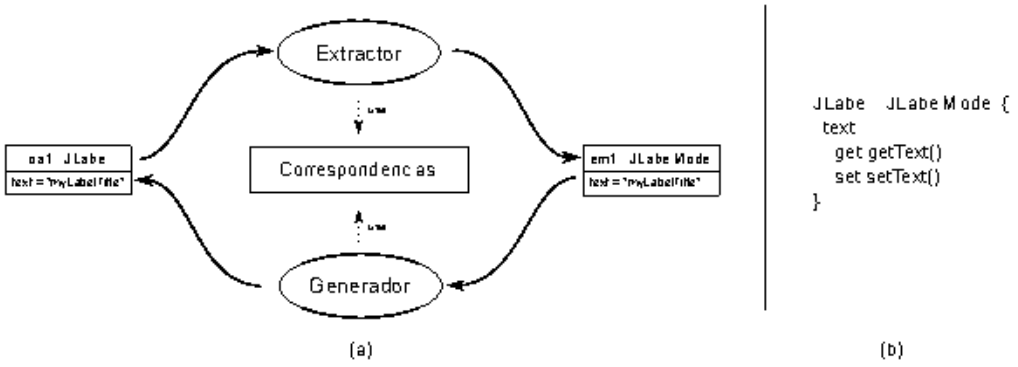


Figura 8.2: (a) Uso de la definición de correspondencias para realizar los procesos de extracción y generación mostrados en las Figuras 8.1a y 8.1b. (b) La definición de correspondencias expresada utilizando API2MoL.

8.1.4. Generación automática de los puentes *apiware-modelware*

Debido a que las API están compuestas normalmente por un número considerable de clases, el esfuerzo para desarrollar un puente *apiware-modelware* no se vería considerablemente reducido utilizando API2MoL, debido a que la creación del metamodelo y de la definición de correspondencias sería una tarea costosa en esfuerzo y en tiempo. Con este problema en mente y dada la cercanía semántica entre el metamodelo y las clases del API, API2MoL incorpora un proceso de descubrimiento para generar automáticamente tanto dicho metamodelo como la definición API2MoL a partir de las clases del API, tal y como se muestra en la Figura 8.3.

Este proceso de descubrimiento podría llevarse a cabo utilizando herramientas de análisis de código como JDT [69] o JastAdd [68] de tal forma que el código del API debería ser analizado para generar tanto el metamodelo como la definición API2MoL. Sin embargo, el desarrollo de este *descubridor* sería una tarea compleja ya que implicaría especificar imperativamente las heurísticas necesarias para analizar el código del API. Además, esta opción solamente es aplicable cuando se dispone del código fuente del API. Una alternativa sería que el *descubridor* analizara las clases del API en tiempo de ejecución. En este caso, la tarea de analizar y comprender la estructura del API sería realizada por reflexión, que debería estar soportada por el lenguaje de programación utilizado para desarrollar el API. Así, la entrada del proceso sería una instancia de cada una de las clases del API. En principio, el desarrollo de un *descubridor* basado en reflexión no debería ser más simple que utilizar un proceso de análisis de código, debido a que todavía sería necesario analizar la información obtenida reflexivamente, sin embargo, este proceso puede ser automatizado utilizando API2MoL y presentado en la sección anterior.

Debido a que la capacidad de reflexión de un lenguaje de programación es ofrecida normalmente por un API reflexiva, sería posible definir las correspondencias en API2MoL para obtener un modelo reflexivo que describa las clases del API. A continuación, este modelo podría ser utilizado para generar automáticamente tanto el metamodelo como la definición

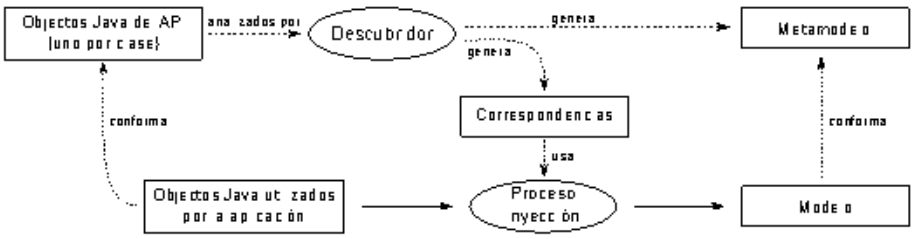


Figura 8.3: Proceso de descubrimiento del metamodelo y de la definición de correspondencias. En la figura, la definición de correspondencias se utiliza como entrada en el proceso de extracción pero también podría ser utilizado por un proceso de generación.

API2MoL para un API determinada. El metamodelo generado tendría normalmente una metaclase para cada clase del API y los atributos de dichas metaclases serían generados a partir de los atributos de las propias clases del API, aunque algunas veces podría ser necesaria la aplicación de heurísticas aplicadas a los métodos del API (p. ej., atributos calculados). Por otro lado, también sería necesario la aplicación de heurísticas para generar la definición de las correspondencias entre las clases del API y las metaclases del metamodelo (p. ej., descubrir los métodos para inicializar los elementos del modelo). Debido a que la implementación actual de API2MoL trata con API Java, hemos implementado este proceso de descubrimiento utilizando el API reflexivo de Java. Este proceso recibe el nombre de proceso *bootstrap* debido a que utilizamos API2MoL para descubrir tanto el metamodelo del API como la definición API2MoL necesarias para aplicar un proceso API2MoL. Este proceso será explicado con más detalle en la sección 8.4.

8.2. El lenguaje API2MoL

Una definición API2MoL está compuesta por un conjunto de reglas donde cada regla define la correspondencia entre una metaclase del metamodelo y una clase del API, incluyendo las correspondencias entre las propiedades de la metaclase y los métodos que tienen que ser invocados para leer/escribir dichas propiedades, tal y como se ha indicado en la sección anterior. Obviamente, el metamodelo destino del API debe estar disponible para poder definir las correspondencias. En la sección 8.4 mostraremos como puede generarse automáticamente este metamodelo.

Para ilustrar los conceptos principales del lenguaje, utilizaremos el API Swing [77] como ejemplo a lo largo de esta sección, la cual está basada en el *Abstract Windows Toolkit* (AWT) y facilita el desarrollo de interfaces gráficas de usuario para aplicaciones Java. Por ejemplo, este API puede ser utilizado para desarrollar una interfaz gráfica como la mostrada en la Figura 8.4a, la cual está compuesta por un elemento `JFrame`, que representa a la ventana de la interfaz, y dos elementos `JButton` del API, que son los botones. La Figura 8.4b muestra una parte del metamodelo al cual deben conformar los modelos utilizados en la extracción o generación. Nótese que el metamodelo Swing tiene una estructura de

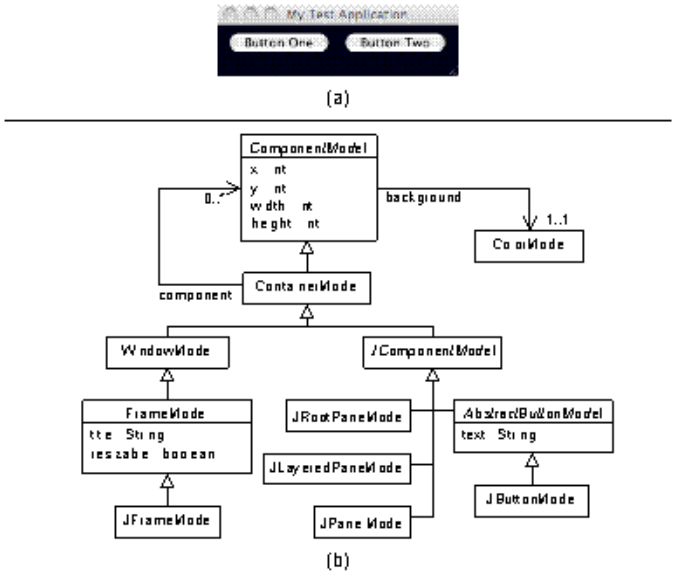


Figura 8.4: Ejemplo Swing utilizado para ilustrar el lenguaje API2MoL. (a) Aplicación Swing de ejemplo a ser extraída y generada. (b) Parte del metamodelo Swing al cual deben conformar los modelos extraídos.

metaclases muy similar a la de las clases del API Swing. Por claridad, las metaclases tienen el sufijo `Model` para evitar la confusión con las clases Java del API. La sección 8.4 mostrará como hemos sido capaces de derivar automáticamente parte de este metamodelo a partir de la especificación del API.

La Figura 8.5 muestra extracto de la sintaxis abstracta de API2MoL, mientras que un ejemplo de su sintaxis textual se muestra en la Figura 8.6. La Figura 8.6a muestra el esqueleto de una definición API2MoL y la Figura 8.6b presenta un ejemplo de sintaxis concreta para el ejemplo de Swing descrito anteriormente. A continuación describiremos los principales elementos del lenguaje así como su notación textual utilizando el ejemplo de Swing para ilustrarlos.

Una definición API2MoL está representada por la metaclase `Definition`, que es el elemento raíz del metamodelo del DSL, el cual incluye el atributo `context`, una sección *Default* (referencia `defaultMetaclass`) más un conjunto de reglas de correspondencia (referencia `mappings`). El contexto define uno o más nombres de paquetes Java los cuales son utilizados para delimitar el proceso de extracción, esto es, el conjunto de clases del API a considerar. Por ejemplo, el contexto del ejemplo Swing está formado por `java.awt.*` y `javax.swing.*`, que son los paquetes que contienen las clases utilizadas por el API Swing. Aquellos objetos Java que no estén incluidos en los paquetes definidos por el contexto serán extraídos conforme a la sección *Default*. Esta sección indica el nombre de la metaclase que se utilizará para representar a los objetos fuera de contexto. Además, la sección *Default* puede especificar el atributo de la metaclase que almacenará el nombre de la clase del objeto fuera de contexto. Así, en el ejemplo Swing, la sección *Default* especifica la metaclase

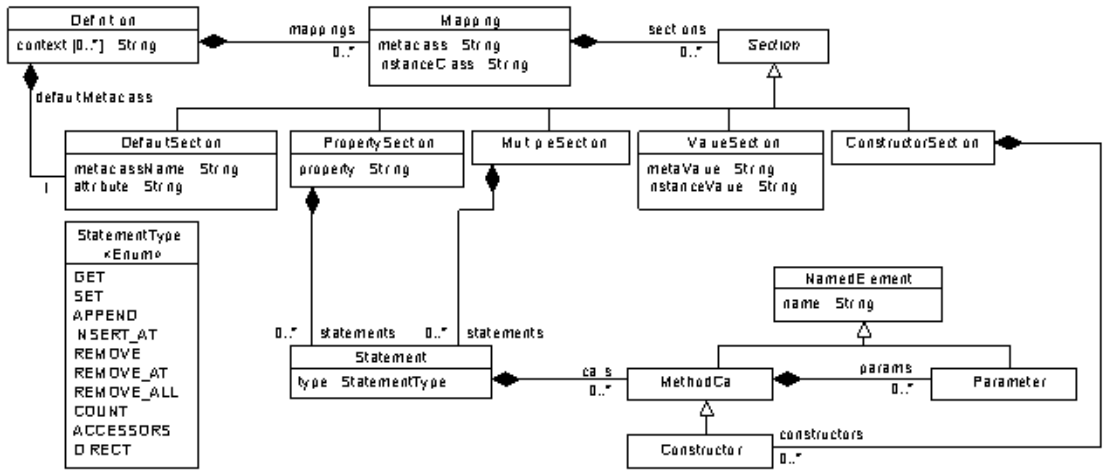


Figura 8.5: Un extracto de la sintaxis abstracta de API2MoL.

`UnknownElement` y el atributo `type`.

Las reglas en API2MoL están compuestas por una cabecera y un conjunto de secciones. La cabecera especifica la clase Java y la metaclase asociadas a la correspondencia definida en la regla (atributos `metaclass` y `instanceClass` de la metaclase `Mapping`). La clase Java es especificada por medio de su nombre canónico (p. ej., la primera regla del ejemplo en la Figura 8.6b especifica la correspondencia entre la metaclase `Component` y la clase Java `java.awt.Component`). Cuando una regla para una determinada clase del API no se ha definido pero dicha clase está incluida en el contexto, se aplica automáticamente una regla predefinida. Las reglas predefinidas definen la correspondencia entre metaclases y clases API (sin el prefijo de paquete) que tienen el mismo nombre (p. ej., si no existiera ninguna regla para la clase `java.awt.Component`, API2MoL aplicaría la regla predefinida entre esta clase del API y la metaclase `Component`). Así, API2MoL puede aplicar reglas normales o predefinidas.

Las secciones de una regla definen cómo se debe aplicar la correspondencia, es decir, cuáles son los métodos de la clase Java indicada en la cabecera que deben utilizarse para leer/escribir las propiedades de la metaclase. Hay cinco tipos de secciones: *Property*, *Default*, *Multiple*, *Constructor* y *Value*, todas ellas subclases de la metaclase `Section`.

Las secciones *Property* (metaclase `PropertySection`) especifican una correspondencia bidireccional entre una propiedad de la metaclase (atributo y referencia) y una propiedad de la clase del API (la mayoría de las veces métodos, pero pueden ser atributos accesibles públicamente). Estas correspondencias son aplicadas durante el proceso de extracción y el de generación (p. ej., las secciones para las propiedades `title` y `resizable` de la tercera regla en el ejemplo). Cada sección *property* especifica el nombre de la propiedad del metamodelo seguida del carácter `:` y un conjunto de elementos *statements* (metaclase `Statement`), los cuales describen el tipo de acceso que ofrece el API para leer/escribir dicha propiedad junto con los nombres de los métodos (metaclase `MethodCall`) que implementan

```
( <ContextSection> )? ';'
( <DefaultMetaclassSection> )? ';'

// mapping rules
<metaclass> ':' <instanceClass> '{'
  ( <DefaultMetaclassSection> |
    <ConstructorSection> |
    <PropertySection> |
    <MultipleSection> |
    <ValueSection> )*
'}'
```

|a|

```
@Context javax.swing.*, java.awt.*;
@DefaultMetaclass UnknownElement(type);

ComponentModel : java.awt.Component {
  x : get;
  y : get;
  width : get;
  height : get;
  background : accessors;
  @Multiple
    set SetBounds(x, y, width, height);
  ...
}

ContainerModel : java.awt.Container {
  component:
    get;
    append add(element);
  ...
}

FrameModel : java.awt.Frame {
  title : accessors;
  resizable : accessors;
  ...
}
```

```
WindowModel : java.awt.Container { ... }

JFrameModel : javax.swing.JFrame { ... }

JComponentModel : javax.swing. JComponent { ... }

JRootPaneModel : javax.swing.JRootPane { ... }

JLayeredPaneModel : javax.swing.JLayeredPane { ... }

JPanelModel : javax.swing.JPanel { ... }

AbstractButtonModel : javax.swing.AbstractButton {
  text : accessors;
}

JButtonModel : javax.swing.JButton { ... }

ColorModel : java.awt.Color {
  @new Color(RGB);
  RGB : get;
}
```

|b|

Figura 8.6: Sintaxis concreta de API2MoL. (a) Esqueleto de una definición API2MoL. (b) Un extracto de la definición API2MoL para el ejemplo Swing.

dicho acceso. En la definición de los elementos *statement* es posible no indicar el método que debe utilizarse, en tal caso, API2MoL infiere el nombre del método a partir del tipo de acceso. Por ejemplo, en un elemento *statement* de tipo `GET` para la propiedad `X`, se infiere el nombre de método `getX`. Si el API define métodos cuyo nombre no sigue esta regla, el desarrollador deberá especificar su nombre manualmente.

Como resultado del análisis de un conjunto de API, se identificaron diez tipos diferentes de elementos *statement*, tal y como se muestra en la Tabla 8.1. Estos elementos *statement* representan los tipos básicos de métodos ofrecidos por un API para acceder y modificar sus objetos, ya sean de tipo primitivo o colección.

Por ejemplo, la aplicación Swing utilizada como ejemplo incluye varios *statement* de diferentes tipos: `GET`, `SET`, `ACCESSOR` y `APPEND`. La primera regla del ejemplo utiliza un elemento *statement* de tipo `GET` en las secciones de las propiedades `x`, `y`, `width` y `height` para obtener el valor de dichas propiedades. Debido a que el API solamente incluye los métodos de tipo *get* para estas propiedades no se ha podido utilizar el elemento *statement* de tipo `SET`. En vez de ello, se utiliza una sección de tipo *Multiple*, tal y como se explicará más adelante. Por otro lado, esta misma regla utiliza el *statement* de tipo `ACCESSOR` para la propiedad `background` ya que existe tanto el método de tipo *get* como *set* para tal propiedad. La segunda regla del ejemplo ilustra el uso de los elementos *statement* de tipo `GET` y `APPEND` para la sección `component`, que hace referencia a la propiedad de tipo colección con el mismo nombre. El primero es utilizado para obtener el valor de dicha propiedad

Tipo	Descripción
`GET`	Obtiene el valor de la propiedad de la clase del API
`SET`	Asigna el/los valores a una o más propiedades del la clase del API
`ACCESSORS`	Especifica la existencia de elementos *statament* de tipo `SET` y `GET`
`APPEND`	Especifica el método del API utilizado para añadir elementos a una colección
`INSERT_AT`	Especifica el método utilizado para incluir un elemento en una determinada posición de una colección
`REMOVE`	Especifica el método utilizado para eliminar un elemento de una colección
`REMOVE_AT`	Especifica el método utilizado para eliminar un elemento de una determinada posición de una colección
`REMOVE_ALL`	Especifica el método utilizado para eliminar todos los elementos de una colección
`COUNT`	Especifica el método que cuenta el número de elementos de una colección
`DIRECT`	Indica que el atributo puede ser accedido directamente. Se utiliza principalmente con atributos de accesibilidad pública.

Tabla 8.1: Tipos de elementos *statement*.

mientras que el segundo permite añadir un nuevo elemento a la colección. La tercera regla del ejemplo también ilustra el uso de los elementos *statement* de tipo `ACCESSOR` para las propiedades `title` y `resizable`.

Las secciones de tipo *Property* utilizadas al aplicar una regla predefinida solamente tienen elementos *statement* de tipo `GET` y `SET`, los cuales no especifican el nombre del método que debe ser utilizado, por lo que se infieren tal y como se ha explicado anteriormente.

La sección *Default* (metaclase `DefaultSection`) de una regla es similar a la sección *Default* de una definición API2MoL pero en este caso, es aplicada solamente a aquellas clases que son subclases de la clase especificada en la regla y para las cuales no se ha definido una regla normal. Esta sección se utiliza normalmente para impedir la aplicación de reglas predefinidas o cuando la clase de la cabecera de la regla no puede ser instanciada al ser abstracta. Por ejemplo, si la primera regla del ejemplo definiera una sección *Default*, cualquier subclase de `Component` sin una regla normal definida sería extraída de acuerdo a dicha sección *default*.

Las secciones de tipo *Multiple* se utilizan solamente durante el proceso de generación cuando el API incorpora métodos que tratan con una o más propiedades al mismo tiempo. Una sección *Multiple* se incluye en una regla haciendo uso de la palabra clave `multiple` seguida de uno o más elementos *statement*. Por ejemplo, la primera regla del ejemplo define una sección *Multiple* que especifica que las propiedades `x`, `y`, `width` y `height` se pueden establecer conjuntamente utilizando el método `setBounds`.

El constructor por defecto, es decir, el constructor sin parámetros es utilizado normalmente para crear los objetos Java a partir de los elementos del modelo en un proceso de generación. Sin embargo, algunas veces es necesario especificar un constructor concreto ofrecido por el API (p. ej., cuando el constructor por defecto no está disponible). En este caso, se utilizan las secciones de tipo *Constructor* (metaclase `ConstructorSection`). Este tipo de sección se incluye en la regla haciendo uso de la palabra clave `@new` seguida por el método constructor (metaclase `Constructor`) que debe ser utilizado. Por ejemplo, la última regla del ejemplo include una sección *Constructor* la cual especifica el constructor a utilizar para crear objetos de tipo `Color`.

API2MoL ofrece soporte a valores enumerados por medio de las secciones de tipo *Value*

```
enum DialogType : int [
  PLAIN     : javax.swing.JRootPane.PLAIN_DIALOG;
  WARNING   : javax.swing.JRootPane.WARNING_DIALOG;
  QUESTION  : javax.swing.JRootPane.QUESTION_DIALOG;
  _
]
```

Figura 8.7: Ejemplo de regla de tipo `enum` y secciones de tipo *Value*.

(metaclase `ValueSection`). Estas secciones se utilizan para definir las correspondencias entre un valor enumerado del metamodelo (atributo `metaValue`) y un valor enumerado del lenguaje de programación (atributo `instanceValue`), y solamente pueden utilizarse como parte de un tipo especial de reglas denominadas reglas `enum`. La Figura 8.7 muestra un elemento de regla `enum` que define las correspondencias entre valores del tipo enumerado `DialogType` y valores del tipo enumerado declarado en la clase `JRootPane` del API Swing. Esta regla incluye varias secciones de tipo *Value* para definir las correspondencias entre cada valor de `DialogType` y `JRootPane`. La regla también especifica el tipo de los valores, es decir, el tipo del valor enumerado utilizado en Java, que en este caso es entero.

Por otro lado, es importante destacar que API2MoL soporta la sobrecarga de métodos y constructores. Al especificar un método en un elemento *statement*, el tipo del argumento puede ser indicado utilizando corchetes para seleccionar el método más apropiado. Por ejemplo, si la clase `Frame` incluyera los métodos `setTitle(String)` y `setTitle(Object)` y debe utilizarse el primero, el elemento *statement* debería indicar el método `setTitle([String])`. Lo mismo ocurre con los constructores. Por ejemplo, si la clase `Color` pudiera ser construida utilizando los constructores `Color(int)` y `Color(Object)` y se quiere especificar que debe utilizarse el primero, la sección *Constructor* debería indicar el constructor `Color([int])`. Si un método/constructor está sobrecargado y no se indican los tipos que deben utilizarse, API2MoL seleccionará aquel cuyos tipos de los argumentos conformen a los tipos de la propiedad del metamodelo.

Por último, nótese que la bidireccionalidad de una definición API2MoL depende de las secciones incluídas en las reglas, las cuales pueden restringir el comportamiento del proceso de extracción y/o generación. Por ejemplo, la primera regla del ejemplo de Swing define una correspondencia bidireccional de la metaclase `Component` debido a que las propiedades de la metaclase pueden ser leídas/escritas desde/hacia las propiedades de la clase Java (la propiedad `background` tiene el `statement` de tipo `ACCESSOR` y las propiedades `x`, `y`, `width` y `height` tienen elementos *statament* de tipo `GET` para leer sus valores desde el objeto así como una sección *Multiple* para establecerlas). Sin embargo, en algunos casos puede ocurrir que las propiedades de una clase API tenga un acceso especial y las correspondientes reglas solamente incluyan secciones para realizar el proceso de extracción o el de generación (p. ej., propiedades de sólo lectura en el caso del proceso de extracción).

8.3. Ejecución del lenguaje API2MoL

En esta sección describiremos cómo se ejecuta la definición API2MoL en los procesos de extracción y generación. Tal y como se explicó en la sección 8.1, nuestra aproximación trata con los objetos del API en tiempo de ejecución, lo que permite ejecutar las definiciones API2MoL *al vuelo*, facilitando su prueba y desarrollo. Sin embargo, precisamente por utilizar los objetos en tiempo de ejecución es necesario utilizar el API reflexivo del lenguaje de programación utilizado para desarrollar el API. De esta forma, la implementación actual de API2MoL soporta el API Reflexivo de Java, aunque puede ser adaptado a otros API reflexivos.

A continuación describiremos la semántica procedural del lenguaje API2MoL por medio de la especificación de forma algorítmica tanto del proceso de extracción como del proceso de generación.

8.3.1. El proceso de extracción

El algoritmo mostrado en la Figura 8.8 describe el comportamiento de API2MoL durante el proceso de extracción de un elemento del modelo denominado `Instancia` a partir de un objeto del API llamado `Objeto`.

Durante el proceso de extracción, este algoritmo es aplicado a todos los objetos en memoria del API que corresponden a una determinada ejecución del programa para obtener su representación como modelo. Estos objetos del API están organizados en forma de grafo, el cual incluye normalmente un elemento que actúa de raíz y que es utilizado para comenzar el proceso, aunque si existieran varios objetos de tipo raíz el proceso se repetiría para cada uno de ellos. Dado un objeto del API, se obtiene su tipo (variable `Clase`) y se busca la regla cuya cabecera especifique dicha clase (variable `Regla`). Esta regla indica la metaclase del metamodelo (variable `Metaclase`) que corresponde a dicha clase del API. De esta forma, se crea una nueva instancia de la metaclase (variable `Instancia`) para representar el objeto que es instancia de la clase del API. Por ejemplo, para extraer un elemento del modelo a partir del objeto `JFrame` en el ejemplo Swing, se localiza la regla cuya cabecera es `JFrameModel : javax.swing.JFrame` y se crea una instancia de la metaclase `JFrameModel`. Nótese que para cada clase del API solamente puede existir una y sólo una regla que defina su correspondencia con una metaclase del metamodelo, la cual puede ser de tipo normal o predefinida.

Una vez que la metaclase ha sido instanciada, el siguiente paso consiste en inicializar sus propiedades (atributos y referencias) por medio de la invocación de los métodos del API que permiten obtener los valores de las propiedades del objeto del API (sentencia `ForEach`). Los métodos a utilizar (variable `Método`) dependen de la información indicada en el correspondiente elemento *statement*, que puede ser de tipo `GET` o `ACCESSOR`, definidos en la sección *Property* de la propiedad (variable `Sección`). Debido a que una regla solamente define las correspondencias para las propiedades declaradas en la metaclase indicada en su cabecera, la inicialización de las propiedades heredadas implica localizar las secciones *Property* de las reglas de las superclases de dicha metaclase. Cada propiedad es inicializada con el valor (variable `Valor`) devuelto por el método aplicado a la instancia de la clase

```
extraer(Objeto) {
  Clase ← Clase API de Objeto
  Regla ← Regla para Clase (localizada en la definición API2MoL)
  // La regla es ejecutada en dos pasos
  // Paso 1. Instanciación de la metaclase
  Metaclase ← Metaclase a ser instanciada (obtenida de la cabecera de Regla)
  Instancia ← Instancia de Metaclase

  // Paso 2. Inicialización de las propiedades de Instancia
  ForEach Propiedad de Metaclase (incluyendo las heredadas)
    Sección ← Sección Property para Propiedad
    Método  ← Método GET a llamar (obtenido de Sección)
    Valor   ← Valor devuelto al llamar a Método
    If el tipo de Propiedad es primitivo
      Asignar Valor a Instancia.Propiedad
    Else
      Asignar el elemento del modelo devuelto por extraer(Valor) a Instancia.Propiedad
}
```

Figura 8.8: Algoritmo del proceso de extracción de API2MoL.

del API. Por ejemplo, para extraer un elemento del modelo a partir de una instancia de `JFrame`, se localizan las reglas que indican en su cabecera las metaclases `JFrameModel`, `WindowModel`, `ContainerModel` y `ComponentModel` para invocar los métodos indicados en sus elementos *statement* de tipo `GET` (p. ej., para las propiedades `x` e `y` de la metaclase `ContainerModel`).

Al inicializar una propiedad de una instancia de metaclase pueden ocurrir dos situaciones dependiendo del tipo de dicha propiedad (sentencia `If`). Si el tipo de la propiedad es primitivo, el valor devuelto por el método es directamente asignado a la propiedad (p. ej., el atributo `title` de la metaclase `Frame`). Sin embargo, si el tipo es una clase del API, el valor devuelto por el método es de nuevo extraído recursivamente (sección `Else` de la sentencia `If`) siguiendo el mismo proceso (p. ej., el atributo `background` al ser de tipo `Color` provocará la ejecución de la regla cuya cabecera es `ColorModel : java.awt.Color`). Por lo tanto, la ejecución de una regla puede provocar el *disparo* de otras reglas de forma parecida a la ejecución de *bindings* en Gra2MoL y ScheMoL. De esta forma, dado un objeto del API, el mecanismo de ejecución de API2MoL también extrae todos los elementos del modelo a partir los objetos que están conectados a dicho objeto, tanto directamente como indirectamente. Es importante destacar que este algoritmo controla no caer en ciclos infinitos por medio del uso de una caché, que no se ha mostrado en el algoritmo por simplicidad.

La Figura 8.9 muestra una parte del modelo extraído dado el objeto de tipo `JFrame` del ejemplo de la aplicación Swing mostrada en la Figura 8.4a. Este modelo conforma al metamodelo Swing mostrado en la Figura 8.4b. Por simplicidad, no se muestran los valores de determinados atributos y referencias. Nótese que la estructura del grafo de objetos es automáticamente extraída a partir de la aplicación Swing junto con los botones y el color de fondo.

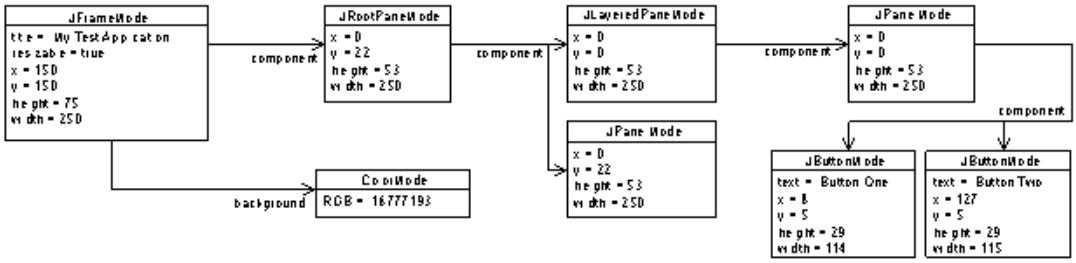


Figura 8.9: Parte del modelo Swing extraído a partir del ejemplo.

8.3.2. El proceso de generación

El procedimiento de generación es muy similar al de extracción. El algoritmo mostrado en la Figura 8.10 describe el comportamiento de API2MoL durante el proceso de generación de una instancia de metaclase identificada como `Instancia`.

En un proceso de generación, la definición API2MoL es utilizada para determinar los objetos del API que deben ser generados a partir de los elementos del modelo que son instancias de las metaclases del metamodelo. De la misma manera que en el proceso de extracción, el proceso de generación arranca con el elemento que actúa como raíz del modelo. Dado un elemento del modelo, en primer lugar se obtiene su metaclase (variable `Metaclase`) y a continuación se localiza la regla (variable `Regla`) que define la correspondencia para dicha metaclase. La cabecera de la regla permite obtener la clase del objeto del API a ser instanciada (variable `Clase`). Este objeto (variable `Objeto`) es creado utilizando el constructor por defecto (p. ej., la instanciación de la clase `Frame`) o, si existe sección *Constructor*, utilizando el constructor definido en dicha sección (p. ej., la instanciación de la clase `Color` utiliza el constructor `Color(RGB)`).

Una vez que el objeto del API se ha creado, sus campos se inicializan a partir de los valores de las propiedades del elemento del modelo. A diferencia del proceso de extracción, los elementos *statement* utilizados en la generación son de tipo `APPEND` o `SET` (también considerados por el elemento *statement* de tipo `ACCESSOR`), dependiendo de si el campo es de tipo colección o no. Por ejemplo, al generar un objeto `JFrame`, el campo `background` será inicializado utilizando el elemento *statement* de tipo `SET`, mientras que el campo `component` se inicializará utilizando el elemento *statement* de tipo `APPEND`. Por otra parte, de la misma manera que en el proceso de extracción, debido a que una regla define solamente las correspondencias para los campos declarados en la clase de la cabecera, la inicialización de los campos heredados provoca la localización de las secciones *Property* o *Multiple* declaradas en las reglas de las superclases de la clase. En primer lugar, se tratan los campos indicados por las secciones *Multiple* y, a continuación, los campos de las secciones *Property*.

Para inicializar los campos indicados por las secciones *Multiple* (paso 2.1 en el algoritmo), se obtiene en primer lugar el elemento *statement* a aplicar (variable `Statement`) y luego se utiliza el método (variable `Método`) indicado por dicho elemento *statement* para inicializar los campos (primera llamada a `generarPropiedad`). Una vez que se han ejecutado las secciones *Multiple* se inicializan el resto de campos (paso 2.2 en el algoritmo) utilizando

```
generar(Instancia) {
  Metaclase  ← Metaclase de Instancia
  Regla      ← La regla de correspondencia para Metaclase
  // La regla es ejecutada en dos pasos
  // Paso 1. Instanciación de la clase
  Clase  ← Clase del API a instanciar
           (obtenida de la cabecera de la regla)
  Objeto ← Instancia creada a partir de Clase
           (utiliza la sección Constructor si existe)

  // Paso 2. Inicialización de las propiedades de la clase
  // Paso 2.1. Secciones Múltiple
  ForEach SecciónMúltiple que involucren a un grupo de campos de Clase
                          (incluyendo secciones Múltiple heredadas)
    ForEach Statement de SecciónMúltiple
      Método ← Método SET/APPEND a llamar (Obtenido de Statement)
      generarPropiedad(Objeto, Método)

  // Paso 2.2. Secciones Property
  ForEach Campo en Clase que no sea parte de una sección Múltiple
                (incluyendo campos heredados)
    Sección ← Sección Property para Campo
    Método  ← Método SET/APPEND a llamar (obtenido de Sección)
    generarPropiedad(Objeto, Metodo)
}

generarPropiedad(Objeto, Método) {
  Llama a Método de Objeto para inicializar los campos del objeto, antes de llamar:
  ForEach Propiedad a ser utilizada como parámetro de Método
    If el tipo de Propiedad es primitivo
      Utilizar el valor de Propiedad como parámetro
    Else
      Utilizar el valor devuelto por generar(valor de Propiedad) como parámetro
}
```

Figura 8.10: Algoritmo del proceso de generación de API2MoL.

las secciones *Property* (variable `Sección`) y el método indicado por el *statement* de dicha sección (segunda llamada a `generarPropiedad`).

Al inicializar un campo (método `generarPropiedad`), el método del elemento *statement* de tipo `SET` o `APPEND` se ejecuta utilizando los valores de las propiedades del elemento del modelo como parámetros. En este punto pueden darse dos situaciones dependiendo del tipo de estas propiedades. Si la propiedad es de tipo primitivo, su valor es directamente utilizado como parámetro al ejecutar el método (p. ej., el campo `title`, que es utilizado en el método de tipo *set*). Por otro lado, si el tipo es una metaclase, se aplica inicialmente la regla cuya cabecera defina la correspondencia para dicha metaclase y, a continuación, el objeto generado por la regla es utilizado como parámetro al ejecutar el método (p. ej., el campo `background`, cuyo tipo es `Color`, provocará la ejecución de la regla cuya cabecera es `Color : java.awt.Color` y a continuación el objeto `Color` creado por la regla se utilizará como parámetro al ejecutar el método indicado por el elemento *statement* de tipo `APPEND`). De esta forma, de la misma manera que en el proceso de extracción, la ejecución de una regla en el proceso de generación puede *disparar* otras reglas, estando también controlada

la recursividad por medio de una caché para evitar ciclos infinitos. Por ejemplo, dada una instancia de la metaclase `JFrameM`, que es la raíz del modelo mostrado en la Figura 8.9, el proceso de generación crearía un grafo de instancias de clases con la misma estructura que la mostrada en la Figura 8.4a.

8.4. El proceso de *bootstrap*: generación automática del metamodelo del API y de las correspondencias

Como se comentó en la sección 8.1.4, hemos definido un proceso de *bootstrap* basado en el propio API2MoL para descubrir la estructura de los elementos del API y generar (casi completamente) tanto el metamodelo del API como la definición API2MoL. Gracias a este proceso de *bootstrap*, el desarrollador simplemente tiene que completar aquellas partes que no han sido descubiertas por este proceso. Mientras que esta sección describe el proceso de *bootstrap*, la siguiente sección se mostrarán algunos resultados empíricos de este proceso acerca de su corrección y cobertura.

El proceso está compuesto por dos fases, que se muestran en la Figura 8.11: (1) las clases del API se representan como un modelo que conforma al metamodelo reflexivo del lenguaje de programación y (2) dicho modelo es transformado por medio de la aplicación de transformaciones *m2m* para generar el metamodelo específico del API así como la correspondiente definición API2MoL. Con este proceso es posible obtener automáticamente estos dos componentes a partir del API siempre que el lenguaje utilizado para desarrollar dicho API soporte reflexión.

La primera fase (fase 1 en la Figura 8.11) utiliza API2MoL para obtener un modelo que represente la información estructural del API de entrada, es decir, los metadatos que representan las clases del API. Este proceso se realiza por medio de la aplicación de un proceso de extracción que utiliza una definición API2MoL entre el API reflexivo y un metamodelo reflexivo creado a mano. Debido a que actualmente API2MoL soporta API Java, hemos desarrollado un metamodelo para el API reflexivo de Java, el cual representa los principales conceptos de dicho API (p. ej., clase, método, atributo, etc), así como la definición API2MoL necesaria para aplicar el proceso de extracción. Las Figuras 8.12a y 8.12b muestran una parte de ambos artefactos, que fueron suficientes para generar un modelo reflexivo de cualquier API Java.

La segunda fase (fase 2 en la Figura 8.11) aplica dos transformaciones *m2m* definidas en ATL, que reciben el modelo reflexivo como entrada y generan (1) el metamodelo del API y (2) la definición API2MoL expresada como un modelo conforme al metamodelo de la sintaxis abstracta de API2MoL. Es importante destacar que para algunos API, la generación podría no ser completa debido que pueden presentar algunas particularidades que deben tratarse de manera especial. Se han identificado un conjunto de correspondencias y heurísticas que cubren la mayoría de las características del API, tal y como se mostrará en la sección 8.5, aunque algunas de ellas puede que no sean descubiertas. En cualquier caso, el pequeño porcentaje de elementos no descubiertos provoca que siempre sea útil aplicar nuestro proceso de *bootstrap* al menos para tener una primera versión. Las transformaciones *m2m* utilizadas en esta fase se desarrollaron tal y como se comenta a continuación.

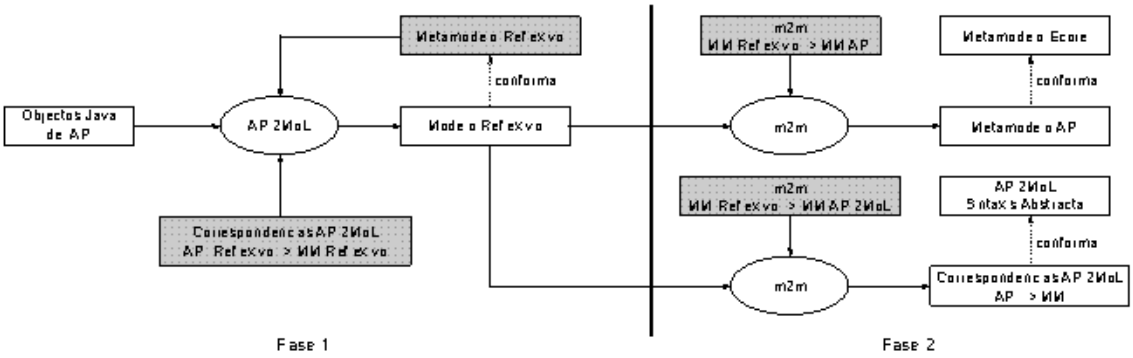


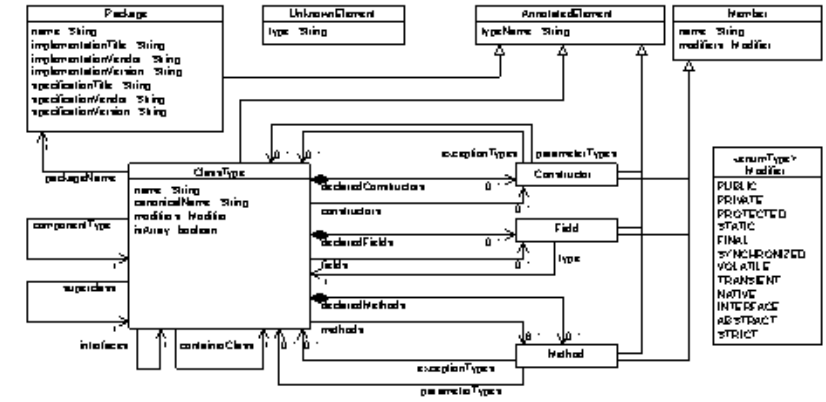
Figura 8.11: Proceso de *bootstrap*. Los elementos representados por cajas grises han sido desarrollados manualmente.

8.4.1. Descubrimiento del metamodelo del API

La transformación utilizada para generar el metamodelo del API transforma cada elemento `ClassType` del modelo reflexivo (Figura 8.12), que representa a una clase del API Java, en una metaclase del metamodelo del API. Estas metaclases tendrán una propiedad por cada campo de la clase del API (referencia `declaredFields`) con accesibilidad pública, es decir, que el campo tenga visibilidad pública (modificador `public`) o que tenga un método de tipo *get*. En particular, cada campo es transformado en un atributo o referencia dependiendo de su tipo. Si el campo es de tipo primitivo, será transformado en un atributo del mismo tipo (p. ej., un campo de tipo entero se transformará en un atributo de tipo entero). Sin embargo, si el tipo es una clase del API (hace referencia a otro elemento `ClassType` del modelo reflexivo), será transformado en una referencia a la metaclase resultado de transformar dicha clase.

Las referencias `interfaces` y `superclass` de los elementos `ClassType` son utilizadas para definir la estructura jerárquica de las clases del API. La primera de ellas representa las interfaces que implementa la clase mientras que la segunda indica su superclase, debido a que Java no permite la herencia múltiple. Sin embargo, debido a que el en el metamodelo sí es posible la herencia múltiple, los elementos de ambas referencias son utilizados como superclases de la metaclase resultante.

La Figura 8.13 ilustra la aplicación de la transformación descrita anteriormente utilizando un ejemplo simple basado en Swing. La Figura 8.13a muestra el modelo reflexivo obtenido al aplicar el proceso de la primera fase a las clases `JButton`, `AbstractButton` y `InSets` del API Swing. En la figura, cada clase del API es representada por un elemento `ClassType`. El elemento `ClassType` cuyo atributo `name` es `AbstractButton` contiene dos campos (`text` y `margin`) y dos métodos (`getText` y `getMargin`). El metamodelo obtenido a partir de la aplicación de la transformación explicada anteriormente se muestra en la Figura 8.13b. Como puede observarse, cada elemento `ClassType` que representa a un clase del API se transforma en una metaclase y los campos de la clase lo hacen en propiedades de la metaclase (p. ej., el campo `text` de la metaclas `AbstractButtonModel`) o en referencias (p. ej., la referencia `margin` de la metaclase `AbstractButtonModel`). Por



(a)

```
ClassType : java.lang.Class {
  name :
    get;
  canonicalName :
    get;
  modifiers :
    get;
  isArray :
    get;
  classLoader :
    get;
  packageName :
    get getPackage;
  constructors :
    get;
  declaredConstructors :
    get;
  ...
}
```

```
Method : java.lang.reflect.Method {
  returnType :
    get;
  parameterTypes :
    get;
  exceptionTypes :
    get;
  ...
}

Member : java.lang.reflect.Member {
  name :
    get;
  modifiers :
    get;
  ...
}
```

```
Package : java.lang.Package {
  name :
    get;
  implementationTitle :
    get;
  implementationVendor :
    get;
  implementationVersion :
    get;
  specificationTitle :
    get;
  specificationVendor :
    get;
  specificationVersion :
    get;
} ...
```

(b)

Figura 8.12: (a) Parte del metamodelo reflexivo utilizado para describir clases de un API Java. (b) Parte de la definición API2MoL para extraer modelos reflexivos a partir de clases de un API Java.

otro lado, la estructura jerárquica también es descubierta (`JButtonModel` es subclase de `AbstractButtonModel`). Igual que anteriormente, las metaclases se nombran con el sufijo `Model` por claridad.

Además de las reglas de transformación anteriores para descubrir el metamodelo del API, se han aplicado dos heurísticas para completar el proceso de descubrimiento del metamodelo del API. La primera de ellas trata con los métodos de tipo *accessor* que se utilizan para acceder a información de la clase del API (p. ej., atributos calculados o derivados). Esta heurística analiza el nombre del método para descubrir nuevas propiedades de la metaclases. Estas propiedades pueden ser también atributos o referencias dependiendo del tipo devuelto por el método. Por ejemplo, si el elemento `ClassType` cuyo atributo `name` es `AbstractButton` contuviera un atributo calculado llamada `perimeter` representado por el método `getPerimeter` cuyo valor devuelto es de tipo entero, la metaclase `AbstractClassModel` creada a partir de dicho `ClassType` contendría un atributo llamado `perimeter`.

La segunda heurística se aplica cuando el campo de la clase del API es de tipo colección. Sin la aplicación de esta heurística, estos campos se transformarían en una propiedad

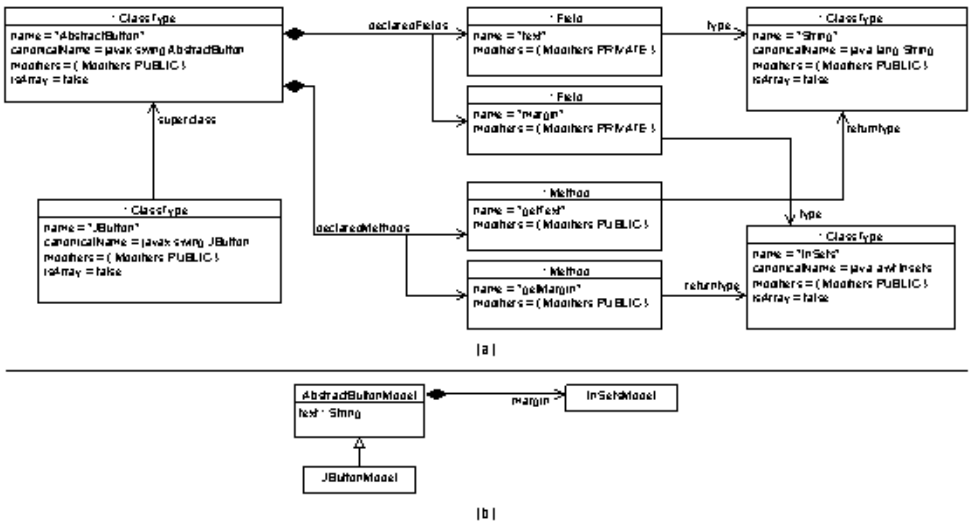


Figura 8.13: (a) Parte del modelo reflexivo para una parte de Swing and (b) metamodelo descubierto a partir de dicho modelo reflexivo.

multivaluada de la metaclase, pero sería necesario descubrir el tipo de elementos del campo colección para generar un atributo o una referencia multivaluada. Por ejemplo, si un elemento `ClassType` contiene un campo cuyo tipo es una lista de elementos `String`, sería transformado en un atributo multivaluado de tipo `String`, mientras que si el tipo del campo es una lista de elementos `ClassType`, sería transformado en una referencia multivaluada cuyo tipo sería la metaclase resultante de transformar el elemento `ClassType`. Para poder aplicar esta heurística se utiliza la información de genericidad ofrecida por el lenguaje. En aquellos casos en los que no sea posible utilizar dicha información, API2MoL no podría descubrir el tipo de las colecciones.

8.4.2. Descubrimiento de la definición API2MoL

La transformación utilizada para generar la definición API2MoL crea una regla para cada elemento `ClassType` y la correspondiente metaclase generada en el metamodelo del API. Esta regla incluye las secciones *Property* necesarias para extraer/generar los campos de la clase del API. Además, cada sección *Property* también incluye los elementos *statement* necesarios, según los métodos ofrecidos por el API. Por el momento, esta transformación solo trata con los elementos *statement* de tipo `SET`, `GET`, `DIRECT` y `APPEND`, los cuales son incluidos siguiendo las siguientes heurísticas:

- Un elemento *statement* de tipo `SET` se añade si existe el método de tipo *set* para el campo de la clase del API.
- Un elemento *statement* de tipo `GET` se añade si existe el método de tipo *get* para el campo de la clase del API.
- Un elemento *statement* de tipo `APPEND` se añade si el campo de la clase del API es

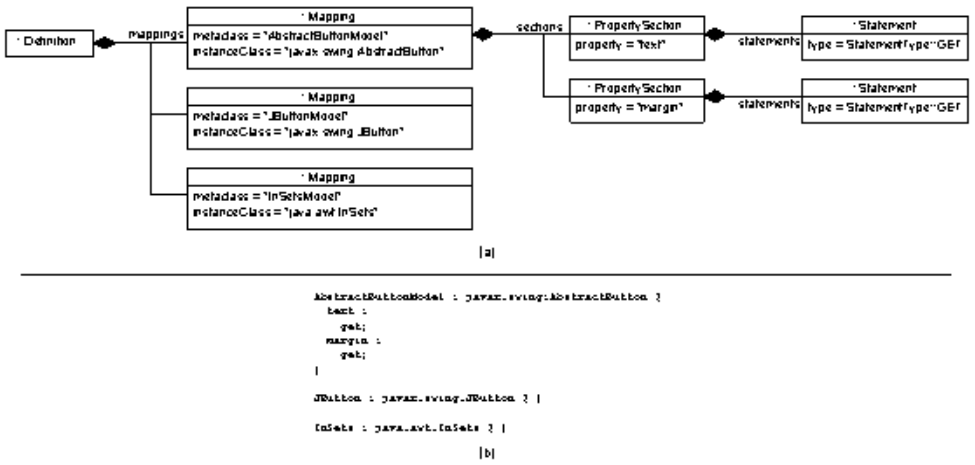


Figura 8.14: Un extracto de la definición API2MoL para el ejemplo de Swing. (a) El modelo conforme al metamodelo de sintaxis abstracta de API2MoL y (b) la sintaxis concreta correspondiente.

de tipo colección y existe el método de tipo *add* para dicho campo.

- Un elemento *statement* de tipo `DIRECT` se añade si la visibilidad del campo de la clase del API es pública, es decir, se puede acceder directamente.

La Figura 8.14 presenta la definición API2MoL para el modelo reflexivo mostrado en la Figura 8.13a. La Figura 8.14a muesta el modelo conforme al metamodelo de sintaxis abstracta de API2MoL, mientras que la Figura 8.14b presenta su correspondiente sintaxis concreta. Esta definición incluye las reglas de correspondencias para los elementos considerados en el ejemplo del modelo reflexivo y el metamodelo del API descubierto en la sección anterior. Como puede observarse, la definición API2MoL incluye tres reglas, una para cada par formado por la clase del API y la correspondiente metaclase del metamodelo descubierto. Además, la regla para la metaclase `AbstractButtonModel` contiene las secciones *Property* para los atributos `text` y `margin`. En este caso, dado que la clase del API solo ofrece el método de tipo *get* para acceder a dichos atributos, las secciones *Property* solamente incluyen elementos *statement* de tipo `GET`.

8.5. Validación

API2MoL ha sido validado conforme se ha ido desarrollando. En primer lugar, desarrollamos el lenguaje de correspondencias de API2MoL y verificamos la corrección de los procesos de extracción y generación. A continuación, implementamos el proceso de *bootstrap* y comprobamos el nivel de cobertura del lenguaje y del proceso de *bootstrap* para tres API Java: Swing, *Standard Widget Toolkit* (SWT) [78] y un API Java para Twitter denominada JTwitter [71]. La Figura 8.15 muestra los pasos seguidos durante el desarrollo y validación de API2MoL. En esta sección describiremos cómo se validó la corrección y el nivel de cobertura de nuestra aproximación.

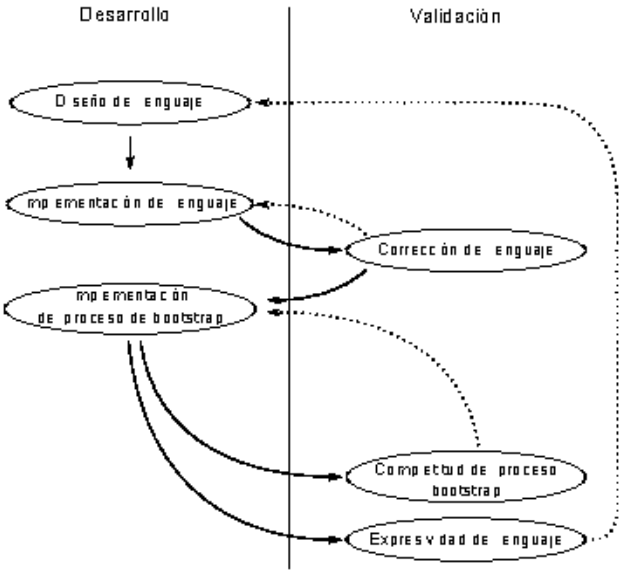


Figura 8.15: Tareas de desarrollo y validación de API2MoL (las líneas punteadas indican el proceso que se siguió al encontrar errores).

Dado un conjunto de objetos de un API de una aplicación (p. ej., el conjunto de objetos de la interfaz gráfica Swing de una aplicación), la estrategia seguida para comprobar la corrección de los procesos de extracción y generación consistió en: (1) aplicar el proceso de extracción a un conjunto de objetos del API para obtener un modelo inicial; (2) aplicar el proceso de generación al modelo inicial para crear un nuevo conjunto de objetos del API; y (3) volver a extraer un modelo a partir los objetos del API generados del modelo inicial, obteniendo de nuevo un modelo que los representa. Como resultado de este proceso, se obtienen dos modelos que representan al mismo conjunto de objetos del API de forma que, si los procesos de extracción y generación son correctos, estos modelos deberían ser idénticos. Para realizar la comparación de estos modelos utilizamos la herramienta EMFCompare [10]. La Figura 8.16 ilustra el proceso. Este proceso fue aplicado satisfactoriamente con un conjunto de aplicaciones Swing de prueba especialmente diseñadas para tratar con cada sección y elemento *statement* del lenguaje (descargables desde `http://modelum.es/api2mol`).

En la segunda fase de validación, tal y como se ha explicado anteriormente, el objetivo es determinar el nivel de cobertura del lenguaje API2MoL así como del proceso de *bootstrap*. Para realizar esta validación se utilizaron las API Swing, SWT y JTwitter. Es importante destacar que el significado de *nivel de cobertura* es diferente para el lenguaje y el proceso de *bootstrap*. Al validar el lenguaje, el nivel de cobertura hace referencia al hecho de que sea suficientemente expresivo como para cubrir todas las posibles correspondencias entre las clases del API y los elementos del metamodelo del API, mientras que en el caso del proceso de *bootstrap*, hace referencia a si el metamodelo del API y la definición API2MoL descubiertos en dicho proceso son completos. A continuación se describirán los resultados

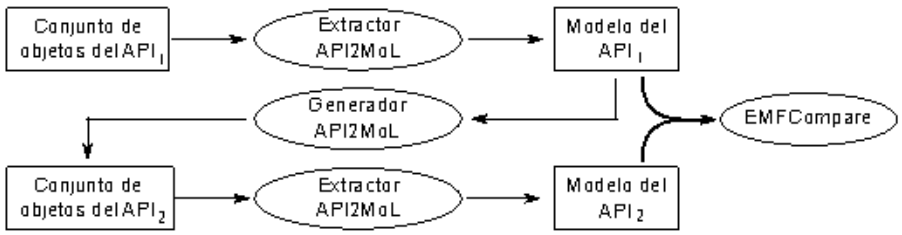


Figura 8.16: Proceso aplicado para verificar la corrección de los procesos de extracción y generación.

de esta segunda fase de validación para cada una de las API consideradas y finalmente se mostrará un resumen del nivel de cobertura obtenido.

API Swing Debido a que el API Swing fue utilizado para validar la corrección del lenguaje, se utilizaron las mismas aplicaciones para comprobar el nivel de cobertura. La validación del lenguaje API2MoL demostró que no existió ningún caso donde el lenguaje no fuera lo suficientemente expresivo.

Una vez la cobertura del lenguaje fue validada, se aplicó el proceso de *bootstrap* para obtener automáticamente el metamodelo Swing y la definición API2MoL para dicho API. Los procesos de extracción y generación fueron de nuevo ejecutados pero utilizando los artefactos del proceso de *bootstrap* y sus resultados fueron comparados con los resultados obtenidos al aplicar el proceso sin *bootstrap*. La comparación del metamodelo del API, así como de la definición de correspondecias se realizó manualmente para detectar posibles partes que faltaran en los elementos resultado del proceso *bootstrap*.

Aunque las definiciones API2MoL fueron suficientes para la mayoría de las aplicaciones simples en Swing, fue necesarrio extenderlas para tratar con los constructores específicos de las clases del API. En Swing, algunos objetos (p. ej., `BevelBorder` o `EmptyBorder`) se crean por medio de constructores que reciben valores de inicialización para los cuales no existe un método de tipo *set*. En estos casos, tuvieron que añadirse manualmente secciones *Constructor* a algunas de las reglas generadas por el proceso de *bootstrap*.

Debido al tamaño del metamodelo Swing y de la definición API2MoL, que incluyen más de mil elementos, no se incluyen más ejemplos particulares de este API. Dichos artefactos pueden ser descargados desde `http://modelum.es/api2mol`.

API SWT SWT es también un API que permite a los desarrolladores crear interfaces gráficas de usuario para aplicaciones Java. Mientras que Swing está integrado en la JDK y es completamente portable, SWT tiene la ventaja que haber sido implementada como una aplicación nativa, mejorando el rendimiento y la compatibilidad. Con respecto a la estructura del API, SWT se diferencia en aspectos como la creación y gestión de los *widgets*.

La aplicación utilizada para validar API2MoL con SWT es el ejemplo `ControlExample`, el cual usa todos los tipos de *widgets* y *layouts* que ofrece el API. Al validar el nivel de cobertura con este API se encontraron dos problemas principales: el descubrimiento

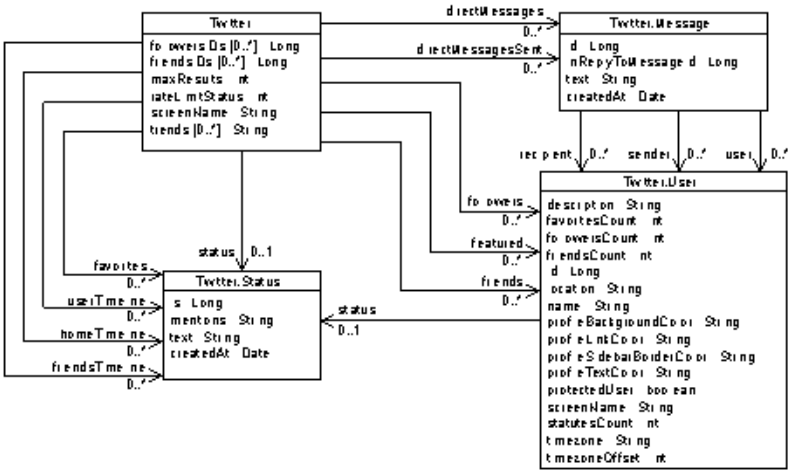


Figura 8.17: Parte del metamodelo descubierto para el API JTwitter.

de (1) atributos públicos y de (2) constructores específicos para los objetos del API. El primero está relacionado con la estructura de elementos de tipo *layout* utilizada por SWT, que define un conjunto de atributos de acceso público para configurar dichos elementos, es decir, no existen métodos de tipo *get* ni *set*. Por lo tanto, el lenguaje tuvo que ser extendido para contemplar un nuevo elemento *statement* de tipo `DIRECT` para soportar la definición de correspondencias con atributos de acceso público de las clases Java. El proceso de *bootstrap* también fue modificado para ser capaz de descubrir este nuevo tipo de elemento *statement* y generarlo automáticamente en las definiciones API2MoL.

El segundo problema afecta principalmente a los elementos *widget*, cuyos constructores reciben normalmente el elemento contenedor de dicho elemento. La solución adoptada es muy parecida a la adoptada en el caso de Swing, añadiéndose manualmente las secciones *Constructor* necesarias en las reglas de algunos elementos. En `http://modelum.es/api2mol` pueden encontrarse el metamodelo del API así como la definición API2MoL.

API JTwitter Finalmente, el lenguaje y el proceso de *bootstrap* fueron validados con el API JTwitter, que es un API *open-source* desarrollada en Java para acceder a los servicios de la red social Twitter. JTwitter permite gestionar la información de las cuentas de usuario, sus seguidores (o *followers*), amigos y estados (o *tweets*). Este tercer ejemplo también permite mostrar la utilidad de API2MoL para modelar el dominio de la Web 2.0 y servir de pivote para representar la información de la actividad de un usuario en esta red social, la cual podría ser utilizada para actualizar la información de otras redes como LinkedIn.

Tanto la validación del lenguaje como el proceso de *bootstrap* fueron aplicados satisfactoriamente al API JTwitter utilizando una cuenta Twitter de prueba llamada `api2moltest`, de esta forma, tanto el metamodelo del API como la definición API2MoL fueron obtenidas automáticamente. La Figura 8.17 muestra una parte del metamodelo descubierto por el proceso de *bootstrap*, el cual no fue necesario modificar ya que fue generado correcta-

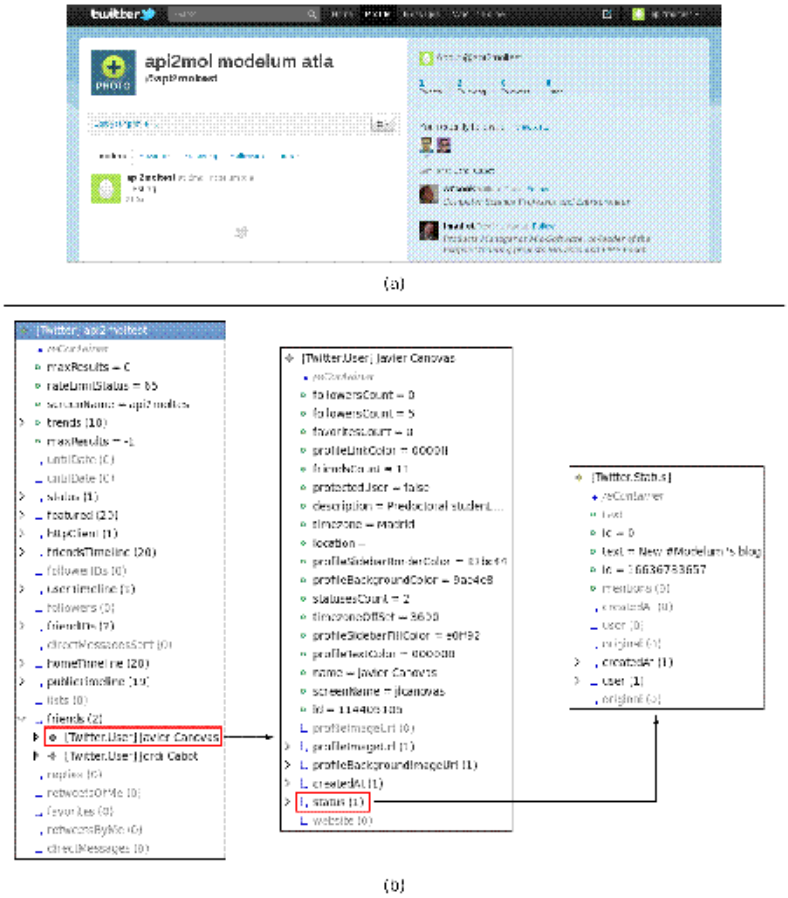


Figura 8.18: Proceso de extracción aplicado a una cuenta Twitter de prueba utilizando el API JTwitter. (a) Captura de la cuenta Twitter de prueba. (b) Parte del modelo extraído.

mente. Las metaclases mostradas corresponden a los conceptos principales del dominio de Twitter. Así, la metaclase `Twitter` representa una cuenta en Twitter y contiene información del usuario. Esta metaclase hace referencia a la información de estado (último *tweet*) por medio de la referencia `status` cuyo tipo es `Twitter.Status`, los usuarios que son *followers* están agrupados en la referencia `friends` cuyo tipo es `Twitter.User` y los mensajes privados del usuario están agrupados en la referencia `directMessages` y representados por la metaclase `Twitter.Message`.

La definición API2MoL generada por el proceso de *bootstrap* contiene las reglas y elementos *statement* necesarios para realizar tanto el proceso de extracción como el de generación, por lo que no fue necesario realizar ningún cambio.

Tanto el metamodelo descubierto como la definición API2MoL fueron utilizados para extraer un modelo que describe la cuenta Twitter utilizada de prueba. La Figura 8.18a muestra la cuenta de prueba utilizada, mientras que la Figura 8.18b presenta una parte del

	Definición API2MoL			Metamodelo del API		
API	***Bootstrap***	**Real**	**Dif.**	***Bootstrap***	**Real**	**Dif.**
Swing	1116 reglas 1053 secciones	1116 reglas 1240 secciones	100 % 84,9 %	1197 clases 446 atributos 486 referencias	1197 clases 446 atributos 486 referencias	100 % 100 % 100 %
SWT	689 reglas 798 secciones	689 reglas 798 secciones	100 % 72,8 %	708 clases 492 atributos 251 referencias	708 clases 492 atributos 251 referencias	100 % 100 % 100 %
JTwitter	53 reglas 160 secciones	53 reglas 160 secciones	100 % 100 %	71 clases 76 atributos 45 referencias	71 clases 76 atributos 45 referencias	100 % 100 % 100 %

Tabla 8.2: Cobertura del proceso de *bootstrap* de API2MoL. Comparativa de la definición de API2MoL y del metamodelo del API obtenidos aplicando el proceso de *bootstrap* con la definición de API2MoL y el metamodelo reales para varias API.

modelo extraído, el cual contiene la instancia `Twitter`, que hace referencia a la instancia `Twitter.User` la cual representa a uno de los amigos de la cuenta de usuario utilizada. Esta instancia de la metaclase `Twitter.User` a su vez contiene una instancia de la metaclase `Twitter.Status`, que representa el último *tweet* de dicho amigo.

Resumen del nivel de cobertura del lenguaje El lenguaje de correspondencias incorporado en API2MoL demostró cubrir la gran mayoría de las correspondencias en las pruebas exceptuando el elemento *statement* de tipo `DIRECT` para el caso del API SWT. Una vez este tipo de elemento *statement* fue incorporado, el nivel de cobertura del lenguaje fue del 100 % para las API utilizadas en el proceso de validación, ya que fue lo suficientemente expresivo para tratar con todas las correspondencias necesarias.

Resumen del nivel de cobertura del proceso de *bootstrap* La Tabla 8.2 muestra el nivel de cobertura obtenido en el proceso de *bootstrap*. Las pruebas fueron realizadas teniendo en cuenta el elemento *statement* de tipo `DIRECT` en el lenguaje. Para cada API se compara el número de elementos generados por el proceso de *bootstrap* con el número real de elementos. En el caso de la definición API2MoL se han considerado el número de reglas y de secciones, mientras que en el caso de los metamodelos del API se ha hecho con el número de clases, atributos y referencias. El número de elementos que no se encuentran en la versión descubierta por el proceso de *bootstrap* se ha contabilizado manualmente.

Como puede observarse, el proceso de generación del metamodelo es completo para las tres API consideradas. Esto es principalmente debido al hecho de que las reglas y heurísticas aplicadas en el proceso de *bootstrap* cubren la mayoría de las particularidades de los API. Por otro lado, la generación de la definición de correspondendias también descubrió un gran número de reglas y secciones. Sin embargo, para algunas API fue necesario añadir secciones de tipo *Constructor* y *Multiple* que fueron específicas para algunos objetos del API, tal y como se ha explicado anteriormente. De acuerdo a la tabla, un 15.1 % de las secciones de Swing tuvieron que ser añadidas manualmente mientras que en el caso del API SWT fue un 27.2 %.

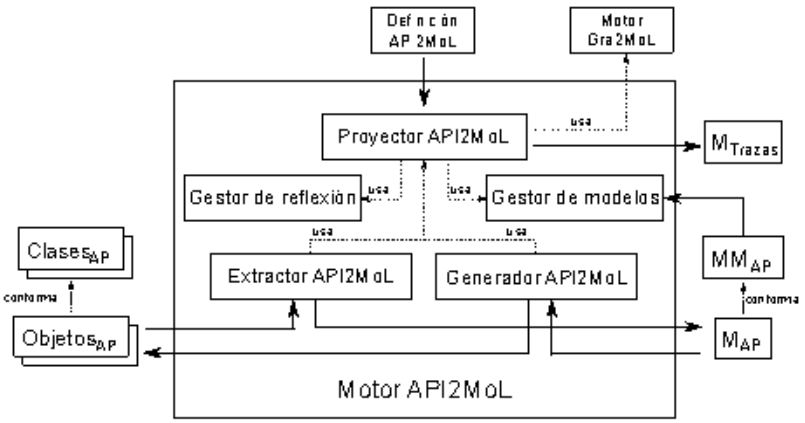


Figura 8.19: Arquitectura del motor API2MoL.

8.6. Implementación

El lenguaje de definición de correspondencias, los procesos de extracción y generación así como el proceso de *bootstrap* han sido desarrollados como *plugins* en la plataforma Eclipse. La Figura 8.19 muestra la arquitectura del motor API2MoL. Los componentes principales son:

- El *extractor*, que se encarga de realizar el proceso de extracción.
- El *generador*, que aplica los procesos de generación.
- El *proyector*, que ofrece servicios comunes tanto al *extractor* como al *generador*. Su tarea principal es crear modelos a partir de la definición de correspondencias, los cuales conforman con el metamodelo de sintaxis abstracta de API2MoL. También utiliza al *gestor de modelos* y al *gestor de reflexión*.
- El *gestor de modelos* permite tratar con modelos genéricamente (p. ej., para crear o inicializar las propiedades de una instancia de cualquier metaclase).
- Finalmente, el *gestor de reflexión* ofrece servicios comunes para gestionar las llamadas a los métodos del API.

Tal y como se muestra en la Figura 8.19, a partir de una definición textual de API2MoL, el *proyector* se encarga de obtener un modelo que conforma con el metamodelo de sintaxis abstracta de API2MoL. Al igual que para el caso de SchcMoL, utilizamos Gra2MoL para realizar este paso y obtener modelos conformes al metamodelo de sintaxis abstracta de API2MoL a partir de las definiciones textuales conformes a la gramática del lenguaje. El modelo conforme al metamodelo de sintaxis abstracta es utilizado posteriormente tanto por el elemento *extractor* para convertir objetos del API en modelos así como por el *generador* para convertir modelos en objetos del API. De esta forma, el elemento *proyector* actua de *front-end* para los elementos *extractor* y *generador*.

Los elementos *gestor de modelos* y *gestor de reflexión* son componentes auxiliares que ofrecen servicios relacionados con la gestion de modelos y con el uso de la reflexión, res-

pectivamente. Dado que el *gestor de modelos* es un componente común a los lenguajes anteriores, en API2MoL también fue reutilizado. Por otro lado, el *gestor de reflexión* permite acceder transparentemente al API reflexiva del lenguaje de programación.

8.7. Trabajo relacionado

Desde nuestro conocimiento, API2MoL es la primera aproximación genérica para la construcción de puentes *apiware-modelware*. Una aproximación parecida se describe en [3, 5], donde se presenta una metodología para crear lenguajes específicos de *frameworks* (*Framework-Specific Modeling Languages*, FSML). Un FSML permite a los desarrolladores representar los conceptos específicos del dominio ofrecidos por el API de un *framework*. Así, la sintaxis abstracta de un FSML es similar al metamodelo descubierto por API2MoL. Sin embargo, a diferencia de API2MoL, la sintaxis abstracta así como el extractor y generador asociados deben ser desarrollados manualmente.

Otras herramientas parecidas a API2MoL son aquellas que: i) extraer ontologías a partir del código fuente [83], ii) extraer modelos a partir del código fuente, como MoDisco [11] y Gra2MoL, o iii) utilizan modelos *at runtime* para crear y manipular dinámicamente los artefactos de una aplicación, tales como los elementos de la interfaz gráfica en Wazaabi [82] o los elementos de gestión de sistemas en SM@RT [94]. A continuación compararemos nuestra aproximación con cada uno de estas alternativas.

La aproximación presentada en [83] permite obtener una ontología común para un conjunto de API que comparten el mismo dominio de aplicación. Estas ontologías pueden ser utilizadas en tareas de ingeniería inversa tales como comprensión de programas o procesos de calidad. Sin embargo, esta aproximación no está destinada a construir puentes *apiware-modelware* ya que no permite definir procesos de extracción y generación.

Tal y como se comentó en la sección 5.2.1, MoDisco es un *framework* extensible para aplicar tareas de ingeniería inversa dirigida por modelos que define el concepto de *discoverer*. Actualmente existen *discoverers* para XML y Java que utilizan las API de Eclipse correspondientes para tratar con estos tipos de ficheros. Sin embargo, los *discoverers* de MoDisco ignoran las interacciones del API al aplicar el proceso de extracción y además solamente pueden aplicarse cuando el código fuente está disponible. De esta forma API2MoL podría ser integrado en MoDisco para ofrecer *discoverers* más completos que sea capaces de obtener modelos donde se consideren las diferentes API utilizadas en el sistema.

Por otro lado, GraMoL (y aproximaciones relacionadas como TCS) puede ser utilizado para el desarrollo de *discoverers* en MoDisco. Sin embargo, de la misma manera que MoDisco, Gra2MoL no ofrece soporte para API, por lo que API2MoL podría ser también utilizado para enriquecer el proceso de obtención de modelos con Gra2MoL.

Wazaabi es una herramienta de construcción de interfaces gráficas para las tecnologías SWT, JSF y Swing. Los desarrolladores definen los modelos de la interfaz gráfica que luego son procesados por un motor para generar los correspondientes objetos de interfaz. A diferencia de API2MoL, Wazaabi solamente es útil para procesos de ingeniería directa que traten con estas API, es decir, generar la interfaz gráfica a partir de los modelos, pero no el proceso contrario. Además, API2MoL añade un nivel extra de automatización ya que

las herramientas necesarias para obtener los modelos son generadas automáticamente a partir de la definición de correspondencias. De esta forma, la parte de Wazaabi que genera los objetos de la interfaz gráfica podría ser reemplazada por un proceso de generación basado en API2MoL.

La aproximación SM@RT permite generar motores de sincronización entre un sistema en ejecución y su modelo. La entrada del proceso de generación son el metamodelo del sistema y el modelo de acceso que describe como llevar a cabo dicha sincronización utilizando el API de gestión del sistema (p. ej., el API JMX [92]). En [93] se describe una aproximación para inferir automáticamente el metamodelo del sistema analizando el código que utiliza el API JMX. Por otro lado, el modelo de acceso es realmente la definición de las correspondencias entre el metamodelo y los elementos del API pero, a diferencia de API2MoL, son definidas imperativamente por medio de plantillas de código. De esta forma, se define una plantilla para cada elemento del metamodelo que especifica el código necesario para manipular dicho elemento en el API. Nótese que los API pueden tener un gran número de elementos, lo que requeriría un considerable esfuerzo para definir estas plantillas de código. Sin embargo, API2MoL ofrece una aproximación más automatizada que permite generar tanto el metamodelo del API como la definición de correspondencias casi completamente. Además, los cambios en estos elementos se ven facilitados por el uso de un lenguaje declarativo.

Fuera del *modelware*, se han propuesto otras aproximaciones para tratar manipular API como por ejemplo en procesos de reingeniería, donde es necesario migrar el código que utiliza un API particular para que utilice otra. Esta adaptación es normalmente llevada a cabo por medio de *wrappers*, tal y como se describe en [102, 103]. API2MoL podría ser utilizado para esta finalidad también, facilitando la migración del uso de API por medio de la automatización de la interacciones con el API. En primer lugar se podrían representar estas interacciones como modelos para, a continuación, definir la transformación entre las API a nivel de modelos.

8.8. Escenarios de aplicación

Mientras que Gra2MoL y ScheMoL están principalmente enfocados a su utilización en modernización de software, API2MoL ofrece una integración completa entre un API y el DSDM. Al ofrecer un puente bidireccional entre el *apiware* y el *modelware*, pueden identificarse un gran número de escenarios de aplicación que necesiten aplicar un proceso de extracción, de generación o ambos, algunos de los cuales son:

- *Agregación de datos web para la integración de aplicaciones.* Por ejemplo, modelos que representen el estado de una cuenta Twitter podrían ser transformados en modelos que representen la plataforma LinkedIn para actualizar automáticamente la cuenta LinkedIn con el último *tweet*. En este caso, los modelos actuan de pivote para la interoperabilidad de las aplicaciones.
- *Análisis de datos.* La herramienta Portolan [74] extrae modelos de servidores virtuales ofrecidos por un proveedores de computación *en la nube* utilizando su API de gestión. Una vez se disponen de estos modelos, las técnicas del DSDM podrían utilizarse para

implementar algoritmos que permitieran mostrar de una forma comprensible y gráfica la forma de optimizar el rendimiento del sistema a los administradores del sistema.

- *Construcción de interfaces gráficas.* Por ejemplo, los desarrolladores podrían definir un modelo conforme a un metamodelo de un API de interfaz gráfica (p. ej., SWT o *Google Web Toolkit*, GWT) a partir del cual generar automáticamente los objetos de la interfaz, tal y como hace Wazaabi [82]. De esta forma, los elementos del modelo de interfaz gráfica actuarían como elementos *proxy* que se encargarían de crear y manipular los objetos de la interfaz de la aplicación. Este escenario permite desarrollar soluciones más mantenibles y adaptables que las soluciones tradicionales basadas en código generado estáticamente.
- *Gestión de modelos en* runtime. El proceso de generar modelos puede se combinado con el proceso de extracción para gestionar modelos en *runtime*. En [94] se muestra una herramienta donde los modelos en *runtime* son utilizados para representar la información de ejecución de un sistema, los cuales se mantienen sincronizados conforme el sistema se ejecuta.
- *Reingeniería* al vuelo. Los objetos del API pueden ser convertidos en modelos como una representación intermedia utilizada para obtener objetos para un API diferente. Este proceso permitiría aplicar tareas de reingeniería *al vuelo*, donde solamente sería necesario tratar con los objetos y los modelos, sin tener que analizar el código. Por ejemplo, los objetos de la interfaz gráfica Swing de una aplicación podrían ser dinámicamente convertidos en objetos de los API SWT o GWT.
- *Evolución de API.* La posibilidad de obtener automáticamente el metamodelo que representa al API permite utilizarlo para comprobar cómo evoluciona un mismo API, comparando los metamodelos de cada versión del API.
- *Comparación de API.* De forma parecida al caso anterior, el proceso de *bootstrap* podría aplicarse a API de un mismo servicio (p. ej., Twitter4J, JTwitter y javatwitter) para identificar patrones y poder construir un metamodelo común.

8.9. Características del lenguaje

La Figura 8.20 muestra las características principales de API2MoL de acuerdo al diagrama de características presentado en [20]. API2MoL es un lenguaje bidireccional que trata con los dominios del *apiware* y del *modelware*. Una transformación API2MoL está compuesta por un conjunto de reglas bidireccionales que definen las correspondencias entre las clases del API y las metaclases. Las reglas se resuelven implícitamente y de forma determinista de forma parecida al uso de *bindings*. La ejecución de un proceso de extracción así como de un proceso de generación crea siempre un modelo nuevo o un conjunto de objetos del API, respectivamente. Además, la estructura de las reglas está divida en diferentes secciones (*Default*, *Property*, *Multiple* y *Constructor*), las cuales permiten clasificar las correspondencias definidas. En cuanto a la información de trazabilidad, el motor de ejecución de ScheMoL crea un modelo de trazas automáticamente.

8.10. Conclusiones

En este capítulo se ha descrito el lenguaje API2MoL, un DSL para la integrar API orientadas a objetos en aplicaciones basadas en modelos. El lenguaje permite aplicar tanto un proceso de extracción de modelos a partir de los objetos del API así como generar dichos objetos a partir del modelo. Además, API2MoL incorpora un proceso de *bootstrap* que permite generar automáticamente la definición de correspondencias entre el API y el metamodelo así como el metamodelo del API. El lenguaje y el proceso de *bootstrap* han sido validados, verificando su corrección y cobertura. En el sitio web `http://modelum.es/api2mol` pueden encontrarse todos los recursos que componen la herramienta.

A continuación, al igual que se ha realizado para los lenguajes anteriores, se presentará la adecuación de API2MoL a los requisitos identificados en [47] así como el nivel de cumplimiento de cada uno de ellos, indicado con los valores 1 a 5 con la misma semántica que en las secciones 5.11 y 7.10.

Conformidad (5). Todos los elementos utilizados en API2MoL usan conceptos del dominio del problema de establecer puentes bidireccionales entre el *apiware* y el *modelware* necesarios para la integración de las API en el DSDM. Las reglas de API2MoL permiten definir las correspondencias entre las clases del API y las metaclases, así como los métodos que deben ser llamados para extraer/generar una propiedad del elemento del metamodelo.

Ortogonalidad (4). El lenguaje API2MoL incluye dos aspectos que afectan negativamente a la ortogonalidad. En primer lugar, para indicar los métodos de tipo *get* y *set* asociados a una propiedad del metamodelo, es posible utilizar los *statement* de tipo `GET` y `SET` pero también el *statement* de tipo `ACCESSOR`. En segundo lugar, el soporte a la sobrecarga de métodos ofrecida por API2MoL permite definir los métodos de dos formas (indicando o no el tipo del parámetro).

Soporte (1). Actualmente, API2MoL solamente ofrece las herramientas necesarias para su ejecución programática desde Java, aunque actualmente está siendo adaptado para ser incorporado como *plugin* de la plataforma Eclipse.

Integración (1). El lenguaje ha sido desarrollado en Java y al igual que ScheMoL, no ofrece mecanismos específicos para su integración con otros lenguajes pero se ha identificado la creacioÌĄn de un *plugin* para la plataforma Eclipse como trabajo futuro.

Longevidad (4). La extracción de modelos a partir de los objetos del API así como la generación de dichos objetos a partir de modelos son tareas necesarias para lograr una completa integración de las API en las aplicaciones y procesos basadas en modelos, favoreciendo la longevidad del lenguaje.

Extensibilidad (1). API2MoL no ofrece todavía ningún mecanismo para extender el lenguaje, aunque actualmente se está estudiando la incorporación de un mecanismos para introducir nuevos tipos de *statement*.

Simplicidad (5). El proceso de definición de una transformación en API2MoL consiste en: (1) definir las reglas e (2) indicar los métodos a utilizar para interactuar con el API.

A diferencia de Gra2MoL o ScheMoL, en API2MoL no se utiliza un lenguaje de consultas. Además, el proceso de *bootstrap* simplifica el proceso de construcción de puentes con API2MoL ya que genera automáticamente la definición API2MoL y el metamodelo de un API determinado, liberando al desarrollador de su implementación. Estas tareas no tienen un grado de complejidad elevado y el tiempo necesario para realizarlas solamente depende del tamaño y complejidad de los elementos (metamodelo y API) involucrados en la transformación.

Calidad (4). API2MoL es un lenguaje que se ejecuta de forma reflexiva en Java, por lo que este requisito no puede aplicarse al código generado. Sin embargo, el proceso de *bootstrap* definido en API2MoL sí genera automáticamente los artefactos necesarios para la construcción de un puente para un API determinado. En este caso, los artefactos son generados siguiendo una reglas y heurísticas que aseguran su corrección y calidad, tal y como se ha validado en la sección 8.5. Además, el nivel automatización que ofrece el lenguaje permite mejorar la calidad de aquellas partes de la aplicación que tratan con el API.

Escalabilidad (2). El único aspecto a considerar en cuanto a la escalabilidad es el tratamiento con modelos grandes que pueden resultar de una extracción del API. En este caso, y al igual que para los lenguajes anteriores, se debería estudiar mecanismos que permitan gestionar estos modelos eficientemente, como un repositorio de modelos.

Usabilidad (4). API2MoL tiene una estructura diferente a Gra2MoL y ScheMoL. Sin embargo, el lenguaje no utiliza construcciones complejas para la definición de las correspondencias que puedan decrementar el nivel de usabilidad, de hecho, su estructura es más simple, ya que ni siquiera utiliza un lenguaje de consultas. Por este motivo, creemos que la usabilidad del lenguaje es de un nivel de complejidad cercano a los desarrolladores de DSDM.

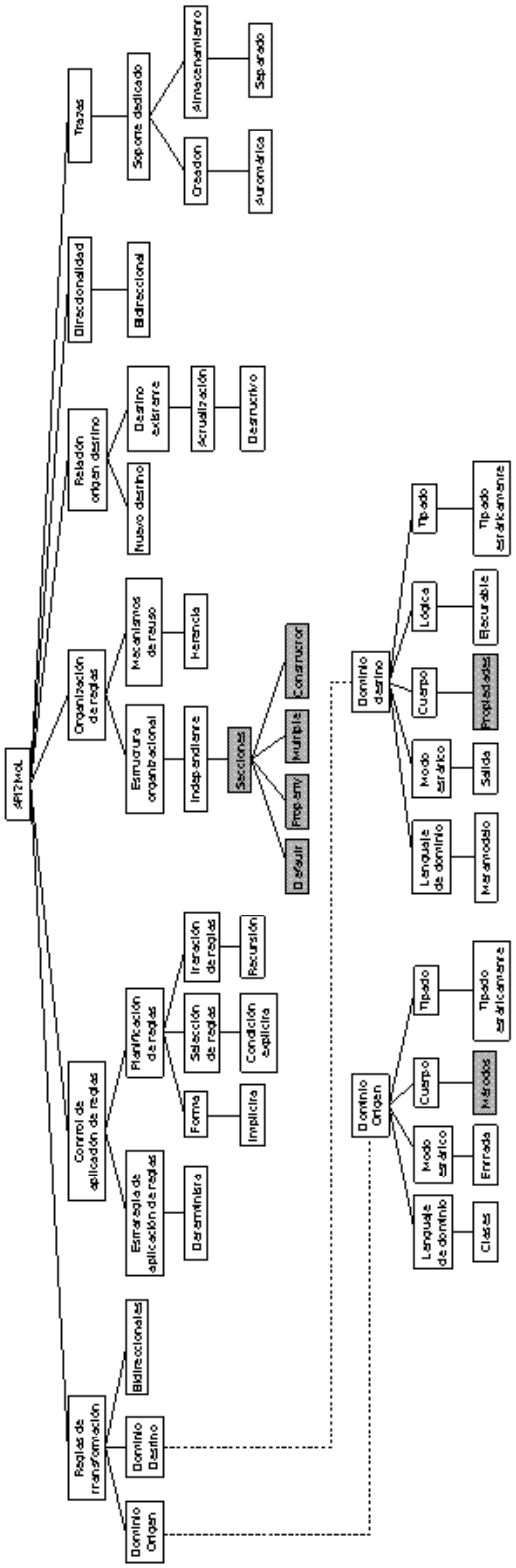


Figura 8.20: Diagrama de características de API2MoL de acuerdo a [20]. Se marcan en gris las características propias del lenguaje.

9

Conclusiones y trabajo futuro

El Alcaudón se tambaleó, cayó de espaldas como un crustáceo de acero.
¡Había caído!
La caída de Hyperion, Dan Simmons.

Este capítulo presenta primero las conclusiones de esta tesis y luego las líneas de trabajo futuro identificadas. Las conclusiones describen las principales lecciones aprendidas durante el desarrollo de Gra2MoL, ScheMoL y API2MoL así como en la experimentación con ADM. Hemos identificado tres líneas principales de trabajo futuro, una para cada DSL desarrollado en esta tesis, pero también dos líneas más, relacionadas con la creación de un nuevo DSL general para la definición de cualquier puente hacia el *modelware* y con la continuación del estudio de la iniciativa ADM. Finalmente, se presenta el trabajo desarrollado a lo largo del trabajo de investigación.

9.1. Conclusiones

La capacidad de elevar el nivel de abstracción al utilizar DSL junto con la automatización ofrecida por las transformaciones de modelos ha demostrado la utilidad del DSDM no sólo en tareas de procesos de ingeniería directa y reingeniería o modernización, sino también en aquellas aplicaciones que utilizan modelos para su funcionamiento, ya sea importando/exportando modelos o usando modelos *runtime*, como las herramientas de cálculo de métricas en modelos, repositorios de modelos o el nuevo Eclipse *e4* [25]. Mientras que los desarrolladores de soluciones DSDM disponen de lenguajes de transformación *m2m* y *m2t*, en la actualidad no existen lenguajes para aplicar transformaciones *t2m*, las cuales permiten extraer modelos a partir de los artefactos que compone un sistema software. Este tipo de transformaciones son especialmente importantes en procesos de modernización basados en modelos, donde existe un paso inicial de extracción de modelos a partir del sistema software a modernizar. En este sentido, esta tesis presenta los lenguajes Gra2MoL

y ScheMoL, los cuales permiten extraer modelos desde texto conforme a una gramática y datos conformes al esquema de una base de datos relacional, respectivamente. Además, el uso de transformaciones *t2m* combinadas con transformaciones *m2t* ofrecen una completa integración del DSDM con otros espacios tecnológicos. Así, los modelos también pueden servir como representación "pivote" de cara a integrar diferentes espacios tecnológicos aprovechando las técnicas del DSDM como las transformaciones de modelos o la comparación de modelos. En este ámbito, esta tesis presenta además el lenguaje API2MoL, el cual no sólo permite la extracción de modelos desde objetos conformes a un API sino también aplicar el proceso contrario, es decir, generar los objetos conformes a un API a partir del modelo, facilitando la integración de los API con el DSDM.

Los lenguajes presentados en esta tesis proporcionan las ventajas propias del uso de DSL, como son el aumento de la productividad y la mejora en la calidad y mantenimiento de las transformaciones definidas. Además, en el caso de Gra2MoL y ScheMoL proporcionan una alternativa a las implementaciones actuales, las cuales están enfocadas al desarrollo de soluciones específicas que requieren un gran esfuerzo de implementación. Por otro lado, API2MoL ofrece una solución completa para la integración de las API con el DSDM que anteriormente no había sido tratada por ninguna otra aproximación, la cual libera al desarrollador de tratar con el código del API y facilita la construcción de puentes entre el *apiware* y el *modelware* gracias al proceso de *bootstrap*.

Los espacios tecnológicos tratados cubren la mayor parte de los artefactos que componen un sistema software, exceptuando los documentos XML, los cuales son ampliamente utilizados en la creación de software para representar metadatos, por ejemplo para representar ficheros de configuración. El espacio tecnológico del XML no fue tratado en esta tesis porque ya existen herramientas para definir puentes bidireccionales entre el *xmlware* y el *modelware*, como los disponibles en la plataforma Eclipse [63]. Otros espacios tecnológicos no considerados en esta tesis son las ontologías o los servicios web. Para estos casos, hemos identificado como trabajo futuro la definición de un lenguaje general para la creación de puentes entre un espacio tecnológico cualquiera y el *modelware*, tal y como se comentará en la sección 9.2.

Para cada uno de los lenguajes presentados se han identificado diferentes escenarios de aplicación que muestran la utilidad de las transformaciones que ofrecen. Así, las secciones 5.9 y 7.8 incluyen diferentes escenarios que muestran el creciente interés y la necesidad de aplicar procesos de extracción de modelos desde el código fuente y datos, respectivamente. Por otro lado, la sección 8.8 identifica varios escenarios de aplicación de API2MoL donde la integración entre los API y el DSDM es requerida.

Tal y como se comentó en la sección 2.3, inicialmente, el objetivo principal del trabajo de investigación fue la extracción de modelos a partir del código fuente. A continuación, gracias a las colaboraciones con los grupos de investigación ONEKIN y AtlanMod se identificó la necesidad de aplicar un proceso de extracción en los espacios tecnológicos del *dataware* y del *apiware*, lo que motivó el desarrollo de ScheMoL y API2MoL. De esta forma, Gra2MoL, ScheMoL y API2MoL pueden ser considerados como una familia de lenguajes para el dominio de la extracción de modelos. Una familia de lenguajes normalmente se identifica por un conjunto de elementos comunes y variables. En el caso de los lenguajes desarrollados,

su diseño ha compartido el uso de la arquitectura de cuatro capas del OMG para la definición de los puentes entre los espacios tecnológicos considerados. En este contexto, los tres lenguajes comparten una misma estructura basada en reglas que establecen las correspondencias entre los elementos de cada espacio tecnológico. De esta forma, existe un alto grado de reutilización en los lenguajes desarrollados, aunque es menor en API2MoL. Esta reutilización se aprecia principalmente en ScheMoL, el cual está fuertemente influenciado por Gra2MoL, del cual reutiliza algunos conceptos de diseño (p. ej., tipos de reglas y uso del mecanismo de *bindings* de ATL [40]) y componentes de implementación (p. ej., el gestor de modelos).

Por otro lado, el elemento variable entre ellos es el uso de un lenguaje de consultas. Así, Gra2MoL y ScheMoL utilizan lenguajes específicos para el espacio tecnológico con el que tratan, mientras que en API2MoL no es necesario ya que las reglas solamente especifican los métodos que deben ser llamados para realizar la transformación. En el caso de Gra2MoL, nos inspiramos en XPath para crear un lenguaje de consultas *structure-shy* que facilitara el recorrido por estructuras de árbol de sintaxis. La principal ventaja de este lenguaje es el uso de los operadores `//` y `///` permiten especificar qué se quiere localizar en la consulta pero no cómo, facilitando en gran medida la definición y la legibilidad de las consultas. Por otro lado, el lenguaje de consultas de ScheMoL utiliza una sintaxis basada en la notación punto para navegar por las relaciones entre las tuplas de la base de datos. La principal característica de este lenguaje es permitir tanto la navegación directa como inversa entre las tablas de la base datos de forma transparente, facilitando la definición de las consultas.

Tanto en Gra2MoL como en ScheMoL, pudimos comprobar que el mecanismo de *bindings* utilizado en ATL se adaptó de forma satisfactoria al problema de la extracción de modelos de código fuente y de bases de datos, respectivamente. A partir de la experiencia en la definición de diferentes transformaciones en estos lenguajes, podemos afirmar que la potencia declarativa ofrecida por los *bindings* facilita la definición y comprensión de las transformaciones. Por otro lado, API2MoL no utiliza el concepto de *binding* como tal, aunque la ejecución de los procesos de extracción y generación es parecido a como se ejecutan las transformaciones basadas en *bindings*, tal y como se ha comentado en las secciones 8.3.1 y 8.3.2.

Para el desarrollo de los tres DSL utilizamos Gra2MoL para extraer modelos conformes a la correspondiente sintaxis abstracta a partir de las definiciones textuales conformes a la gramática de cada lenguaje. Esta decisión se tomó inicialmente porque, en el momento de desarrollar Gra2MoL, las herramientas de definición de DSL como Xtext no estaban lo suficientemente maduras, tal y como se ha explicado en la sección 5.6. A continuación, para ScheMoL y API2MoL tomamos la misma decisión ya que la existencia de Gra2MoL nos facilitaba enormemente su implementación. Además, esta aproximación también ilustra un escenario de aplicación para Gra2MoL fuera del contexto de la modernización y nos permitió experimentar en la definición de DSL externos. Sin embargo, al no utilizar herramientas de creación de DSL en el desarrollo de nuestros lenguajes, es necesario implementar individualmente las herramientas de soporte, es decir, un entorno de trabajo que facilite la definición y ejecución de transformaciones, como se ha hecho para Gra2MoL y se ha identificado como trabajo futuro para ScheMoL y API2MoL. En la actualidad, dado

que las herramientas de definición de DSL han evolucionado considerablemente, hubiera sido más adecuado su uso para la definición de los DSL.

Gra2MoL ha sido probado en un buen número de escenarios en los que fue necesario extraer modelos de código Java, Delphi, PL/SQL, Maude, IDL, especificaciones EBNF, gramáticas ANTLR y *scripts* Bash. El desarrollo de las transformaciones en estos escenarios de aplicación nos permitió estudiar la adecuación del lenguaje al dominio del problema de la extracción de modelos desde código fuente así como mejorar sus capacidades, validando positivamente los requisitos de calidad del lenguaje.

Un valor añadido del trabajo con Gra2MoL ha sido la valoración de la iniciativa ADM. En concreto, se definió un proceso de extracción de modelos KDM el cual se aplicó posteriormente para calcular métricas. Con esta experiencia pudimos observar que los metamodelos ASTM y KDM utilizados permiten una representación estándar de gran parte de los artefactos de un sistema software, favoreciendo la interoperabilidad entre aplicaciones dedicadas a la modernización. Sin embargo, la interoperabilidad se ve afectaba negativamente al representar con precisión los sistemas software (p. ej., representando cada una de las sentencias de programación de un aplicación) ya que se requieren conceptos propios de la arquitectura software que se está modelando (p. ej., tipos de sentencias propias del lenguaje) que no son cubiertos por estos metamodelos. Aunque los metamodelos ofrecen mecanismos para su extensión, precisamente estas extensiones decrementan la capacidad de interoperar de los modelos creados. Por otro lado, la falta de ejemplos y su complejidad impiden considerablemente su adopción, siendo en la mayoría de los casos más útil trabajar con metamodelos propios y exportar a ASTM o KDM cuando sea necesario interoperar con otras herramientas. Además, el hecho de que no existan todavía herramientas estables que soporten estos metamodelos también afecta negativamente a su adopción.

La estructura y funcionamiento de API2MoL difiere a la de Gra2MoL y SchoMoL. API2MoL ofrece soporte para la definición de transformaciones bidireccionales, permitiendo una integración completa entre el *apiware* y el *modelware*. De esta forma, la estructura de las reglas se vio alterada principalmente para soportar esta bidireccionalidad pero también por la propia naturaleza del problema. Cuando se definen las correspondencias entre los elementos del API (clases) y los del metamodelo (metaclases), en vez de utilizar un lenguaje específico de consultas, se especifican los métodos necesarios para poder extraer/generar la información necesaria en el proceso de extracción/generación.

Un aspecto importante a destacar en API2MoL es la capacidad de generar automáticamente el puente para un determinado API por medio del denominado proceso de *bootstrap*. Este proceso de *bootstrap*, en vez de utilizarse para el desarrollo del DSL en sí mismo (como se ha aplicado en Gra2MoL), realmente se utiliza para generar automáticamente los artefactos requeridos en un puente para un API (definición API2MoL y metamodelo del API).

Los lenguajes presentados en esta tesis han sido validados con diferentes casos de estudio, lo que nos ha permitido identificar limitaciones y solucionar algunas deficiencias. Además, en el caso de API2MoL también se ha aplicado un proceso de validación para comprobar la corrección del lenguaje así como la cobertura tanto del lenguaje como del proceso *bootstrap*. Con este proceso de validación en API2MoL hemos comprobado que el proceso de *bootstrap*

genera prácticamente el total de los artefactos necesarios para construir el puente, siendo solamente necesario enriquecer la definición API2MoL generada. Sin embargo, a pesar de requerir esta intervención manual, el proceso facilita en gran medida la definición de puentes para cualquier API, siendo un buen punto de partida para su creación.

En general, los lenguajes y herramientas definidos como resultado del trabajo de investigación de esta tesis ofrecen a los desarrolladores los mecanismos necesarios para integrar diferentes artefactos software en soluciones basadas en modelos. De esta forma, creemos que estos lenguajes pueden llegar a ser un paso más para facilitar la adopción del DSDM.

9.2. Trabajo Futuro

A partir de las tareas desarrolladas en esta tesis se han identificado las siguientes líneas de trabajo futuras, clasificadas según el lenguaje o área de interés que trata.

Gra2MoL

Dado que el lenguaje depende del CST para representar el código fuente y aplicar las consultas definidas en la transformación, su gestión es un aspecto fundamental que afecta a la eficiencia de Gra2MoL. Actualmente Gra2MoL soporta la gestión de CST en memoria o en un repositorio de modelos como CDO, el cual es recomendable cuando se trata con un gran número de ficheros de código. En este sentido, estamos interesados en estudiar la escalabidad de ambas alternativas así como el estudio de otras, como la creación de CST *al vuelo* conforme se va reconociendo el código. Además, dado que las consultas se aplican sobre el CST, también estamos interesados en el impacto que tiene la forma de gestionar el CST en la ejecución de las consultas.

Por otro lado, los modelos que se extraen del código fuente pueden tener un tamaño considerable que depende del tamaño del sistema software. Por este motivo, es necesario estudiar mecanismos que permitan a Gra2MoL gestionar los modelos que se extraen de forma eficiente. En este sentido, también sería útil incorporar el soporte a CDO para almacenar los modelos extraídos del ódigo.

También creemos conveniente estudiar mecanismos de modularidad para favorecer la composición y el mantenimiento de las transformaciones Gra2MoL. Por ejemplo, vemos interesante estudiar la incorporación de un mecanismo de fases al lenguaje similar al proporcionado por RubyTL. El uso de fases también motiva el estudio de un mecanismo de consulta de trazas de ejecución para ofrecer un mejor soporte al control del flujo de transformaciones.

Finalmente, dado que Gra2MoL solamente soporta el uso de gramáticas ANTLR, estamos interesados en ampliar el número de formatos de gramáticas para facilitar la definición de transformaciones reutilizando las gramáticas ya existentes.

ScheMoL

Para facilitar la adaptación del lenguaje a diferentes problemas de extracción de modelos a partir de datos almacenados en una base de datos relacional, estamos interesados en

incorporar un mecanismo de extensibilidad de forma parecida al definido en Gra2MoL. De esta forma, los desarrolladores podrían mejorar las capacidades del lenguaje con nuevos operadores para las consultas.

Por otro lado, al igual que se ha comentado para el caso de Gra2MoL, dado que los modelos extraídos de la base de datos pueden ser de gran tamaño, también vemos muy interesante la adaptación del lenguaje para trabajar con repositorios de modelos. Por ejemplo, podría adaptarse el lenguaje para almacenar el modelo resultante de la transformación en un repositorio CDO.

Actualmente el lenguaje soporta bases de datos MySQL y se está trabajando en el soporte a Firebird [64]. Sería necesario ampliar el soporte a otros gestores de bases de datos, como por ejemplo Oracle, dada su amplia utilización en la industria.

Finalmente, para facilitar la integración y el soporte de la herramienta, SchcMoL debería ofrecer un entorno de desarrollo con herramientas que faciliten la definición y ejecución de transformaciones así como su integración con otras herramientas y lenguajes. En este sentido, sería deseable implementar un *plugin* para la plataforma Eclipse, de la misma manera a como se ha realizado en Gra2MoL.

API2MoL

Actualmente, el modo de funcionamiento de API2MoL es de tipo *batch*, es decir, los procesos de extracción y generación se realizan en un solo paso para todos los objetos del API y del modelo, respectivamente. Otros modos de funcionamiento pueden plantearse permitiendo que, además de soportar la extracción y generación en un solo paso, se soporte la extracción y/o generación incremental. Por ejemplo, en escenarios de reingeniería de interfaces gráficas, podría ser interesante utilizar un extractor de un solo paso y un generador incremental. En estos escenarios se extraería inicialmente un modelo del conjunto de objetos que corresponden a la interfaz gráfica de forma que posteriores cambios en dichos modelos se reflejarían en la interfaz incrementalmente. Otro ejemplo se daría en escenarios que utilicen modelos *runtime*, donde sería útil disponer tanto de un extractor como un generador incremental de forma que tanto los cambios en el sistema como en el modelo que lo representa se extraerían/generarían incrementalmente, permitiendo que el modelo se mantenga siempre sincronizado.

Dado que la actual implementación de API2MoL solamente puede acceder reflexivamente a los objetos que se ejecutan en la misma máquina virtual que el proceso API2MoL, también estamos estudiando incorporar el soporte para el API *Java Debug Interface* (JDI) con el objetivo de poder manipular objetos Java en memoria que no son accesibles por medio de reflexión. Por ejemplo, este soporte permitiría acceder a los objetos de aplicaciones J2EE que están siendo ejecutados por un servidor de aplicaciones. Además, dado que API2MoL actualmente soporta API Java, sería interesante su adaptación para otros lenguajes de programación como C#. De esta forma, también se podría estudiar la posibilidad de independizar el motor de API2MoL del lenguaje de programación en el que esté implementado el API.

Otra línea de trabajo futura es la adaptación a API no orientadas a objetos, como descripciones de servicios web o *mashups*. En este sentido, estamos particularmente interesados

en extender API2MoL para facilitar la interacción entre aplicaciones basadas en modelos y servicios web, los cuales pueden ser vistos como un API externa.

Al igual que en ScheMoL, API2MoL requiere un entorno de desarrollo que facilite la definición de procesos de extracción y ejecución así como la configuración del proceso de *bootstrap*. Actualmente estamos trabajando en un *plugin* para la plataforma Eclipse para facilitar estas tareas, favoreciendo el soporte del lenguaje y la integración con otras herramientas.

Otro aspecto a considerar como línea de trabajo futura en API2MoL sería la incorporación de mecanismos para extender el lenguaje. Actualmente estamos estudiando la posibilidad de incorporar un mecanismo para introducir nuevos tipos de elementos *statement* que mejoren el soporte para adaptarse a las funciones ofrecidas por un API.

Finalmente, queremos experimentar con los diferentes escenarios de aplicación identificados en 8.8. Este estudio también permitiría analizar la necesidad de incorporar mecanismos de gestión de grandes modelos ya que, al igual que en los casos anteriores, los modelos extraídos pueden tener un tamaño considerable. Esta línea de trabajo relativa a los escenarios de aplicación también afecta a Gra2MoL y ScheMoL, cuyos escenarios de aplicación se han identificado en 5.9 y 7.8, respectivamente.

Lenguaje general de extracción de modelos

A partir de nuestra experiencia en el desarrollo de los DSL presentados en esta tesis, una línea de trabajo sería la definición de un nuevo DSL más general que permitiera crear cualquier puente hacia el *modelware*, el cual reutilizaría gran parte del conocimiento adquirido durante el desarrollo de los DSL de esta tesis y permitiría experimentar con el concepto de familia de DSL. Este nuevo lenguaje permitiría definir las correspondencias entre los elementos del espacio tecnológico considerado y el *modelware*.

El desarrollo de este nuevo DSL requeriría estudiar la adecuación del uso de reglas para definir las correspondencias entre los elementos. También deberían estudiarse características básicas utilizadas en Gra2MoL y ScheMoL, como es el uso del mecanismo de *binding* o la necesidad de un lenguaje de consultas; o la posibilidad de dar soporte a la bidireccionalidad, tal y como se ha hecho con API2MoL, dependiendo del dominio de aplicación que se esté tratando.

El motor de transformación del nuevo DSL podría ser parametrizado en dos puntos: (1) el tipo de los elementos origen y (2) el lenguaje de consultas a utilizar, si fuera necesario. Esta parametrización requeriría, en primer lugar, ofrecer mecanismos para que el DSL pueda interactuar con los elementos origen, es decir, sea capaz de resolver los *bindings* entre el elementos origen y la propiedad del metamodelo. Por otro lado, también sería necesario estudiar mecanismos de composición de lenguajes, ya que el lenguaje base debería ser extendido para expresar el posible lenguaje de consultas necesario para obtener información del dominio origen.

Experimentación con ADM

La iniciativa ADM todavía se encuentra en desarrollo. Prueba de ello es la falta de algunos de los metamodelos planificados. En este contexto, estamos interesados en continuar estudiando la iniciativa con casos reales. Actualmente están disponibles las especificaciones de los metamodelos ASTM, KDM y SMM, algunos de los cuales han sido tratados en el contexto de esta tesis. En el contexto de ASTM y KDM hemos identificado algunas líneas de trabajo futuro. Con respecto a ASTM, estamos interesados en el estudio de definir transformaciones *m2m* modulares, de forma que las transformaciones que traten con el GASTM sean reutilizables para cualquier lenguaje, cuyos elementos particulares se representan con SASTM.

En cuanto al metamodelo KDM, estamos interesados en estudiar en profundidad los paquetes de la capas de recursos y abstracciones, los cuales permiten definir vistas arquitecturales de partes concretas de un sistema como la interfaz gráfica o los procesos de negocio, dado que en la actualidad los ejemplos existentes son muy simples. Nuestro objetivo sería modelar sistemas software reales utilizando estos paquetes pero también estudiar las relaciones entre los elementos de los modelos de cada capa en KDM.

Finalmente, tal y como se ha comentado en la sección 3.4.1, el metamodelo IPMSS se encuentra en la fase final de su aprobación y publicación en la iniciativa ADM. Como trabajo futuro en este ámbito, vemos interesante el estudio de aplicar transformaciones Gra2MoL para extraer modelos IPMSS a partir del código fuente y comparar estar aproximación con otra que utilice transformaciones *m2m* desde los modelos ASTM extraídos con Gra2MoL para obtener modelos IPMSS.

9.3. Publicaciones

Revistas con índice de impacto

- J. L. Cánovas y J. García Molina. An Architecture-Driven Modernization Tool for Calculating Metrics. *IEEE Software*, 27:37–43, 2010.
  Índice de Impacto: 2,039 (1^{er} tercio).
- Ó. Díaz, G. Puente, J. L. Cánovas y J. García Molina. Harvesting Models from Web 2.0 Databases. *Software and System Modeling*, 11, 2011.
  Índice de Impacto: 1,533 (1^{er} tercio)
- J. L. Cánovas, F. Jouault, J. Cabot y J. García Molina. API2MoL: Automating the Building of Bridges between API and Model-Driven Engineering. *Journal on Information and Software Technology*, 2011. En proceso de revisión, *major revision*.
  Índice de Impacto: 1,821 (1^{er} tercio)
- J. L. Cánovas y J. García Molina. Extracting Models from Source Code in Software Modernization. *Software and System Modeling, Model Evolution Issue*, 2011. En proceso de revisión.
  Índice de Impacto: 1,533 (1^{er} tercio)

Congresos de calidad similar a una revista por su grado de aceptación

- J. L. Cánovas y J. García Molina. A Domain Specific Language for Extracting Models in Software Modernization. En European Conference on Model Driven Architecture - Foundations and Applications, vol. 5562, pags. 82-97, Berlin, Heidelberg, 2009. Springer-Verlag.
  Grado de aceptación: 33 %.

Conferencias / Talleres Internacionales

- J. L. Cánovas, J. Sánchez Cuadrado y J. García Molina. Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization. En Workshop Model Driven Software Evolution, 2008.
- J. L. Cánovas y J. García Molina. Gra2MoL put into practice. En Tools and Consultancy Track de la European Conference on Modelling Foundations and Applications, 2010.

Conferencias / Talleres Nacionales

- J. L. Cánovas, B. Cruz Zapata y J. García Molina. Definición y ejecución de métricas en el contexto de ADM. En Taller sobre Desarrollo de Software Dirigido por Modelos, 2010.
- J. L. Cánovas y J. García Molina. Gra2MoL: Una herramienta para la extracción de modelos en modernización de software. Demostración en Jornadas de Ingeniería del Software y Bases de Datos, 2009.
- J. L. Cánovas y J. García Molina. Extracción de modelos en una modernización basada en ADM. En Taller sobre Desarrollo de Software Dirigido por Modelos, 2009.
- J. L. Cánovas, Ó. Sánchez Ramón, J. Sánchez Cuadrado y J. García Molina. DSL para la extracción de modelos en modernización. En Taller sobre Desarrollo de Software Dirigido por Modelos, 2008.
- J. L. Cánovas, Ó. Sánchez Ramón, J. Sánchez Cuadrado y J. García Molina. Utilidad de las transformaciones modelo-modelo en la generación de código. En Jornadas de Ingeniería del Software y Bases de Datos, 2007.

Tutoriales

- J. L. Cánovas y J. García Molina. Una aplicación práctica de Architecture-Driven Modernization (ADM). Jornadas de Ingeniería del Software y Bases de Datos, 2010.

9.4. Proyectos fin de carrera dirigidos

- B. Cruz Zapata. Medea: Definición y ejecución de métricas basadas en SMM. Directores: J. García Molina y J. L. Cánovas. Junio 2010.

9.5. Estancias de investigación

El trabajo de investigación se vio complementado con dos colaboraciones con los siguientes grupos de investigación:

- Estancia de investigación en el grupo de investigación AtlanMod (Department of Computer Science, École des Mines de Nantes and INRIA) desde el 11 de abril de 2010 al 18 de julio de 2010, bajo supervisión del Dr. Jordi Cabot. El trabajo de la estancia se centró en la implementación del lenguaje API2MoL.
- Dos visitas al grupo de investigación ONEKIN (Universidad de País Vasco) bajo supervisión del Dr. Óscar Díaz. La primera visita fue del 11 de septiembre de 2009 al 18 de septiembre de 2009 y la segunda fue del 19 de septiembre de 2010 al 24 de septiembre de 2010. EL trabajo realizado se centró en el desarrollo del lenguaje ScheMoL.

9.6. Herramientas desarrolladas

Todas las herramientas desarrolladas a lo largo del trabajo de investigación están accesibles *online* desde sus respectivos sitios web, los cuales son:

- **Gra2MoL**. `http://modelum.es/gra2mol`
  El sitio web de Gra2MoL contiene todos los recursos necesarios para trabajar con la herramienta: código fuente, ejemplos, casos de estudio, manual de uso y *plugin* de Eclipse. Los casos de uso basados en ADM también pueden ser descargados desde esta web. Además, la herramienta ha sido recientemente licenciada como EPL.
- **ScheMoL**. `http://modelum.es/schemol`
  El sitio web de ScheMoL incluye los diferentes casos de uso donde se utilizó ScheMoL así como el código fuente para su descarga.
- **API2MoL**. `http://modelum.es/api2mol`
  El sitio web de API2MoL incluye los ejemplos utilizados en la validación del lenguaje así como el código fuente para su descarga.

9.7. Proyectos que han utilizado los resultados de esta tesis

Los resultados de esta tesis han sido utilizados en los siguientes proyectos:

- **Herramienta orientada a la migración basada en modelos.**
  Este proyecto está financiado por fondos del CDTI (Centro para el Desarrollo Tecnológico Industrial) y se realiza conjuntamente con la empresa Sinergia Tecnológica.

Comenzó el 28 de julio de 2010 y tiene una duración de 2 años.
El objetivo de este proyecto es la construcción de una herramienta para asistir la migración automática de aplicaciones Oracle Forms. Gra2MoL fue utilizado para la extracción de modelos a partir del código fuente PL/SQL de los *triggers* de una aplicación Oracle Forms así como para extraer modelos a partir de la definición del esquema de base de datos utilizado por la aplicación.

- **Mancoosi.** `http://www.mancoosi.org/`
  Mancoosi es un proyecto europeo financiado por el *7th Research Framework Programme (FP7)* de la comisión europea. Comenzó el 1 de febrero de 2008 y tuvo una duración de 3 años.
  El objetivo de este proyecto es ofrecer soporte a la simulación de actualizaciones de sistemas de información para predecir fallos que puedan afectar a sistemas reales. Gra2MoL fue utilizado para la extracción de modelos desde ficheros *script* escritos en Bash, permitiendo representar aspectos estáticos (p. ej., incoherencias en la configuración) como dinámicos (p. ej., la ejecución de *scripts* de mantenimiento).
- **MOMO: Un entorno de modernización de software dirigida por modelos en escenarios de migración de plataformas.**
  Este proyecto fue financiado por la Fundación Séneca. Comenzó el 1 de enero de 2009 y finalizó el 31 de diciembre de 2010.
  El objetivo de este proyecto es la construcción de un entorno de modernización basado en modelos para escenarios de migración de plataformas. Gra2MoL se incorporó como lenguaje de transformación *t2m* para llevar a cabo la extracción de modelos a partir del código fuente y se utilizó en el caso de estudio para extraer modelos desde código Delphi.

9.8. Becas

Durante el desarrollo de esta tesis el candidato disfrutó de las siguientes becas:

- **Beca-contrato predoctoral de formación del personal investigador.**
  Financiada por la Fundación Séneca, tienen una duración de cuatro años y permitió al candidato dedicarse íntegramente al trabajo de investigación.
- **Ayuda para estancias cortas en centros distintos al de aplicación de los becarios-contratados FPI.**
  Financiada por la Fundación Séneca, esta beca permitió realizar la estancia de investigación en el grupo AtlanMod.
- **Becas para asistencia a congresos científicos**
  Financiada por la Fundación Séneca, permitió asistir a las Jornadas de Ingeniería del Software y Bases de Datos en el 2007.

Bibliografía

[1] L. F. Andrade, J. Gouveia, M. Antunes, M. El-Ramly, and G. Koutsoukos. Forms2Net - Migrating Oracle Forms to Microsoft .NET. In *Generative and Transformational Techniques in Software Engineering conference*, pages 261–277, 2006.

[2] AndroMDA. `http://www.andromda.org/`.

[3] M. Antkiewicz and K. Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. In *Models conference*, pages 692–706, 2006.

[4] M. Antkiewicz, K. Czarnecki, and M. Stephan. Engineering of Framework-Specific Modeling Languages. *IEEE Transactions on Software Engineering*, 36(6):795–824, 2009.

[5] M. Antkiewicz, K. Czarnecki, and M. Stephan. Engineering of Framework-Specific Modeling Languages. *IEEE Transactions on Software Engineering*, 36, no. 6:795–824, 2009.

[6] Antlrworks. `http://www.antlr.org/works`.

[7] C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[8] J. Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE, the European Journal for the Informatics Professional*, V(2):21–24, 2004.

[9] J. Bézivin, M. Barbero, and F. Jouaul. On the Applicability Scope of Model Driven Engineering. In *Model-based Methodologies for Pervasive and Embedded Software workshop*, 2007.

[10] C. Brun and A. Pierantonio. Model Differences in the Eclipse Modelling Framework. *UPGRADE, the European Journal for the Informatics Professional*, Vol. IX, issue No. 2:29–34, 2008.

[11] H. Brunelière, J. Cabot, F. Jouault, and F. Madiot. MoDisco: a Generic and Extensible Framework for Model Driven Reverse Engineering. In *Automated Software Engineering conference*, pages 173–174, 2010.

[12] C. Cetina, P Giner, J. Fons, and V. Pelechano. Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. *IEEE Computer*, 42(10):37–43, 2009.

[13] P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database System*, 1:9–36, 1976.

[14] E. J. Chikofski and J. H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[15] A. Cicchetti, D. Di Ruscio, L. Lovino, and A. Pierantonio. Managing the Evolution of Data-Intensive Web Applications by Model-Driven Techniques. *Software and Systems Modeling*, 10:1–31, 2011.

[16] T. Clark, P. Sammut, and J. Willians. *Applied Metamodelling. A Foundation for Language Driven Development.* Ceteva, 2008.

[17] OCL constraint language. `http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf`.

[18] S. Cook, G. Jones, S. Kent, and A. W. Cameron. *Domain-Specific Development with Visual Studio DSL Tools.* Addison Wesley Professional, 2007.

[19] K. Czarnecki and U. W. Eisenecker. *Generative Programming.* Addison Wesley, 2000.

[20] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Generative Technique in the Context of the Model Driven Architecture workshop*, 2003.

[21] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[22] R. De Groot. Implementation of the Java-swul language. A Domain-specific Language for the SWING API Embedded in Java.

[23] A. V. Deursen and P. Klint. Little languages: Little maintenance?, 1998.

[24] Rivera. J. E. *On the Semantics of Real-Time Domain Specific Modeling Languages.* PhD thesis, Universidad de Málaga, 2010.

[25] Eclipse *e4* project. `http://www.eclipse.org/e4`.

[26] Enterprise JavaBeans 3.0 Specification (JSR 220). `http://jcp.org/en/jsr/detail?id=220`.

[27] Epsilon. http://www.eclipse.org/gmt/epsilon.

[28] J. M. Favre. Foundations of Model (Driven) (Reverse) Engineering - Episode i: Story of The Fidus Papyrus and the Solarus. In *Dagsthul Seminar 04101 on Language Engineering for Model-Driven Software Development*, 2004.

[29] J. M. Favre. Foundations of Model (Driven) (Reverse) Engineering - Episode ii: Story of Thotus the Baboon. In *Dagsthul Seminar 04101 on Language Engineering for Model-Driven Software Development*, 2004.

[30] M. Fowler. *Domain-Specific Languages.* Addison Wesley, 2011.

[31] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison Wesley Professional, 1999.

[32] GCC. `http://gcc.gnu.org`.

[33] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories. Assembling Applications Frameworks.* Wiley, 2004.

[34] hCard Microformat specifications. `http://microformats.org/wiki/hcard`.

[35] R Heckel, R. Correia, C. Matos, M. El-Ramly, G. Koutsoukos, and L. F. Andrade. Architectural Transformations: From Legacy to Three-Tier and Services. In *Software Evolution.* Springer, 2008.

[36] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and Refinement of Textual Syntax for Models. In *European Conference on Model Driven Architecture - Foundations and Applications*, volume 5562, pages 114–129, 2009.

[37] Hibernate. `http://www.hibernate.org`.

[38] ADM initiative. `http://adm.omg.org`.

[39] JDBC. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html`.

[40] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):21–39, 2008.

[41] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented Domain Analisys (foda) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[42] S. Kelly and R. Pohjonen. Worst Practices for Domain-Specific Modeling. *IEEE Software*, 26:4, 2009.

[43] S. Kelly and J. P. Tolvanen. *Domain-Specific Modeling*. Wiley-IEEE Computer Society, 2008.

[44] A. Kleppe. Towards the Generation of a Text-Based IDE from a Language Metamodel. In *European Conference on Model Driven Architecture - Foundations and Applications*, pages 114–129, 2007.

[45] A. Kleppe. *Software Language Engineering*. Addison Wesley, 2008.

[46] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained. The Model Driven Architecture. Practice and Promise*. Addison Wesley, 2003.

[47] D. Kolovos, R. Paige, T. Kelly, and F. Polack. Requirements for Domain-Specific Languages. In *Domain-Specific Program Development workshop*, 2006.

[48] A. Kunert. Semi-automatic Generation of mMetamodels and Models from Grammars and Programs. In Theorical Computer Science, editor, *Graph Transformation and Visual Modeling Techniques worshop*, volume 211, pages 111–119, 2008.

[49] I. Kurtev, J. Bézivin, and M. Aksit. Technological Spaces: An Initial Appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial Track*, 2002.

[50] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computer Survey*, 37(4):316–344, 2005.

[51] ASTM metamodel specification. `http://www.omg.org/spec/ASTM`.

[52] KDM metamodel specification. `http://www.omg.org/spec/KDM`.

[53] SMM metamodel specification. `http://www.omg.org/spec/ASTM`.

[54] MOFScript. `http://www.eclipse.org/gmt/mofscript`.

[55] MPS. `http://www.jetbrains.com/mps`.

[56] Newcode. Migrating Visual Basic Applications to VB.NET using the NewCode extension for Microsoft Visual Studio, 2008.

[57] G Nordstrom, J. Sztipanovits, G. Karsai, and A. Ledeczi. Metamodeling-Rapid Design and Evolution of Domain-Specific Modeling Environments. *IEEE International Conference on the Engineering of Computer-Based Systems*, 0:68, 1999.

[58] Open Architecture Ware. `http://www.openarchitectureware.org`.

[59] R. Pérez-Castillo, I. García Rodríguez, M. Piattini, and O. Ávila-García. MARBLE: A Modernization Approach for Recovering Business Process from Legacy Systems. In *Reverse Engineering Models from Software Artifacts workshop*, pages 17–20, 2009.

[60] R. Pérez-Castillo, I. García Rodríguez, M. Piattini, and O. Ávila-García. On the use of ADM to Contextualize Data on Legacy Source Code for Software Modernization. In *Working Conference on Reverse Engineering*, pages 128–132, 2009.

[61] ANTLR project. `http://http://www.antlr.org`.

[62] CDO project. `wiki.eclipse.org/CDO`.

[63] EMF Eclipse project. `http://www.eclipse.org/emf`.

[64] Firebird project. `http://www.firebirdsql.org`.

[65] GMP project. `http://www.eclipse.org/modeling/gmp`.

[66] GMT project. `http://www.eclipse.org/gmt`.

[67] JAMOPP project. `http://jamopp.inf.tu-dresden.de`.

[68] JastAdd project. `http://www.jastadd.org`.

[69] JDT Eclipse project. `http://www.eclipse.org/jdt`.

[70] JET project. `www.eclipse.org/emft/projects/jet`.

[71] JTwitter project. `http://www.winterwell.com/software/jtwitter.php`.

[72] Kiama project. `http://code.google.com/p/kiama/`.

[73] MetaEdit project. `http://www.metacase.com/mep`.

[74] Portolan project. `http://code.google.com/a/eclipselabs.org/p/portolan`.

[75] Spoofax project. `http://spoofax.org`.

[76] Stratego project. `http://strategoxt.org`.

[77] Swing project. `http://java.sun.com/products/jfc/tsc/articles/architecture`.

[78] SWT Eclipse project. `http://www.eclipse.org/swt`.

[79] TCS project. `http://www.eclipse.org/gmt/tcs`.

[80] Teneo project. `http://wiki.eclipse.org/Teneo`.

[81] TXL project. `http://www.txl.ca`.

[82] Wazaabi project. `http://wazaabi.org`.

[83] D. Ratiu, M. Feilkas, and J. Jürjens. Extracting Domain Ontologies from Domain Specific APIs. In *European Conference on Software Maintenance and Reengineering*, pages 203–212, 2008.

[84] RDFa specification. `http://rdfa.info/wiki/Introduction`.

[85] J. Sánchez Cuadrado and J. García Molina. Building Domain-Specific Languages for Model-Driven Development. *IEEE Software*, 24(5):48–55, 2007.

[86] J. Sánchez Cuadrado and J. García Molina. A Model-Based Approach to Families of Embedded Domain-Specific Languages. *IEEE Transactions on Software Engineering*, 35(6):825–840, 2009.

[87] J Sánchez Cuadrado, J. García Molina, and M. M. Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *European Conference on Model Driven Architecture - Foundations and Applications*, volume 4066, pages 158–172, 2006.

[88] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems*. Addison Wesley, 2003.

[89] B. Selic. MDA Manifestations. *UPGRADE, the European Journal for the Informatics Professional*, Vol. IX, issue No. 2:7–11, 2008.

[90] B. Selic. Personal Reflections on Automated Programming, Culture and Model-based Software Engineering. *Automated Software Engineering*, 15:3–4, 2008.

[91] D. Sergey. Language Oriented Programming: The Next Programming Paradigm, 2004.

[92] JMX server. `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

[93] H. Song, G. Huang, Y. Xiong, F. Chauvel, Y. Sun, and H. Mei. Inferring Meta-models for Runtime System Data from the Clients of Management APIs. In *Models conference*, pages 168–182, 2010.

[94] H. Song, Y. Xiong, F. Chauvel, G. Huang, Z. Hu, and H. Mei. Generating Synchronization Engines between Running Systems and their Model-Based Views. In *Models conference workshops*, pages 140–154, 2009.

[95] MOF specification. `http://www.omg.org/mof`.

[96] QVT specification. `http://www.omg.org/spec/QVT/1.0/PDF`.

[97] UML specification. `http://www.omg.org/spec/UML`.

[98] MDA specifications. `http://www.omg.org/mda/specs.htm`.

[99] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[100] ADM Task Force. Why Do We Need Standards For The Modernization Of Existing Systems? White Paper.

[101] ADM Task Force. Architecture-Driven Modernization Scenarios. White Paper, 2006.

[102] T. Tonelli, K. Czarnecki, and R. Lämmel. Swing to SWT and Back: Patterns for API Migration by Wrapping, 2010.

[103] T. Tonelli, K. Czarnecki, R Lämmel, and T. van der Storm. Study of an API Migration for Two xml APIs. In *Postproceedings of Software Language Engineering conference*, 2009.

[104] W. Ulrich and P. Newcomb. *Information Systems Transformation: Architecture Driven Modernization Case Studies*. Morgan Kaufmann, 2010.

[105] M. Völter. MD*/DSL Best Practices, 2011.

[106] J. Wijngaarden. Code Generation from a Domain Specific Language. MSc Thesis, 2003.

[107] J. Wijngaarden and E. Visser. Program Transformation Mechanics. A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems. Technical report, Department of Information and Computing Sciences, Utrecht University, 2003.

[108] M. Wimmer and G. Kramler. Bridging Grammarware and Modelware. In *Satellite Events at the Models conference*, pages 159–168, 2006.

[109] Xpand. `http://wiki.eclipse.org/Xpand`.

[110] Xpath. `http://www.w3.org/TR/xpath`.

[111] OpenArchitectureWare 4.1 xText language reference. `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents`.

[112] J. Øyvind and I. Solheim. New Roles in Model-Driven Development. In *Model Driven Architecture with an emphasis on Methodologies and Transformations workshop*, 2004.