DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CARRERA DE INGENIERÍA EN SISTEMAS E INFORMÁTICA

PROYECTO DE TITULACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN SISTEMAS E INFORMÁTICA

AUTOR: FRANCISCO XAVIER EREMIEV BURNEO

TEMA: ESTUDIO Y COMPARACIÓN DE LOS LENGUAJES DE GENERACIÓN AUTOMÁTICA DE CÓDIGO QVT Y ATL

DIRECTOR: ING. HINOJOSA, CECILIA

CODIRECTOR: ING. CORAL, HENRY

SANGOLQUÍ, SEPTIEMBRE 2014

CERTIFICACIÓN

Certifico que el presente trabajo fue realizado en su totalidad por el Sr. FRANCISCO XAVIER EREMIEV BURNEO como requerimiento parcial a la obtención del título de INGENIERO EN SISTEMAS E INFORMÁTICA

Sangolquí, Septiembre de 2014

_____	_____
ING. CECILIA HINOJOSA	ING. HENRY CORAL
DIRECTORA DE TESIS	CODIRECTOR DE TESIS

DECLARACIÓN

Yo, Francisco Xavier Eremiev Burneo, declaro que el presente trabajo es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación personal y que he consultado las referencias bibliográficas que se incluyen en el documento.

La Universidad de las Fuerzas Armadas ESPE puede hacer uso de los derechos correspondientes a este trabajo, según lo establecido por la Ley de Propiedad Intelectual, por su reglamento y por la normativa institucional vigente.

Sangolquí, Septiembre de 2014

_____

Francisco Xavier Eremiev Burno

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CARRERA DE INGENIERÍA EN SISTEMAS E INFORMÁTICA

AUTORIZACIÓN

Yo:

Francisco Xavier Eremiev Burneo

Autorizo a la Universidad de las Fuerzas Armadas ESPE la publicación, en la biblioteca virtual de la Institución, del trabajo "ESTUDIO Y COMPARACIÓN DE LOS LENGUAJES DE GENERACIÓN AUTOMÁTICA DE CÓDIGO QVT Y ATL" cuyo contenido, criterio e ideas son de mi exclusiva responsabilidad y autoría.

Sangolquí, Septiembre de 2014

_____

Francisco Xavier Eremiev Burneo

No. 110471405-8

DEDICATORIA

A mi madre.

FRANCISCO XAVIER EREMIEV BURNEO

AGRADECIMIENTOS

A mis profesores a quienes les debo gran parte de mis conocimientos, en especial a la ingeniera Cecilia Hinojosa y al ingeniero Henry Coral por su invaluable apoyo durante el desarrollo del presente trabajo.

FRANCISCO XAVIER EREMIEV BURNEO

Índice de Contenidos

LISTADO DE FIGURAS

LISTADO DE TABLAS

RESUMEN

Con la evolución del desarrollo de software, han surgido metodologías que incorporan dentro de sus procesos la transformación de modelos, con el fin de incrementar su productividad, maximizando la compatibilidad entre los sistemas, para lo cual hacen uso de modelos estándar que pueden ser migrados entre las diferentes tecnologías. Tanto QVT como ATL, son lenguajes de transformación de modelos, QVT enfocado en el desarrollo de software y ATL en la ingeniería de datos. Con el propósito de analizar las propiedades de ambos lenguajes, se realizó su estudio, utilizando como referencia las definiciones formales de éstos. En base a las características deseables de un buen lenguaje de transformación de modelos y relacionándolas con las dimensiones: Uso, Organización y Formalización del lenguaje, se desarrolló el modelo de comparación. Con dicho modelo se evaluaron de forma técnica los lenguajes QVT y ATL, obteniendo como resultado que QVT mantiene un buen nivel de cumplimiento en cuanto a las características deseables, especialmente porque cuenta con una alta expresividad, un buen manejo de condiciones, y cumpliendo con el requerimiento de ser declarativo, en cuanto a ATL sus puntos fuertes radican en su alta ejecutabilidad, y el poder manejar construcciones gráficas de forma eficiente. El modelo comparativo propuesto es genérico, y puede ser utilizado como base para el estudio de lenguajes de transformación de modelos.

Palabras Clave: Lenguajes de Transformación de Modelos, QVT, ATL, estudio comparativo.

ABSTRACT

With the evolution of software development, methodologies have emerged that incorporate within their process model transformation, which make use of standard models that can be migrated between different technologies, this, in order to increase productivity by maximizing compatibility between systems. Both QVT and ATL, are model transformation languages, QVT focused on software development and ATL in data engineering. In order to analyze the properties of both languages, a study was conducted, using as reference the formal definitions of these. Based on the desirable characteristics of a good language model transformation and relating dimensions: Purpose, Organization and Formalization of language, the comparison model was developed. With this model were evaluated technically the ATL and QVT languages, resulting that QVT maintains a good level of compliance in terms of desirable features, especially because it has a high expressiveness, a good condition management, and complying with the requirement of being declarative, about ATL its strengths lie in its high executability, and its capacity to handle graphic constructions efficiently. The comparative model proposed is generic and can be used as basis for the study of model transformation languages.

Key words: Transformation Model Languages, QVT, ATL, comparative study.

CAPÍTULO 1

ESTUDIO Y COMPARACIÓN DE LENGUAJES DE GENERACIÓN AUTOMÁTICA DE CÓDIGO QVT Y ATL.

1.1 INTRODUCCIÓN

Según la definición de la OMG (OMG, 2010), MDE (MODEL-DRIVEN ENGINEERING) es una metodología de diseño de software que se enfoca en la generación de modelos de dominio, éstos son representaciones abstractas del conocimiento y actividades que se manejan en un determinado ambiente, en lugar de enfocarse en conceptos computacionales o algorítmicos.

El objetivo de MDE (MODEL-DRIVEN ENGINEERING) es el de incrementar la productividad al maximizar la compatibilidad entre los sistemas, esto mediante la utilización de modelos estándar que pueden ser migrados entre las diferentes tecnologías.

Dichos modelos, independientes del lenguaje y plataforma donde el programa será ejecutado, son conocidos como PIM (platform-indepentent model), los cuales a través de un lenguaje de transformación de modelos, como QVT (Query View Transformation) o ATL (Atlas Transformation Language), permiten la generación automática de modelos específicos de plataforma PSMs (Platform-specific models).

Tanto QVT como ATL son lenguajes para la transformación de modelos, los cuales tienen como objetivo producir modelos a partir de otros modelos de origen.

1.2 PLANTEAMIENTO DEL PROBLEMA

QVT es propuesto por la OMG (OBJECT MANAGEMENT GROUP), en el contexto del enfoque MDA (MODEL-DRIVEN ARCHITECTURE), teniendo como objetivo el desarrollo de software, mientras que ATL se centra en resolver problemas inter operacionales en la ingeniería de datos, que puede ser utilizado así mismo para resolver problemas de desarrollo de software.

Dado que los modelos se encuentran desarrollados en un número de lenguajes limitados (UML), la definición de un lenguaje correcto de transformación facilitará la implementación de un sistema.

El OMG (OBJECT MANAGEMENT GROUP), responsable de la definición de MDA (MODEL-DRIVEN ARCHITECTURE), proporciona los lineamientos que dictan las características deseables con las cuales un buen lenguaje de transformación de modelos debe contar, sin embargo no existe una metodología que permita la evaluación de los lenguajes de transformación de modelos con el fin de conocer el nivel de cumplimiento de dichas características, por esta razón, la ejecución de un proyecto en el cual se desarrolle un modelo comparativo que permita determinar dicho nivel de cumplimiento, facilitaría la selección de un lenguaje de transformación de modelos, al momento de desarrollar un proyecto que contemple la transformación de modelos siguiendo los lienamientos de MDA.

1.3 JUSTIFICACIÓN E IMPORTANCIA

Ya que ambos lenguajes comparten las mismas bases y principios de la orientación a objetos, así como, la utilización de modelos PIM (PLATFORM-INDEPENTENT MODEL), para la transformación a modelos PIMs (PLATFORM-

SPECIFIC MODELS), es factible realizar un estudio y comparación del comportamiento de cada lenguaje. Permitiendo de esta forma, generar un modelo con el cual se pueda realizar la comparación de forma técnica, de lenguajes de transformación de modelos, para finalmente entender las ventajas y desventajas de cada uno de los lenguajes analizados.

1.4 OBJETIVOS

1.4.1 Objetivo General

- Realizar el estudio y comparación de los lenguajes de transformación de modelos QVT y ATL, con el fin de identificar sus fortalezas y debilidades.

1.4.2 Objetivos Específicos

- Determinar las características relevantes de los lenguajes de transformación de modelos QVT y ATL.
- Realizar el desarrollo de un aplicativo en base a los lineamientos de MDA
- Utilizar herramientas que permitan la generación automática de código a partir de modelos PIM (PLATFORM-INDEPENTENT MODEL), con el fin de obtener un modelo PSM (PLATAFORM SPECIFIC MODEL).
- Identificar las diferencias entre los modelos generados utilizando los dos lenguajes de conversión de modelos QVT y ATL.
- Poner a prueba los lenguajes QVT y ATL, mediante el análisis de una implementación en la que se transforme un modelo basado en un módulo de manejo de agendas.

1.5 ALCANCE

El proyecto de tesis abarcará el estudio y comparación de los lenguajes de transformación de modelos, mediante la implementación de un modelo de comparación que permita la evaluación de sus características de forma técnica, con el fin de establecer sus fortalezas y debilidades. En primera instancia realizando una investigación de MDE y MDA, para entender su función y como estas metodologías permiten la incorporación de los lenguajes de transformación de modelos en el desarrollo de software. Adicional se verificara los resultados obtenidos al realizar una transformación en los lenguajes QVT y ATL, llevando un modelo PIM (PLATFORM-INDEPENTENT MODEL), a un modelo PSMs (Platform-specific models).

Finalmente se realizará la comparación de ambas soluciones, en dónde se observará las diferencias entre ambos lenguajes, así como sus ventajas y desventajas.

CAPíTULO 2

2.1 INGENIERÍA DIRIGIDA POR MODELOS – MDE

La Ingeniería Dirigida por Modelos (Model Driven Engineering), MDE, para abreviar, tiene como función incrementar la abstracción en las especificaciones del software y elevar el nivel de automatización en su desarrollo. Esto al utilizar diferentes modelos, cada uno con diferentes niveles de abstracción.

Esto permite obtener un nivel elevado de automatización en el desarrollo ya que se transforman los modelos de alto nivel, a modelos de nivel inferior, y a su vez éstos nuevos modelos pueden ser nuevamente transformados, hasta que se obtenga modelos resultado que puedan ser utilizados para generar código.

Los lenguajes de los modelos son específicos de cierto dominio, por lo cual se los llama lenguajes específicos de dominio (DSL), pudiendo ser éstos de tipo textual o visual.

Los modelos en los cuales se indican la utilización DSL se conocen como modelos específicos de dominio (DSM). Para aquellos sistemas de gran complejidad es común tener varios DSMs. especificados en diferentes lenguajes DSL. En estos casos los modelos hacen referencia entre si y se combinan en la ejecución, para esto es necesario llevar una estructuración del espacio de modelado.

2.1.1 Enfoque de MDE

El objetivo de MDE es el de incrementar la productividad de una empresa, al mejorar la productividad a corto plazo del proceso de desarrollo, al aumentar el valor de los módulos de software disponibles, en comparación a la cantidad de funcionalidad que éstos ofrecen.

Así mismo, está concebido con el propósito de mejorar la productividad a largo plazo del equipo de desarrolladores, mediante la reducción de la velocidad a la que los módulos de un sistema principal se vuelven obsoletos.

Por esto según Engels (Engels, 2007) se consideran los siguientes factores, los cuales pueden ser controlados mediante este tipo de ingeniería:

Personal: el conocimiento vital del desarrollo no debe ser almacenado sólo en la mente de los desarrolladores, esta información se perderá en caso de que existan frecuentes fluctuaciones de personal. Por lo tanto, esta información debe estar fácilmente accesible para los demás miembros del equipo, a más de los creadores iniciales del artefacto software. De ser necesario, este conocimiento debe poder ser transmitido de una forma comprensible para la mayor cantidad de usuarios de esta, esto incluye a los clientes.

Requisitos: el cambio en los requisitos originales de una solución de software a medida que el negocio evoluciona, es un problema común dentro del medio de la ingeniería de software. Este escenario en el que los requerimientos cambian, aun antes de que el sistema esté terminado, es cada vez más común. Con este enfoque se facilita la

actualización de requerimientos, sin tener un grande impacto en el tiempo de actualizar el código relacionado.

Implicación técnica: facilita la integración de nuevos requerimientos, sin la necesidad de deshabilitar el sistema.

Plataformas de desarrollo: El software ya no puede estar atado a una plataforma en específica, las nuevas tendencias impulsan el desarrollo de multiplataforma y la alta portabilidad.

Plataformas de implementación: Migrar una solución de una tecnología a otra a medida que estas están disponibles.

En la mayoría de los casos cuando se habla de MDA o MDE el único objetivo asociado con estos términos es reducir la sensibilidad de los artefactos software, para el cambio en las plataformas de despliegue utilizando un PIM y PSM.

En síntesis, el objetivo del MDE es aumentar tanto la productividad a corto plazo, por ejemplo, la cantidad de funcionalidad entregada por un artefacto de software, y la productividad a largo plazo, por ejemplo, la reducción de la sensibilidad de los artefactos software a los cambios de personal, requisitos, plataformas de desarrollo y plataformas de despliegue.

2.1.2 Proceso de ingeniería de software

Los artefactos (modelos) necesarios en MDE sólo constituyen una parte de lo que se requiere para el proceso de ingeniería. Es importante que exista algún tipo de

proceso, que por lo menos dicte algunas directrices sobre el orden en el que se generarán los modelos objetivos. Para esto existen los procesos macro y micro, que se encuentran asociados a la ingeniería de los diferentes modelos.

Los macro procesos se refieren al orden en que se producen los modelos destino, y cómo se coordinan, en el caso de que un modelo dependa de otro para ser generado, este macro proceso debe indicar cual modelo debe generarse primero.

Los micro procesos definen las instrucciones para la producción de un modelo en particular, es decir todo el proceso con el cual se crea cada modelo (abstracción, modelamiento, validación del modelo, etc.).

2.1.3 Desarrollo de software dentro de MDE

Un proceso de desarrollo de software es una actividad de resolución de problemas, que transfiere un conjunto de problemas en un conjunto de soluciones ejecutables. Estos problemas pueden ser vistos como los requerimientos que definen la solución.

Teniendo así dos tipos de requisitos funcionales y no funcionales.

Funcionales: describe lo que el sistema debe hacer, las necesidades del usuario final que deben ser cubiertas, y como interactuar con otros sistemas.

No funcionales: define los requisitos que no indican ninguna función a realizar por el sistema, ejemplos de esto serían: rendimiento, seguridad, accesibilidad, etc.

Escenarios

Según Kurtev (Kurtev, 2005) se puede definir dos tipos de escenarios en la creación de sistemas de software. La primera opción indica que existen soluciones alternativas para ciertos requisitos. En términos de un proceso de transformación, esta situación puede presentarse como se muestra en la Figura 1.

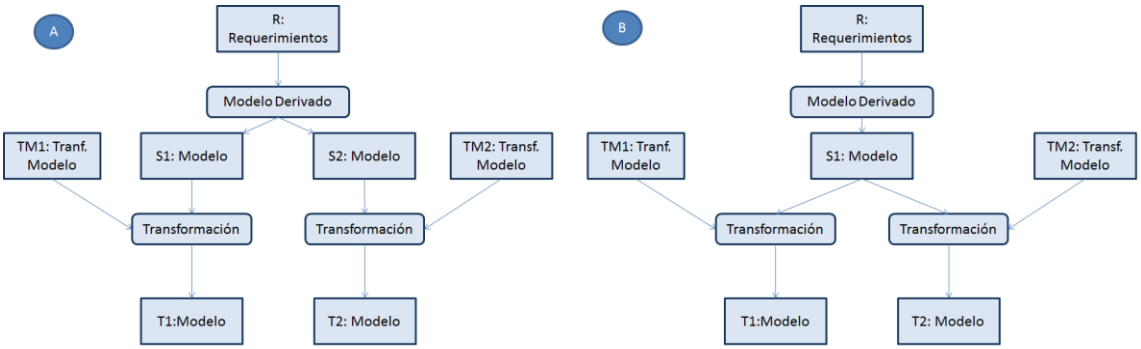


Figura 1. Alternativas de transformación de modelos

Fuente: (Kurtev, 2005)

En la Figura 1a se crean dos modelos de fuentes alternativas (S1 y S2) para los requisitos de (R). Existe una diferencia entre S1 y S2, ya que ambos de modelaron de forma diferente, dando así también lugar a diferentes transformaciones TM1 y TM2.

En la Figura 1b se muestra un escenario diferente, donde un modelo de una sola fuente (S) se puede transformar en dos modelos de destino diferentes (T1 y T2) mediante el uso de diferentes modelos de transformación (TM1 y TM2). En ese caso (eventualmente otros modelos de transformación) de TM1 y TM2 transforman el modelo independiente de plataforma S, a diferentes modelos específicos de plataforma (T1, T2, etc.).

Este proceso de transformación se puede ejecutar las veces que sea necesario hasta obtener el modelo que se busca.

Otro escenario es la composición y descomposición, en este caso se pueden descomponer los requisitos, para a continuación componer una o varias soluciones. Normalmente un sistema complejo existe gracias al conjunto de requisitos provenientes de diferentes partes interesadas, la descomposición de estos requisitos, creando sub-conjuntos viables, es una de las estrategias para hacer frente a la complejidad que presenta el desarrollo de dichos sistemas.

El primer caso se muestra en la Figura 2, si se supone que un conjunto de requisitos iniciales se descomponen en dos sub-conjuntos, y que éstos se basan en subconjuntos que se derivan dos modelos S1 y S2. Hay dos posibilidades para componer dos modelos de origen diferentes en un modelo de destino:

La composición de los modelos de origen antes de que sean transformadas (Figura 2a) o primero realizar la transformación de S1 y S2 para T1 y T2 que a su vez se componen para formar el resultado modelo de destino (Figura 2b).

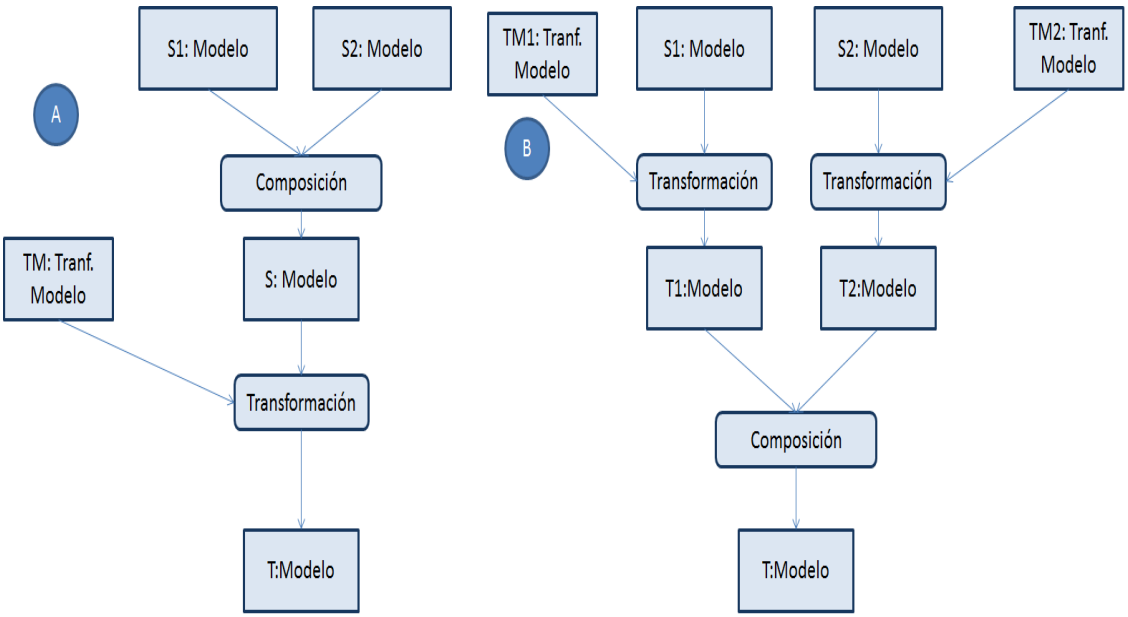


Figura 2. Composición y Descomposición de modelos

Fuente:(Kurtev, 2005)

2.1.4 Evolución de los sistemas de software

Los sistemas de software suelen cambiar durante su tiempo de vida, a esto se le llama evolución de un sistema de software, la cual puede darse de tres formas:

- Mantenimiento correctivo: Indica los cambios realizados en un sistema para resolver los errores de procesamiento, su performance o ejecución.
- Mantenimiento adaptativo: es la migración de una n un sistema de software a otro, como resultado de la evolución de la empresa o del proceso.
- Mantenimiento perfectivo: normalmente se produce al ver la necesidad de mejorar la calidad de un sistema, ya sea mejorando el rendimiento, corrigiendo ineficiencias, etc.

En cuanto a la evolución de un sistema, se consideran tres posibles situaciones:

- Adición
- Sustitución
- Sustracción

Estos escenarios describen el curso de la vida de un componente del sistema, una vez que se añade se puede cambiar durante su tiempo de vida hasta que se retira.

Evolución por adición

En el escenario de la evolución por adición, un nuevo componente de sistema se añade a un sistema de software existente. El ejemplo que se muestra en la Figura 3, tomado de Kurtev (Kurtev, 2005) se ilustra un posible escenario al usar la ingeniería dirigida por modelos. Al instante t1 el sistema inicial se muestra. En el momento t2 se muestra una adición de un modelo de fuente S1 que está compuesto por un modelo de fuente de S ' con S antes de que se transforma en un nuevo modelo objetivo T'. El momento t3 muestra una adición de un modelo de destino T1 al sistema existente T ' resulta en un sistema nuevo T ".

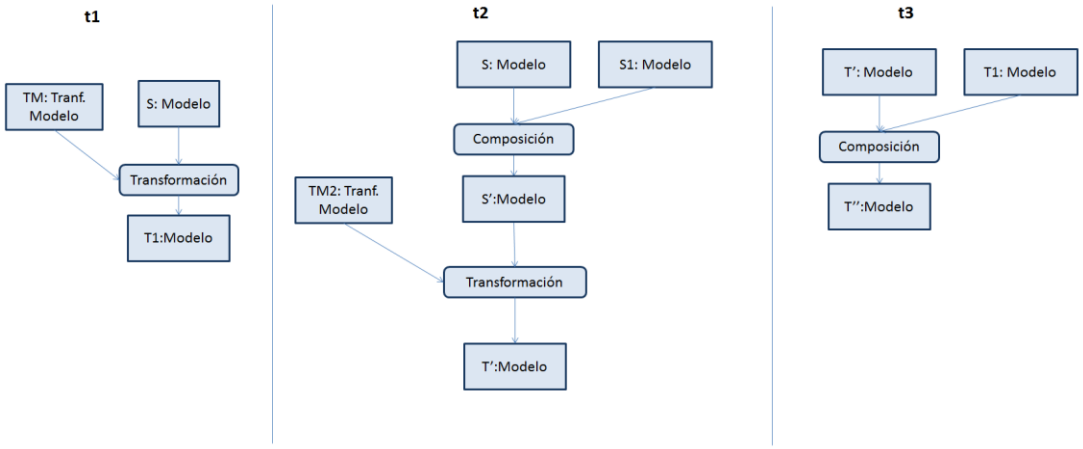


Figura 3. Evolución por adición a través del tiempo

Fuente: (Kurtev, 2005)

En la Figura 3 se visualiza una única adición de la composición. Por supuesto, es posible definir muchos más escenarios basados en la adición de la fuente, ya sea de destino o de transformación de los modelos que utilizan composición o transformación.

Sustitución de un componente del sistema

El mantenimiento correctivo o perfectivo lleva sobre todo a las sustituciones de los componentes del sistema mediante una versión modificada de ese componente. Los cambios en un componente se pueden hacer por la adaptación del modelo de origen, o también por el cambio de la definición de transformación o cambiando directamente el modelo de destino. Modificar un componente al actualizar el modelo de origen puede requerir un cambio de transformaciones propagado, en contraste a las transformaciones sin cambio de estado, que sólo admiten volver a ejecutar una transformación desde cero, después de la ejecución inicial.

El uso de un cambio de transformación por multiplicación permite la sustitución de un componente en tiempo real o con la preservación de los datos de tiempo de ejecución. Un claro ejemplo de este escenario se presenta cuando se cambia la estructura de una base de datos utilizando como entrada un modelo de dominio sin que exista una pérdida de información en la base de datos.

Evolución de los componentes

Dentro de un sistema, la evolución de sus componentes se ha categorizado de la siguiente manera:

• Pulsar.

Este tipo de componente cambia de tamaño durante su vida, teniendo etapas en las cuales es más grande, debido a un incremento en sus funciones, o disminuye, como resultado de una optimización.

• Supernova.

Un Supernova es un componente que en un momento explota en tamaño. Las razones para esto pueden variar, algunas posibilidades son una importante refactorización de todo el sistema o porque era una llamada del durmiente - componente que se definió hace mucho tiempo y en un instante se llena de funcionalidad.

• White Dwarf (enana blanca).

Componente que en un momento mantenía una funcionalidad específica, pero que sin embargo, debido a la evolución de todo el sistema, fue siendo irrelevante y ya no tiene un propósito.

• Red Giant.

Un gigante rojo es un gran componente que tiende a ser muy grande sobre varias versiones. Gigantes rojas tienden a aplicar demasiada funcionalidad y son muy difíciles de re factorizar.

•Estancado.

Un componente estancado es uno que no cambia durante varias versiones del sistema de software al que pertenece. Las razones pueden ser, por ejemplo, código muerto, el buen diseño o porque pertenece a una parte del sistema que no está bajo el cambio.

• Dayfly.

Componentes que fueron creados para realizar pruebas de concepto, normalmente no existen más que en una versión del sistema.

• Persistente.

Componentes que se crearon desde un inicio y se mantienen vigentes hasta la actualidad. Éstos pueden contener código muerto que ningún desarrollador se

atreve a quitar ya que no hay una forma de explicar el propósito del componente.

• Evolución sustractiva

Al final de la vida de un componente, éste se retira del sistema de software, los escenarios en los cuales ocurre esto se les conoce como evolución de sustracción, los cuales son los opuestos a los escenarios descritos para la evolución por adición.

2.2 ARQUITECTURA DIRIGIDA POR MODELOS – MDA

MDA, Model Driven Architecture o Arquitectura Dirigida por Modelos, fue mencionada por primera vez en 2000 en un ensayo de la OMG (Soley, 2000). Con base en este ensayo, miembros de la OMG decidieron formar un equipo de arquitectura para producir una declaración más formal de la MDA. Esta definición formal, pero aún incompleta de la MDA fue presentada en el 2001 en el documento "Model Driven Architecture - Una Perspectiva Técnica" (OMG, 2001). Los miembros de la OMG votaron a favor de establecer MDA como la arquitectura base para las normas de su organización a finales de 2001. En 2003 una definición más detallada de MDA, presentada en el documento "MDA Guide Version 1.0.1" (OMG, 2003), fue adoptado por los miembros de OMG.

2.2.1 Enfoque de MDA

MDA es un enfoque para el uso de los modelos de desarrollo de software. El Model-Driven Architecture prescribe ciertos tipos de modelos que se utilizarán, cómo se pueden preparar los modelos y las relaciones de los diferentes tipos de modelos.

Según la OMG, la arquitectura dirigida por modelos propone el separar la operación de un sistema, según como éste utiliza la plataforma con la cual es definida (OMG, 2007).

De esta forma MDA ofrece un enfoque con el cual la especificación del sistema, no está ligada a la plataforma donde éste reside, para esto, se utiliza otro enfoque que permite dicha especificación.

Como objetivos, MDA tiene:

Portabilidad

Interoperabilidad

Reutilización

2.2.2 Conceptos básicos de MDA

En la siguiente sección se detallan algunos conceptos que definen MDA

Sistema

Un sistema dentro de MDA puede ser un programa, un sistema que está conformado por diferentes partes (personas, empresas, áreas, etc.)

Modelo

Un modelo es la descripción de un sistema mediante la utilización de texto y gráficos, los cuales son adaptados a un propósito específico.

En cuanto a MDA, los modelos representan el funcionamiento, comportamiento y estructura de un sistema en específico, mediante la abstracción de los procesos que involucran el sistema del mundo real.

Se dice que MDA es un lenguaje dirigido por modelos, ya que utiliza éstos directamente para el diseño, construcción, despliegue y mantenimiento de los sistemas.

Puntos de vista de MDA

El Model-Driven Architecture especifica tres puntos de vista sobre un sistema:

- Punto de vista independiente de cálculo.
- Punto de vista independiente de la plataforma.
- Punto de vista específico de la plataforma.

Punto de vista independiente de cálculo.

El punto de vista independiente cálculo se enfoca en el medio ambiente del sistema, así como en los requisitos del sistema, donde los detalles de la estructura y procesamiento del sistema están ocultos o aún por determinar (OMG, 2001).

Punto de vista independiente de plataforma

El punto de vista independiente de la plataforma se centra en el funcionamiento de un sistema al tiempo que oculta los detalles necesarios para una plataforma en particular. Una visión independiente de la plataforma muestra parte de la especificación completa que no cambia de una plataforma a otra. Una vista independiente de la plataforma puede utilizar un lenguaje de modelado de propósito general, o un lenguaje específico para la zona en la que se usará el sistema (OMG, 2001).

Punto de vista específico de plataforma

El punto de vista específico de la plataforma combina el punto de vista independiente de plataforma con un enfoque adicional en el detalle, donde se detalla la utilización de una plataforma específica por un sistema (OMG, 2001; Juan de Lara, 2012) .

2.2.3 Modelos MDA

MDA especifica tres modelos por defecto de un sistema correspondiente a los tres puntos de vista de MDA, además también especifica un modelo de plataforma.

Modelo independiente de cálculo (CIM)

Un modelo independiente de cálculo expresa un sistema sin tomar en cuenta los cálculos que se realizan en él. Los detalles de la estructura no se pueden observar en un CIM o también conocido como modelo de dominio.

Normalmente quien utiliza los CIM son los expertos del negocio, sin necesidad de que éstos tengan el conocimiento de cómo implementar dicha solución.

El CIM permite definir la función de un sistema, sin tener que detallar el cómo va a ser esta funcionalidad implementada, donde su función principal es facilitar la comunicación entre los expertos del negocio, diseñadores y programadores.

Modelo Independiente de Plataforma (PIM)

Utilizando el concepto de punto de vista independiente de plataforma, nacen los modelos PIM. Éstos ya cuentan con la información necesaria para describir como un

sistema va a resolver los diferentes problemas o requerimientos, sin especificar la plataforma donde éste se ejecutará.

Un PIM refleja la estructura de un sistema desde el punto de vista ontológico, es decir detalla la estructura más no los detalles de implementación. El modelo ontológico define que algo es un sistema si cumple con las siguientes condiciones:

- Composición: se tiene que dentro del sistema existe al menos un conjunto de elementos que pertenecen a una categoría especifica (numérica, física, económica, etc.).

- Medio ambiente: dentro del sistema existe un medio ambiente donde los elementos existen, es decir un conjunto de elementos que pertenecen a una categoría.

- Producción: interacción entre los elemento que conforman la composición, para producir un resultado, el cual es consumido por aquellos elementos que se encuentren en el medio ambiente.

- Estructura: define la forma en la cual los elementos interactúa entre sí.

Modelo específico de plataforma (PSM)

El punto de vista especifico de plataforma permite definir el concepto con el cual los PSM son creados, esto implica que un PSM extiende los modelos PIM, incorporando detalles que indican como un sistema utiliza los diferentes componentes que ofrece una plataforma.

Modelo de plataforma

El modelo de una plataforma expone técnicamente la función de esta. También proporciona, para su uso en un modelo específico de la plataforma, los conceptos que representan los diferentes tipos de elementos que se utilizan en la especificación de la utilización de la plataforma por una aplicación (Markus Völter, 2013).

2.2.4 Proceso básico para la construcción de software mediante MDA

En principio, el proceso de construcción se inicia con la definición del Modelo Independiente de Computación o modelo de dominio. Esta CIM es definida por un analista de negocios en colaboración con el usuario de negocios. La CIM se transformará en un modelo independiente de plataforma, donde se incorpora el detalle arquitectónico de la CIM, sin mostrar los detalles de la plataforma utilizada. El PIM resultante tiene que ser dirigida a una plataforma para completar el proceso de generación. Por lo tanto se necesita un modelo detallado de la plataforma. El PSM resultante puede ser una aplicación si se proporciona toda la información necesaria para construir un sistema y ponerlo en funcionamiento. El proceso descrito se visualiza en la Figura 4.

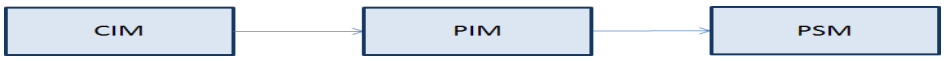


Figura 4. Flujo de Conversión de Modelos

Fuente: (Engels, 2007)

En este proceso, el principal desafío es la transformación entre los diferentes modelos. Los enfoques tradicionales de estas transformaciones, son en su mayoría según

la OGM muy ineficientes, ya que no se utilizan modelos formales, sin modelos formales no es posible definir una transformación formal que pueda ser (parcialmente) automatizada.

2.2.5 Clasificación de los métodos de transformación de modelo

Según la OMG (OMG, 2007), al realizar una transformación de modelos, se deben considerar los siguientes puntos:

- Reglas de transformación
- Ámbito de aplicación de reglas
- Relaciones entre el origen y el destino
- La estrategia de aplicación de reglas
- Reglas de programación
- Reglas de la organización
- Reglas de localización
- Reglas de direccionalidad.

Reglas de transformación

Las reglas de transformación describen cómo los elementos de un modelo de origen deben traducirse en elementos de un modelo de destino. Una regla de transformación se compone de dos partes: un lado de la mano izquierda (LHS – Left hand statement) y un lado derecho (RHS - Right hand statement).

El LHS tiene acceso al modelo de origen, mientras que el lado derecho se expande en el modelo de destino. Ambos pueden ser descritos por variables, patrones, lógica, consultas, etc.

Ámbito de aplicación de reglas

El ámbito de aplicación de reglas permite a una transformación definir el espacio de destino de una transformación. Dando lugar a la creación de restricciones, las cuales pueden estar definidas tanto para los modelos de origen como de destino.

Relación entre el origen y el destino

Cuando se habla de la transformación de modelos, el hecho es que se crea un nuevo modelo de destino sobre la base de la información contenida en el modelo de origen. Además se pueden tener transformaciones en las cuales el objetivo no es la creación de nuevos modelos, si no actualizar aquellos ya existentes. Estos enfoques de actualización pueden ser destructivos, por ejemplo, pueden eliminar elementos.

Estrategia de aplicación de reglas

Es necesario definir el alcance de una regla, ya que pueden existir escenarios donde varios elementos de una fuente cumplen la condición para ejecutar dicha regla, con una estrategia de aplicación de reglas se pueden controlar el orden con el cual estas van a ser aplicadas sobre los diferentes elementos del dominio.

Reglas de programación

En las transformaciones de modelos más complejas, la cantidad de reglas va a ser alta. Por esto, se puede utilizar un mecanismo de programación para determinar el orden en el que se aplican las reglas individuales.

Los enfoques también pueden diferir en la forma en que se seleccionan las reglas, cómo se realiza la iteración de cada regla y si hacen uso de diferentes fases de transformación.

Reglas de la organización

Las reglas pueden estar compuestas y estructuradas de diferentes maneras. Los enfoques de la transformación de modelos principalmente tienen tres áreas de variación: mecanismos de modularidad (reglas de envasado en módulos), los mecanismos de reutilización (definición de reglas basadas en una o más reglas) y estructura organizacional (la organización de normas basadas en el idioma de origen, de destino o de otra organización independiente).

Reglas de localización

Las transformaciones pueden grabar vínculos entre sus elementos de origen y de destino. Estos enlaces pueden ser útiles en la realización de análisis de impacto, la sincronización entre modelos, la depuración basada en el modelo y determinar el destino de una transformación.

Reglas de direccionalidad

Una transformación puede ser en una dirección o bidireccional, esto significa que de un modelo destino se puede modificar el modelo origen, así como el modelo destino es modificado al realizar cambios en el modelo origen.

Las reglas de direccionalidad permiten definir en qué escenario nuestra regla estará ubicada.

2.2.6 Transformación de modelo mediante MDA

El OMG presenta un par de los enfoques que se utilizan para la transformación de los modelos en su guía de MDA (OMG, 2003). Esta clasificación no es completa debido a que se han expuesto los planteamientos de la OMG como de impresión general, mas no están completamente resueltos.

Marcas o Señalización

Para realizar una transformación desde un PIM hacia un PSM utilizando el concepto de marcado de PIM, es necesario en primera instancia elegir la plataforma. A continuación se realiza el mapeo para dicha plataforma, esto se lo hace mediante un conjunto de marcas.

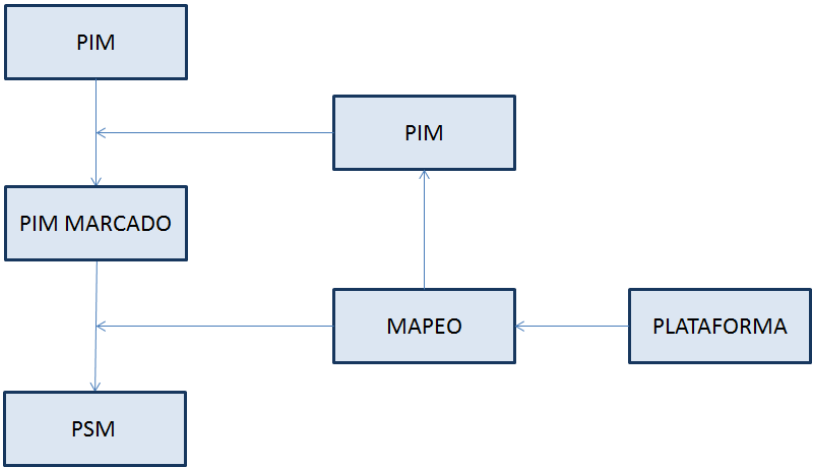


Figura 5. Ejemplo de transformación utilizando marcas

Fuente:(Kurtev, 2005)

En este proceso se utilizan los concepto de left hand side(LHS) y right hand side(RHS), el primero define las marcas sobre los elementos de origen, y el segundo las marcas sobre los elementos de destino, adicional se definen las reglas a las cuales se debe regir el marcado de elementos.

Modelo

En la Figura 6 se muestra un enfoque de transformación de modelos basado en el mapeo de la plataforma de tipos independientes sobre los tipos específicos de la plataforma. Los elementos del PIM son subtipos de los tipos especificados en un modelo tipo independiente de la plataforma. Los elementos en el PSM son subtipos de los tipos especificados en un modelo de tipo dependiente de la plataforma.

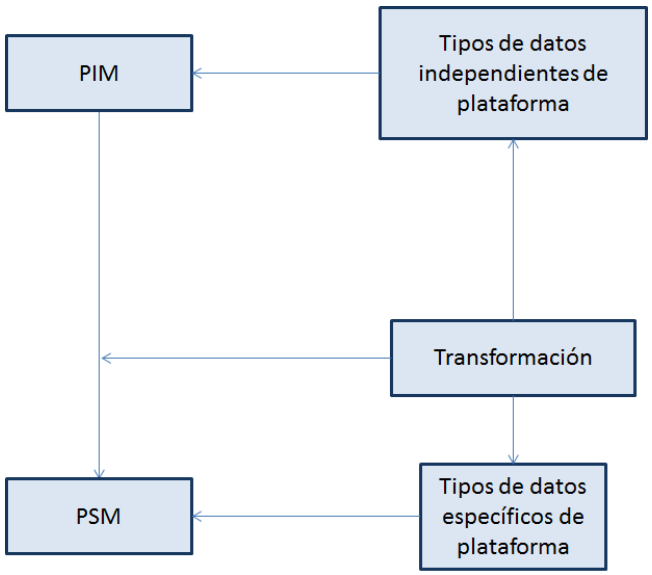


Figura 6. Conversión de PIM a PSM

Fuente: (Engels, 2007)

Las reglas de transformación definidas en la especificación de transformación, asignan directamente los tipos independientes de plataforma sobre los tipos específicos de la plataforma. El LHS de una regla selecciona un tipo independiente de la plataforma, mientras que el lado derecho selecciona un tipo específico de la plataforma. No se necesita una definición del alcance de aplicación de reglas específicas, la asignación es válida para todo el modelo. En la mayoría de los casos, un nuevo modelo de objetivo se creó en base al modelo de código. La estrategia de aplicación de la regla es determinista, a menos que las asignaciones complejas se definen en el que una plataforma de tipo independiente tiene más posibles tipos específicos de la plataforma.

Meta modelo

Existe una sutil diferencia entre el método de asociación subtipo y el enfoque meta modelo. La estructura básica de ambos métodos es la misma. En la Figura 6 se

visualiza el enfoque de transformación de modelos meta modelo. El PIM se expresa en el idioma definido en la plataforma meta modelo independiente. El PSM se expresa mediante un lenguaje dependiente de la plataforma especificada por un meta modelo. La especificación de transformación define la asignación entre los meta modelos.

En principio, este enfoque no se puede clasificar, es demasiado general. Mientras que en el método de asociación subtipo se definen y asignan a cada otro tipo, este enfoque define los conceptos en un meta modelo que se asigna, la diferencia es que en el primer caso se modelan los objetos de dominio basados en el hecho que se puede especificar un PIM. En el segundo caso se define un lenguaje de propósito general (por ejemplo UML) en la que se expresa el PIM.

Aplicación del modelo

Una adición al método de asociación subtipo puede ser el uso de patrones. Los patrones se utilizan para indicar los grupos de elementos en el modelo de fuente y para asignarlos a los grupos de elementos en el modelo de destino. Esta adición sólo cambia la definición del alcance de aplicación de reglas. En lugar de aplicar reglas para todo el modelo, las reglas pueden ser definidas sólo si son aplicables a los patrones preestablecidos.

Otros enfoques

Además de los enfoques presentados por el OMG existe un par de otros métodos que se utilizan en la práctica. Los enfoques directos de manipulación ofrecen una representación del modelo interno. Se llevan a cabo por lo general como un marco orientado a objetos. Los enfoques sobre la estructura se basan en dos fases distintas: la

primera fase se utiliza para crear la estructura jerárquica del modelo de destino, mientras que la segunda fase establece los atributos y referencias en el modelo de destino.

2.2.7 Características deseables de un lenguaje de transformación de modelos

La elección de un enfoque de transformación de modelos en particular depende en gran medida de la situación en la que se pretende utilizar. Los diferenciadores mencionados, todos tienen sus propias ventajas y desventajas. Sin embargo, además de estos diferenciadores, un número de características generales se puede definir que son deseables tener para un lenguaje de transformación de modelos.

MDA (Markus Völter, 2013) recomienda que un lenguaje de transformación de modelos que soporta el desarrollo de software dirigido por modelos, tenga las siguientes características:

- Ser ejecutable.
- Poder aplicarse de una manera eficiente.
- Ser plenamente expresivo, pero inequívoco, de las transformaciones que modifican los modelos existentes, así como crear modelos completamente nuevos.
- Facilitar la productividad del desarrollador con descripciones precisas, concisas y claras.
- El lenguaje debe diferenciar claramente la descripción de las reglas de selección de modelo de la fuente de las normas para la producción del modelo de destino.

- El lenguaje debe ofrecer construcciones gráficas en los casos en que los conceptos representados son más concisos e intuitivos, en forma gráfica en comparación con un textual.
- El lenguaje debe ser declarativo, haciendo implícito cualquier concepto o mecanismos que se puede interpretar de manera intuitiva por el contexto.
- Proporcionar un medio para combinar transformaciones, para formar otros compuestos, que ofrecen al menos los operadores para la secuenciación, la selección condicional y la repetición de las transformaciones.
- Proporcionar un medio para definir las condiciones bajo las cuales se permite la ejecución de una transformación.

2.2.8 Lenguajes de Modelado

Los lenguajes de modelado permiten tener una visión simplificada de un proceso u objeto real, con lo cual el diseño y construcción de software se vuelve más ágil y efectivo. Para esto existen varios lenguajes en los cuales un sistema puede ser modelo, tal como UML, sin embargo dado la naturaleza de este lenguaje, puede ocurrir que no siempre sea el más idóneo para definir un sistema, o al menos solamente una parte de éste, teniendo la necesidad de recurrir a otros lenguajes de modelado como DSL (Domain Specific Language). La sintaxis de dicho lenguaje es un modelo en sí, y esto puede ser visto como el modelo de un modelo, esto es lo que se conoce como un meta-modelo.

Donde el modelo es una instancia del meta-modelo, de esta forma se pueden modelar de muchas formas un sistema, y dependiendo de las necesidades específicas puede ser necesario utilizar uno u otro tipo de modelo.

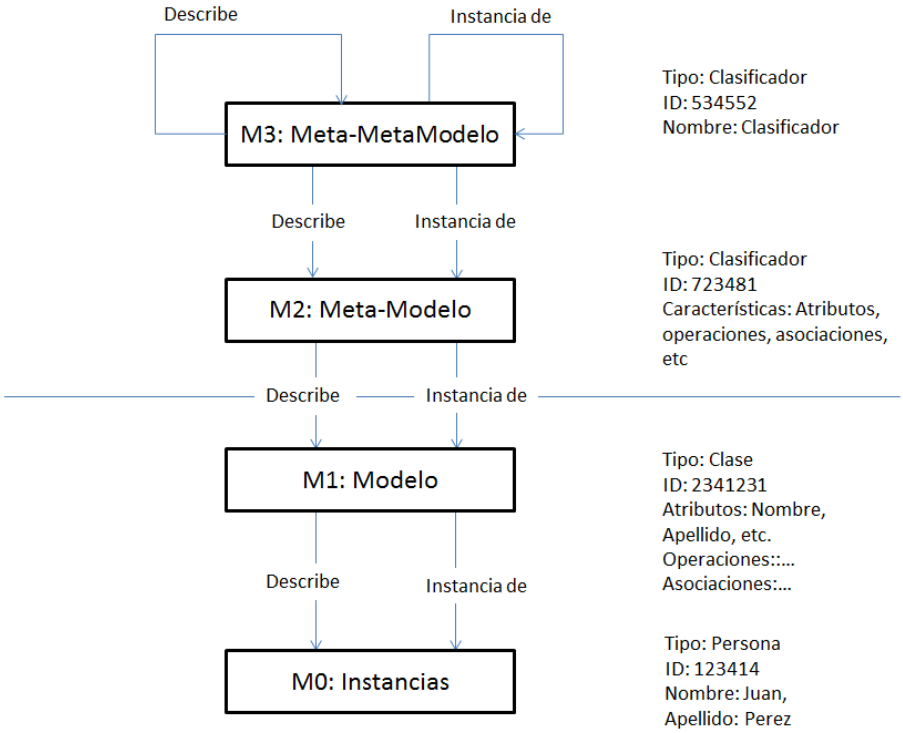


Figura 7. Los cuatro meta niveles de la OMG

Fuente: (Favre, 2010)

En la Figura 7 se puede observar los diferentes niveles de abstracción en los que los modelos operan, los modelos que se encuentran bajo la línea punteada (M1 y M0) son los aquellos que normalmente son trabajados por los desarrolladores, en cambio M2 y M3 son meta modelos, los cuales permiten instanciar los modelos de los niveles inferiores, en este caso, M2 corresponde a la especificación de UML y M3 es la especificación del MOF (el cual a su vez está definido por sí mismo).

En este caso conocer la definición de MOF permitiría, en el caso de ser necesario, realizar una extensión del lenguaje UML, a través de estereotipos, de su definición, con lo cual se crearía un DSL.

2.2.9 Ecore

Dentro de esta tesis se utiliza el framework de modelado de eclipse EMF (Eclipse Modeling Framework), donde existe la librería ecore, ésta es utilizada para representar los modelos UML. Dentro del concepto de MDE, el lenguaje Ecore es una especialización de un lenguaje (UML), por lo tanto es de tipo DSL (Domain Specific Language).

Además es necesario conocer que Ecore es solamente una parte de UML, dada la arquitectura de este DSL, Ecore es un modelo en sí mismo, siendo su propio meta modelo, con esto se puede comprender que no existirán niveles de abstracción mayores.

Es decir Ecore es una especialización de UML, un PIM, con el cual se podrá definir los modelos PSM.

Ecore define 4 clases:

- *EClass*
- *EAttribute*
- *EDatatype*
- *EReference*

Donde EClass representa una clase que ha sido modelada, la cual tiene un nombre, cero o más referencias, y cero o más atributos.

EReference es una asociación en una vía, entre dos Eclasses (una origen y una destino), con un nombre, un límite superior e inferior que definen la cardinalidad de la referencia, la propiedad isContainment, la cual indica si una clase destino está contenida dentro de una clase inicio.

EAttribute representa un atributo modelado.

Finalmente EDataType representa un tipo básico de dato, como String o Integer.

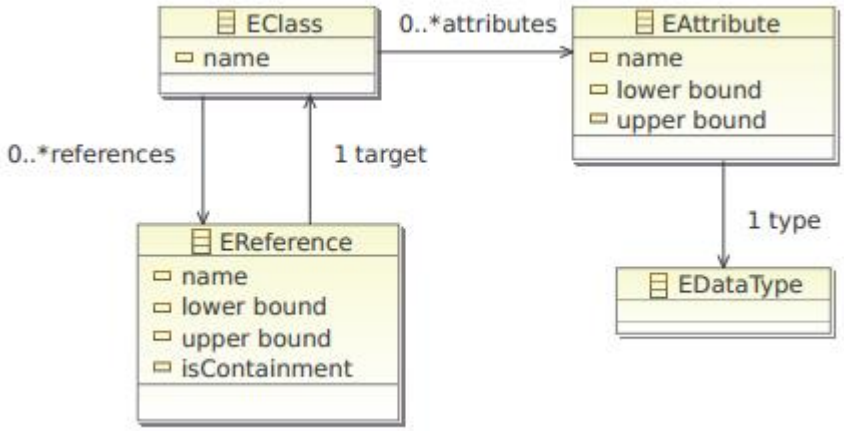


Figura 8. Meta modelo simple de Clases editor EcoreDiagram

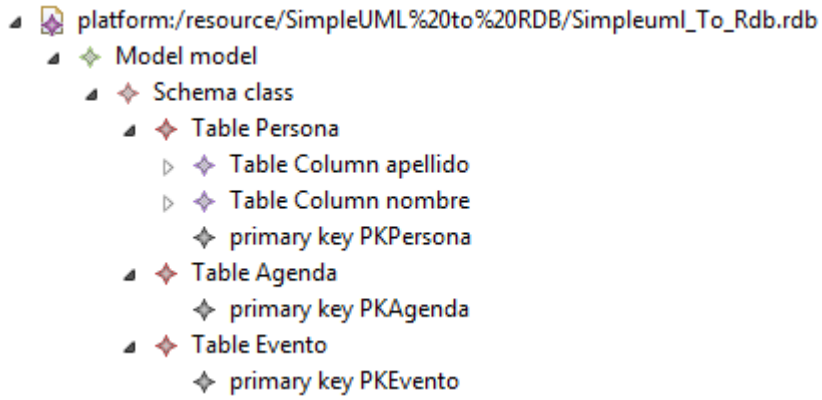


Figura 9. Meta modelo RDBMS editor Ecore

2.2.10 Generación de Código fuente

Una vez que se han realizado las trasformaciones de PIM a PSM, es decir, de modelo a modelo, es necesario llevar a cabo la transformación de un modelo PSM a texto, con lo cual se obtendría el código fuente, el cual será ejecutado sobre la plataforma de destino. Para esto existen varios tipos de generadores de código, los cuales se pueden clasificar de la siguiente manera. (Favre, 2010):

- Code munging: Utiliza como entrada el código y genera varios ficheros de salida, este es el caso de la documentación que puede ser generada a raíz de los comentarios agregados en el código fuente.
- Inline-code expander: Reemplaza secciones del código fuente de entrada, donde existen notaciones especiales, con otro código, por ejemplo: SQLj.
- Mixed-code generation: En este caso el código de entrada es reescrito por el generador de código, su función es igual al de Inline-code expander.

Es necesario considerar que estas herramientas de generación de código, recibirán como datos de entrada, los modelos que se obtuvieron como resultado de

varias transformaciones, siendo éstos lo suficientemente detallados para que el proceso de generar código no requiera una lógica muy compleja.

2.2.11 META-OBJECT FACILITY (MOF)

Meta-Object Facility (MOF) es un estándar del Object Management Group, con el cual UML es definido, esto implica que el MOF es el meta modelo de UML, en otras palabras UML es una instancia del meta modelo MOF.

Definición de meta modelo

Los meta modelo son modelos que abstraen modelos, es decir, un meta modelo es una abstracción de segundo nivel de un modelo, ya que el modelo en si es una abstracción. El objetivo del meta modelo es describir las propiedades con las cuales el modelo es definido. Éstos pueden utilizarse en los siguientes escenarios:

- Como esquemas de datos para ser consumidos por otros procesos.
- Para soportar métodos o procesos particulares.
- Para extender la semántica de un lenguaje

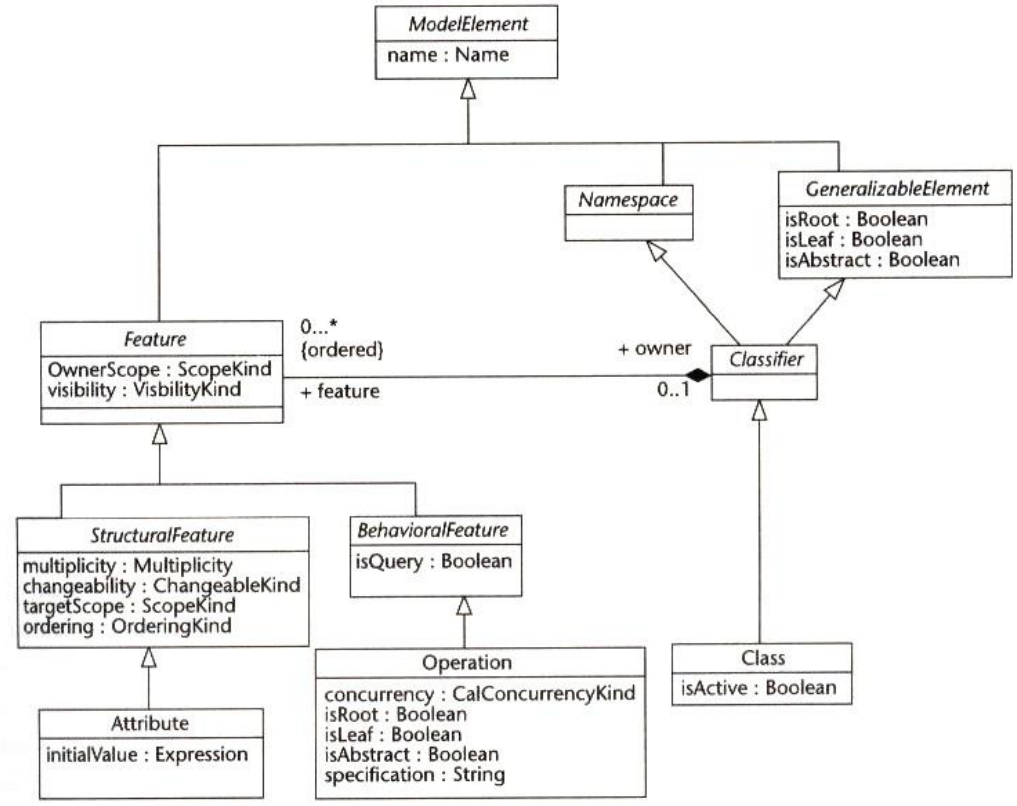


Figura 10. Parte del meta modelo de UML

Fuente: (OMG, 2010)

Para comprender este concepto, se puede utilizar UML, si se define un modelo para un banco, éste contaría con elementos tipo Cuenta, Fondo, Cliente entre otros, el meta modelo de este modelo contaría con elementos tipo Clase, Funciones, Atributos, etc.

Definición de MOF

MOF fue creado para definir de forma estándar UML. En la figura 11, se observa una parte del meta modelo con el cual se instancia UML, en este caso la sección que define la construcción de clases. Como resultado se tiene que este meta modelo permite

definir una sintaxis abstracta de UML, el cual a su vez utiliza la definición de MOF para realizarlo de forma estándar.

Según la definición de MOF de la OMG (OMG, 2007), este grupo de definiciones de forma informal tanto como formal se conoce como MOF meta modelo o "MOF model".

MOF comprende que no existe un único buen modelo, y por esto es más que un lenguaje de modelado(Favre, 2010), como resultado se tiene que es necesario tener varios tipos de modelos para diferentes funciones.

En el caso de tener un grupo de definiciones de modelado, como el caso de un modelo relacional, requiere definir elementos como tabla, columna, clase etc. Por otro lado, en un modelo de clases de UML debe incluir elementos como clase, atributo, operación, asociación, etc. De esta forma, para crear un tipo de modelo especifico, es necesario definir el meta modelo con el cual estas construcciones serán definidas. Para esto MOF proporciona un ambiente donde se pueden crear estas definiciones de modelado, sin importar el ámbito de acción de las mismas.

MOF al ser definido, visto la necesidad de estandarizar UML, utiliza el mismo concepto de clases para describir la forma en la cual se define la construcción de modelos en otros lenguajes. En otras palabras, los meta modelos con los que se define MOF, son en realidad modelos de clase utilizando UML

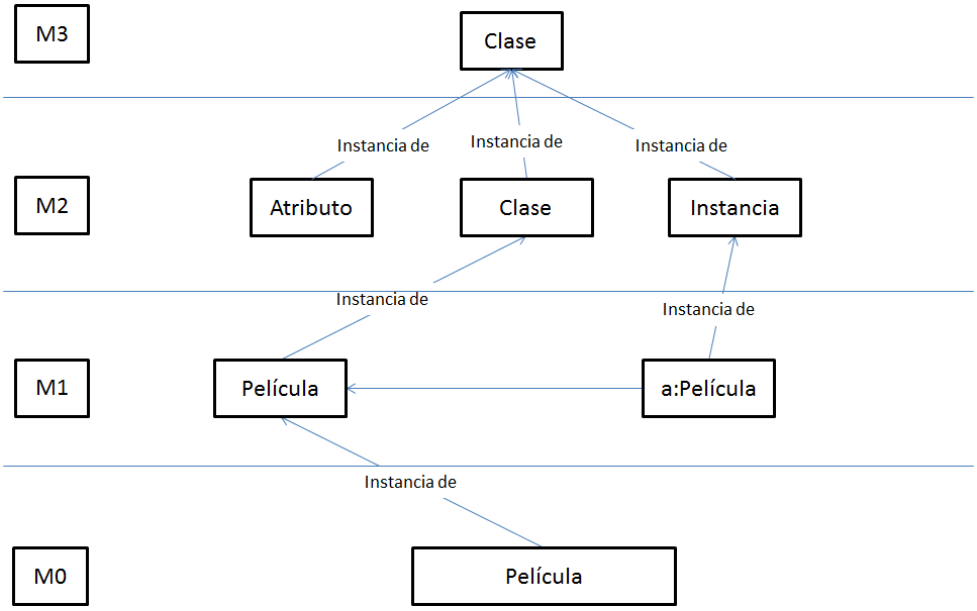


Figura 11. Niveles MOF

Fuente: (OMG, 2007)

Meta nivel MOF

MOF está formado por una arquitectura que se divide en cuatro meta niveles: M0, M1, M2, M3

Tabla 1**.**

Meta niveles de MOF

Meta Nivel	Descripción
M3	MOF
M2	Meta Modelos
M1	Modelos
M0	Objetos y Datos

Fuente: (OMG, 2007)

• **Meta nivel M3**. Este meta nivel es el MOF en sí, define los elementos que se pueden utilizar para construir meta modelos.

• **Meta nivel M2**: En este meta nivel se encuentran los meta modelos, tales como UML, que a su vez ya son una instancia del meta nivel M3.

• **Meta nivel M1**: En el meta nivel M1 se encuentran los modelos en sí, así como en el M2 son instancias de M3, los modelos de M1 son instancias de M2.

• **Meta nivel M0**: Finalmente el nivel donde se definen los Objetos y Datos, instancias de los modelos definidos en el nivel anterior M1.

2.3 QUERY – VIEW– TRANSFORMATION QVT

2.3.1 QVT Visión General

QVT es un lenguaje estandarizado para la transformación de modelos, el cual fue definido por la OMG (Object Management Group).

La definición de QVT lo especifica como un lenguaje de naturaleza híbrida, entre declarativo e imperativo, donde la sección declarativa tiene una arquitectura de dos niveles.

2.3.1.1 Arquitectura declarativa de dos niveles

La arquitectura de la definición declarativa de QVT se divide en dos capas:

- Un meta modelo de relaciones que es amigable con el usuario y un lenguaje que soporta patrones complejos de objetos y plantillas.
- Un meta modelo “Core” y un lenguaje que se define utilizando extensiones de EMOF y OCL.

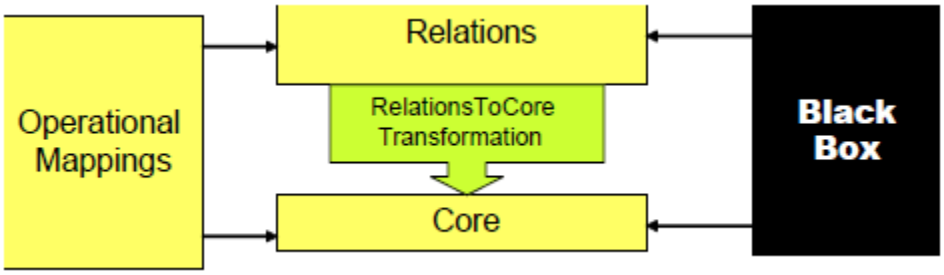


Figura 12. Arquitectura de QVT

Fuente: (Object Management Group, 2008)

2.3.1.2 Relaciones

Las relaciones son una especificación declarativa de las relaciones entre los modelos MOF (Meta-Object-Facility), este lenguaje soporta un complejo nivel de concordancia entre los patrones de los objetos, y sus instancias, para llevar un registro claro de los resultados de la transformación.

2.3.1.3 Core

Lenguaje de modelos que acepta la comparación de patrones sobre variables planas, utilizando para eso condiciones.

Una de sus características más importantes es su simplicidad, sin dejar de ser por esto poderoso.

Un modelo básico puede ser utilizado para implementar directamente el proceso, o simplemente se lo utilizara, como una referencia en otro modelo.

2.3.1.4 Máquina Virtual QVTd

La máquina virtual que se define en QVT es similar al lenguaje Java ™, donde el lenguaje Core es el equivalente al Kava Byte Code, la semántica del lenguaje Core se

puede expresar como la especificación de comportamiento de la Máquina Virtual de Java.

El lenguaje de relaciones tiene el mismo objetivo que el lenguaje Java, y la transformación con Core es cumple la misma función que el compilador.

2.3.1.5 Implementaciones Imperativas

Adicional a las relaciones declarativas y el lenguaje Core que permiten realizar la abstracción de un modelo en dos niveles diferentes, existe la posibilidad de llamar a implementaciones imperativas de una transformación ya sea definida mediante el lenguaje Core o de transformaciones relacionales: utilizando un lenguaje estándar, que es lo que se conoce como asignaciones operacionales, así como implementaciones tipo caja negra utilizando operaciones MOF.

2.3.1.6 Lenguaje de Mapeo Operacional

Lenguaje estándar que proporciona las implementaciones imperativas.

El lenguaje operacional permite la implementación de relaciones más complejas que no pueden ser descritas únicamente utilizando el lenguaje de relaciones.

Una transformación que esté definida únicamente utilizando operaciones de asignación es conocida como una transformación de tipo operativa.

2.3.1.7 Implementaciones de Caja Negra

Según la definición formal de QVT (Object Management Group, 2008), cualquier tipo de relación puede ser descrita utilizando MOF, esto permite crear funciones que encapsulen un proceso específico y su resultado sea consumido por otra operación QVT, esto permite:

- Crear algoritmos complejos en cualquier lenguaje de programación que será ejecutado desde un lenguaje que esté vinculado con MOF
- Utilizar librerías específicas de un dominio en particular para realizar cálculos de dicho dominio. Por ejemplo, matemática, física, química, ingeniería, entre otros, donde se han desarrollado ya grandes algoritmos con bibliotecas específicas que contienen procesos de dichos dominios, los cuales serían complicados o imposibles de implementar utilizando OCL.
- Encapsular procesos específicos de una transformación.

Así mismo la OMG en la definición de QVT (Object Management Group, 2008), recomienda la utilización de una caja negra solamente en el caso de ser necesaria ya que puede ser compleja y exponer el código, si no se controla bien éste, podría repercutir negativamente en la transformación, al alterar valores de forma arbitraria.

Una implementación de tipo caja negra no va a poseer una referencia implícita a una relación, por lo cual solamente se implementan de forma explícita en una relación, la cual servirá para mantener las relaciones entre los elementos de un modelo y la

operación. Una vez ejecutado el proceso de caja negra, el elemento procesado puede ser utilizado en una relación para su procesamiento y transformación.

2.3.1.8 Escenarios de ejecución

Los siguientes escenarios pueden presentarse al momento de ejecutar un modelo en el lenguaje Core o de relaciones:

- Verificar las relaciones ya existentes entre modelos
- Transformaciones unidireccionales.
- Transformaciones bidireccionales.
- Establecer relaciones entre modelos que ya han sido creados.
- Actualización de modelos.
- Definir nuevos objetos o valores a ser creados en un modelo o así mismo eliminar aquellos que ya existen.
- El enfoque de mapeo operacional y caja negra, incluso cuando ejecutados en conjunto con las relaciones, restringe escenarios, permitiéndose sólo la especificación de transformaciones en una sola dirección.

Las transformaciones bidireccionales sólo son posibles si una implementación operativa inversa es proporcionada por separado, sin embargo, todas las otras capacidades definidas anteriormente están disponibles con las ejecuciones imperativas e híbridas.

2.3.2 Meta modelos MOF

La especificación de QVT define los siguientes paquetes para los lenguajes definidos dentro de QVTCore:

- QVTBase, el cual define una estructura estándar para las transformaciones.
- QVTRelation, que utiliza expresiones de patrones de diseño, mismos que son definidos en el paquete QVTTemplateExp.
- QVTOperational, al utilizar el mismo *framework* que QVTRelation, se convierte en una extensión de éste, agregando el uso de expresiones imperativas definidas en el paquete ImperativeOCL.

Todos los paquetes de QVT están basados en el paquete EssentialOCL del OCL 2.0.

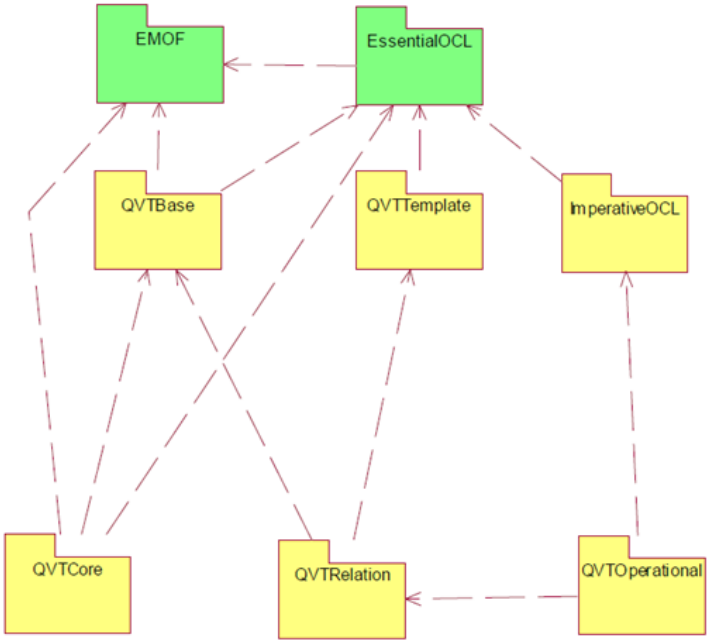


Figura 13. Dependencias de QVT

Fuente: (Object Management Group, 2008)

2.3.2.1 Transformaciones y Modelos

En el lenguaje de relaciones, una transformación entre dos modelos (origen y destino) se especifica como un conjunto de relaciones que se deben establecer entre ambos modelos para que se considere exitosa. Para las transformaciones se utiliza un modelo origen, el cual debe estar especificado en un metalenguaje, el mismo que define la estructura del modelo, un ejemplo de transformación sería la siguiente:

transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {}

En esta declaración "umlRdbms", se tiene un modelo de tipo "uml" y uno de tipo "rdbms", los cuales se encuentran definidos en los meta modelos SimpleUML y SimpleRDBMS respectivamente.

Nota: En una transformación se puede invocar un proceso para que verifique la consistencia entre ambos modelos o para forzarla en el caso que ésta no se cumpla.

2.3.2.2 Dirección de ejecución de transformaciones

Cuando en una transformación se invoca un proceso para forzar la consistencia entre los modelos origen y destino, ésta se ejecuta en una dirección específica, primero se selecciona el modelo destino, el cual puede estar vacío o contener información previa a la ejecución de la transformación.

Primeramente la transformación verificara si existe la consistencia entre ambos modelos, para aquellos que la regla no se cumpla, la transformación intentara ajustar el modelo destino para forzar esta relación, ya sea creando, borrando o modificando el modelo destino, ajustándolo así a la transformación.

2.3.2.3 Relaciones y Dominios

Las relaciones en el contexto de una transformación, definen las condiciones que deben cumplirse en los modelos a transformar (origen y destino)

Un dominio especifica el ámbito en el cual se va a trabajar, como UML, y sobre el cual se declara un patrón, el cual puede ser visto como un objeto del dominio, mismo que deriva sus propiedades y restricciones del dominio.

En el siguiente ejemplo se muestran dos dominios, los cuales están especificados para UML y RDBMS, cada uno con un patrón simple, un paquete y un esquema con un nombre, ambos ligados a la variable pn, lo cual implícitamente define que ambas propiedades tendrán el mismo valor:

```
relation PackageToSchema /* mapea cada paquete del dominio uml a un esquema del dominio rdbms */
{
domain uml p:Package {name=pn}
domain rdbms s:Schema {name=pn}}
```

2.3.2.4 Clausulas When y Where

Dentro de una relación se puede definir dos tipos de cláusulas, where y when, las cuales permiten definir condiciones que deben cumplirse dentro de la transformación.

En el siguiente ejemplo se define la relación ClassToTable, donde, la cláusula when especifica la condición que ClassToTable se realizara siempre y cuando PackageToSchema se realice.

La cláusula where define las condiciones que deben satisfacer todos los elementos de los modelos que intervengan en el proceso, ejemplo:

```
relation ClassToTable /* mapea cada clase persistente a una tabla */
{
domain uml c:Class
  {namespace = p:Package {},kind='Persistent',name=cn}
domain rdbms t:Table {schema = s:Schema {},name=cn,column = cl:Column
                                                {name=cn+'_tid',type='NUMBER'},
      primaryKey = k:PrimaryKey {name=cn+'_pk',column=cl}
}
when {PackageToSchema(p, s);}
where {AttributeToColumn(c, t); }}
```

2.3.2.5 Relaciones de alto nivel (top-level)

Dentro de una transformación se pueden encontrar dos tipos de relaciones: top-level y non-top-level. Este tipo de relaciones permiten definir dos escenarios, en el primero (top-level) la ejecución de la transformación exigirá que todas sus relaciones se cumplan, mientras que en el segundo caso (non-top-level), las relaciones exigen que se cumplan solamente cuando son invocadas directamente.

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
top relation PackageToSchema {…}
top relation ClassToTable {…}
relation AttributeToColumn {…}}
```

En el ejemplo presentado, ClassToTable y PackageToSchema son ambas relaciones de alto nivel, mientras que AttributeToColumn es una relación non-top-level.

2.3.2.6 Enforce y Check

Las clausulas checkonly y enfoce cumplen la siguiente función:

- Checkonly: Verifica solamente si la relación existe.
- Enforced: Obliga a que la relación se cumpla (borrando, creando o actualizando elementos del modelo).

```
relation PackageToSchema /* mapea cada paquete en un esquema */
{
checkonly domain uml p:Package {name=pn}
enforce domain rdbms s:Schema {name=pn}
}
```

En este caso si se ejecuta en la dirección de uml, solamente se verificará si existe un esquema creado con el mismo nombre de cada paquete, en el caso de no existir solamente alertará un error de inconsistencia de modelos.

Si se ejecuta en dirección de rdbms, como se está forzando con la cláusula enforce, en el caso de que se encuentre un paquete para el cual no exista un equivalente esquema, la transformación lo creará.

Una posibilidad adicional en este último caso sería que se encuentre un esquema para el cual no existe un paquete correspondiente, en esta situación el esquema sería borrado, para de esta forma forzar la consistencia entre ambos modelos.

2.3.2.7 Expresiones Object Template

Las expresiones object template permiten definir patrones que deben cumplir los elementos de un modelo que va a entrar al proceso de transformación, en el caso de ClassToTable se definen varios object templates uno de éstos es el siguiente:

```
domain uml  c:Class { namespace = p:Package {},
kind='Persistent',
name=cn}
```

En este ejemplo, se tiene un object template en el dominio de uml, este patrón va a ligar todas las propiedades definidas en la expresión ("c", "p", and "cn"), la variable de dominio "c" de tipo Class. La variable "p" ya se encontrará ligada dentro de la transformación gracias a la expresión PackageToSchema(p, s) de la cláusula when. La sección kind=’Persistent’ permite eliminar cualquier objeto que no cumpla esa condición.

Para el caso de la variable ‘cn’ pueden suscitarse dos escenarios, en el caso de que la variable ya posea un valor, se filtraran todos los objetos que no posean en su propiedad name el mismo valor que tiene asignado “cn”, o, si la variable se encuentra vacía, en ese caso e l valor que se encuentre en la propiedad name del objeto se asociará a la variable “cn”, misma que podría utilizarse en otro dominio para trasladar ese valor a ese nuevo dominio.

La expresión "namespace = p:Package {}" va a buscar aquellas clases donde su propiedad "namespace" no tenga una referencia a un paquete nulo. Así mismo, la variable "p" hará referencia a un paquete, sin embargo como a "p" se le crea una referencia a un paquete en la cláusula when, el patrón solamente buscará aquellas clases que su "namespace" haga referencia al paquete asociado a la variable "p".

Nota: el conjunto de last tres variables: "c", "p" and "cn", conforman lo que se conoce como tupla, donde cada conjunto de estas tres variables que realicen las asociaciones con éxitos representaran una referencia valida.

Un object template expresión también permite la creación de objetos en el modelo destino según las reglas previamente vistas.

2.3.2.8 Cambio en la propagación.

Para realizar un cambio en la propagación de la transformación, se debe reescribir el código teniendo en cuenta que el modelo origen ahora se convertirá en destino, y el destino en origen, esto se lo puede realizar siempre y cuando la semántica sea consistente con aquella definida por el lenguaje relacional.

2.3.2.9 Operaciones de caja negra y relaciones

Una relación puede tener una implementación de caja negra operacional para forzar un tipo de dominio específico. Este tipo de implementaciones son invocadas al momento en que una relación entre dos modelos no se cumple, en ese momento, la función que se ejecuta es la encargada de ajusta el entorno para que exista una relación exitosa entre ambos modelos.

Las relaciones que sean implementadas con el uso de mapeos operacionales de caja negra tendrán las siguientes restricciones:

- El dominio debe ser primitivo o contener solamente un simple object template.
- Las condiciones when – where no deben definir variables.

2.3.2.10 Semántica

Para entender la descripción de la semántica, se puede observar a una relación con la siguiente estructura abstracta:

```
Relation R
{
Var <R_variable_set> //
[checkonly | enforce] Domain:<typed_model_1>
<domain_1_variable_set> // subset of <R_variable_set>
{
    <domain_1_pattern> [<domain_1_condition>]
}
...
[checkonly | enforce] Domain:< typed_model_n>
<domain_n_variable_set> // subset of <R_variable_set>
{
<domain_n_pattern> [<domain_n_condition>]
} // n >= 2
```

[when <when_variable_set> <when_condition>]

[where <where_condition>]

}

(Object Management Group, 2008)

2.3.2.11 Semántica de los patrones

La semántica de la especificación del patrón de construcción que está soportado por el lenguaje relacional se muestra en la siguiente figura.

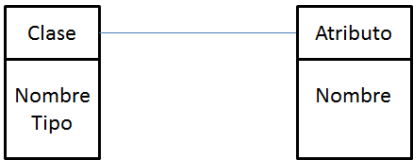


Figura 14. Semántica relacional

Fuente: (Object Management Group, 2008)

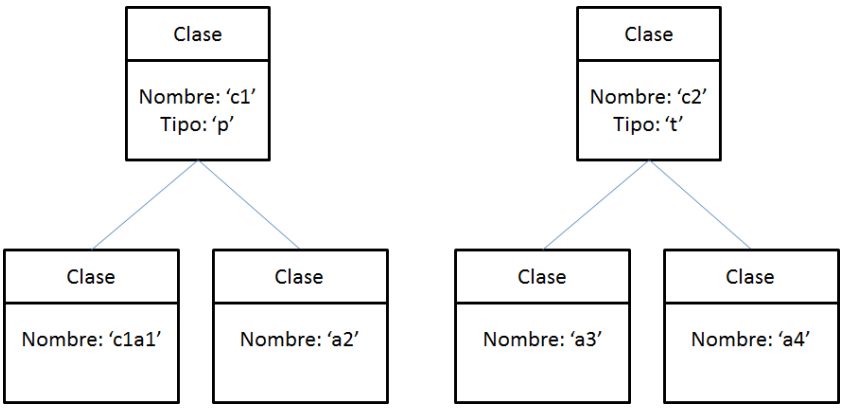


Figura 15. Ejemplo de instancia según la semántica relacional

Fuente: (Object Management Group, 2008)

2.3.2.12 Estructura de los patrones

Dentro de esta definición, se tiene que un patrón tendrá la siguiente estructura abstracta:

Pattern ={e1: <classname1>, e2: <classname2> …. en:<classnameN>

l1 : <assoc1> (ei, ej) …. lm:<assocM>(eu, ew) where <predicate>}

Un patrón puede ser visto como un grafo en el cual cada uno de sus elementos e1, e2,…the tipo <classname1>, <classname2>…<classnamen>, son los nodos, y los links l1, l2,…lm son las aristas.

El predicado es una expresión booleana que puede hacer referencia a cualquier elemento del patrón o variables fuera de éste.

Los patrones son utilizados para encontrar sub-grafo dentro de un modelo. Un sub-grafo que esté formado de objetos (o1, o2,..., on) será relacionado con un patrón, cuando las siguientes reglas se cumplan:

• on es del tipo <classnamen> o uno de sus subtipos

• Cada objeto está asociado a otro a través de un <assoc1>

Una vez que se han establecido las relaciones entre los sub-graphs y el patrón, cada oi tendrá una correlación a cada elemento ei, por ejemplo en el siguiente código:

Pattern

{ c1: Class, a1: Attribute l1: attrs (c1, a1) where c1.name = X and a1.name = X + Y }

Las variables X y Y corresponde al predicado, estas son las que definen la búsqueda del sub-graph, encontrando solamente 1, cual sería <c1, c1a1>. Esta relación establece una referencia X = c1 y Y = a1.

2.3.2.13 Colecciones dentro de patrones

Dentro de un patrón se puede utilizar colecciones de cualquier tipo soportado por el OCL: Set, OrderedSet, Bag o Sequence.

Ejemplo:

Pattern {

c1: Class, a1: Set(Attribute)

l1: attrs (c1, a1)

where TRUE }

En este caso los dos sub-graphs <c1, {c1a1, a2}> y <c2, {a3, a4}> del modelo en la figura 11 entran dentro de este patrón.

2.3.2.14 Sintaxis y semántica abstractas

El meta modelo relacional está compuesto de tres paquetes: QVTBase, QVTTemplate, and QVTRelation

2.3.2.15 QVTBase

Este paquete contiene una serie de conceptos básicos que son heredados de las definiciones EMOF y OCL, como lo son las estructuras de las transformaciones, sus reglas, y los modelos de entrada y salida, adicionalmente introduce el concepto de utilizar un patrón como un conjunto de predicados sobre variables en las expresiones OCL. Estas clases son extendidas en los paquetes específicos del lenguaje, las cuales proveen la semántica requerida.

2.3.2.16 Transformaciones

Una transformación se define como un conjunto de modelos de entrada que van a ser transformados en un conjunto de modelos de salida. Se puede decir que una trasformación es a su vez un paquete y una clase, como paquete contiene la definición

del espacio en el cual va a trabajar, para que las reglas que se encuentran dentro de él puedan ser ejecutadas, y como clase posee una serie de propiedades y operaciones.

Superclass

Package

Class

Associations

modelParameter: TypedModel [*] {composes}

Conjunto de modelos y sus tipos, que participaran en la transformación.

rule: Rule [*] {composes}

Conjunto de reglas de la transformación.

ownedTag: Tag [*] {composes}

Conjunto de tags asociados a la transformación.

extends: Transformation [0..1]

Una transformación puede ser extendida de otra transformación.

/ownedType: Type [0..*] {composes, ordered} (from Package)

Especifica los tipos definidos por la transformación.

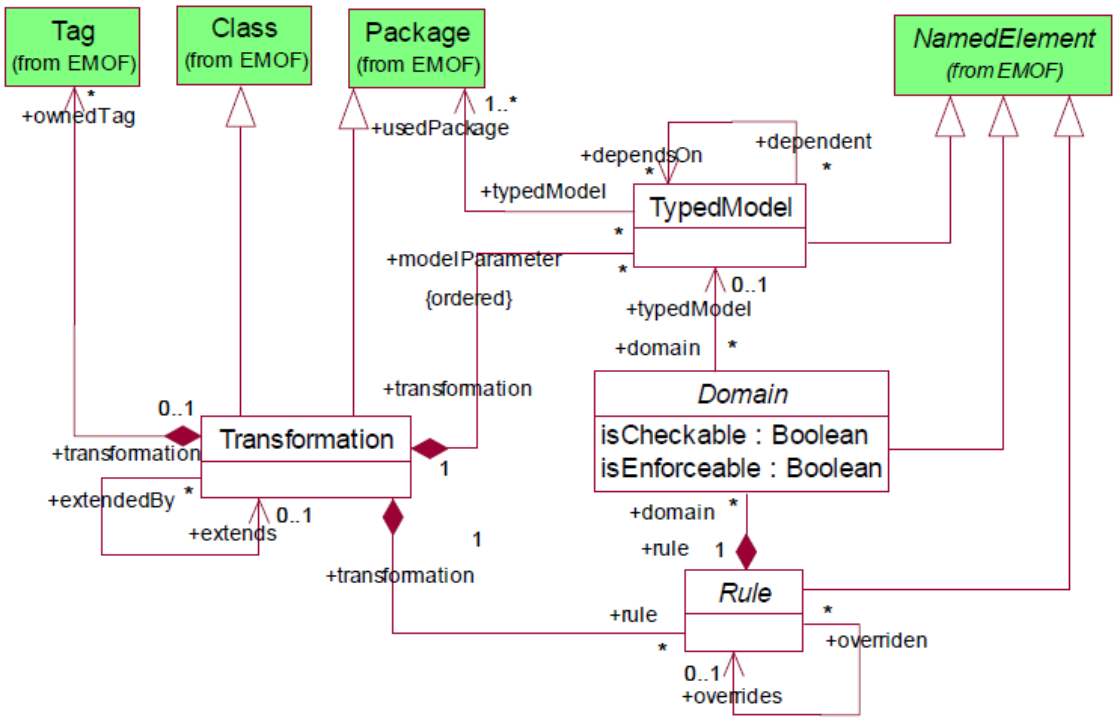


Figura 16. Paquete base de QVT, transformaciones y reglas

Fuente: (Object Management Group, 2008)

TypedModel

Un typed model especifica el tipo de cada elemento que debe contener un modelo para ser transformado.

Superclasses

NamedElement

Associations

transformation: Transformation [1]

La transformación que posee el *typed model*.

usedPackage: Package [1..*]

El meta modelo que especifica los tipos utilizados por los modelos que sean del tipo definido por el *typedmodel.*

dependsOn: TypedModel [*]

Conjunto de *typedmodels* con los cuales este *typedmodel* guarda dependencia.

Dominio

El dominio (domain) especifica el conjunto elementos de un typedmodel que van a ser utilizados por una regla. En este caso domain se define como una clase abstracta, la cual al ser heredada permite definir el mecanismo por el cual se especifican los elementos de un dominio, el cual se puede especificar mediante diferentes métodos (un grafo, un patrón, un conjunto de typedvariables).

Un dominio puede ser definido como checkable o enforceable.

Superclasses

NamedElement

Attributes

isCheckable : Boolean

Indica si el dominio es *checkable*

isEnforceable : Boolean

Indica que el dominio es *enforceable*

Associations

rule: Rule [1]

La regla que rige el dominio.

typedModel: TypedModel [0..1]

El *typedmodel* el cual contiene la declaración de los tipos de cada elemento de los modelos del dominio.

2.3.2.17 Rule

Una regla (rule), permite especificar como los elementos especificados dentro del dominio se relacionan entre sí, así como definir, la forma en la cual los elementos de un dominio se computan hacia otro dominio.

Rule es una clase abstracta, sus subclases son las responsables de definir como estas relaciones se establecen.

Superclasses

NamedElement

Associations

domain: Domain [*] {composes}

El dominio sobre el cual la regla actúa.

transformation: Transformation[1]

La transformación que posee la regla

overrides: Rule [0..1]

La regla que sobrescribe.

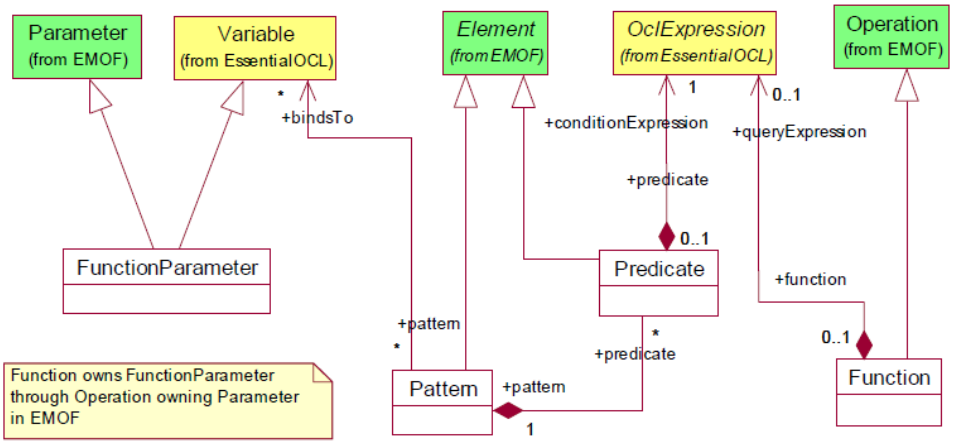


Figura 17. Paquete básico de QVT, Patrones y funciones

Fuente: (Object Management Group, 2008)

2.3.2.18 Función

Una función tiene como objetivo producir el mismo resultado cada vez que sea invocada con los mismos argumentos, se las considera libres de efectos secundarios por esta razón. Adicionalmente estas pueden ser especificadas como una expresión de OCL, o tener su propia implementación tipo caja negra.

Superclasses

Operation

Associations

queryExpression: OclExpression [0..1] {composes}

La expresión OCL con la cual esta función está especificada, en el caso de omitir esta referencia se sobreentiende que la función tiene su propia implementación.

Predicado

Un predicado es una expresión de tipo booleana, la cual debe encontrarse definida dentro de un patrón. Se encuentra definido dentro de una expresión OCL,

misma que puede hacer referencia a variables que se encuentren dentro del patrón, con el cual el predicado es instanciado.

Superclasses

Element

Associations

conditionExpression: OclExpression [1] {composes}

La expresión OCL que define el predicado.

pattern: Pattern [1]

Patrón que posee el predicado.

Patrón

Un patrón es un conjunto de declaraciones de variables y predicados, las cuales al ser evaluadas dentro de un modelo, se obtiene como resultado un conjunto de asociaciones para las variables.

Superclasses

Element

Associations

bindsTo: Variable [*]

Conjunto de variables a ser evaluadas.

predicate: Predicate [*] {composes}

Conjunto de predicados que deben ser evaluados como verdaderos para que las relaciones entre las variables sean establecidas.

2.3.2.19 QVTTemplate

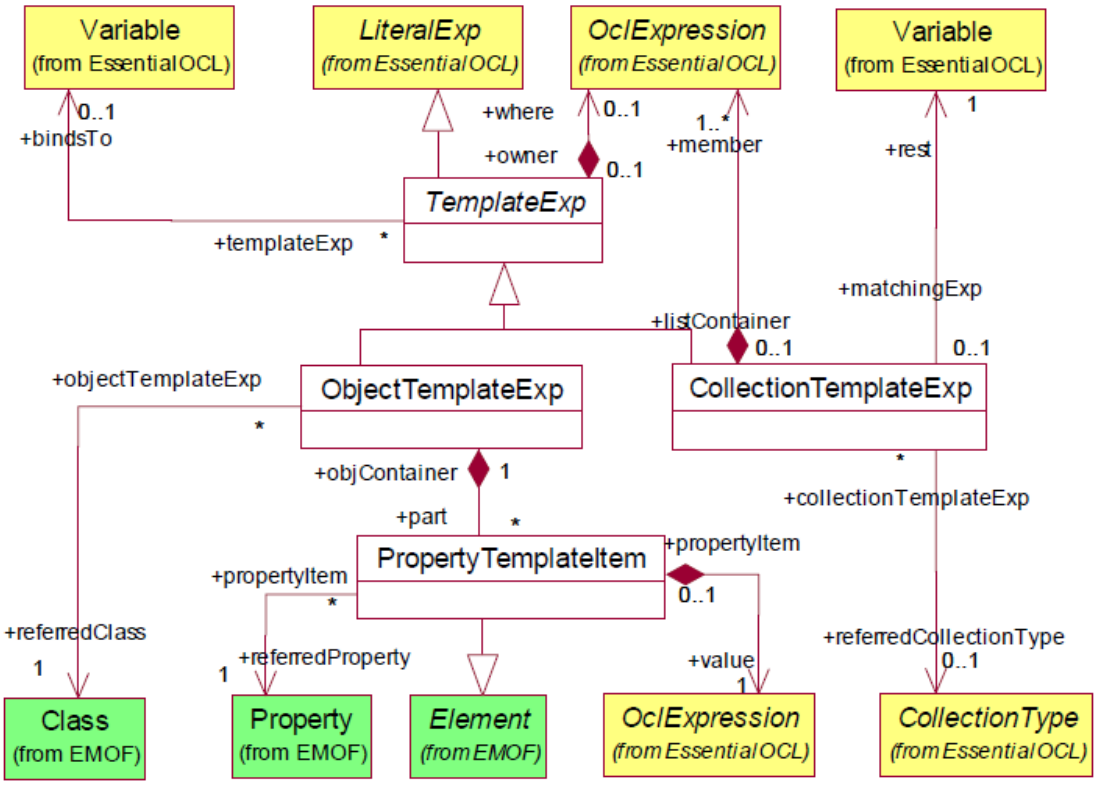


Figura 18. Paquete de plantillas de QVT, QVTTemplate

Fuente: (Object Management Group, 2008)

TemplateExp

Un template especifica un patrón el cual debe establecerse en un modelo a ser transformado, así las variables de dicho modelo pueden ser utilizadas en cualquier parte de la transformación.

Superclasses

LiteralExp

Associations

bindsTo: Variable [0..1]

La variable a la el elemento del modelo hará referencia.

where: OclExpression [0..1] {composes}

Una expresión OCL que debe cumplirse para el templeate.

ObjectTemplateExp

Las expresiones de tipo *object template* como se vio previamente, especifican un tipo de patrón de un objeto de un modelo.

Superclasses

TemplateExp

Associations

referredClass: Class [1]

La clase EMOF que especifica cuáles son los tipos de los objetos que serán buscados por la expresión.

part: PropertyTemplateItem [*] {composes}

Expresión que especifica las condiciones que deben cumplirse por el objeto que esté siendo evaluado para pertenecer a este tipo de patrón.

2.3.2.20 QVTRelation Package

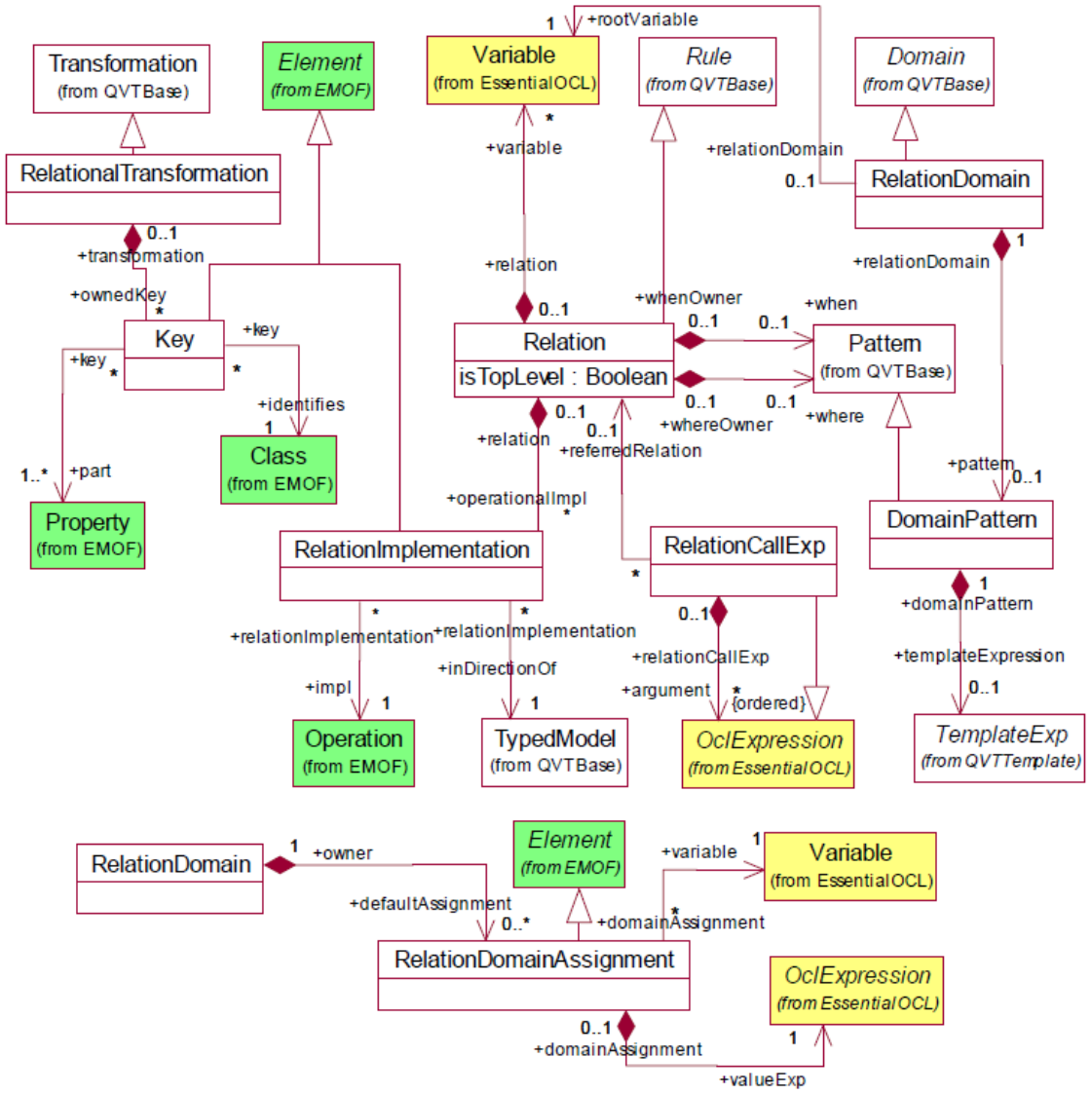


Figura 19. Paquete relacional de QVT

Fuente: (Object Management Group, 2008)

El paquete QVRelation está compuesto de los siguientes componentes: RelationalTransfomation: Especialización de una transformación, misma que se encuentra representando una transformación definida en el paquete relacional (QVT-Relation).

Relation: Define el comportamiento de una transformación, misma que es una clase concreta que hereda sus características de Rule. Permite definir una relación que existirá entre el modelo origen y destino

RelationDomain: Clase concreta de la clase abstracta *Domain*, misma que sirve para definir el dominio sobre el cual la relación tendrá efecto.

DomainPattern: Subclase de la clase *Pattern*, la cual permite especificar un grafo conformado por expresiones tipo template (*object template expresión*, *collection template expresions*, *template expresión*).

Key: Identificador único de cada instancia de una clase en el contexto de un dominio, similar a la utilización de llaves primarias en bases de datos.

RelationImplementation: Implementaciones específicas que permiten forzar una transformación dentro de un dominio.

RelationCallExp: Se utiliza para definir la invocación a una relación específica. Adicional se tiene que la librería estándar de QVT para el paquete relacional es la misma de OCL.

2.3.3 Sintaxis Concreta

La siguiente sección muestra la sintaxis tanto grafica del lenguaje Relacional de QVT según su especificación.

2.3.1.1 Sintaxis grafica

La sintaxis grafica del lenguaje relacional utiliza la especificación de UML, extendiéndola mediante la utilización de anotaciones, en la siguiente imagen se muestra

una transformación *UML2Rel* que va de un modelo de clases UML a un modelo relacional con tablas y columnas.

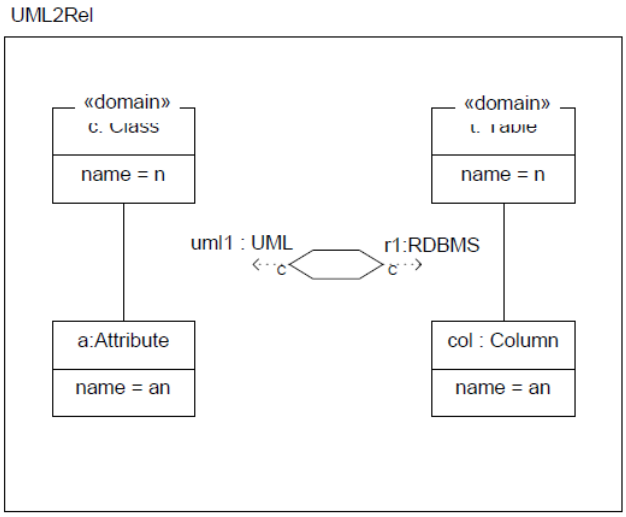


Figura 20. Relación entre una clase UML y una tabla relacional

Fuente: (Object Management Group, 2008)

El siguiente código corresponde a la sintaxis textual del grafico anterior:

relation UML2Rel {

checkonly domain uml1 c:Class {name = n, attribute = a:Attribute{name = an}}

checkonly domain r1 t:Table {name = n, column = col:Column{name = an}}}

La cláusula where se grafica mediante una caja como indica la siguiente figura:

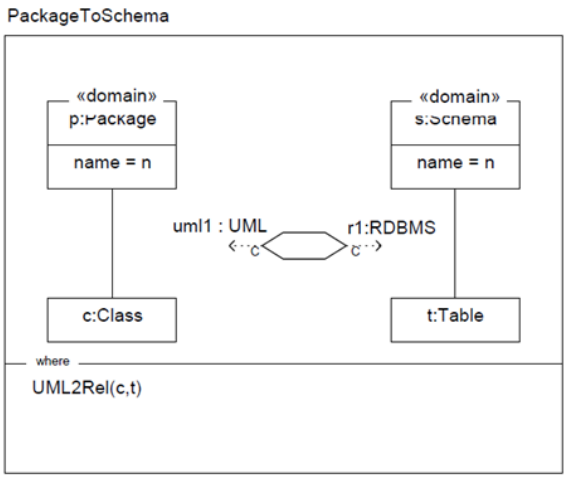


Figura 21. Ejemplo de clausula where

Fuente: (Object Management Group, 2008)

Las condiciones se incorporan mediante la utilización de comentarios de UML, en la siguiente imagen se muestra como los objetos *col* y el patrón de UML con condicionados mediante un comentario.

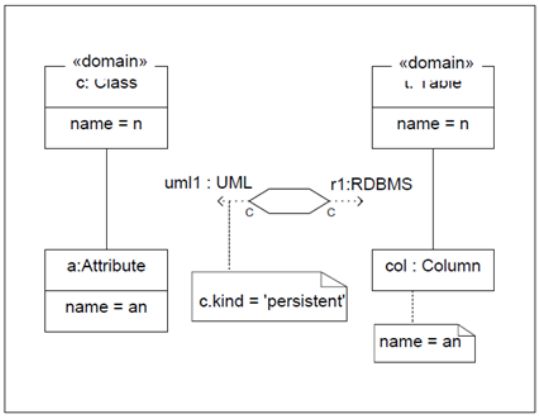


Figura 22. Ejemplo de transformación con condiciones

Fuente: (Object Management Group, 2008)

En la siguiente imagen se muestra una relación entre los objetos, en este caso se relaciona el valor de la *totcols* con el valor del atributo a de la clase.

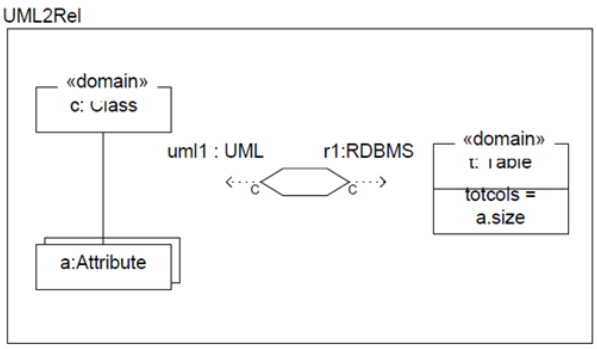


Figura 23. Ejemplo de set

Fuente: (Object Management Group, 2008)

El siguiente ejemplo muestra un patrón, el cual al ser evaluado para un objeto, éste debe cumplir la condición que no existan atributos ligados a la clase c, esto mediante la utilización de la anotación {not}

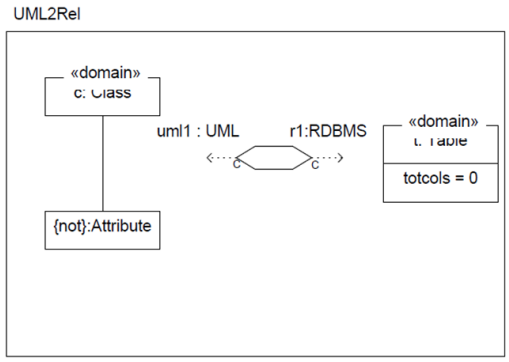


Figura 24. Ejemplo utilizando not

Fuente: (Object Management Group, 2008)

El código de la figura anterior sería el siguiente:

relation UML2Rel {

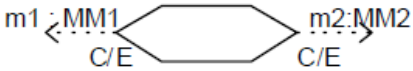
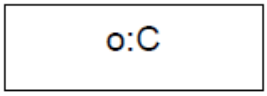
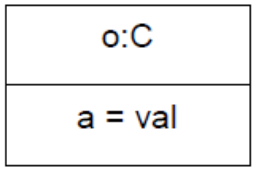
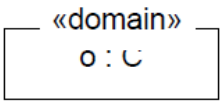
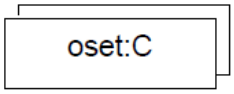
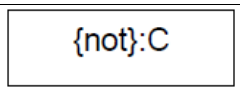
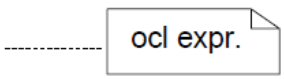
checkonly domain uml1 c:Class {attribute = Set(Attribute){}}{attribute->size()=0}

checkonly domain r1 t:Table {totcols = 0 }}

Elementos de anotación

La siguiente tabla muestra una descripción de los elementos gráficos que permiten realizar anotaciones:

Tabla 2**.**
Notaciones gráficas

Anotación	Descripción
	Relación que se define entre m1 y m2, MM1 es el meta modelo de m1, y MM2 es el meta modelo de m2, C/E indica si el modelo en esa dirección puede tener la propiedad check o enforce.
	Plantilla de objeto que tiene como tipo C, con el nombre o.
	Plantilla de objeto que tiene como tipo C, con el nombre o, el cual indica que la propiedad a debe tomar el valor de val.
	<<domain>> indica el dominio de la relación
	oset es un tipo de objeto, que corresponde a un conjunto de objetos de tipo C
	Una plantilla not, indica que la correlación se realiza solo cuando no existe un objeto de tipo C que cumpla con las condiciones asociadas a este tipo de objeto.
	Una condición asociada a un dominio o a un tipo de objeto.

Fuente: (Object Management Group, 2008)

Mapeos Operacionales

El lenguaje operacional de QVT permite definir transformaciones de forma imperativa, o realizar transformaciones relacionales con operaciones imperativas.

Transformaciones Operacionales.

Las transformaciones operacionales se realizan de forma unidireccional, mismas que son expresadas imperativamente.

Este tipo de transformaciones requieren de un punto de ejecución definido por la palabra *main.*

Tipos de Modelos

Un tipo de modelo debe ser definido por un meta modelo, para esto se utiliza el lenguaje MOF, encontrando su definición en los diferentes paquetes que conforman dicho lenguaje.

Librerías

La declaración de librerías permite utilizar sus componentes a lo largo de la transformación.

Se las puede agregar dentro de una transformación mediante dos métodos:

- *access:* Da acceso a los métodos de una librería.
- *extension*: Permite extender los métodos de una librería.

Operaciones de mapeo

Una operación de mapeo es la implementación en sí de la relación entre los modelos de origen y los modelos de destino.

Según la definición de QVT operacional, una operación de mapeo debe contar con:

Un *guard* (*when*),

Un cuerpo

Una post condición (*where*).

La sintaxis general de un mapeo operacional es la siguiente:

mapping <dirkind0> X::mappingname

(<dirkind1> p1:P1, <dirkind2> p2:P2) : r1:R1, r2:R2

when { … } where { … } { init { … } population { … } end { … } }

init: definición de código que debe ejecutarse previo a la instanciación de las variables de salida.

- **population:** Código que puebla las variables de salida
- **end:** Código adicional a ejecutar antes de salir de la operación.

Si lo que se desea es retornar un objeto existente, y no crear uno nuevo, se debe especificar en la sección intermedia entre el *init* y *population* la asignación de los parámetros resueltos a dicho objeto.

Objetos

Dentro del QVT operacional, se define un *facility* que permite la reacción y actualización de los elementos de un modelo:

object s:Schema {

name := self.name;

table := self.ownedElement->**map** class2table();}

En este ejemplo, se define una variable existente ‘s’ de tipo Schema, la semántica de este código expresa lo siguiente: si ‘s’ es null, una instancia de Schema es creada y asignada a la variable ‘s’.

Si la variable ‘s’, posee ya una referencia a un objeto creado en el modelo de destino, en lugar de crear un nuevo objeto simplemente lo actualizará con el nuevo valor.

2.3.1.1.1 Helpers

Un *helper* es un tipo de función que realiza una serie de operaciones sobre un objeto y que devuelve un resultado concreto. Se dice que un helper puede tener efectos secundarios, esto significa que un objeto que ha sido enviado a un helper puede ver sus atributos alterados.

Un query es una operación tipo helper sin efectos secundarios.

2.3.1.1.2 Paquete QVT Operational

El paquete de QVTOperational permite definir los conceptos que son utilizados para especificar las transformaciones escritas de forma imperativa.

Estos conceptos se clasifican en dos categorías:

- Definición de transformaciones.
- Operaciones dentro de las transformaciones.
- Definición de transformaciones:

Este grupo se encuentra formado por 7 clases y 2 enumeraciones:

Clases:

- OperationalTransformation: La transformación en sí, define los modelos de entrada y de salida, teniendo como punto de entrada a la transformación una operación main. Esta clase no puede ser heredada.
- Library: Una librería es considerada como un módulo (conjunto de operaciones y definiciones) que sirven para ser reutilizadas.
- Module: Un módulo es una unidad compuesta por diferentes operaciones definidas para operar sobre un modelo.
- ModuleImport: Un module import expresa la integración de un módulo dentro de una transformación, ya sea mediante importándolo o extendiéndolo.
- ModelType: Cada parámetro dentro de un modelo tiene un tipo asociado, éste se encuentra definido dentro del meta modelo del modelo en cuestión.
- ModelParameter: Parámetros con los cuales el modelo es definido o invocado
- VarParameter: Definición de un parámetro
- Enumeraciones:
- ImportKind: Indica el orden y dirección en la cual los parámetros serán manejados(in, out)
- DirectionKind: Define la dirección de la transformación.

2.3.1.1.3 Lenguaje Core

El lenguaje Core maneja el mismo concepto de pattern-matching que el lenguaje relacional, haciéndolo igual de efectivo, sin embargo es un lenguaje más simple, aunque semánticamente hablando el lenguaje es más simple, al momento de describir una transformación, ésta puede requerir de mayor detalle.

Dado que el lenguaje Core realiza el seguimiento entre los modelos inicio y fin de forma implícita (guarda una relación entre el modelo inicio y destino), soporta una mayor cantidad de relaciones que el lenguaje relacional.

2.3 ATLAS TRANSFORMATION LANGUAGE - ATL

Atlas Transformation Language, ATL, nace como una respuesta del grupo INRIA a la propuesta de la OMG / QVT, ATL se centra principalmente en la transformación de modelo a modelo. El cual puede ser utilizado para realizar transformaciones tanto semánticas como sintácticas. Siendo un lenguaje especificado con una sintaxis textual concreta y como un meta modelo, además de ser declarativo e imperativo.

ATL permite la definición de tres tipos de unidades: los módulos de transformación ATL, librerías ATL y queries ATL, los cuales a su vez estarán compuestos por ATL *helpers,* atributos y reglas. La sintaxis de cada una de estas unidades está basada en el OCL (Object Constraint Language).

2.3.1 Data Types

El esquema que define los tipos de datos de ATL es similar al de OCL, el cual está detallado en la siguiente figura:

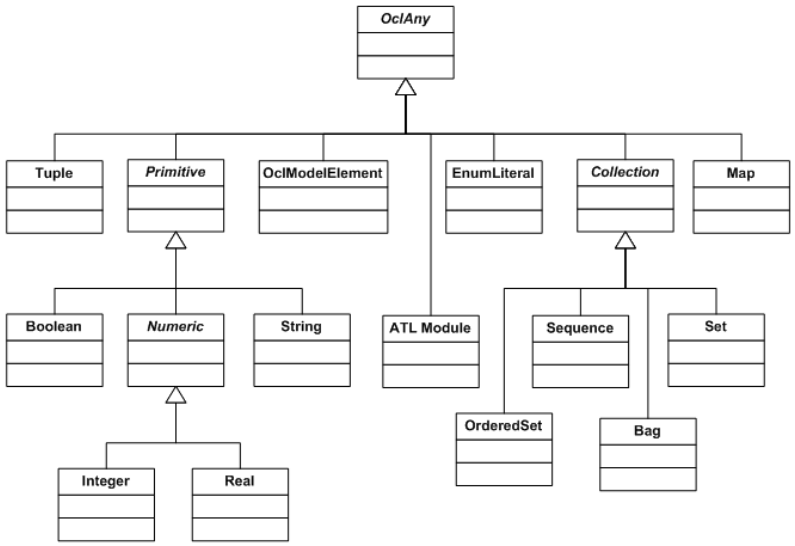


Figura 25. Tipos de datos OCL

Fuente: (Tom Armstrong, 2000)

Operaciones OclType

Retorna todas las instancias que existan del tipo *self.*

Operaciones OclAny

ATL al momento soporta las siguientes operaciones OCL:

- Operadores comparativos: =, <>;
- oclIsUndefined() retorna un valor verdadero o falso dependiendo si self no está definido;
- oclIsKindOf(t : oclType) retorna un valor verdadero o falso dependiendo si self es una instancia de t o de uno de sus subtipos;

- oclIsTypeOf(t : oclType) devuelve un valor verdadero o falso dependiendo si self es una instancia de t.
- toString() retorna una cadena que representa a self;
- oclType() retorna el tipo al que pertenece self con referencia a OCL;
- output(s : String) escribe la cadena que se le envía en la consola.
- debug(s : String) retorna el valor de self, además escribe el valor de "s : self_value" como una cadena en la consola;
- refSetValue(name : String, val : oclAny) es una operación que permite asignar el valor value a una propiedad de self;
- refGetValue(name : String) retorna el valor de la propiedad que se solicite de self;

Colecciones

Las colecciones están basadas en OCL, las cuales se definen a continuación:

- Set: es una colección sin duplicados y sin orden.
- OrderedSet: es una colección sin duplicado y con un orden.
- Bag: es una colección en la cual los duplicados están permitidos y sin orden.
- Sequence: es una colección en la cual los duplicados están permitidos y con un orden.

2.3.2 Módulo ATL

Un módulo ATL es un modelo de transformación, el cual permite definir la forma en la cual se generaran los modelos objetivo, partiendo de un conjunto de modelos iniciales. Para esto, los dos modelos, tanto origen como objetivo, deben ser "descritos" por sus respectivos meta-modelos. Además, los módulos ATL, deben recibir un numero especifico de modelos de entrada para ser procesados y convertidos en modelos de salida, los cuales también están ya definidos. De esta forma, siempre se conoce cuantos modelos de entrada se necesitan para una transformación y cuantos modelos de salida se obtendrá como consecuencia de dicho proceso.

Un módulo ATL facilita una función (resolveTemp) que permite hacer referencia a un elemento del modelo destino que será generado, teniendo la siguiente declaración

resolveTemp(var, target_pattern_name)

El parámetro *var* corresponde a una variable que se encuentre dentro del modelo origen, El parámetro *target_pattern_name* es una referencia al patrón de destino que mapea el elemento provisto (var) dentro del modelo destino.

Ejemplo de resolveTemp:

```
rule AtoAnnotedB {
        from
                a : MMA!A
        to
                ann : MMB!Annotation (),
```

```
        b : MMB!B (
            annotation <- ann)}
```

En esta regla un objeto tipo A es transformado en un objeto tipo B anotado.

En la siguiente regla se hace referencia al elemento B que fue creado anteriormente

```
rule ARefToBRef {
    from
        aRef : MMA!ARef
    to
        bRef : MMB!BRef (
            ref <- thisModule.resolveTemp(aRef.ref, 'b')
        )
}
```

2.3.2.1 Estructura de un módulo ATL

La estructura de un módulo ATL es la siguiente:

- Encabezado: definición de atributos.
- Importación de bibliotecas
- Definición de métodos.
- Definición de los modelos generados.

La definición de reglas y helpers puede darse en cualquier parte del módulo ATL.

2.3.2.2 Encabezado

La sintaxis del encabezado tiene la siguiente estructura:

module module_name;

create output_models [from|refining] input_models;

Tipos de datos primitivos

Dado que ATL está definido con OCL, se manejan los siguientes tipos de datos

- Datos primitivos
- Boolean
- Integer
- Real
- String
- Colecciones
- OrderedSet
- Secuence
- Bag
- Set
- Enumeraciones
- Tuplas
- Elementos de Modelo
- Tipos de datos definidos por el usuario

2.3.2.3 Expresiones declarativas OCL

Además de las expresiones declarativas soportadas por los tipos de datos ya mencionados, ATL utiliza las siguientes que se encuentran definidas dentro de OCL

Expresiones If

Expresiones Let

Otras Expresiones

Además de las mencionadas, se puede definir expresiones que se acoplen dentro de la especificación de OCL, estas pueden ser:

- Expresiones constantes, que corresponden a un dato específico soportado.
- Expresiones que llaman a un *helper*, ya sea que esté definido dentro del contexto de un módulo ATL o como parte de un elemento de un modelo origen.
- Expresiones que invoquen a una operación.

2.3.2.4 ATL Helpers

Dentro de ATL un método es conocido como un helper, y dependiendo de su sintaxis se puede encontrar de dos tipos: funcionales y de atributos, los funcionales permiten el uso de parámetros, a diferencia de los helpers de atributos que no lo soportan.

Semántica:

helper [context context_type]? def : helper_name(parameters) : return_type = exp;

Cada helper se define con dentro de un contexto(context_type), su nombre(helper_name), un conjunto de parámetros de entrada(parameters) y el tipo de dato que se obtiene como resultado de su ejecución(return_type). Al definir un contexto,

lo que se indica es que tipos de elementos pueden hacer una llamada a dicho método (helper).

El nombre de un helper es definido con la palabra reservada *def*.

La definición de los parámetros incluye tanto su nombre como su tipo:

parameter_name : parameter_type

El siguiente es un ejemplo de un helper:

```
helper def : averageLowerThan(s : Sequence(Integer), value : Real) : Boolean =
        let avg : Real = s->sum()/s->size() in avg < value;
```

Teniendo que su nombre es *averageLowerThan*, definido dentro del contexto del módulo ATL ya que no se especifica el contexto, mediante el cual se obtendrá un valor booleano que represente si el promedio de los valores contenidos dentro de la secuencia de valores s es menor al valor que contenga la variable value.

La expresión "let" define he inicializa la variable avg, la cual es a continuación comparada contra value, para conocer si es menor o no.

La palabra reservada super, permite invocar a un helper de la clase padre dentro de un helper de la clase hijo, ejemplo:

Si se tiene la siguiente sentencia:

```
class A {}
class B extends A {}
```

Se puede llamar al helper test() de la clase A desde el helper test() de la clase B

```
helper context A def: test() : Integer = 1;

helper context B def: test() : Integer = super.test() + 1;
```

2.3.2.5 Atributos

Un atributo dentro del contexto de ATL puede ser visto como una constante, misma que es especificada dentro de un contexto, su mayor diferencia con un helper es la carencia de parámetros de entrada, siendo su sintaxis la siguiente: helper [contexto context_type]? def : attribute_name : return_type = exp;

El siguiente es un ejemplo relacionado con el meta modelo de transformación MMPerson (ECLIPSE GROUP):

```
helper def : getYoungest : MMPerson!Person =

    let allPersons : Sequence(MMPerson!Person) =

MMPerson!Person.allInstances()->asSequence() in

allPersons->iterate(p; y : MMPerson!Person = allPersons->first() |

    if p.age < y.age

    then p  else y   endif);
```

El atributo llamado *getYoungest* se lo define dentro del contexto de un módulo ATL, define que el meta modelo *MMPerson* debe contener un modelo *Person* así como sus elementos. El objetivo de este atributo es encontrar la persona más joven del modelo origen (por lo que el resultado es un elemento de tipo MMPerson!Person). El código de este atributo está compuesto por la expresión let que define la variable *allPersons* de

tipo set(para contener todas las personas que se encuentren definidas dentro del modelo origen). Una vez se tiene este listado, se realiza una iteración, siendo la variable p la cual representa la persona actual de la iteración, la variable y representa la primera persona de la secuencia, y la sección if, va a retornar la persona más joven, ya sea que se encuentre en y o en p.

En cuanto a la ejecución de un helper y un atributo existe una gran diferencia, dado que el helper va a ser invocado desde cualquier contexto que lo soporte, el motor de ATL recibirá la orden de ejecutarlo bajo demanda, mientras que los atributos, ya que siempre retornaran el mismo valor, el motor solamente lo ejecutará una vez en el momento de inicialización del programa ATL.

2.3.2.6 Reglas ATL

Dentro de ATL una regla es quien define la transformación en si entre dos modelos(origen y destino), pudiendo ser estas de dos tipos:

Matched rules: Éste tipo de reglas buscan un modelo que cumpla con las condiciones definidas por la regla, para generar a partir de éste el modelo destino, es de tipo declarativo.

Called rules: Éste tipo de regla debe ser invocada desde otro proceso para que pueda ser ejecutada, es de tipo imperativo.

Lazy rules: Al igual que las *Called rules*, deben ser invocadas desde otro proceso, sin embargo estas son de tipo declarativo.

2.3.2.7 Código imperativo ATL

ATL soporta la especificación de código imperativo, esto dentro de bloques de código definidos en las reglas.

Asignaciones

Para realizar una asignación se utiliza la siguiente sintaxis:

```
target <- exp;
```

En el siguiente código se realiza una adición de 1 al valor que se encuentra en la variable counter del modelo que se esté analizando:

```
thisModule.counter <- thisModule.counter + 1;
```

Sentencia IF

Sintaxis:

```
if(condition) {statements1}
[else {statements2}]?
```

Ejemplo:

```
if(aPerson.gender = #male) {  thisModule.fullName <- 'Mr. ' + aPerson.name + ' ' +
aPerson.surname;}
else {   if(aPerson.isSingle) {  thisModule.fullName <- 'Miss ' + aPerson.name;
        thisModule.surname <- aPerson.surname; }
```

```
else {   thisModule.fullName <- 'Mrs. ' + aPerson.name;
         thisModule.surname <- aPerson.marriedTo.surname;}
thisModule.fullName <- thisModule.fullName + ' ' + thisModule.surname;}
```

Sentencia For

Sintaxis

```
for(iterator in collection) {statements}
```

Ejemplo:

```
for(p in MMPerson!Person.allInstances()) {  if(p.gender = #male)   thisModule.men-
>including(aPerson);  else      thisModule.women->including(aPerson);}
```

2.3.2.8 Matched Rules

Al momento de crear una regla en ATL se debe especificar primeramente cuáles serán los elementos de los modelos de entrada, cuáles serán los elementos y modelos objetivos y la forma en la cual los elementos deben ser inicializados, para lo cual se tiene la siguiente sintaxis:

```
rule rule_name {
        from
                in_var : in_type [in model_name]? [( condition)]?
        [using {var1 : var_type1 = init_exp1;
```

```
        ...

        varn : var_typen = init_expn;}]?

    to

        out_var1 : out_type1 [in model_name]? (bindings1),

        out_var2 : distinct out_type2 foreach(e in collection)(bindings2),

        ...

    out_varn : out_typen [in model_name]? (bindingsn) [do {statements}]?

}
```

La sentencia *rule_name* específica el nombre (único) de la regla, una sección *from* y una sección *to*, así como dos adicionales *using* y *do*.

Patrón de origen

La sección from de la regla contiene la definición del patrón del modelo origen, éste indica el tipo de modelo que la regla espera.

Ejemplo:

```
from

    p : MMPerson!Person (

        p.name = 'Smith')
```

Variables locales

La sección opcional *using* permite declarar variables locales para ser utilizadas dentro del *helper*

Ejemplo:

```
from
        c : GeometricElement!Circle
using {
        pi : Real = 3.14;
        area : Real = pi * c.radius.square();
}
```

Patrón de destino

La sección *to* de una regla define el patrón del modelo de destino que se generará al ejecutarse la regla, pudiendo ser éste un patrón simple o iterativo (varios elementos de un modelo son creados)

La última sección *do* de una regla ATL permite definir código que será ejecutado una vez que la sección de inicialización ha concluido. Este tipo de código puede ser utilizado para inicializar elementos adicionales o modificar características de aquellos que ya fueron creados.

```
helper def : id : Integer = 0;

...

rule Journal2Book {

    from    j : Biblio!Journal to b : Biblio!Book (...)

    do {thisModule.id <- thisModule.id + 1;b.id <- thisModule.id;}}
```

En este ejemplo la variable global id se define dentro del módulo de ATL con un valor de 0, con lo cual el objetivo es asociar a cada elemento del modelo generado un id único, por lo que el bloque imperativo (sección *do*) incrementa el valor de la variable global *id* y lo asigna a la propiedad *id* del elemento generado.

2.3.2.9 Lazy Rules

Ejemplo de invocación de una *Lazy rule*:

```
lazy rule getCross {

  from    i: ecore!EObject   to      rel: metamodel!Relationship (     ) }
```

Código que invoca la regla

```
rule Example {   from      s : ecore!EObject   to      t : metamodel!Node ( name <-
s.toString(),     edges <- thisModule.getCross(s)    ) }
```

2.3.2.10 Called Rules:

Ejemplo:

```
[entrypoint]? rule rule_name(parameters){
    [using {var1 : var_type1 = init_exp1;... varn : var_typen = init_expn;}]?
    [to out_var1 : out_type1 ( bindings1  ),
        out_var2 : distinct out_type2 foreach(e in collection)(bindings2),
        ...
        out_varn : out_typen (bindingsn)]?[do {statements}]?z
```

Nota: El nombre de una regla invocada debe ser única dentro del contexto de la transformación.

Al igual que los *helpers*, una regla puede contener parámetros, para lo cual éstos deben ser especificados, para lo cual existen 3 secciones opcionales que se pueden definir dentro de la regla:

using: Declaración he inicialización de variables

to: el patrón objetivo de la regla.

do: definición imperativa de instrucciones a ejecutar.

2.3.2.11 Herencia de reglas

Herencia de reglas:

```
abstract rule A {    from [fromA]   using [usingA]   to [toA]   do [doA]
 }
 rule B extends A {
   from [fromB]    using [usingB]    to [toB]    do [doB]
 }
 rule C extends B {   from [fromC]   using [usingC]   to [toC]   do [doC] }
```

Cuando el motor de ATL compila este código, el resultado sería similar a tener el siguiente código:

```
rule B {
   from [fromB]
   using [usingB]
   to [toA.bindings union toB.bindings]
   do [doB]
 }
 rule C {   from [fromC]   using [usingC]   to [toA.bindings union toB.bindings union
toC.bindings]   do [doC] }
```

2.3.2.12 Uso de reglas

A continuación se detalla cómo cada una de las 3 tipos de reglas son utilizadas:

- **Matched rules** se invocan cada vez que se encuentra un elemento de un modelo que cumpla con las condiciones necesarias.
- **Lazy rules** se utilizan cada vez que sean mencionadas por otros procesos.
- **Unique lazy rules** son invocadas una sola vez para cada elemento que cumpla las condiciones necesarias, aun si son llamadas desde varios procesos diferentes.

La siguiente tabla muestra un resumen del número de veces que una regla es ejecutada dependiendo de diferentes factores:

Tabla 3.
Ejecución de reglas ATL

Regla	Referencias al patrón de origen	Número de veces que el elemento destino es creado
Standard	0	1
	1	1
	n > 1	1
Lazy	0	0
	1	1
	n > 1	N
Unique lazy	0	0
	1	1
	n > 1	1

Fuente: (ECLIPSE GROUP)

2.4 ATL QUERIES

ATL también permite definir un query sobre uno o varios modelos origen para producir un resultado de un tipo de dato soportado.

Un query estará conformado por su definición, helpers y atributos.

Sintaxis:

query *query_name = exp*;

El cuerpo de un query debe estar definido dentro de las expresiones OCL

En ocasiones puede ser necesario conocer el resultado del query, para esto se puede utilizar la siguiente expresión para escribir el resultado en un documento de texto.

```
query PersonNb =
    MMPerson!Person.allInstances()->size().toString().writeTo('result.txt');
```

Este ejemplo toma el set de entidades tipo Person y calcula el tamaño total del set, para a continuación escribirlo en un archivo llamado result.txt

CAPÍTULO 3

3.1 CARACTERÍSTICAS GENERALES DE QVT y ATL:

Antes de comenzar con la comparación de ambos lenguajes, se identificaron las características más relevantes de ATL y QVT, esto con el fin de comprender como estos lenguajes están compuestos, y sentar los precedentes para su comparación.

La siguiente tabla muestra las características generales de ambos:

Tabla 4.
Características generales de QVT y ATL

Característica Generales		ATL	QVT
Estructura del lenguaje	**Tipo de Sintaxis**	Textual	Textual - Gráfica
	Estructura de la sintaxis	Verbosa	Verbosa
Transformación	**Meta - Modelos**	Soporta varios meta modelos	Soporta varios meta modelos
	Modelos	Todo es un modelo	Acepta modelos MOF(transformaciones de XMI a XMI)
	Query		Maneja querys, que devuelven un sub conjunto de elementos de un tipo especifico
	Estereotipos		Se pueden definir plantillas que indican un tipo de elemento
Nivel de abstracción	**2**		Relacional
	1	ATL	Core, Operacional
	0	VM	VM
Escenarios de transformación	**Sincronización**		Relacional, Core
	Validación		Relacional, Core
	Transformación	AMW, ATL, VM	Relacional, Core, Operacional
	Condiciones		Claúsulas where y when permiten definir condiciones de ejecución
Paradigma	**Declarativo**		Relacional, Core
	Hibrido	ATL	
	Imperativo		Operacional
	Herencia	ATL	Relacional, Core, Operacional
	Librerias Externas	ATL	Relacional, Core, Operacional
Direccionalidad	**Multidireccional**		Relacional, Core
	Unidireccional	ATL	Operacional
Cardinalidad	**Varios a Varios**	ATL	Operacional, Relacional, Core(como checkonly)
	Varios a Uno		Relacional y Core (enforce)
Traceabilidad	**Automática**	ATL	Relacional, Operacional
	Manual		Core

Fuente: (Object Management Group, 2008; ECLIPSE GROUP)

3.2 CRITERIOS DE COMPARACIÓN:

En la siguiente sección se detallan los aspectos y criterios de comparación que se utilizaron en este trabajo para evaluar ambos lenguajes.

3.2.1 Características deseables

La siguiente tabla está derivada de las características deseables definidas por MDA (Juan de Lara, 2012), en la cual se encuentran una lista de condiciones con las cuales un buen lenguaje de transformación debería contar.

Tabla 5.
Criterios de comparación según las características deseables

Condiciones	Definición
Ejecutabilidad	Ser ejecutable.
Ser eficiente	Poder aplicarse de una manera eficiente.
Expresividad	Ser plenamente expresivo, pero inequívoco, de las transformaciones que modifican los modelos existentes, así como crear modelos completamente nuevos.
Precisión	Facilitar la productividad del desarrollador con descripciones precisas, concisas y claras.
Definición clara de reglas	El lenguaje debe diferenciar claramente la descripción de las reglas de selección de modelo de la fuente de las normas para la producción del modelo de destino.
Construcciones gráficas	El lenguaje debe ofrecer el manejo de construcciones gráficas en los casos en que los conceptos representados son más concisos e intuitivos, en forma gráfica en comparación con un textual.
Declarativo	El lenguaje debe ser declarativo, haciendo implícito cualquier concepto o mecanismos que se puede interpretar de manera intuitiva por el contexto.
Reutilización	Proporcionar un medio para combinar transformaciones, para formar otros compuestos, que ofrecen al menos los operadores para la secuenciación, la selección condicional y la repetición de las transformaciones.
Manejo de condiciones	Proporcionar un medio para definir las condiciones bajo las cuales se permite la ejecución de una transformación.

Fuente : (Juan de Lara, 2012)

Utilizando el estudio realizado en el documento de Juan Herrera (Juan C. Herrera, 2010), se tiene las siguientes dimensiones con las cuales los lenguajes de modelado pueden ser analizados.

3.2.2 Según su uso

Comprende características relacionadas a la utilización del lenguaje, la forma en la cual éstos interpretan los modelos para luego ser transformados, la siguiente tabla muestra los parámetros de evaluación relacionados a esta dimensión que fueron tomados para la comparación realizada en este trabajo:

Tabla 6.
Criterios de comparación según el uso del lenguaje

Características	Atributos	Descripción
Traducción	Migración	Un modelos es transformado en otro en el mismo nivel de abstracción
	Síntesis	Un modelo se transforma en otro a un nivel de abstracción inferior
	Ing. Inversa	Un modelo es transformado en otro a un nivel superior de abstracción
	Endógeno	Todos los modelos son conforme al mismo meta modelo.
	Exógeno	Múltiples meta modelos son utilizados
Interpretación	Reemplaza	Modelo de entrada se modifica y convierte en modelo de salida
	Crea	El modelo de entrada es solo de lectura, con el cual se genera el modelo de salida.
	Normalización	Un modelo es transformado en un sub lenguaje
	Refactorización	Especialización de un modelo
	Corrección	Corrección de errores del modelo
	Adaptación	Un modelo es actualizado según los nuevos requerimientos
Sincronización	Parcial	Parte de un modelo es modificado
	Comprobación	Valida la relación entre modelos
	Doble sentido	Genera un modelo a partir de un código fuente
	Vistas	Un modelo que se deriva de otro modelo

Fuente: (Juan C. Herrera, 2010)

3.2.3 Según la forma como el lenguaje se encuentra organizado

En esta dimensión se evalúan los lenguajes tomando en cuenta su ejecución, las diferentes posibilidades al momento de ejecutar una transformación, y como los modelos

van a interactuar entre sí dentro del flujo, la siguiente tabla contiene las áreas y propiedades a evaluar en esta dimensión:

Tabla 7.
Criterios de comparación según la organización del lenguaje

Características	Atributos	Descripción
Modularidad	Composición	Las transformaciones se manejan mediante modelos
	Combinarse	Existe la posibilidad de combinar varios módulos con el fin de generar otro modulo.
Composición	Encadenarse	Varias transformaciones se encadenan consecutivamente
	Componerse	Composición de transformaciones con el fin de obtener una nueva
	Combinarse	Combina varias transformaciones
Definición de Condiciones de ejecución	Precondición	Las reglas se aplican según condiciones previas
	Determinístico	La ejecución está dirigida según un orden determinado
	No determinístico	No es posible controlar el orden de ejecución de reglas
	Interactivo	El usuario es quien define el orden y forma en la cual las reglas son ejecutadas
Mecanismos para reutilizar e integrar	Importando	Permite importar librerías
	Por parámetros	Según el uso de parámetros es posible modificar una transformación
	Deshabilitar	Limita la transformación a un conjunto definido de objetos
	Reemplazo	Sobrecarga de reglas
	Herencia	Herencia de reglas
Definición de elementos destino	Múltiples elementos Por una sola regla	Varios elementos son manejados por una sola regla
	Múltiples reglas para un solo elemento	Un elemento es manejado a través de varias reglas.
Direccionalidad	Muchos a muchos	Varios modelos de origen se transforman en varios modelos de salida
	Uno a uno	Un modelo de origen se transforma en un modelo de salida
	Muchos a uno	Varios modelos de origen se transforman en un modelo de salida
	Uno a muchos	Un modelo de origen se transforma en varios de salida

Fuente: (Juan C. Herrera, 2010)

3.2.4 Según la formalización del lenguaje

Dimensión que evalúa la sintaxis y semántica del lenguaje, enfocándose en la usabilidad y adaptabilidad de éste, el objetivo del análisis es conocer el grado de flexibilidad del lenguaje al momento de llevar a cabo una transformación:

Tabla 8.
Criterios de comparación según la formalización del lenguaje

Características	Atributos	Descripción
Notación sintaxis abstracta	Textual	Notación Textual
	Meta modelo	El lenguaje puede ser escrito utilizando modelos
Notación sintaxis concreta	Textual	Las transformaciones son escritas sin herramientas graficas
Estilo de la definición	Declarativo	Describe relaciones entre elementos
	Imperativo	Describe los pasos a seguir para producir un resultado
Expresión	Conciso	Pocas construcciones sintácticas
	Verboso	Variedad de construcciones sintácticas
	General	Grupo general de construcciones
Correspondencia de elementos	Por colección	Existe relación entre el conjunto de elementos del modelo origen y el modelo destino
	Por tipo	Los elementos son seleccionados según un tipo, incluidos los subtipos
	Por tipo-preciso	Los elementos son seleccionados según un tipo, excluidos los subtipos
	Filtrado condicional	Permite filtrar los conjuntos de objetos mediante asociaciones, atributos, etc.
	Reglas sobre clases	Indica como inicializar un objeto en el modelo destino
	Reglas sobre atributos	Indica el valor a asignar a los atributos en el modelo destino

Fuente: (Juan C. Herrera, 2010)

3.3 EVALUACIÓN DE LOS LENGUAJES QVT Y ATL:

El criterio de evalución de los lenguajes QVT y ATL, será el nivel de cumplimiento de las características deseables para cada lenguaje, dichas características se encuentran definidas en la tabla 4 (Richard F. Paige, 2009). El método seleccionado para verificar el cumplimiento es el siguiente:

- **Relación atributos – características generales de QVT y ATL**: En primer lugar se toman las características generales de los lenguajes QVT y ATL, y se analizan contra las dimensiones, de esta forma es posible conocer qué atributos poseen los lenguajes en cada dimensión.
- **Relación atributos – características deseables de los lenguajes:** A continuación se toman los atributos por cada dimensión, y se relacionan con las características deseables, de esta forma es posible conocer los atributos que contribuyen a incrementar el cumplimiento de una característica deseable.

Las características deseables se encuentran descritas en la sección anterior en las tablas 6, 7 y 8.

3.3.1 Relación atributos – características generales de los lenguajes

En las siguientes tablas se realiza la relación entre los atributos de una dimensión y las características generales de QVT y ATL.

Tabla 9.
Relación entre atributos de la dimensión según el uso del lenguajes y las características de QVT y ATL

				Atributos según el uso del lenguaje									
Característica Generales		**ATL**	**QVT**	Endógeno	Exógeno	Reemplaza	Crea	Refactorización	Corrección	Adaptación	Parcial	Comprobación	Vistas
Transformación	**Meta - Modelos**	Soporta varios meta modelos	Soporta varios meta modelos	X	X								
	Query		Maneja querys, que devuelven un sub conjunto de elementos de un tipo especifico										X
Escenarios de transformación	**Sincronización**		Relacional, Core							X	X		
	Validación		Relacional, Core						X			X	
	Transformación	AMW, ATL, VM	Relacional, Core, Operacional			X	X	X					

Tabla 10.
Relación entre los atributos de la dimensión organización del lenguajes y las características de QVT y ATL.

				Atributos según la organización del lenguaje										
Característica Generales		**ATL**	**QVT**	Composición	Precondición	Determinístico	No Determinístico	Interactivo	Por Importación	Deshabilitar	Herencia	Muchos a muchos	uno a uno	Muchos a uno
Transformación	**Meta - Modelos**	Soporta varios meta modelos	Soporta varios meta modelos	X										
Escenarios de transformación	**Condiciones**		Clausulas where y when permiten definir condiciones de ejecución		X					X				
Paradigma	**Herencia**	ATL	Relacional, Core, Operacional								X			
	Librerías Externas	ATL	Relacional, Core, Operacional						X					
Direccionalidad	**Multi- Direccional**		Relacional, Core					X						
	Uni-Direccional	ATL	Operacional			X	X							
Cardinalidad	**Varios a Varios**	ATL	Operacional, Relacional, Core(como checkonly)									X	X	
	Varios a Uno		Relacional y Core (enforce)											X

Tabla 11**.**
Relación entre los atributos de la dimensión formalización del lenguaje y las características de QVT y ATL

				Atributos según la formalización del lenguaje										
Característica Generales		**ATL**	**QVT**	Textual	Meta modelo	Textual	Declarativo	Imperativo	Verboso	Por colección	Por tipo	Por tipo-preciso	Reglas sobre clases	Reglas sobre atributos
Estructura del lenguaje	**Tipo de Sintaxis**	Textual	Textual - Gráfica	X	X	X								
	Estructura de la sintaxis	Verbosa	Verbosa						X					
Transformación	**Query**		Maneja querys, que devuelven un sub conjunto de elementos de un tipo específico								X	X		
	Estereotipos		Se pueden definir plantillas que indican un tipo de elemento										X	X
Paradigma	**Declarativo**		Relacional, Core				X							
	Híbrido	ATL						X						
Traceabilidad	**Manual**		Core							X				

3.3.2 Relación atributos – características deseables de los lenguajes

En este paso el objetivo es conocer en qué características deseables influyen los atributos de las dimensiones. Para establecer esta relación entre características y atributos, se utilizaron los siguientes criterios:

3.3.2.1 Ejecutabilidad

Se relacionan los atributos que afecten la ejecución de la transformación, obteniendo como resultado los siguientes:

Tabla 12.
Relación atributos según la formalización del lenguajes - Ejecutabilidad

Dimensión	Características	Atributos
Formalización del lenguaje	**Notación sintaxis abstracta**	Textual
	Notación sintaxis concreta	Textual
	Estilo de la definición	Declarativo
		Imperativo
	Expresión	Conciso
		Verboso
		General

3.3.2.2 Eficiencia

Se toma en cuenta aquellos atributos que ayudan a manejar la transformación de forma eficiente, reducir código y reutilizar.

Tabla 13.
Relación atributos según la formalización del lenguaje - Eficiencia

Dimensión	Características	Atributos
Uso del lenguaje	**Traducción**	Migración
		Síntesis
		Ing. Inversa
		Endógeno
		Exógeno
	Interpretación	Reemplaza
		Crea
		Corrección
	Sincronización	Parcial
		Comprobación
Organización del lenguaje	**Modularidad**	Combinarse
	Composición	Encadenarse
		Componerse
		Combinarse
	Mecanismos para reutilizar e integrar	Por Importación
		Por parámetros
		Reemplazo
		Herencia
Formalización del lenguaje	**Expresión**	Conciso
	Correspondencia de elementos	Por colección
		Por tipo
		Por tipo-preciso
		Filtrado condicional

Expresividad

Atributos que permitan que el lenguaje sea más exacto al momento de definir transformaciones.

Tabla 14.
Relación atributos según la formalización del lenguaje - Expresividad

Dimensión	Características	Atributos
Uso del lenguaje	**Interpretación**	Crea
	Sincronización	Comprobación
		Vistas
Organización del lenguaje	**Definición de Condiciones de ejecución**	Precondición
		Determinístico
		Interactivo
	Mecanismos para reutilizar e integrar	Por parámetros
	Direccionalidad	Uno a uno
Formalización del lenguaje	**Notación sintaxis abstracta**	Textual
		Meta modelo
	Notación sintaxis concreta	Textual
	Estilo de la definición	Declarativo
		Imperativo
	Expresión	Conciso
	Correspondencia de elementos	Por colección
		Por tipo
		Por tipo-preciso
		Filtrado condicional
		Reglas sobre clases
		Reglas sobre atributos

3.3.2.3 Precisión

Atributos relacionados con la productividad, ayudan en la definición e implementación; facilitan la creación de transformaciones.

Tabla 15.
Relación atributos según la formalización del lenguaje – Precisión

Dimensión	Características	Atributos
Uso del lenguaje	**Traducción**	Endógeno
	Interpretación	Crea
		Corrección
		Adaptación
	Sincronización	Parcial
		Comprobación
		Vistas
Organización del lenguaje	**Modularidad**	Composición
		Combinarse
	Composición	Encadenarse
		Componerse
		Combinarse
	Definición de Condiciones de ejecución	Precondición
		Interactivo
	Mecanismos para reutilizar e integrar	Por Importación
		Por parámetros
		Reemplazo
		Herencia
	Definición de elementos destino	Múltiples elementos por una sola regla
		Múltiples reglas para un solo elemento
	Direccionalidad	Muchos a muchos
		Uno a uno
		Muchos a uno
		Uno a muchos
Formalización del lenguaje	**Notación sintaxis abstracta**	Meta modelo
	Estilo de la definición	Declarativo
		Imperativo
	Expresión	Conciso
		Verboso
		General
	Correspondencia de elementos	Por tipo
		Por tipo-preciso
		Filtrado condicional
		Reglas sobre clases
		Reglas sobre atributos

3.3.2.4 Definición clara de reglas

Atributos asociados a la aplicación de reglas sobre transformaciones.

Tabla 16.
Relación atributos según la formalización del lenguaje – Definición clara de reglas

Dimensión	Características	Atributos
Organización del lenguaje	**Definición de Condiciones de ejecución**	Precondición
		Determinístico
		Interactivo
	Mecanismos para reutilizar e integrar	Por Importación
		Por parámetros
		Deshabilitar
		Reemplazo
		Herencia
	Definición de elementos destino	Múltiples elementos por una sola regla
		Múltiples reglas para un solo elemento
Formalización del lenguaje	**Estilo de la definición**	Imperativo
	Expresión	Verboso
	Correspondencia de elementos	Filtrado condicional
		Reglas sobre clases
		Reglas sobre atributos

3.3.2.5 Construcciones gráficas

Atributos relacionados con construcciones gráficas, meta modelos y modelos.

Tabla 17**.**
Relación atributos según la formalización del lenguaje – Construcciones gráficas

Dimensión	Características	Atributos
Uso del lenguaje	**Traducción**	Endógeno
		Exógeno
Formalización del lenguaje	**Notación sintaxis abstracta**	Meta modelo
	Estilo de la definición	Declarativo
	Expresión	Conciso
		Verboso
		General

3.3.2.6 Declarativo

Atributos que definen si el lenguaje es declarativo.

Tabla 18.
Relación atributos según la formalización del lenguaje - Declarativo

Dimensión	Características	Atributos
Formalización del lenguaje	**Estilo de la definición**	Declarativo

3.3.2.7 Reutilización

Atributos asociados a la reutilización de elementos.

Tabla 19.
Relación atributos según la formalización del lenguaje – Reutilización

Dimensión	Características	Atributos
Uso del lenguaje	**Traducción**	Migración
		Síntesis
		Ing. Inversa
		Endógeno
		Exógeno
	Interpretación	Reemplaza
		Normalización
		Refactorización
	Sincronización	Parcial
		Doble sentido
Organización del lenguaje	**Modularidad**	Composición
		Combinarse
	Composición	Encadenarse
		Componerse
		Combinarse
	Mecanismos para reutilizar e integrar	Por Importación
		Por parámetros
		Reemplazo
		Herencia
	Definición de elementos destino	Múltiples elementos por una sola regla
		Múltiples reglas para un solo elemento
	Direccionalidad	Uno a muchos

3.3.2.8 Manejo de condiciones

Atributos que expresan la flexibilidad del lenguaje en cuanto a cómo ejecutar una transformación, es decir, condiciones de ejecución.

Tabla 20.
Relación atributos según la formalización del lenguaje – Manejo de condiciones

Dimensión	Características	Atributos
Uso del lenguaje	**Traducción**	Migración
		Síntesis
		Ing. Inversa
		Endógeno
		Exógeno
	Interpretación	Reemplaza
		Crea
		Normalización
		Refactorización
		Corrección
		Adaptación
	Sincronización	Doble sentido
Organización del lenguaje	**Definición de Condiciones de ejecución**	Precondición
	Mecanismos para reutilizar e integrar	Deshabilitar
Formalización del lenguaje	**Correspondencia de elementos**	Reglas sobre clases
		Reglas sobre atributos

En las siguientes tablas, a más de construir la relación entre los atributos y las características deseadas descritas anteriormente, se muestra a través de colores, la relación que existe entre el atributo y el lenguaje QVT o ATL.

Para identificar el cumplimiento de los atributos de cada dimensión, por parte de los lenguajes QVT y ATL, se utilizó la letra “Q”, y la letra “A” respectivamente, cuando el lenguaje cumple con el atributo de la dimensión.

La tabla 21 presenta la relación entre las características deseables en la dimensión según el uso del lenguaje, el color gris es utilizado para indicar que existe una relación entre el atributo y una característica deseable.

Tabla 21.
Características deseables en la dimensión según el uso del lenguaje.

			CARACTERÍSTICAS DESEABLES SEGÚN OMG								
	Características	Atributos	Ejecutabilidad	Eficiencia	Expresividad	Precisión	Definición clara de reglas	Construcciones gráficas	Declarativo	Reutilización	Manejo de condiciones
Uso del lenguaje	Traducción	Migración		QA						QA	QA
		Síntesis		QA						QA	QA
		Ing. Inversa									
		Endógeno		QA		QA		QA		QA	QA
		Exógeno		QA				QA		QA	QA
	Interpretación	Reemplaza		QA						QA	QA
		Crea		QA	QA	QA					QA
		Normalización									
		Refactorización								QA	QA
		Corrección		Q		Q					Q
		Adaptación				Q					Q
	Sincronización	Parcial		Q		Q				Q	
		Comprobación		Q	Q	Q					
		Doble sentido									
		Vistas			Q	Q					
QVT				9	3	7		2		7	9
ATL				6	1	2		2		6	7

La tabla 22 contiene la relación entre las características deseables y la dimensión según la organización del lenguaje.

Tabla 22.
Características deseables en la dimensión según la organización del lenguaje

			CARACTERÍSTICAS DESEABLES SEGÚN OMG								
	Características	**Atributos**	Ejecutabilidad	Eficiencia	Expresividad	Precisión	Definición clara de reglas	Construcciones gráficas	Declarativo	Reutilización	Manejo de condiciones
Organización del lenguaje	**Modularidad**	Composición				**QA**				**QA**	
		Combinarse									
	Composición	Encadenarse									
		Componerse									
		Combinarse									
	Definición de Condiciones de ejecución	Precondición			**Q**	**QA**	**Q**				**Q**
		Determinístico			**QA**		**QA**				
		No determinístico									
		Interactivo			**Q**	**Q**	**Q**				
	Mecanismos para reutilizar e integrar	Por Importación		**QA**		**QA**	**QA**			**QA**	
		Por parámetros									
		Deshabilitar					**QA**				**Q**
		Reemplazo									
		Herencia		**QA**		**QA**	**QA**			**QA**	
	Definición de elementos destino	Múltiples elementos por una sola regla									
		Múltiples reglas para un solo elemento									
	Direccionalidad	Muchos a muchos				**QA**					
		Uno a uno			**QA**	**QA**					
		Muchos a uno				**Q**					
		Uno a muchos									
QVT				2	4	9	6			3	2

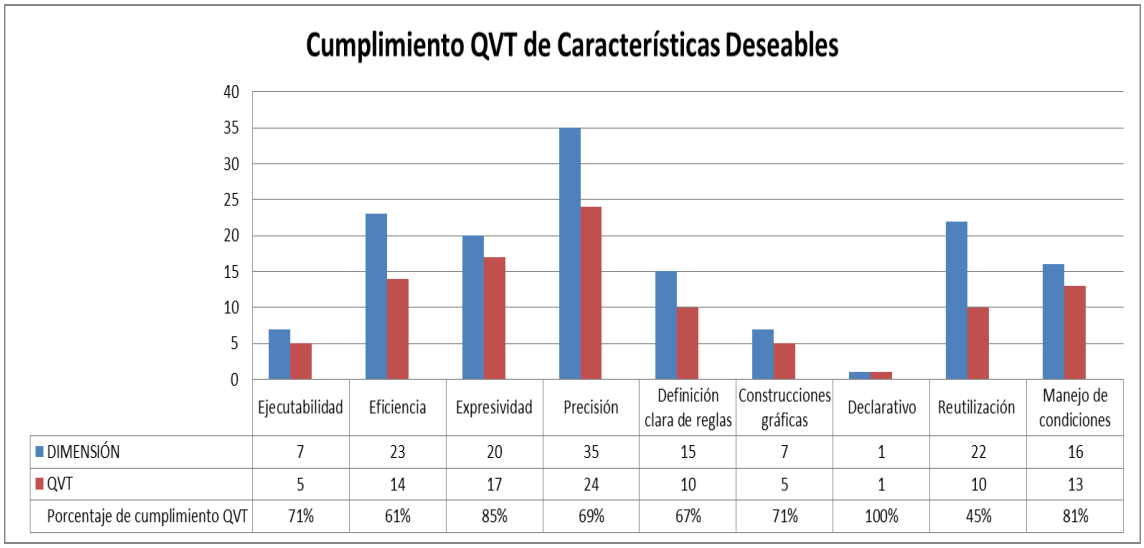
La tabla 23 presenta la relación entre las características deseables y la dimensión según la formalización del lenguaje.

Tabla 23.
Caracteristicas deseables en la dimensión según la formalización del lenguaje

			CARACTERÍSTICAS DESEABLES SEGÚN OMG								
	Características	**Atributos**	Ejecutabilidad	Eficiencia	Expresividad	Precisión	Definición clara de reglas	Construcciones gráficas	Declarativo	Reutilización	Manejo de condiciones
Formalización del lenguaje	**Notación sintaxis abstracta**	Textual	**Q A**		**Q A**						
		Meta modelo			**Q A**	**Q A**		**Q A**			
	Notación sintaxis concreta	Textual	**Q A**		**Q A**						
	Estilo de la definición	Declarativo	**Q A**		**Q A**	**Q A**		**Q A**	**Q A**		
		Imperativo	**Q A**		**Q A**	**Q A**	**Q A**				
	Expresión	Conciso									
		Verboso	**Q A**			**Q A**	**Q A**	**Q A**			
		General									
	Correspondencia de elementos	Por colección		**Q**	**Q**						
		Por tipo		**Q**	**Q**	**Q**					
		Por tipo-preciso		**Q**	**Q**	**Q**					
		Filtrado condicional									
		Reglas sobre clases			**Q**	**Q**	**Q**				**Q**
		Reglas sobre atributos			**Q**	**Q**	**Q**				**Q**
QVT			5	3	10	8	4	3	1		2
ATL			5	0	5	4	2	3	1		0

3.3.3 Análisis de resultados:

La figura 26, presenta la comparación entre los lenguajes QVT y ATL, la cual se obtuvo al contabilizar los atributos de dichos lenguajes.



	Ejecutabilidad	Eficiencia	Expresividad	Precisión	Definición clara de reglas	Construcciones gráficas	Declarativo	Reutilización	Manejo de condiciones
DIMENSIÓN	7	23	20	35	15	7	1	22	16
QVT	5	14	17	24	10	5	1	10	13
Porcentaje de cumplimiento QVT	71%	61%	85%	69%	67%	71%	100%	45%	81%

Figura 26. Cumplimiento de QVT de las características deseables

A continuación se analizan cada una de las características deseables, según los resultados obtenidos de la comparación entre QVT y ATL.

Ejecutabilidad: QVT es un lenguaje que posee una alta ejecutabilidad gracias a su sintaxis, la cual es declarativa he imperativa, esto ayuda a crear un escenario en el cual varios tipos de transformaciones pueden ser definidos, incrementando así el campo de acción del lenguaje, por estas razones los escenarios donde puede ser ejecutable son mayores.

Eficiencia: La eficiencia de QVT se ve afectada debido a la limitada capacidad que tiene el lenguaje para reutilizar elementos, la herencia y la inclusión de librerías son las dos opciones que ofrece el lenguaje para cumplir este cometido, sin embargo, las otras

alternativas, como lo son la habilidad para combinar, componer o encadenar módulos, no se encuentran presentes.

Expresividad: QVT es un lenguaje altamente expresivo, esto gracias a su sintaxis que permite la definición de varios tipos de mecanismos de control, como lo son condiciones de ejecución, filtros sobre objetos, definición de precondiciones y la creación de estereotipos que permiten la creación automática de objetos.

Precisión: QVT maneja un nivel de precisión considerable, esto gracias a que permite no solamente controlar las ejecuciones a través de condiciones, también permite realizar validaciones de la transformación, y correcciones de la misma.

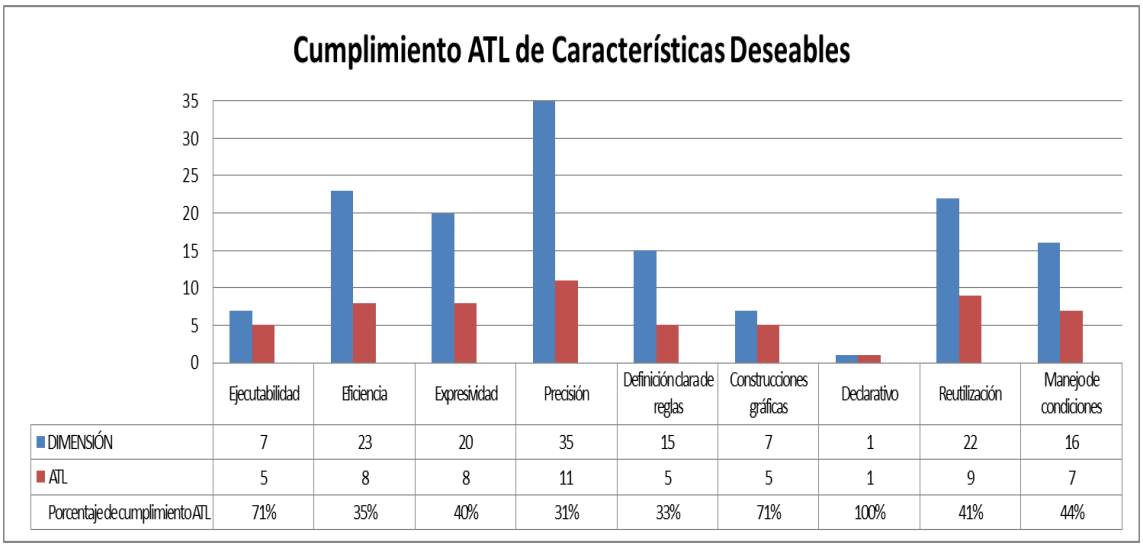
Definición clara de reglas: A pesar de cumplir con más de la mitad de atributos, QVT no posee un alto nivel de definición clara de reglas, debido a su falta de variedad en el manejo de reglas sobre elementos.

Construcciones gráficas: La gran debilidad de QVT en este punto es su característica de ser un lenguaje verboso, y no poseer una opción de construcciones más simples, esto supone un reto al momento de crear transformaciones simples, puesto que es necesario recurrir a complejas estructuras para obtener un resultado.

Declarativo: QVT es un lenguaje que se encuentra definido de forma tanto declarativa como imperativa.

Reutilización: Al igual que la eficiencia, este punto se ve afectado en QVT por las limitadas opciones que se ofrecen para reciclar elementos.

Manejo de condiciones: Otro punto fuerte de QVT, que se ve afectado solamente por su incapacidad de llevar a cabo ingeniería inversa, obtener un meta modelo a partir de un modelo.



	Ejecutabilidad	Eficiencia	Expresividad	Precisión	Definición clara de reglas	Construcciones gráficas	Declarativo	Reutilización	Manejo de condiciones
DIMENSIÓN	7	23	20	35	15	7	1	22	16
ATL	5	8	8	11	5	5	1	9	7
Porcentaje de cumplimiento ATL	71%	35%	40%	31%	33%	71%	100%	41%	44%

Figura 27. Cumplimiento de ATL de las características deseables

Ejecutabilidad: ATL posee una sintaxis con un tipo de paradigma híbrido, es decir declarativo e imperativo, que le permite tener una alta ejecutabilidad, permitiéndole de esta forma, ser ejecutable en transformaciones simples, como en complejas.

Eficiencia: La eficiencia de ATL es un punto importante a tener en cuenta, al ser su sintaxis de tipo verbosa, y no facilitar una opción más concisa de esta, no facilita la creación de transformaciones más eficientes, la falta de construcciones que permitan la reutilización de elementos previamente creados también afectan el desempeño del lenguaje en esta categoria.

Expresividad: ATL no incorpora mayores mecanismos de control sobre los modelos, objetos de un modelo y transformaciones, el lenguaje carece de la habilidad para comprobar los modelos.

Precisión: La precisión de ATL se ve afectada dada su limitada capacidad para controlar la ejecución de una transformación, el no contar con precondiciones de ejecución es la principal causa para que su precisión disminuya.

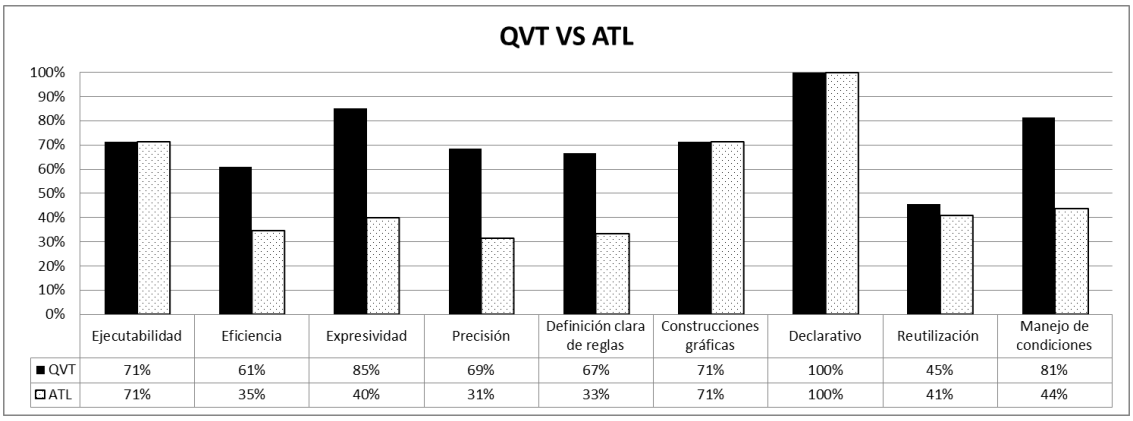
Definición clara de reglas: ATL permite definir dentro de un módulo los helpers que actúan como reglas, sin embargo éstos no permiten la implementación de precondiciones, es decir validaciones previas que deben cumplirse antes que una transformación sea ejecutada.

Construcciones gráficas: A pesar que ATL es un lenguaje netamente textual, permite manipular modelos creados gráficamente, la definición de meta modelos, y su naturaleza híbrida mejora su capacidad de expresar transformaciones que si bien son de forma textual, pueden ser comprendidas en parte al observar los meta modelos de forma gráfica.

Declarativo: ATL es un lenguaje que se encuentra definido de forma híbrida, tanto declarativa como imperativa.

Reutilización: Al igual que la eficiencia, este punto se ve afectado en ATL por las limitadas opciones que se ofrecen para reciclar elementos.

Manejo de condiciones: ATL cumple con esta característica solamente a nivel superficial, ya que no cuenta con opciones para realizar filtrados, precondiciones o correcciones de las transformaciones.



	Ejecutabilidad	Eficiencia	Expresividad	Precisión	Definición clara de reglas	Construcciones gráficas	Declarativo	Reutilización	Manejo de condiciones
■ QVT	71%	61%	85%	69%	67%	71%	100%	45%	81%
□ ATL	71%	35%	40%	31%	33%	71%	100%	41%	44%

Figura 28. Comparación QVT-ATL

Ejecutabilidad: Tanto QVT como ATL son lenguajes que poseen un buen nivel de ejecutabilidad, ya que comparten características similares, ambos lenguajes pueden ser expresados en forma textual, y poseen características tanto de un lenguaje declarativo como imperativo.

Eficiencia: QVT resulta ser un lenguaje más eficiente que ATL, con un nivel de cumplimiento del 61%, mientras que ATL solamente alcanza el 35%, la diferencia entre ambos lenguajes es la capacidad de QVT para manejar comprobaciones de los modelos y transformaciones, así como la capacidad que tiene de realizar correspondencia entre elementos.

Expresividad: ATL alcanza un 40% de expresividad dada su limitada capacidad de control, en contraste con QVT, el cual tiene un 85%, esto gracias a su estructura que

incorpora los condicionantes where y when, que facilitan el control de las transformaciones, incorporando mecanismos de control sobre los modelos, objetos de un modelo y transformaciones.

Precisión: La capacidad de QVT de definir precondiciones que dicten si una transformación se ejecuta o no, supera a ATL en este punto, ya que el segundo no dispone de dichos mecanismos, convirtiéndolo en un lenguaje menos preciso.

Definición clara de reglas: Las reglas de QVT, permiten un campo de acción más amplio que los helpers de ATL, estas incorporan la capacidad de definir si una transformación va a ser ejecutada o no, en ATL simplemente la transformación se ejecuta, y las condiciones dictan como se realiza la transformación.

Construcciones gráficas: Ambos lenguajes poseen características que les permite trabajar con modelos de forma gráfica. Sin existir mayor diferencia entre ambos.

Declarativo: Ambos lenguajes poseen un paradigma declarativo.

Reutilización: QVT ofrece ligeramente mayor capacidad para reutilizar elementos que lo que ofrece ATL, la diferencia radica en que con QVT se puede actualizar un elemento parcialmente debido a la sincronización de elementos, mientras que con ATL no es factible este escenario.

Manejo de condiciones: QVT supera a ATL en este punto ya que tiene un campo de acción más amplio en cuanto a sus condiciones, permite definir validaciones previas, así como filtrar objetos dentro de una transformación.

3.4 CASO PRÁCTICO

El objetivo del caso práctico, es comprobar las características generales definidas, tanto para QVT, como ATL, con las cuales se realizó el estudio comparativo de los lenguajes, en 3.1, 3.2 y 3.3.

3.4.1 Transformación Simple

El siguiente ejemplo muestra cómo realizar una transformación de elementos, en el archivo origen se tienen familias, cada familia cuenta con un apellido (*lastName*) y cada miembro de la familia cuenta con un nombre (*name*) el objetivo es convertir a cada miembro de la familia en una persona, con género, nombre y apellido.

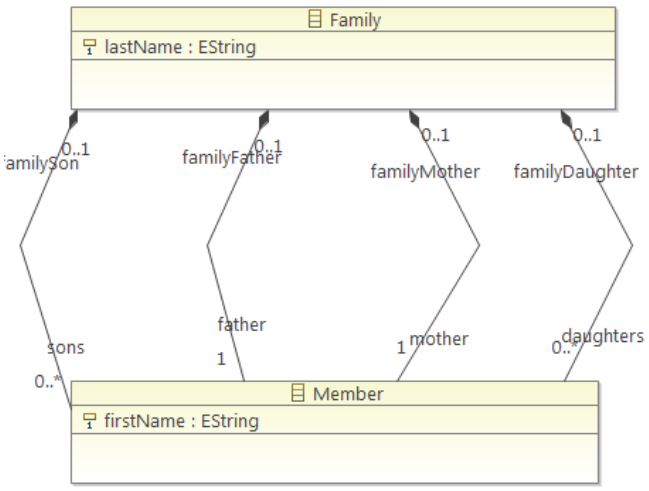


Figura 27. Ejemplo Familias a personas Modelo Origen

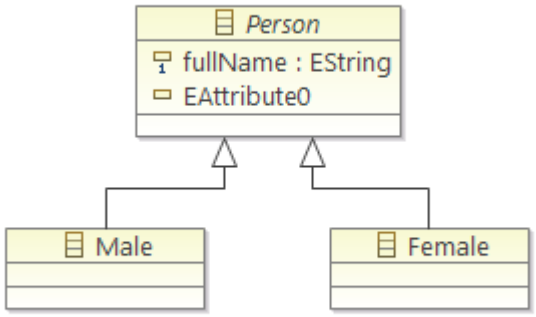


Figura 28. Ejemplo Familias a personas Modelo Destino

Transformación:

```
-- @path Families=/Families2Persons/Families.ecore
-- @path Persons=/Familiest2Persons/Persons.ecore

module Families2Persons;
create OUT: Persons from IN: Families;

helper context Families!Member def: isFemale(): Boolean =
                if not self.familyMother.oclIsUndefined() then
                        true
                else
                        if not self.familyDaughter.oclIsUndefined() then
                                true
                        else
                                        false
                        endif
                endif;
--Helper que verifica si un miembro de una familia es mujer, hija o madre.
helper context Families!Member def: familyName: String =
                if not self.familyFather.oclIsUndefined() then
                        self.familyFather.lastName
                else
                        if not self.familyMother.oclIsUndefined() then
                                self.familyMother.lastName
                        else
```

```
                        if not self.familySon.oclIsUndefined() then
                                self.familySon.lastName
                        else
                                        self.familyDaughter.lastName
                        endif
                endif
        endif;
--Helper que verifica si un miembro de una familia es padre, madre, o hijo

rule Member2Male {
        from
                s: Families!Member (not s.isFemale())
        to
                t: Persons!Male
                (
                        fullName <- s.firstName + ' ' +
s.familyName
                )
}
--Transforma un hombre en un miembro de la familia
rule Member2Female {
        from
                s: Families!Member (s.isFemale())
        to
                t: Persons!Female
                (
                        fullName <- s.firstName + ' ' +
s.familyName
                )
}
```

Transforma una mujer en un miembro de la familia

El archivo origen que será la entrada para la transformación, es el siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns="Families">

        <Family lastName="Perez">
```

```
        <father firstName="Luis"/>
        <mother firstName="Ana"/>
        <sons firstName="Daniel"/>
        <daughters firstName="Luisa"/>
    </Family>
    <Family lastName="Samaniego">
        <father firstName="Pedro"/>
        <mother firstName="Jenny"/>
        <sons firstName="David"/>
        <sons firstName="Diego"/>
        <daughters firstName="Daniela"/>
    </Family>
</xmi:XMI>
```

Una vez ejecutada la transformación, dará como resultado un archivo con la siguiente información:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns="Persons">
  <Male fullName="Luis Perez"/>
  <Male fullName="Daniel Perez"/>
  <Male fullName="Pedro  Samaniego"/>
  <Male fullName="David Samaniego"/>
  <Male fullName="Diego Samaniego"/>
  <Female fullName="Ana Perez"/>
  <Female fullName="Luisa Perez"/>
```

<Female fullName=" Jenny Samaniego"/>

<Female fullName="Daniela Samaniego"/>

</xmi:XMI>

Análisis de la transformación

Esta transformación permite comprender el proceso de transformación utilizando el lenguaje ATL, cada objeto que se encuentra, es decir miembro de una familia, es evaluado para encontrar que valores tiene, y según esto, crear un nuevo objeto, dependiendo el tipo que se haya definido.

El resultado es una lista de personas, que se obtuvo a partir de un archivo que cuenta con una lista de familias y sus integrantes.

3.4.2 Transformación de Clases a RDBMS

En el siguiente caso, se realiza la transformación de un modelo de clases UML a un modelo RDBMS, tanto para QVT como ATL, para lo cual se utilizará el siguiente modelo y meta modelo:

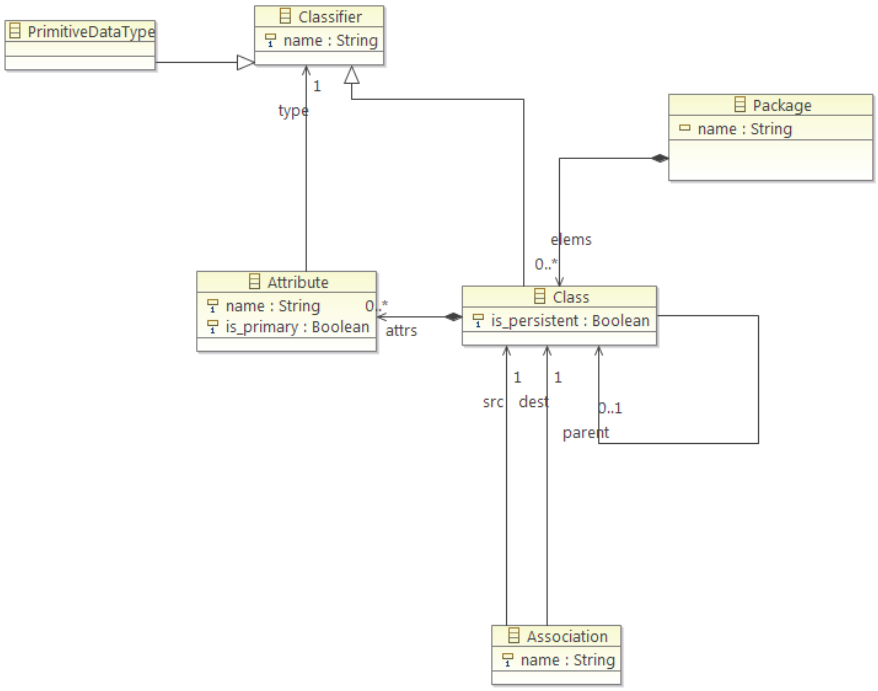


Figura 29. Meta modelo de clases

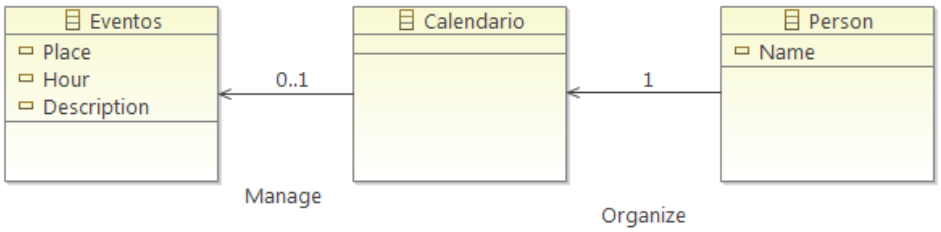


Figura 30. Ejemplo diagrama de clases

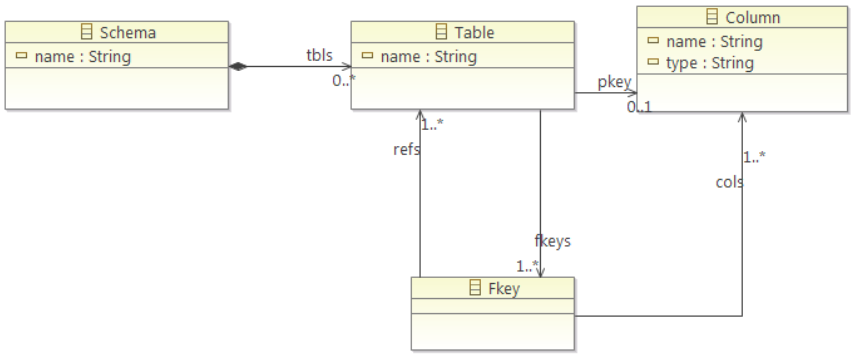


Figura 31. Meta modelo RDBMS

El objetivo de esta transformación es llevar un modelo de clases a un modelo de base de datos, para esto la relación que se genera es la siguiente:

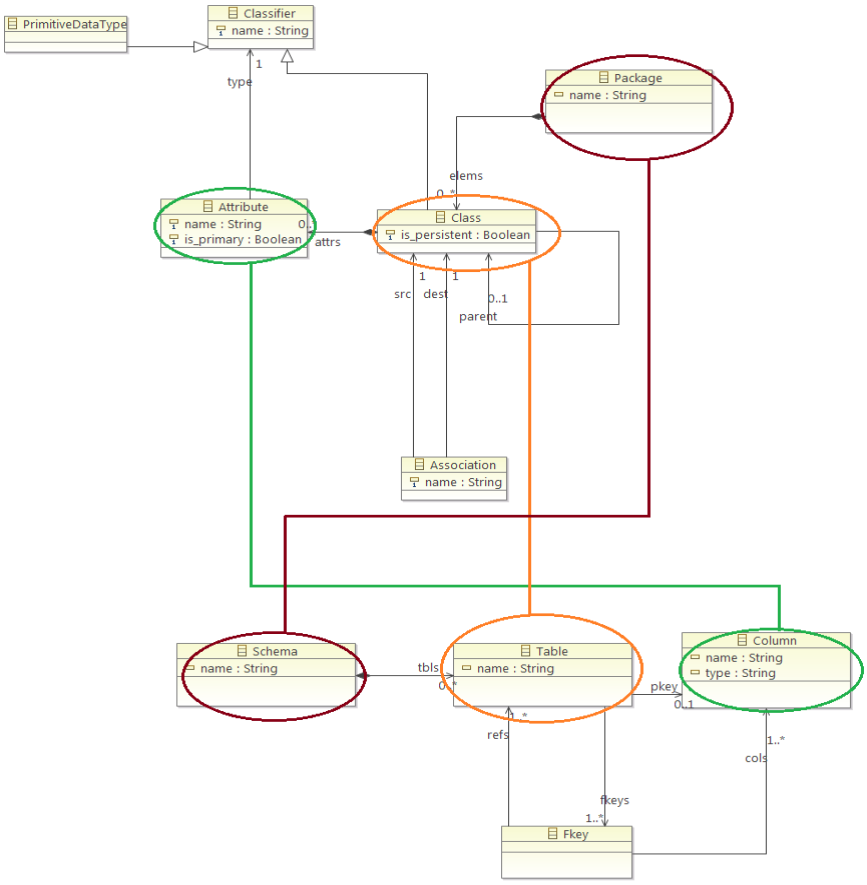


Figura 32. Transformacion de modelo UML a RDBMS

3.4.2.1 Transformación QVT

```
/*
La transformación SimpleUML a RDB demuestra cómo utilizar el lenguaje QVT para llevar un modelo
PIM a un PSM.
En este código se utilizan las siguientes características de QVT:
helper queries, mapping guards, y resolution operations.
/*
modeltype UML uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/simpleuml';
modeltype RDB uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/rdb';

/*
 * Se declara que un modelo UML será el input, mientras que un modelo de tipo RDB será el resultado de
la transformación
/*
transformation Simpleuml_To_Rdb(in uml : UML, out RDB);

/*
 * Punto de entrada de la transformación.
 */
main() {
        uml.rootObjects()[UML::Model]->map model2RDBModel();
}

/*
 * Retorna una instancia del modelo RDB del UML que fue pasado por la función principal main()
*/
mapping UML::Model::model2RDBModel() : RDB::Model {
        name := self.name;
        schemas := self.map package2schemas()->asOrderedSet();
}

/*
 * Este mapeo invoca recursivamente el mapeo package2schema() para producir una secuencia de objetos
a partir del paquete de UML.
 */
mapping UML::Package::package2schemas() : Sequence(RDB::Schema) {
        init {
                result := self.map package2schema()->asSequence()->
                        union(self.getSubpackages()->map package2schemas()->flatten());
        }
}

mapping UML::Package::package2schema() : RDB::Schema
        when { self.hasPersistentClasses() }
{
        name := self.name;
        elements := self.ownedElements[UML::Class]->map persistentClass2table()->asOrderedSet()
}

/*
```

```
 * Mapeo que produce un objeto tipo table en el modelo RDB, a partir de una clase persistente del modelo
UML. La cláusula when utiliza la condición isPersistent() para verificar que la clase sea de tipo
persistente.
 *
 */
mapping UML::Class::persistentClass2table() : RDB::Table
        when { self.isPersistent() }
{
        name := self.name;
        columns := self.map class2columns(self)->sortedBy(name);
        primaryKey := self.map class2primaryKey();
        foreignKeys := self.attributes.resolveIn(
                                        UML::Property::relationshipAttribute2foreignKey,
                                        RDB::constraints::ForeignKey)->asOrderedSet();
}

/*
 * Un objeto de tipo PrimaryKey se crea a partir de una clase al agregarle el prefijo 'PK'.
 */
mapping UML::Class::class2primaryKey() : RDB::constraints::PrimaryKey {
        name := 'PK' + self.name;
        includedColumns := self.resolveoneIn(UML::Class::persistentClass2table,
RDB::Table).getPrimaryKeyColumns()
}

/*
 * Mapeo que crea un conjunto ordenado de objetos tipo TableColumn
 */
mapping UML::Class::class2columns(targetClass: UML::Class) : OrderedSet(RDB::TableColumn) {
        init {
                result := self.map dataType2columns(targetClass)->
                        union(self.map generalizations2columns(targetClass))->asOrderedSet()
        }
}

/*
 * Para el tipo de dato que el mapeo recibe, un objeto de tipo TableColumn es creado
 */
mapping UML::DataType::dataType2columns(in targetType : UML::DataType) :
OrderedSet(RDB::TableColumn) {
        init {
                result := self.map primitiveAttributes2columns(targetType)->
                        union(self.map enumerationAttributes2columns(targetType))->
                        union(self.map relationshipAttributes2columns(targetType))->
                        union(self.map associationAttributes2columns(targetType))->asOrderedSet()
        }
}

mapping UML::DataType::dataType2primaryKeyColumns(in prefix : String, in leaveIsPrimaryKey :
Boolean, in targetType : UML::DataType) : OrderedSet(RDB::TableColumn) {
        init {
                result := self.map dataType2columns(self)->select(isPrimaryKey)->
                        collect(c | object RDB::TableColumn {
                                name := prefix + '_' + c.name;
```

```
                                        domain := c.domain;
                                        type := object RDB::datatypes::PrimitiveDataType {
                                                name := c.type.name;
                                        };
                                        isPrimaryKey := leaveIsPrimaryKey
                                })->asOrderedSet();
        }
}

mapping UML::DataType::primitiveAttributes2columns(in targetType: UML::DataType) :
OrderedSet(RDB::TableColumn) {
        init {
                result := self.attributes->map primitiveAttribute2column(targetType)->asOrderedSet()
        }
}

mapping UML::Property::primitiveAttribute2column(in targetType: UML::DataType) :
RDB::TableColumn
        when { self.isPrimitive() }
{
        isPrimaryKey := self.isPrimaryKey();
        name := self.name;
        type := object RDB::datatypes::PrimitiveDataType { name :=
umlPrimitive2rdbPrimitive(self.type.name); };
}
mapping UML::DataType::enumerationAttributes2columns(in targetType: UML::DataType) :
OrderedSet(RDB::TableColumn) {
        init {
                result := self.attributes->map enumerationAttribute2column(targetType)-
>asOrderedSet()
        }
}
mapping UML::Property::enumerationAttribute2column(in targetType: UML::DataType) :
RDB::TableColumn
        when { self.isEnumeration() }
{
        isPrimaryKey := self.isPrimaryKey();
        name := self.name;
        type := object RDB::datatypes::PrimitiveDataType { name := 'int'; };
}

mapping UML::DataType::relationshipAttributes2columns(in targetType: UML::DataType) :
OrderedSet(RDB::TableColumn) {
        init {
                result := self.attributes->map relationshipAttribute2foreignKey(targetType)->
                        collect(includedColumns)->asOrderedSet();
        }
}

mapping UML::Property::relationshipAttribute2foreignKey(in targetType: UML::DataType) :
RDB::constraints::ForeignKey
        when { self.isRelationship() }
{
        name := 'FK' + self.name;
```

```
	includedColumns := self.type.asDataType().map dataType2primaryKeyColumns(self.name,
self.isIdentifying(), targetType);
	referredUC := self.type.late resolveoneIn(UML::Class::class2primaryKey,
RDB::constraints::PrimaryKey);
}

mapping UML::DataType::associationAttributes2columns(targetType : UML::DataType) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.attributes[isAssociation()]->
			collect(type.asDataType()->map dataType2columns(targetType))-
>asOrderedSet()
	}
}

mapping UML::Class::generalizations2columns(targetClass : UML::Class) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.generalizations.general->map class2columns(targetClass)->flatten()-
>asOrderedSet();
	}
}

query UML::Package::getSubpackages() : OrderedSet(UML::Package) {
	return self.ownedElements[UML::Package]->asOrderedSet()
}

/*
 * Este query realiza una validación de tipos entre UML a UML
 */
query UML::Type::asDataType() : UML::DataType {
	return self.oclAsType(UML::DataType)
}

/*
 * Query que retorna true si la lista de cadenas incluye'primaryKey'.
 */
query UML::Property::isPrimaryKey() : Boolean {
	return self.stereotype->includes('primaryKey')
}

/*
 * Query que retorna true si la lista de cadenas incluye 'identifying'.
 */
query UML::Property::isIdentifying() : Boolean {
	return self.stereotype->includes('identifying')
}

/*
 * Query que retorna true si el tipo del atributo de la propiedad es equivalente al tipo UML
PrimitiveType.
 */
query UML::Property::isPrimitive() : Boolean {
	return self.type.oclIsKindOf(UML::PrimitiveType)
```

```
}

/*
  * Query que retorna true si el tipo del atributo de la propiedad es    equivalente al tipo UML
  Enumeration.
 */
query UML::Property::isEnumeration() : Boolean {
        return self.type.oclIsKindOf(UML::Enumeration)
}
query UML::Property::isRelationship() : Boolean {
        return self.type.oclIsKindOf(UML::DataType) and self.type.isPersistent()
}

query UML::Property::isAssociation() : Boolean {
        return self.type.oclIsKindOf(UML::DataType) and not self.type.isPersistent()
}
query RDB::Table::getPrimaryKeyColumns() : OrderedSet(RDB::TableColumn) {
        return self.columns->select(isPrimaryKey)
}
query UML::ModelElement::isPersistent() : Boolean {
        return self.stereotype->includes('persistent')
}
query UML::Package::hasPersistentClasses() : Boolean {
        return self.ownedElements->exists(
                let c : UML::Class = oclAsType(UML::Class) in
                        c.oclIsUndefined() implies c.isPersistent())
}

helper umlPrimitive2rdbPrimitive(in name : String) : String {
        return if name = 'String' then 'varchar' else
                if name = 'Boolean' then 'int' else
                        if name = 'Integer' then 'int' else
                                name
                        endif
                endif
        endif
}
```

Como resultado se tiene la transformación de cada clase persistente en una tabla, el resultado se obtiene en un archivo en formato XMI:

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:rdb="http://www.eclipse.org/qvt/1.0.0/Operational/examples/rdb">
  <rdb:Model name="model">
    <schemas name="class">
      <elements xsi:type="rdb:Table" name="Persona">
        <columns name="apellido" isPrimaryKey="false">
          <type name="varchar"/>
        </columns>
        <columns name="nombre" isPrimaryKey="true">
          <type name="varchar"/>
```

```
      </columns>
      <primaryKey name="PKPersona" includedColumns="/0/@schemas.0/@elements.0/@columns.1"/>
    </elements>
    <elements xsi:type="rdb:Table" name="Agenda">
      <primaryKey name="PKAgenda"/>
    </elements>
    <elements xsi:type="rdb:Table" name="Evento">
      <primaryKey name="PKEvento"/>
    </elements>
  </schemas>
 </rdb:Model>
 <rdb:TableColumn name="Evento" isPrimaryKey="true">
  <type name="int"/>
 </rdb:TableColumn>
 <rdb:TableColumn name="nombre" isPrimaryKey="false">
  <type name="varchar"/>
 </rdb:TableColumn>
 <rdb:TableColumn name="Descripción" isPrimaryKey="true">
  <type name="int"/>
 </rdb:TableColumn>
 <rdb:TableColumn name="Fecha" isPrimaryKey="false">
  <type/>
 </rdb:TableColumn>
 <rdb:TableColumn name="Fecha" isPrimaryKey="false">
  <type name="int"/>
 </rdb:TableColumn>
</xmi:XMI>
```

3.4.2.2 Transformación ATL

Para realizar la misma transformación utilizando ATL, se genera el siguiente código:

Módulo ATL:

```
module SimpleClasstoSimpleRDBMS;
create OUT : SimpleRDBMS from IN : SimpleClass;

rule PersistentClasstoTable{
        from
                c : SimpleClass!Class (
                        c.is_persistent and c.parent->oclIsUndefined()
                       )

        using {
                primary_attributes : Sequence(TupleType(name : String,
                                                        type : SimpleClass!Classifier,

isPrimary : Boolean)
                                                        ) =
                                        c.flattenedFeatures->select(f | f.isPrimary);
```

```
persistent_features : Sequence(TupleType
                (
                name : String,
                class : SimpleClass!Class,
                offcet : Integer,
                nofAttrs : Integer
                )
        ) =     c.flattenedFeatures->iterate(tuple; acc : Sequence(
                TupleType(
                        name : String,
            class : SimpleClass!Class,
                        offcet : Integer,
                        nofAttrs : Integer))=Sequence{} |
            if tuple.type->oclIsKindOf(SimpleClass!Class)
                        then
                                acc->append
                                (
                                        Tuple{
                                                name=tuple.name,
                                                class = tuple.type,
                                                offcet=if acc-
>size()=0 then 1
                                                        else acc-
>last().offcet + acc->last().nofAttrs
                                                        endif,

        nofAttrs=tuple.type.topParent.flattenedFeatures->select(t | t.isPrimary)->size()
                                                }
                                )
                        else
                                acc
                        endif
        );
        foreign_key_attributes : Sequence(TupleType
                (
                        name : String,
                        type : SimpleClass!Classifier
                )
        ) = persistent_features->collect
                (tuple |
                        tuple.class.topParent.flattenedFeatures->select(t |
t.isPrimary)->collect
                        (a |
                                Tuple
                                        {
```

```
                                                    name=tuple.name + '_' +
a.name,
                                                    type=a.type
                                                }
                                        )
                                )->flatten();

                rest_of_attributes : Sequence(TupleType
                                (
                                        name : String,
                                        type : SimpleClass!Classifier
                                )
                        ) =     c.flattenedFeatures->select
                                (tuple | not tuple.isPrimary and
                                        not tuple.type-
>oclIsKindOf(SimpleClass!Class)
                                );
                }

        to
                t : SimpleRDBMS!Table (
                        name<-c.name,
                                cols<-primary_key_columns-
>union(foreign_key_columns)->union(rest),
                                pkey<-primary_key_columns,
                                fkeys<-foreign_keys
                        ),

                primary_key_columns : distinct SimpleRDBMS!Column foreach
(primAttr in primary_attributes)
                        (
                                name<-primAttr.name,
                                type<-primAttr.type.name
                        ),

                foreign_keys : distinct SimpleRDBMS!FKey foreach (persAttr in
persistent_features)
                                (
                                        references<-persAttr.class.topParent,
                                        cols<-persistent_features->iterate
                                                (tuple;
                                                        acc :
Sequence(Sequence(SimpleRDBMS!Column))=Sequence{} |
                                                        acc->append
                                                                (
```

```
        foreign_key_columns.subSequence(
                                                        tuple.offcet,
                                                        tuple.offcet +
tuple.nofAttrs-1)
                                                        )
                                                )
                                ),

                foreign_key_columns : distinct SimpleRDBMS!Column foreach (attr in
foreign_key_attributes)
                                (
                                        name<-attr.name,
                                   type<-attr.type.name
                                ),

                rest : distinct SimpleRDBMS!Column foreach (attr in rest_of_attributes)
                                (
                                        name<-attr.name,
                                   type<-attr.type.name
                                )
}



helper context SimpleClass!Class def :
        allAttributes : Sequence(SimpleClass!Attribute) =

        self.attrs->union(
                if not self.parent.oclIsUndefined() then
                        self.parent.allAttributes->select(attr |
                          not self.attrs->exists(at | at.name = attr.name)
                        )
                else
                        Sequence {}
                endif
        )->flatten();



helper context SimpleClass!Class def :
        allAssociations : Sequence(SimpleClass!Association) =

        let defAssoc : Sequence(SimpleClass!Association) =
                SimpleClass!Association.allInstances()->select(assoc |
```

```
                    assoc.src = self) in
        defAssoc->union(
                if not self.parent.oclIsUndefined() then
                  self.parent.allAssociations
                else
                  Sequence {}
      endif
        )->flatten();


helper context SimpleClass!Class def :
        attributesOfSubclasses : Sequence(SimpleClass!Attribute) =

        let attrsInSubclasses : Sequence(SimpleClass!Attribute) =
                SimpleClass!Class.allInstances()->select(c |
                        c.parent=self
                )->collect(directSubclass |
                        directSubclass.attributesOfSubclasses
                )->flatten() in
        attrsInSubclasses->union(
                self.attrs->select(attr |
                        not attrsInSubclasses->exists(a |
                                a.name = attr.name)
        ))->flatten();


helper context SimpleClass!Class def :
        associationsOfSubclasses : Sequence(SimpleClass!Association) =

        SimpleClass!Association.allInstances()->select(assoc |
                        assoc.src = self)->union(
                        SimpleClass!Class.allInstances()->select(c |
                                c.parent = self)->collect(subclass |
                        subclass.associationsOfSubclasses)->flatten()
        )->flatten();


helper context SimpleClass!Class def :
        topParent : SimpleClass!Class =
        if self.parent.oclIsUndefined() then
                self
        else
```

```
                self.parent.topParent
        endif;
helper context SimpleClass!Class def :
        flattenedFeatures : Sequence
                (
                        TupleType
                                (
                            name : String,
                                 type : SimpleClass!Classifier,
                                 isPrimary : Boolean
                                )
                ) =
   if self.topParent.is_persistent then
                self.topParent.attributesOfSubclasses->union(
                        self.topParent.associationsOfSubclasses)
        else
                self.allAttributes->union(self.allAssociations)
        endif->collect(f |
                let feature : TupleType
                                        (
                                    name : String,
                                    type : SimpleClass!Classifier,
                                            isPrimary : Boolean
                                       ) =
                if f.oclIsKindOf(SimpleClass!Attribute) then
                        Tuple{name = f.name, type = f.type, isPrimary = f.is_primary}
                else
                        Tuple{name = f.name, type = f.dest, isPrimary = false}
                endif in
        if feature.type.oclIsKindOf(SimpleClass!PrimitiveDataType) then
                feature
        else if not feature.type.topParent.is_persistent then
                feature.type.flattenedFeatures->collect (f | Tuple{name=feature.name+ '_'
+ f.name, type=f.type, isPrimary=f.isPrimary})
        else feature
        endif endif
        )->flatten();
```

Como resultado se tiene la transformación de cada clase persistente en una tabla, el resultado se obtiene en un archivo en formato XMI:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:rdb="http://www.eclipse.org/qvt/1.0.0/Operational/examples/rdb">
  <rdb:Model name="model">
    <schemas name="class">
      <elements xsi:type="rdb:Table" name="Persona">
        <columns name="apellido" isPrimaryKey="false">
          <type name="varchar"/>
        </columns>
        <columns name="nombre" isPrimaryKey="true">
          <type name="varchar"/>
        </columns>
        <primaryKey name="PKPersona" includedColumns="/0/@schemas.0/@elements.0/@columns.1"/>
      </elements>
      <elements xsi:type="rdb:Table" name="Agenda">
        <primaryKey name="PKAgenda"/>
      </elements>
      <elements xsi:type="rdb:Table" name="Evento">
        <primaryKey name="PKEvento"/>
      </elements>
    </schemas>
  </rdb:Model>
  <rdb:TableColumn name="Evento" isPrimaryKey="true">
    <type name="int"/>
  </rdb:TableColumn>
  <rdb:TableColumn name="nombre" isPrimaryKey="false">
    <type name="varchar"/>
  </rdb:TableColumn>
  <rdb:TableColumn name="Descripción" isPrimaryKey="true">
    <type name="int"/>
  </rdb:TableColumn>
  <rdb:TableColumn name="Fecha" isPrimaryKey="false">
    <type/>
  </rdb:TableColumn>
  <rdb:TableColumn name="Fecha" isPrimaryKey="false">
    <type name="int"/>
  </rdb:TableColumn>
</xmi:XMI>
```

3.5 RESULTADO DE LAS TRANSFORMACIONES

Una vez obtenidos los resultados de las transformaciones, estos archivos se pueden migrar a modelos mediante el uso de herramientas que sean compatibles con el formato XMI, en este caso las herramientas de modelado de ECLIPSE, que se encuentran dentro del “framework” de modelado, “ecore”, pueden ser utilizadas con este propósito.

3.5.1 Transformación UML Class a RDBMS en QVT:

En este caso se observa como QVT lleva a cabo la transformación para obtener una tabla a partir de una clase, primero cada elemento u objeto que se encuentra instanciado en el archivo origen, el cual debe corresponder al meta modelo que lo describe, es evaluado, con el fin de definir si es un objeto de tipo "clase" y que tenga la propiedad "persistente", a continuación el código procede a ejecutar la transformación, creando una tabla con el nombre de ese elemento evaluado, las columnas serán creadas únicamente cuando la anterior relación fuese establecida con éxito para ese objeto.

Con este ejemplo se puede comprobar las siguientes características generales de QVT descritas en la tabla 4.

3.5.1.1 Tipo de sintaxis en QVT:

El lenguaje QVT permite la definición de transformaciones mediante una *sintaxis textual* a través de su *lenguaje operacional*, con el cual, el caso práctico fue realizado, sin embargo las construcciones presentadas por el lenguaje son complejas, convirtiéndolo en un lenguaje con *sintaxis verbosa*, esto se puede observar en la siguiente sección del código:

```
result := self.map dataType2columns(self)->select(isPrimaryKey)->
                        collect(c | object RDB::TableColumn {
                                name := prefix + '_' + c.name;
                                domain := c.domain;
                                type := object RDB::datatypes::PrimitiveDataType {
                                        name := c.type.name;
                                };
                                isPrimaryKey := leaveIsPrimaryKey
                        })->asOrderedSet();
```

Este código permite asignar un resultado a la variable “result”, el cual, es el mapeo entre un tipo de dato a columnas, para lo cual se seleccionan los objetos que cumplan las siguientes validaciones: sean llaves primarias, correspondan al tipo descrito, cuenten con un nombre, cuenten con un dominio y cuenten con un tipo.

3.5.1.2 QVT Soporta Meta Modelos:

La siguiente sección de codigo define la ubicación de los meta modelos con los cuales la transformación define la estructura de los modelos de entrada y salida:

```
modeltype UML uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/simpleuml';
modeltype RDB uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/rdb';
```

3.5.1.3 Manejo de Query en QVT:

El lenguaje maneja Querys o consultas, para obtener subconjuntos de elementos, ejemplo:

```
/*
 * Este query realiza una validación de tipos entre UML a UML
 */
query UML::Type::asDataType() : UML::DataType {
        return self.oclAsType(UML::DataType)
}
```

3.5.1.4 Manejo de Estereotipos en QVT

El lenguaje es capaz de definir plantillas o estereotipos con los cuales se define un tipo de dato, el siguiente código muestra una plantilla de un objeto tipo tabla en el ámbito de RDB, el cual indica los elementos con los cuales debe contar una columna: nombre (name), dominio (domain), tipo (type):

```
object RDB::TableColumn {name := prefix + '_' + c.name;domain := c.domain;  type := object
RDB::datatypes::PrimitiveDataType {name := c.type.name;};
```

3.5.1.5 Nivel de abstracción de QVT

Con QVT Operacional es posible realizar transformaciones en el mismo nivel de abstracción (M1), en este caso de estudio se llevó un modelo de clases en UML a un modelo de base de datos RDBMS.

3.5.1.6 Sincronización en QVT

QVT permite manejar sincronización, el modelo destino se actualiza si se realizan cambios en el modelo origen.

3.5.1.7 Transformaciones en QVT

Al ejecutar el módulo, se obtuvo como resultado un archivo XMI, con los elementos transformados, de clases a tablas.

3.5.1.8 Condiciones en QVT

La cláusula "when", permite aplicar condiciones al código, el siguiente es un ejemplo:

```
mapping UML::Property::enumerationAttribute2column(in targetType: UML::DataType) :
RDB::TableColumn
when { self.isEnumeration() }
{isPrimaryKey := self.isPrimaryKey();
        name := self.name;type := object RDB::datatypes::PrimitiveDataType { name := 'int'; };}
```

En esta sección de código, el mapeo se realiza si la función "isEnumeration" se evalúa de forma positiva.

3.5.2 Transformación UML Class a RDBMS en ATL:

Esta transformación permite observar el proceso que ATL lleva a cabo para realizar una transformación, cada elemento u objeto que se encuentra instanciado en el archivo origen, el cual debe ser conforme al meta modelo que lo describe, es evaluado,

con el fin de conocer si es un objeto de tipo “clase” y de tipo “persistente”, en el caso que la evaluación resulte positiva, se crea una tabla con el nombre de esa clase.

Con este ejemplo se pudo comprobar las siguientes características generales de ATL descritas en la tabla 4.

3.5.2.1 Tipo de sintaxis en ATL:

El lenguaje ATL permite la definición de transformaciones de forma textual, con el cual, el caso práctico fue realizado, sin embargo las construcciones presentadas por el lenguaje son complejas, convirtiéndolo en un lenguaje con sintaxis verbosa, esto se puede observar en la siguiente sección del código:

```
helper context SimpleClass!Class def :
        topParent : SimpleClass!Class =
        if self.parent.oclIsUndefined() then   self      else self.parent.topParent
endif;
```

En esta sección de código se define un helper, el cual es el equivalente a una función, su objetivo es validar si una clase está definida como clase padre, en el caso de que no lo sea, la transformación buscará la clase padre del objeto, esto con el objetivo de obtener los atributos de ese objeto, tanto de su clase, como de las clases padre de las cuales herede atributos.

3.5.2.2 ATL Soporta Meta Modelos:

ATL utiliza meta modelos para definir la estructura de los modelos que serán utilizados para realizar las transformaciones.

3.5.2.3 Nivel de abstracción de QVT

Con ATL es posible realizar transformaciones en el mismo nivel de abstracción (M1), en este caso de estudio se llevó un modelo de clases en UML a un modelo de base de datos RDBMS.

3.5.2.4 Transformaciones en QVT

Al ejecutar el módulo, se obtuvo como resultado un archivo XMI, con los elementos transformados, de clases a tablas.

3.5.2.5 Condiciones en ATL

ATL permite la implementación de condiciones mediante la sentencia lógica if, esto, mediante el uso de helpers, permite guiar la transformación y condicionarla, sin embargo ésta se ejecuta sin existir la posibilidad de definir condiciones previas.

```
helper context SimpleClass!Class def :
        topParent : SimpleClass!Class =
        if self.parent.oclIsUndefined() then
                self
        else
                self.parent.topParent
        endif;
```

3.5.2.6 Paradigma Híbrido de ATL:

Sección imperativa: Indica al compilador como realizar una acción, ejemplo:

if acc->size()=0 **then** 1 **else** acc->last().offcet + acc->last().nofAttrs **endif**,

Si el tamaño de la variable acc es = 0 a función retorna 1, caso contrario evalúa la variable para el siguiente valor.

Sección declarativa: Indica al compilador lo que se busca, mas no se precisa como realizarlo:

select(c |c.parent=**self**)->collect(directSubclass|directSubclass.attributesOfSubclasses)->flatten() **in**

Código que solicita realizar una búsqueda de los objetos contenidos en c.

CAPÍTULO 4

4.1 CONCLUSIONES

Se cumplieron los objetivos planteados en el presente proyecto:

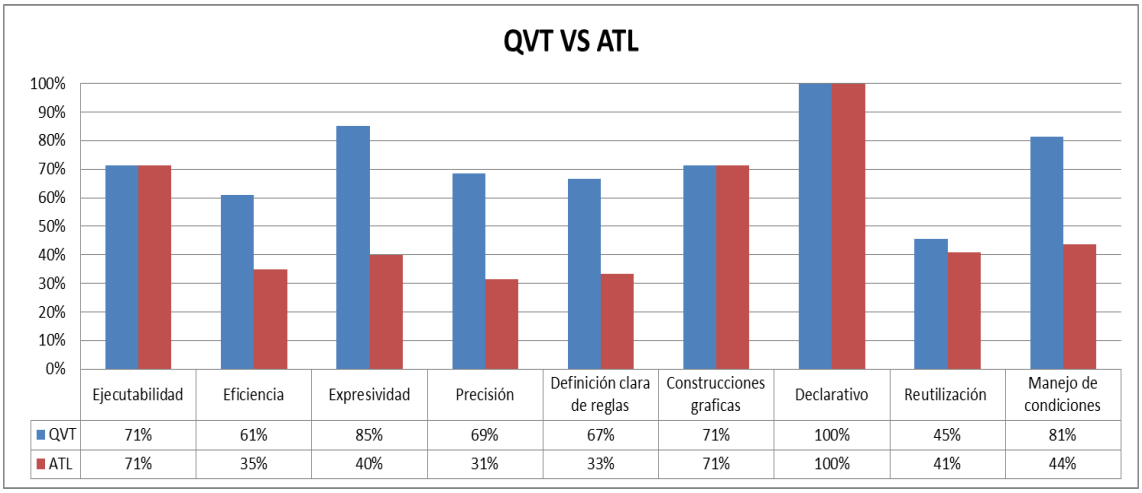
- Se realizó el estudio y comparación de los lenguajes de transformación de modelos QVT y ATL.
- Se llegó a un nivel adecuado de comprensión de los lenguajes estudiados, QVT se enfoca en el desarrollo de software, puesto que su campo de acción es más extenso dadas sus características de eficiencia y expresividad, ATL al ser un lenguaje híbrido, que combina los paradigmas imperativos y declarativos, está pensando para manejar procesos relacionados con la ingeniería de datos.
- Se utilizó como base la implementación de un módulo de agendas para realizar un modelo de clases, el cual permitió el estudio de los lenguajes ponerlos en práctica.
- Se transformó un modelo PIM a PSM básico, realizando la transformación de un modelo de clases a un modelo de base de datos.

El aporte del presente proyecto radica en el desarrollo de un modelo de evaluación comparativo base genérico para lenguajes de transformación de modelos, el cual toma como punto principal las características deseables que MDA define para dichos lenguajes, utilizando como criterios de evaluación los atributos que se encuentran asociados a las siguientes dimensiones: según el uso del lenguaje, según la organización del lenguaje y según la formalización del lenguaje.

Para el desarrollo del modelo en primer lugar fue necesario evaluar las características deseables de un lenguaje de transformación de modelos y asociar los

atributos de cada dimensión a dichas características, de esta forma se obtuvo una primera relación.

En el siguiente paso se analizaron las características definidas para los lenguajes, con dicho análisis fue posible indentificar los atributos en base a las dimensiones definidas, que de cada lenguaje posee. Una vez conocidos los atributos con los cuales cuentan los lenguajes, se procedió a evaluar el nivel de cumplimiento de cada característica deseable, utilizando la primera relación establecida, con lo cual se obtuvo los siguientes resultados:



	Ejecutabilidad	Eficiencia	Expresividad	Precisión	Definición clara de reglas	Construcciones graficas	Declarativo	Reutilización	Manejo de condiciones
QVT	71%	61%	85%	69%	67%	71%	100%	45%	81%
ATL	71%	35%	40%	31%	33%	71%	100%	41%	44%

Figura 33. QVT vs ATL comparación

QVT resultó ser un lenguaje que satisface de forma más completa las condiciones de MDA, encontrándose con una mayor madurez que ATL, especialmente en las áreas de eficiencia, expresividad, precisión, manejo de reglas y condiciones, como puede ser observado en la figura 33.

Ejecutabilidad: Tanto QVT como ATL son lenguajes que poseen un buen nivel de ejecutabilidad.

Eficiencia: QVT cuenta con un nivel de cumplimiento del 61%, mientras que ATL alcanza el 35% de eficiencia deseado por MDA

Expresividad: ATL posee un 40% de expresividad, en contraste con QVT tiene un 85% de expresividad deseada.

Precisión: QVT cumple con un 69% de precisión, ATL cumple con el 31%

Definición clara de reglas: QVT posee un 67% de cumplimiento, puesto que sus condiciones permiten un campo de acción más amplio que los helpers de ATL que solamente permiten alcanzar un 33% de cumplimiento en éste punto.

Construcciones gráficas: Ambos lenguajes poseen características que les permite trabajar con modelos de forma gráfica. Sin existir mayor diferencia entre ambos.

Declarativo: Ambos lenguajes poseen un paradigma declarativo.

Reutilización: QVT ofrece ligeramente mayor capacidad para reutilizar elementos que lo que ofrece ATL.

Manejo de condiciones: QVT supera a ATL en este punto ya que tiene un campo de acción más amplio en cuanto a sus condiciones, permite definir validaciones previas, así como filtrar objetos dentro de una transformación.

Del estudio comparativo realizado se puede observar que las fortalezas de QVT radican en su expresividad con un cumplimiento del 85%, su manejo de condiciones que le atribuyen un 81% de cumplimiento con esta característica y el ser un lenguaje que

maneja un paradigma declarativo. Los puntos fuertes de ATL, son su alta ejecutabilidad con un 71% de complimiento y su capacidad de manejar construcciones gráficas.

QVT mantiene un nivel de completitud mayor que ATL, dada su arquitectura divida, gracias a esto posee una mayor cantidad de atributos en sus diferentes dimensiones, que le permiten cumplir de mejor manera con los requerimientos que MDA presenta para un lenguaje de transformación de modelos, es por esto que QVT puede ser utilizado de forma óptima en el desarrollo de software, mientras que con las características de ATL, lo que se busca es resolver problemas de transformaciones alrededor de la ingeniería de datos.

Las siguientes diferencias fueron encontradas entre los lenguajes al realizar su estudio y comparación:

- Tanto ATL como QVT tienen características que les permiten realizar transformaciones, sin embargo el ámbito de acción de cada uno es diferente.
- QVT posee una mayor cantidad de atributos en sus diferentes dimensiones, que le permiten cumplir de mejor manera con los requerimientos que MDA presenta para un lenguaje de transformación de modelos, es por esto que QVT puede ser utilizado de forma óptima en el desarrollo de software, mientras que con las características de ATL, lo que se busca es resolver problemas de transformaciones alrededor de la ingeniería de datos.
- Las transformaciones implementadas con ATL representaron un reto mayor, ya que aun si la comunidad de ATL es mayor, se encuentra al momento desactualizada, los foros suelen tener un tiempo de respuesta entre 2 a 3

semanas, y en casos existen posts que no han tenido respuesta. Adicional a esto, los ejemplos que se encontraron de ATL se encuentran desactualizados, éstos utilizan definiciones de meta modelos que residen en la web, sin embargo estos sitios ya no cuentan con la definición del meta modelo, aun en la página del proyecto más grande de ATL que es Eclipse.

- Las transformaciones realizadas en QVT fueron más sencillas de implementar, esto debido a que los ejemplos de transformación encontrados (UML a RDBMS) se encontraban ampliamente documentados y actualizados. En cuanto al plug-in de Eclipse, éste mostraba información relacionada con errores de sintaxis o consistencia, por lo cual el proceso de depuración de las transformaciones se llevó a cabo de forma sencilla.
- En cuanto a las transformaciones de ATL, estas tuvieron un mayor nivel de dificultad al momento de su implementación, en gran parte dada la carente información técnica que facilite su implementación y aprendizaje, además, el plug in de eclipse no facilita el depurado, al no contar con un módulo de validación de código y consistencia, en varios vasos se tenían errores de consistencia en el código, sin embargo la transformación se ejecutaba sin mostrar los posibles errores, volviendo muy complicado el ejecutar la transformación de forma correcta.

Si bien las transformaciones pudieron ser ejecutadas, ambos lenguajes no poseen aun un nivel de madurez optimo, el proceso de aprendizaje necesario para implementar dichas transformaciones requiere de un amplio conocimiento en otras áreas, como lo

sería el modelado de diagramas, intercambio de información entre modelos a través de archivos XMI, conceptos y paradigmas de transformación necesarios para implementar dichas transformaciones, a más de comprender ampliamente el lenguaje utilizado.

En cuanto a las metodologías estudiadas, se puede concluir que, si bien MDA comprende que el proceso de modelado es más amplio que simplemente convertir de PIM a PSM, no comprende un enfoque de ingeniería real como lo es MDE, ya que éste comprende una fase de análisis de procesos y arquitectura. Por esta razón MDE tiene como objetivo los siguientes:

Incrementar la productividad a corto plazo.

Incrementar la productividad a largo plazo.

Incrementar la funcionalidad alcanzada por el producto software.

Reducir el impacto por:

Cambios de personal

Modificación de requerimientos

Actualización de plataformas.

En base a la expreciencia adquirida en la realización del presente proyecto, se concluye que ATL es el lenguaje de transformación más usado, con un zoológico de transformaciones considerable y con una comunidad mayor, sin embargo una de sus desventajas es que existen muchos foros sin respuestas, y las transformaciones

encontradas, no se encuentran documentadas a nivel técnico de forma extensa y concisa, lo cual dificulta su aprendizaje.

QVT se encuentra estandarizado al utilizar el enfoque declarativo, las transformaciones fueron más sencillas de implementar ya que se encuentran actualizadas, sin embargo no existe una comunidad que facilite su aprendizaje, esto, dado el hecho que, el proyecto más avanzado de implementación de los módulos de QVT, es el que se encuentra bajo el proyecto de "Model to Model Transformation" por parte del grupo Eclipse, en el cual, el avance que se produce es gracias al aporte voluntario de sus miembros, los cuales poseen un gran nivel de conocimiento en el area, y por lo tanto, la documentación y los casos de estudio que se encuentran documentados, poseen un elevado nivel técnico, creando una brecha de aprendizaje que puede representar un reto considerable de superar para un individuo que comience a manejar este lenguaje.

4.2 RECOMENDACIONES

- El modelo que se propone es genérico y una base para el estudio comparativo de otros lenguajes de transformación de modelos, adicional, el estudio presentado en este trabajo puede ser extendido mediante el análisis de los lenguajes en un entorno de desarrollo, utilizando como base la metodología descrita por el "Model Driven Software Development", el cual dicta los requerimientos necesarios para desarrollar software utilizando modelos.

- Comprender los conceptos de MDA y MDE al momento de manejar lenguajes de transformación de modelos como lo son QVT o ATL, contribuye a un mejor entendimiento del campo de acción de los lenguajes en cuestion.
- La aplicación de MDE viene recomendada para proyectos que tendrán un ciclo de vida largo.
    - Dada la complejidad de generar un meta modelo, normalmente éstos son generados a partir de un sistema ya existente.
- Existe un zoo de modelos tanto para QVT como ATL, con transformaciones ya definidas que puede utilizarse.
- Utilizar un lenguaje que posea un editor capaz de detectar errores de sintaxis, incompatibilidad de tipos, etc.
- El lenguaje a utilizar debe estar seleccionado dependiendo del ámbito en el cual se va a trabajar, en el caso de que se requiera desarrollar software la mejor opción es QVT, dado que está basado en MOF y por lo tanto maneja un estándar más estable, en el caso de requerir realizar ingeniería de datos, ATL sería la mejor opción, puesto que éste soporta utilizar cualquier tipo de modelo, aun si no está definido conforme al MOF.

GLOSARIO

Programación Imperativa

La programación imperativa se realiza en base a describir los estados de un programa y las sentencias que dictan el cambio de un estado a otro, es decir, en la programación imperativa se indica al computador como realizar una tarea.

Ejemplos de lenguajes imperativos son: Basic, C, Pascal, Java, Etc.

Programación Declarativa

La programación declarativa se basa en la declaración de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen un problema y detallan la solución al mismo, ésta es encontrada internamente por el lenguaje, sin la necesidad de especificar como hacerlo.

SQL es considerado un lenguaje declarativo.

Nociones del sistema

Existen dos nociones diferentes del sistema: la teleológica y la noción de sistema ontológico (Dietz y Hoogervorst , 2007). La noción de sistema teleológica es acerca de la función y el comportamiento (externo) de un sistema. Esta noción puede ser visualizada como un modelo de caja negra. La noción de sistema teleológica es adecuada para el propósito de utilizar o controlar un sistema. La noción de sistema ontológica, en el otro lado, se puede utilizar para la construcción o el cambio de un

sistema. Se trata de la construcción y operación de un sistema y puede ser entendido como un modelo de caja blanca.

Tanto la teleológica y la noción de sistema ontológico son relevantes para el diseño de un sistema (Dietz y Hoogervorst , 2007), en otras palabras : tanto el funcional y la perspectiva de la construcción de un sistema son relevantes.

Punto de vista

Un punto de vista sobre un sistema es una técnica para la extracción usando un conjunto seleccionado de los conceptos arquitectónicos y reglas de estructuración, con el fin de centrarse en las preocupaciones particulares dentro de ese sistema. Aquí "abstracción" se utiliza para referirse al proceso de supresión de detalle seleccionado para establecer un modelo simplificado (OMG, 2003).

Plataforma

Una plataforma es un conjunto de subsistemas y tecnologías que proporcionan un conjunto coherente de funcionalidad, a través de interfaces y patrones de usos específicos, que cualquier aplicación soportada por dicha plataforma, puede ser utilizada sin preocuparse por los detalles de cómo se implementa la funcionalidad para la plataforma proporcionada (OMG, 2003).

Independencia de la plataforma

La independencia de plataforma es una cualidad que un modelo puede exhibir. Esta cualidad indica que el modelo es independiente de las características de una

plataforma de cualquier tipo particular, como la mayoría de las cualidades de los modelos, la independencia de plataforma es una cuestión de grado. Por lo tanto, un modelo sólo podría asumir disponibilidad de las características de un tipo muy general de la plataforma, como la invocación remota, mientras que otro modelo podría suponer la disponibilidad de un determinado conjunto de herramientas para otra plataforma.

Del mismo modo, un modelo podría asumir la disponibilidad de una característica de un tipo particular de la plataforma, mientras que otro modelo podría estar plenamente comprometidos con este tipo de plataforma (OMG, 2003) .

Servicios generalizados

Servicios generalizados son los servicios disponibles en una amplia gama de plataformas (OMG , 2003) . En la MDA servicios generalizados se modelan como una plataforma independiente. Ejemplos de ello son los servicios de directorio, manejo de eventos, persistencia, transacciones y seguridad (Soley , 2000) .

Aplicación

El termino aplicación se utiliza para referirse a la funcionalidad que entregan los programas informáticos.

Implementación

Una aplicación es una especificación, que ofrece toda la información necesaria para construir un sistema y ponerlo en funcionamiento (OMG, 2003).

Modelo

El término " modelo " se deriva de la palabra latina módulo, lo que significa medida, norma, modelo, ejemplo a seguir. Una definición formal de modelo puede ser: Cualquier sujeto utilizando un sistema de A que no es ni directa ni indirectamente a la interacción con un sistema B , para obtener información sobre el sistema B, está utilizando una como modelo para B.

Esta definición es bastante genérica, se puede describir el modelo de concepto más precisa mediante la presentación de tres criterios para un modelo. Para que un modelo se defina como tal debe tener las siguientes características:

• Mapeo: un modelo está basado en un original. El original (sistema) puede ser algo aún por construir o puede permanecer completamente imaginario.

• Reducción: no todas las propiedades de la materia se asignan en el modelo, pero el modelo se reduce de alguna manera. Sin embargo, un modelo debe reflejar, al menos, algunas de las propiedades de la materia.

• Pragmático: un modelo tiene que ser utilizable en lugar de un objeto con respecto a algún propósito.

Transformación de modelo

La transformación de un modelo a otro modelo significa que un modelo de fuente se transforma en un modelo de destino sobre la base de algunas reglas de transformación. Diferentes métodos pueden ser utilizados para definir las reglas de

transformación. En 2007, el OMG libera QVT, una especificación del lenguaje de transformación de modelos.

Se puede hacer una distinción entre las transformaciones (o mejoras) la adición de información computacional y transformaciones añadiendo la tecnología (o plataforma) de la información. El primer caso precisa de la intervención humana, el último se puede hacer de forma automática si se alimenta la transformación con la información de la plataforma.

Aunque ambos ejemplos son transformaciones entre capas de abstracción, difieren en ejecutabilidad. Es por eso que se puede distinguir entre las transformaciones automáticas y manuales.

Otro tipo de transformación que se destaca, es una transformación donde se propaga el cambio. En lugar de volver a ejecutar la transformación completa, sólo los cambios se propagan al modelo de destino. El uso de una transformación de propagación de cambio permite la sustitución de un componente en tiempo de ejecución o con la preservación de los datos de tiempo de ejecución. Un ejemplo de tal transformación es la actualización de la estructura de una base de datos (el modelo de destino) en base a un modelo de objetos (el modelo de origen) sin perder datos. Esto, por supuesto, no es posible para cada cambio.

Bibliografía

Brent Rector, C. S. (1999). *ATL internals.* Addison-Wesley.

ECLIPSE GROUP. (s.f.). *ATL/User Guide - The ATL Language.*

Engels, G. (2007). *Model Driven Engineering Languages and Systems.* Nashville, USA: Springer.

Favre, L. (2010). *Model Driven Architecture for Reverse Engineering Technologies.* IGI Global Snippet.

Jeff Z. Pan, S. S. (2012). *Ontology-Driven Software Development.*

Jesse Russell, R. C. (2012). *Atlas Transformation Language.*

Juan C. Herrera, F. L. (2010). *Una evaluación por dimensiones para los lenguajes de transformación de modelos.*

Juan de Lara, A. Z. (2012). *Fundamental Approaches to Software Engineering.* Tallinn, Estonia: Springer.

Kurtev, I. (2005). *Adaptability of Model Transformations.* University of Twente.

Lambert M. Surhone, M. T. (2010). *SmartQVT.* VDM Publishing.

Marco Brambilla, J. C. (2012). *Model-driven Software Engineering in Practice.* Morgan & Claypool Publishers.

Markus Völter, T. S. (2013). *Model-Driven Software Development.* John Wiley & Sons.

MOF. (2008). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.*

OMG. (2001). *Model Driven Architecture - Una Perspectiva Técnica.*

OMG. (2007). *Meta Object Facility*.

OMG. (2010). *Object Constraint Language.*

Paige, R. F. (2009). *Theory and Practice of Model Transformations.* Zürich, Switzerland: Springer.

Richard F. Paige, A. H. (2009). *Model Driven Architecture - Foundations and Applications.* Springer.

Roland Petrasch, F. F. (2009). *Model Driven Software Engineering - Transformations and Tools.* Logos Verlag Berlin.

Simmonds, D. M. (2007). *Transforming UML Class Models.* Colorado State University.

Tom Armstrong, R. P. (2000). *ATL developer's guide.* M&T.

Biografía

Nombre: Francisco Xavier Eremiev Burneo

Nacionalidad: Ecuatoriana

Lugar de nacimiento: Loja

Fecha de nacimiento: 09 de Marzo de 1988

Instrucción Primaria

Nombre: La Salle

Período: 1994-2000

Instrucción Secundaria

Nombre: Colegio Particular Eugenio Espejo

Período: 2001-2006

HOJA DE LEGALIZACIÓN DE FIRMAS

ELABORADO POR

_____

Francisco Xavier Eremiev Burneo

DIRECTOR DE LA CARRERA

_____

Ing. Mauricio Campaña

Sangolquí, Septiembre de 2014