UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Reglas de traducción de restricciones entre OCL y LN
Autores: Lopez, Danae Claudia – Ibargüengoytia, María Amalia
Director: Dra. Pons, Claudia
Codirector:
Asesor profesional:
Carrera: Licenciatura en Sistemas

Resumen

El Desarrollo de Software Dirigido por Modelos es un paradigma que ayuda a las compañías de desarrollo con la gestión de los sistemas que construyen y mantienen; para adaptarse rápidamente a los cambios tecnológicos. Permite generar modelos altamente abstractos, utilizando modelos gráficos como Ecore. Si bien estos modelos son expresivos no permiten describir toda la información que debería mostrar el modelo. Para reducir este problema, los lenguajes formales permiten incrementar la expresividad, aunque resultan más complejos.
El lenguaje formal OCL es difícil de entender por personas que no posean conocimientos sobre matemáticas, lógica e inclusive orientación a objetos lo que hace compleja su utilización como extensión de un modelo con un nivel de abstracción muy alto. Esta desventaja hace que se deban realizar tareas manuales extras, como traducir las restricciones OCL a lenguaje natural para poder presentar un modelo completo a alto nivel a personas sin conocimientos técnicos. El objetivo principal de la presente tesina es generar una herramienta que permite realizar la traducción de dichas restricciones a lenguaje natural de forma automática mediante el uso de transformación de modelos utilizando una gramática de lenguaje natural reducida. Lo que se intenta lograr es fomentar el uso de OCL restringiendo sus limitaciones.

Palabras Claves

Desarrollo dirigido por modelos (MDD), Metamodelos, Modelos.
Transformación de Modelos, Transformaciones modelo a modelo (M2M), Transformaciones modelo a texto (M2T).
Gramáticas, BNF, EBNF, Xtext.
OCL, ATL.
Eclipse, Ecore, MOF.
Lenguaje Natural.
Traducción OCL a Lenguaje Natural.

Trabajos Realizados

Investigación teórica sobre MDD, Lenguaje Natural, OCL, ATL y Xtext.
Creación de una gramática de Lenguaje Natural reducido al dominio del problema que se intenta resolver mediante el uso de la herramienta Xtext.
Desarrollo de un plugin Eclipse para la traducción de restricciones en OCL a Lenguaje Natural Reducido utilizando reglas de transformación definidas en ATL.

Conclusiones

En está tesina hemos mencionado el problema de usabilidad de OCL debido a lo dificultosa que resulta su escritura y entendimiento para usuarios que no poseen los conocimientos técnicos necesarios. Hemos presentado una herramienta para la traducción automática de restricciones en OCL, definidas por el usuario, a lenguaje natural. Está herramienta es el resultado de nuestra investigación con el fin de mejorar la usabilidad de OCL. El desarrollo se ha basado en MDD y transformaciones mediante el uso de ATL.

Trabajos Futuros

Extender la implementación de manera de permitir la traducción de invariantes que utilicen definition, let, reject(), allInstances() y de métodos junto con sus precondiciones, postcondiciones y body.
Mejorar la detección automática de género y número en la transformación.
Generar las reglas de traducción ATL que permitan la traducción inversa a la desarrollada en este trabajo, desde invariantes en lenguaje natural a su equivalente en OCL.

Fecha de la presentación: Mayo 2017

Agradecimientos

Agradezco especialmente a mi compañero, y esposo, Alex por toda su paciencia y su apoyo desde que decidí realizar esta tesina que deje de lado cuando decidimos mudarnos a vivir a Londres.
A mis abuelas que, aunque hoy ya no están, fueron el impulso más importante para terminar está carrera porque sin todos los mates que me cebaron cada mañana al despertarme para ir a la facultad y durante largas jornadas de estudio no lo hubiera logrado.
A mis padres por brindarme la posibilidad de estudiar, creer en mí y por sobre todo por los valores que me enseñaron que hacen de mi la persona que soy.

Danae

Agradezco a Papá y Mamá por el gran esfuerzo que hicieron para darme la posibilidad de estudiar, no lo podría haber logrado sin ustedes, gracias por enseñarme a ser quien soy y por inculcarme los valores que tengo. A mis hermanos y sobrinos por darme siempre amor.
A mi novio, mi compañero incondicional que siempre me alentó para que no baje los brazos y se bancó todos mis momentos y me supo entender y acompañar.
Pero sobre todas las cosas se lo dedico a mi Mamá, que me enseñó a nunca bajar los brazos y nunca dejar de luchar, sos un ejemplo de que todo se puede lograr, no dejes nunca de sonreír, gracias.

Amalia

Agradecemos a Claudia por todo su apoyo, dedicación y paciencia.

Amalia y Danae

Índice

Índice figuras

Capítulo 1: Introducción

1.1 - Objetivo

Esta tesina no posee un único objetivo sino un conjunto de ellos que deben ser cumplidos con el fin de completar el objetivo final de desarrollo. El conjunto de objetivos definidos son:

- Afianzar y resumir conceptos utilizados como base de esta tesina: Desarrollo de Software basado en modelos, metamodelos, modelos, lenguajes específicos del dominio, transformaciones de modelos.
- Investigar sobre lenguajes formales, específicamente sobre el Lenguaje de Restricciones de Objetos (OCL).
- Investigar características de los lenguajes naturales, específicamente castellano. Analizar su composición sintáctica y semántica.
- Investigar la factibilidad de la traducción automática entre Lenguaje de Restricciones de Objetos y Lenguaje Natural.
- Prototipar una herramienta para el proceso de traducción entre Lenguaje de Restricciones de Objetos y Lenguaje Natural.

1.2 - Motivación

En los últimos años se ha experimentado un crecimiento muy grande alrededor de lo que se conoce como Desarrollo de Software dirigido por Modelo (MDD, *Model Driven Development*) un nuevo paradigma que surge como respuesta a los principales problemas a los que se enfrentan actualmente las compañías de desarrollo de software:

1. Gestionar la creciente complejidad de los sistemas que construyen y mantienen.
2. Permitir adaptarse a la rápida evolución de las tecnologías software.

Las razones por las que MDD, y en particular la Arquitectura dirigida por modelos (MDA, *Model Driven Architecture*) es una propuesta adecuada para enfrentarse a los problemas presentados anteriormente son:

1. Se utilizan modelos para representar todos los artefactos de software, no sólo los sistemas. Cada uno de los modelos se enfoca en un aspecto del sistema que puede ser definido a un nivel de abstracción elevado e independiente de la plataforma tecnológica utilizada. Al disminuir la dependencia de los modelos de las plataformas logramos que la evolución de las plataformas tecnológicas no afecten al sistema.
2. Se consigue también proteger gran parte de la inversión que se realiza en la informatización de un sistema, pues los modelos son los verdaderos artífices de su funcionamiento final y, por tanto, no será necesario empezar desde cero cada vez que se plantee un nuevo proyecto o se desee realizar algún tipo de mantenimiento sobre el producto.
3. Se disminuyen los costos asociados con la creación de documentación ya que los modelos constituyen la documentación del sistema. Esto también aumenta la mantenibilidad ya que las implementaciones se realizan automáticamente a partir de la documentación.
4. Como se mencionó anteriormente una de las características de MDD es que permite generar modelos con altos niveles de abstracción, tal es el caso de la representación gráfica de modelos como los diagramas de clases del lenguaje unificado de modelado

(UML, *Unified Modeling Language*) o los modelos gráficos en Ecore. Estas representaciones son muy apropiadas para representar modelos de forma visual pero no poseen la expresividad suficiente como para describir toda la información con la que debería contar un modelo. Para incrementar la expresividad de estos modelos se suelen utilizar lenguajes formales que permitan la definición de restricciones sobre dichos modelos.

El Lenguaje de Restricciones de Objetos (OCL, *Object Constraint Language*) es un lenguaje formal, tipado, basado en teoría de conjuntos y lógica de primer orden que permite la definición de restricciones sobre un modelo. Este lenguaje posee ciertas desventajas:

1. Debido a su naturaleza, OCL es difícil de entender por personas que no posean los conocimientos sobre matemáticas, lógica e inclusive orientación a objetos, lo que hace compleja su utilización como extensión de un modelo con un nivel de abstracción muy alto.
2. Por las razones presentadas en 1., resulta muy difícil escribir restricciones OCL si no se poseen conocimientos técnicos necesarios.

Estas desventajas hacen que se deban realizar tareas extras y manuales, como traducir las restricciones OCL a lenguaje natural para poder presentar un modelo completo a alto nivel, y contar con un miembro especializado en OCL para la escritura de restricciones. En cierto modo estas desventajas de la utilización de OCL contradicen las ventajas presentadas por la utilización de modelos con respecto a la reducción de costos y tiempos.
Una posible solución a estos problemas sería contar con un conjunto de reglas que definan cómo traducir restricciones OCL a lenguaje natural y partiendo de ellas desarrollar un prototipo de una herramienta que automatice todo el proceso de traducción.

1.3 - Desarrollos propuestos

- Realizar transformación de modelos (traducción) mediante el uso de un lenguaje de transformación.
- Desarrollo de un modelo de prueba con restricciones en OCL para analizar el proceso de generación de lenguaje natural.
- Prototipación de una herramienta que permita procesar un archivo con restricciones OCL y traducirlas al lenguaje natural.

1.4 - Resultados esperados

Se pretende fomentar el uso de OCL como lenguaje de definición de restricciones sobre modelos gráficos proveyendo una herramienta para la traducción a lenguaje natural permitiendo así que personas con bajo o nulo conocimiento de OCL sean capaces de interpretar modelos con dichas restricciones.

1.5 - Organización de la tesina

En el **capítulo 2** se realiza una introducción al desarrollo de software dirigido por modelos que incluye la definición de los conceptos de modelo, metamodelo, transformaciones que se consideran fundamentales. Se describe la arquitectura de 4 capas y se introduce MOF.
En el **capítulo 3** se presenta un marco teórico del lenguaje natural.
El lenguaje formal OCL se presenta en el **capítulo 4** donde se busca mostrar los conceptos claves, sintaxis y semántica de forma de obtener los conocimientos necesarios del lenguaje.

Los diferentes tipos de transformaciones se exponen en el **capítulo 5.** Se define el concepto de transformación, se describen los distintos tipos de transformaciones ya sea transformaciones de modelo a modelo (M2M) o transformaciones de modelo a texto (M2T) y se presentan las diferentes herramientas que se investigaron para realizar las transformaciones.
Las herramientas seleccionadas para el desarrollo de la tesina se muestran en el **capítulo 6**, donde lo que se busca es profundizar en los conceptos basando el enfoque en las herramientas elegidas.
En el **capítulo 7** se analizan diferentes tipos de invariantes en OCL y el proceso de traducción desde el metamodelo OCL al metamodelo de Lenguaje Natural Reducido mediante el uso de reglas definidas en el lenguaje de transformación de modelos ATL.
La arquitectura de la herramienta se expone en el **capítulo 8** junto con un caso de estudio para mostrar cómo utilizar la herramienta desarrollada.
En el **capítulo 9** se presentan trabajos relacionados y se realiza una comparación con el método desarrollado en está tesina.
Se presentan las conclusiones en el **capítulo 10** y se mencionan los trabajos futuros que pueden realizarse a partir del trabajo presentado.

Capítulo 2: Desarrollo de Software dirigido por modelos

2.1- Desarrollo de software dirigido por modelos (MDD)

MDD es un nuevo paradigma de desarrollo de software que surge ante la necesidad de mejorar el proceso de construcción de software basado en modelos (MBD *Model Based Development*). La diferencia principal entre estos paradigmas es que en MDD los modelos son parte fundamental del proceso, mientras que en el MBD los modelos son solo parte del proceso.
En MDD los modelos se generan partiendo de los más abstractos a los más concretos a través de diferentes fases que los transforman y/o refinan hasta conseguir en la última fase la generación del código fuente. Debido a esto, podemos decir que la transformación de modelos es uno de los pilares más importantes de MDD.

2.2.1 - Elementos clave

2.2.1.1 - Abstracción

En MDD para obtener una mayor abstracción se definen lenguajes de modelado específicos del dominio, los cuales muestran los conceptos del dominio del problema, y ocultan o minimizan los aspectos relacionados con las tecnologías de implementación. Las formas sintácticas utilizadas por estos lenguajes, dejan ver la naturaleza de los conceptos del dominio. Además, si bien el desarrollo de aplicaciones se ve afectado por los cambios tecnológicos, el impacto se ve reducido a través de la abstracción mediante el uso de modelos que permiten que un mismo modelo abstracto se pueda representar en diferentes plataformas de software.

2.2.1.2 - Automatización

Para aumentar la productividad y calidad el método más eficiente es la automatización. Por lo que la intención de MDD es recolectar todas las tareas repetitivas que puedan ser realizadas con mayor eficiencia usando una computadora y automatizarlas. Entre otras cosas, esto incluye, la capacidad de transformar modelos expresados mediante conceptos de alto nivel, específicos del dominio, en su equivalente programa informático ejecutable sobre la plataforma tecnológica correspondiente. Por otra parte, las transformaciones se realizan utilizando herramientas que permiten aplicar patrones y técnicas de reconocido éxito favoreciendo la fiabilidad del producto.

2.2.1.3 - Estándares

MDD se implementa mediante estándares industriales abiertos. Estos estándares son normas que permiten el intercambio de especificaciones entre herramientas que sean complementarias, o herramientas que son equivalentes y corresponden a diferentes proveedores.
Por lo tanto, los estándares permiten que los fabricantes de herramientas enfoquen sus esfuerzos en su área de experiencia sin tener que recrear funcionalidad que ha sido producida por otro fabricante. Por lo que una herramienta que transforma modelos no necesita enfocar sus esfuerzos en la edición de modelos, si no que puede utilizar una herramienta de edición de modelos que fue generada por otro fabricante y ajustarse a sus estándares.

2.2 - Problemas que el MDD viene a resolver

***Problema de productividad, mantenimiento y documentación*.**
En un desarrollo basado en modelos, a medida que se avanza en la codificación se va perdiendo la relación entre los diagramas y el código. Como se muestra en la figura 2-1, suele ocurrir que los cambios se realizan sobre el código y, por falta de tiempo, los desarrolladores dejan de actualizar los diagramas y documentos relacionados.

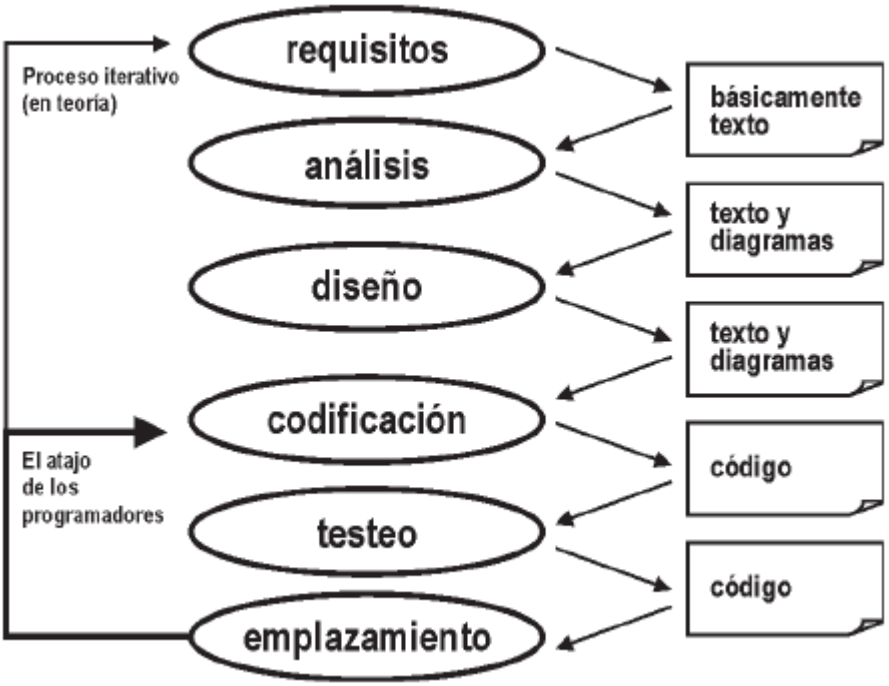


Figura 2-1. Ciclo de vida del desarrollo de software basado en modelos

Existe la creencia de que escribir código es productivo pero realizar modelos y generar documentación no lo es, lo que genera que a medida que el sistema crece el mantenimiento se vuelve complejo ya que resulta dificultoso poder entender el sistema basándose únicamente en el código. Además, se debe considerar que las personas que se encargaron del desarrollo pueden dejar de ser empleados en algún momento o que nuevos empleados necesitan entender cómo funciona el sistema y el negocio en general; y es en estos casos donde la documentación y los diagramas se transforman en herramientas muy valiosas si se encuentran actualizadas.
Una posible solución, aunque sólo parcial, es la posibilidad de generar documentación directamente sobre el código pero solo resuelve el problema en los niveles más bajos ya que los documentos y diagramas de más alto nivel deben ser actualizados manualmente.

El problema de la flexibilidad a los cambios tecnológicos.
A diferencia de lo que ocurre en la mayoría de las industrias, la industria de software se caracteriza por ser la que se renueva más a menudo. Nuevas tecnologías surgen constantemente, y algunas llegan a ser muy populares (algunos ejemplos son Elasticsearch, Scala, ruby, Node.js, AngularJS, etc), lo que lleva a que las compañías migren a estas nuevas tecnologías. Existen diferentes razones por las cuáles estas migraciones suceden:

- Las nuevas tecnologías se adaptan mejor al problema que se intenta resolver.
- Las herramientas utilizadas actualmente dejan de proveer soporte.
- Por exigencia de los clientes.

- Por un compromiso constante de la compañía de utilizar siempre lo último en tecnología.

Como consecuencia de estos cambios, las inversiones que se realizan en tecnología se renuevan casi constantemente. También puede suceder que no se requiera una migración del sistema actual a nuevas tecnologías, sino que sea necesario comunicarse con sistemas que estén implementados en nuevas tecnologías.
El proceso de adopción de nuevas tecnologías suele acarrear un alto costo para las compañías de desarrollo se software y el hecho de que estos cambios se requieren cada vez más seguido hace surgir el problema de la flexibilidad tecnológica que MDD viene a resolver de manera exitosa.

2.3 - Tipos de modelos

2.3.1 - Modelo independiente de la computación (CIM)

Un modelo independiente de la computación (CIM, *Computational Independent Model*) como su nombre lo indica, brinda una visión del sistema independiente de la computación, es decir no muestra detalles de la estructura de los sistemas. También se lo suele llamar modelo de dominio y para su construcción se utiliza un vocabulario conocido por los experimentados en este dominio.
Se supone que las personas experimentadas en el dominio no conocen los modelos o artefactos que se utilizaran en la implementación del sistema. El rol principal del CIM es disminuir la grieta entre los expertos en el dominio y sus requerimientos, y los expertos en el diseño y la construcción de los componentes.

2.3.2 - Modelo independiente de la plataforma (PIM)

Un modelo independiente de la plataforma (PIM, *Platform Independent Model*), es un modelo que posee un alto nivel de abstracción que lo hace independiente de la tecnología y/o el lenguaje que se utilizara en la implementación.
Los modelos PIM pueden ser implementados sobre cualquier plataforma gracias a su alto nivel de abstracción y se modela enfocándose en cómo el sistema será utilizado para brindar soporte al negocio, lo que hace a este modelo totalmente independiente de las aplicaciones informáticas o tecnológicas que serán utilizadas para proveer dicho soporte.

2.3.3 - Modelo específico de la plataforma (PSM)

Un modelo específico de la plataforma (PSM, *Platform Specific Model*) es un modelo que se obtiene como resultado de refinar un modelo PIM para adaptarlo a los servicios y mecanismos ofrecidos por una plataforma concreta. Un PIM puede ser transformado en uno o múltiples PSMs (cada PSM es definido para una tecnología diferente pero interactúan entre sí para proveer una solución completa).
Un PSM es una combinación entre las especificaciones definidas en el PIM y los detalles que definen cómo ese sistema usa un tipo particular de plataforma. Este modelo puede proveer más o menos detalles dependiendo del propósito del mismo, es decir será una implementación, si provee toda la información necesaria para construir un sistema y ponerlo en operación, o puede actuar como un PIM que es usado para realizar otro refinamiento y obtener un PSM que puede ser directamente implementado.
Una plataforma es un conjunto de subsistemas y tecnologías que describen la funcionalidad de una aplicación a través de interfaces y patrones específicos, facilitando que cualquier

sistema que vaya a ser implementado sobre dicha plataforma pueda hacer uso de estos recursos sin tener en consideración aquellos detalles que son relativos a la funcionalidad ofrecida por la plataforma concreta [10].

2.3.4 - El modelo de implementación (Código)

El modelo de implementación (IM, *Implementation Model*), es un modelo que describe de qué manera un modelo PSM se implementará en términos de una tecnología en particular. Es la transformación de un modelo ya definido para una plataforma específica a código fuente.

2.3.5 - Ciclo de vida

En la figura 2-2 se muestra el ciclo de vida del desarrollo de software usando MDD. En este ciclo se identifican las mismas fases que en el ciclo de vida tradicional, lo que los diferencia son los tipos de modelos formales que se generan durante el proceso. Los tipos de modelos son los detallados anteriormente CIMs, PIMs, PSMs, IMs.

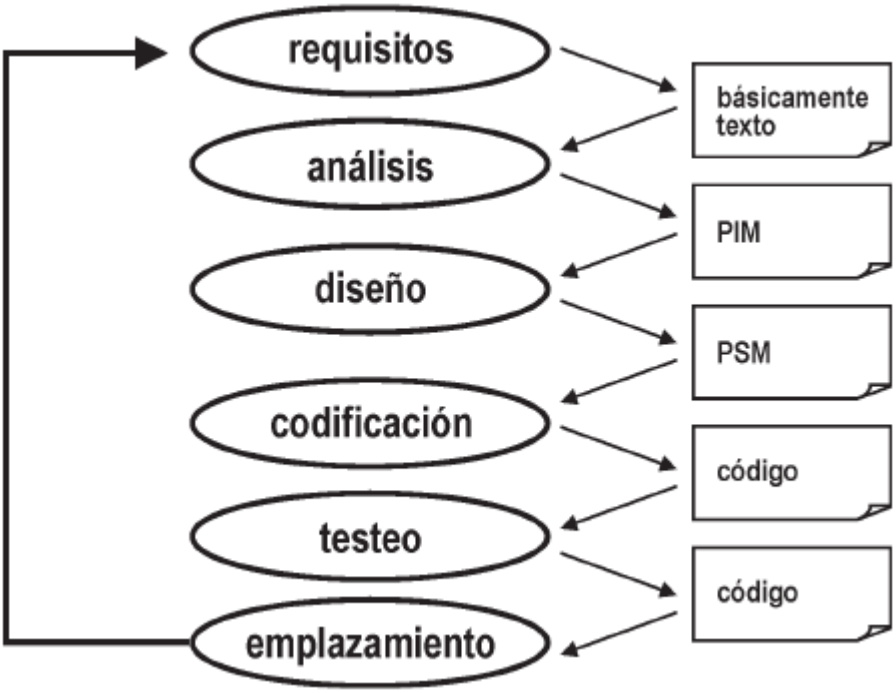


Figura 2-2. Ciclo de vida del desarrollo de software dirigido por modelos

2.3.6 - Otras clasificaciones

Modelos abstractos y detallados. Está es una clasificación subjetiva ya que en general, las herramientas de modelado permiten observar un mismo modelo en distintos niveles de detalle agregando u ocultando información. Por ejemplo, en un diagrama de clases podemos limitar la información visualizada a solo los nombres de clases y limitar los niveles de jerarquía a un número máximo. Luego podemos agregar más detalle incluyendo las operaciones, eliminando la restricción en la cantidad de niveles de la jerarquía a ser visualizados, etc.

Modelos de negocio y modelos de software. Los primeros describen un negocio o empresa aunque también pueden representar solo una porción de ellos y el lenguaje

utilizado contiene un vocabulario que permite especificar los procesos del negocio, los clientes, los departamentos, las dependencias entre procesos, etc sin describir detalles acerca del software utilizado. El modelo de software en cambio, es una descripción del sistema de software que se utiliza para soportar el negocio o cierta parte de él. Usualmente los requisitos del sistema (o sistemas) de software son derivados del modelo del negocio al cual el software dará soporte, por lo tanto existe una relación entre estos modelos.

Modelos estáticos y dinámicos. Los modelos estáticos son aquellos modelos que representan un grupo de objetos que integran un sistema junto con sus propiedades y sus conexiones. También se los conoce como modelos estructurales.
Los modelos dinámicos representan el comportamiento de los objetos del sistema, por lo que también son llamados modelos de comportamiento.
Para clarificar estos conceptos analicemos el siguiente ejemplo donde como modelo estructural contamos con un negocio de indumentaria que posee sucursales donde en cada sucursal se registran ventas. Por otro lado, los clientes compran en una sucursal y también pueden cambiar la mercadería en una sucursal distinta pero de la misma cadena de indumentaria, esta dinámica define el comportamiento del sistema.
Estos modelos se encuentran muy relacionados, uno representa la estructura y otro representa el comportamiento de un sistema, y en conjunto constituyen el modelo global del sistema.

2.4 - Metamodelos

2.4.1 - Qué es un metamodelo?

Tras definir los modelos, es importante señalar el hecho de que el lenguaje utilizado para describir a los mismos debe estar bien definido y ofrecer un nivel de abstracción adecuado para expresar y razonar sobre él. Si bien se contaba con mecanismos útiles y precisos para definir la sintaxis de los lenguajes, estos estaban pensados para lenguajes textuales y no tanto así para los gráficos (como lo son los lenguajes de modelado), surgiendo la necesidad de recurrir a un mecanismo distinto para definirlos. La idea compartida por todos los paradigmas encerrados dentro del MDD es la conveniencia de utilizar, para el modelado, lenguajes de mayor nivel de abstracción que los lenguajes de programación, que manejen conceptos más cercanos al dominio del problema. Estos se llaman lenguajes específicos de dominio (DSL, *domain-specific language*), los cuales requieren una descripción precisa, siendo lo más apropiado definirlos también como un modelo. Surge entonces una técnica específica para facilitar la definición de los lenguajes gráficos, llamada "*metamodelado*".
Un metamodelo es un modelo que especifica los conceptos de un lenguaje, las relaciones entre ellos y las reglas estructurales que restringen los posibles elementos de los modelos válidos, así como aquellas combinaciones entre elementos que respetan las reglas semánticas del dominio [3]. Un metamodelo es también un modelo, por lo que debe estar escrito en un lenguaje bien definido; este lenguaje se denomina metalenguaje (por ejemplo, BNF).
Por ejemplo, el metamodelo de UML es un modelo que posee los elementos para describir modelos UML, como Package, Classifier, Class, Operation, Association, etc. El mismo también define las relaciones entre estos conceptos, y las restricciones de integridad de los modelos UML (una de ellas, la que obliga a que las asociaciones sólo puedan conectar classifiers, y no paquetes u operaciones).

2.4.2 - Transformaciones

En la transformación de modelos se busca obtener un nuevo modelo a través de la transformación del modelo existente (sobre las transformaciones se profundizará más adelante). Entonces, los modelos son elementos de primer orden, a partir de esto aparece la noción de metamodelos y transformaciones.
Un modelo se define respetando la semántica de su metamodelo; un modelo se dice que conforma a su metamodelo. De la misma manera, un metamodelo debe conformar a su meta-metamodelo. En esta arquitectura de tres capas (modelos, metamodelos y meta-metamodelos), el meta-metamodelo usualmente conforma a su propia semántica; esto es, se define utilizando sus propios conceptos.
Considerar los modelos como entidades de primer orden demanda contar con un conjunto de herramientas para definir transformaciones sobre los mismos. Una transformación de modelos provee facilidades para generar un modelo Mb, conforme a un metamodelo MMb, desde un modelo Ma conforme a un metamodelo MMa.
En la figura 2-3, muestra la relación entre modelos y metamodelos.

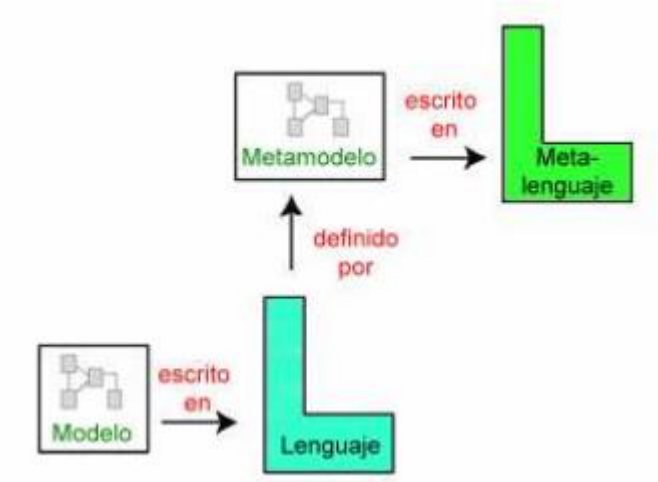


Figura 2-3. Relación entre modelos, metamodelos y meta-metamodelos

Un metamodelo define la sintaxis abstracta del lenguaje, esta sintaxis es lo que se utiliza como base en el procesamiento automatizado de los modelos. La sintaxis concreta es definida mediante otros mecanismos, los cuales no son relevantes para las herramientas de transformación de modelos. La sintaxis concreta es lo que se considera la interfaz del modelador e influye en la legibilidad de los modelos. Es por esto que el metamodelo y la sintaxis concreta de un lenguaje puede mantener una relación 1:n, es decir, la misma sintaxis abstracta que define el metamodelo puede ser vista a través de diferentes sintaxis concretas.
Analizaremos la relación entre los modelos y sus metamodelos a través de un ejemplo. En la figura 2-4 se muestra la estructura de dos metamodelos los cuales se expresan como diagramas de clase UML. La imagen (*a)* de la figura, muestra un metamodelo que puede usarse en modelos de clase. En este metamodelo está incluido el concepto abstracto de clasificadores, que involucra tipos de datos primitivos y clases. Dentro de los paquetes se definen las clases y dentro de las clases los atributos. Para todos los elementos del modelo es necesario especificar un nombre y las clases podrían ser persistentes. En la imagen (*b)*, de la figura, se presenta un posible metamodelo sencillo que permite definir esquemas de bases de datos relacionales. Entonces, este metamodelo define que, un esquema tiene tablas y dentro las tablas tienen columnas. Los tipos de dichas columnas se representan como cadenas. En cada tabla se define una clave primaria que está representada a través

de una columna, esta es pkey. Además, el concepto de claves foráneas es definido por FKey, ya que relaciona columnas de clave externa con tablas.

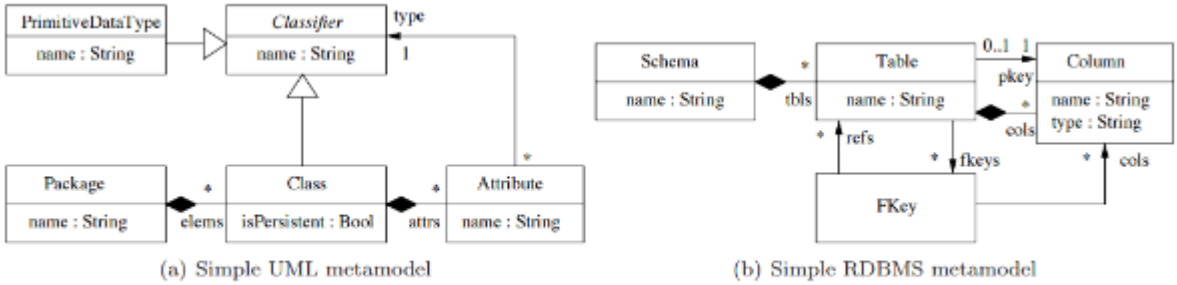


Figura 2-4. Metamodelos de muestra

En la figura 2-5 se muestra ejemplos de modelos que corresponden a la muestra de los metamodelos de la figura anterior. En la imagen (*a)* de la figura, tenemos un modelo de clase con un paquete App, este paquete contiene dos clases: Cliente y Dirección, el cliente es persistente, la dirección no. La imagen (*b)* de la figura, corresponde a una instancia del metamodelo del esquema. Esta instancia corresponde a un esquema que permite persistir objetos del cliente.

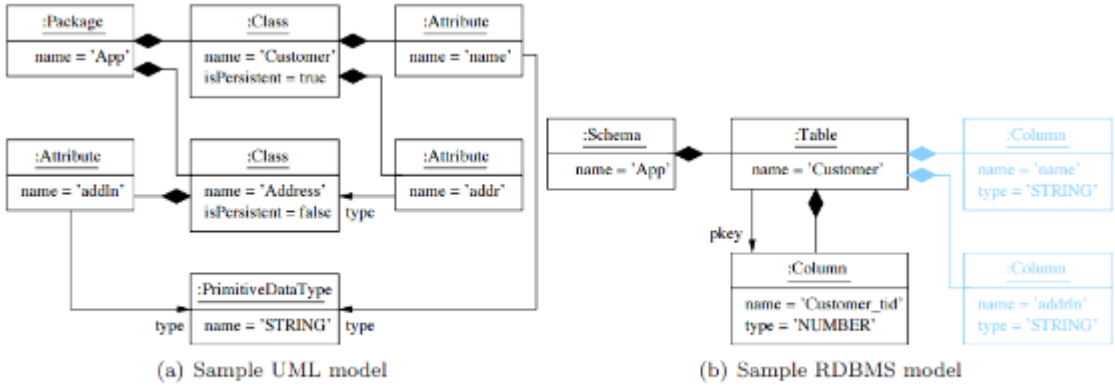


Figura 2-5. Modelos de muestra

2.4.3 - Arquitectura de 4 capas

La OMG propuso en el 2000 una arquitectura de cuatro capas de modelado, orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos.

Existen 4 niveles en esta arquitectura: M3, M2, M1 y M0 (del más abstracto al más concreto), como se observa en la figura 2-6.

En el modelo **M0** se encuentran las instancias "reales" del sistema, los objetos de la aplicación. Por encima de la capa M0 se sitúa la capa **M1**, que representa el modelo de un sistema de software. Sus conceptos representan categorías de las instancias de M0: cada elemento de M0 es una instancia de un elemento de M1.

Análogamente a como ocurre entre M0 y M1, los elementos de M1 son a su vez instancias del nivel **M2**, recibiendo la capa el nombre de Metamodelo. Por último, podemos ver los elementos de M2 como instancias de la capa **M3** o capa de meta-metamodelo, un modelo que define el lenguaje para representar un metamodelo. Este es el nivel más abstracto, que

permite definir metamodelos concretos. Dentro del OMG, MOF es su lenguaje estándar, siendo entonces que todos metamodelos de M2 son instancias de MOF.

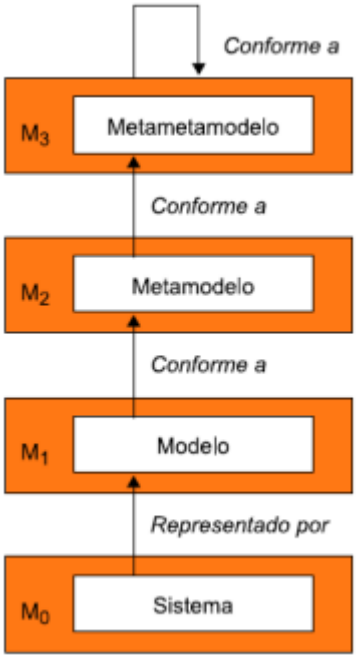


Figura 2-6. Arquitectura de 4 capas de modelado del OMG

2.4.4 - MOF

El lenguaje MOF es el lenguaje de la OMG para describir metamodelos, siendo UML un lenguaje de modelado conforme a él. Es un estándar para la ingeniería dirigida por modelos ubicada, como se dijo recientemente, en la capa superior de la arquitectura de 4 capas.
MOF provee un meta-metalenguaje que permite definir metamodelos en la capa M2. El ejemplo más conocido de un elemento en la capa M2 es el metamodelo UML, que describe al lenguaje UML.
Comprende una arquitectura de metamodelado cerrada y estricta: cerrada, porque el metamodelo de MOF se define en términos de sí mismo; y estricta, pues cada elemento de un modelo en cualquiera de las capas posee una correspondencia estricta con un elemento del modelo de la capa superior.
MOF puede ser usado para definir metamodelos de lenguajes orientados a objetos, como es el caso de UML, y también para otros lenguajes no orientados a objetos, como es el caso de las redes de Petri o los lenguajes para servicios web.

2.4.4.1 - ECORE

Ecore es un lenguaje común basado en EMOF que forma parte de la especificación MOF. Ecore es usado por EMF para la definición de metamodelos. EMF define los metamodelos y modelos con documentos XML.
Básicamente, es la forma en la que Eclipse [16] implementa a través de un plugin el metamodelo MOF.

2.4.4.2 - EMF

EMF es un framework de modelado para Eclipse y generador de código utilizado para el desarrollo de herramientas y aplicaciones. Partiendo de una especificación de un modelo descripto en XMI, EMF provee herramientas y soporte en tiempo de ejecución para producir

un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y edición basada en comandos del modelo y un editor básico.[2]
El EMF contiene una implementación de Service Data Objects (SDO) y XML Schema Infoset Model (XSD, un nuevo componente del proyecto Model Development Tools (MDT). XML Schema Definition (XSD) brinda un modelo y una API para manipular componentes de un esquema XML, con acceso a representación Document Object Model (DOM) del documento del esquema.
Los modelos pueden ser especificados usando notación Java, documentos XML, o herramientas de modelado como Rational Rose, y después ser exportados a EMF. Lo más interesante, es que EMF suministra las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.
EMF está compuesto por tres pilares fundamentales:

- **EMF-Core:** un meta modelo (*ECore*) donde se definen los modelos y los soportes en tiempo de ejecución para dichos modelos incluyendo notificaciones de cambio, una API para manipular objetos EMF genéricamente y soporte de persistencia con serialización XMI(*XML Metadata Interchange*) por defecto.
- **EMF-Edit:** compuesto por un conjunto de clases reutilizables genéricas que sirven para construir editores para modelos EMF.
- **EMF-Codegen:** El generador de código EMF es capaz de generar todo lo necesario para construir un editor completo para un modelo EMF. Esto incluye un GUI desde el cual pueden ser especificadas las opciones de generación, y los generadores a ser invocados. La generación se basa en Java Development Tooling (JDT), un componente de Eclipse. Hay tres tipos de generación de código que son soportadas:
  - *Modelo*: Proporciona clases Java de implementación e interfaz para todas las clases del modelo, además una clase de implementación de factory y package (metadata).
  - *Adaptadores*: Genera clases de implementación, que adaptan las clases del modelo para que se puedan editar y visualizar.
  - *Editor*: Produce un editor estructurado adecuadamente que se ajusta al estilo recomendado por los editores de modelos EMF Eclipse y sirve como punto de partida al comienzo de la personalización.

Capítulo 3: Lenguajes

3.1 - Definición

Un lenguaje es un sistema de comunicación compuesto por un conjunto de sonidos o símbolos escritos que permiten la abstracción y comunicación de conceptos. Un lenguaje consta de una estructura predefinida en la que se establecen las posibles combinaciones. Aunque en sentido estricto, el lenguaje sería la capacidad humana para comunicarse mediante lenguas, se suele usar para denotar los mecanismos de comunicación no humanos (el lenguaje de las abejas o el de los delfines), o los creados por los hombres con fines específicos (los lenguajes de programación, los lenguajes de la lógica, los lenguajes de la aritmética, entre otros).

3.2 - Clasificación

Los lenguajes se pueden clasificar en dos grandes ramas: los lenguajes naturales como por ejemplo el castellano, inglés, francés y los lenguajes formales como por ejemplo los lenguajes de programación.

3.2.1 - Lenguaje Natural

El lenguaje natural (o lenguaje humano), se basa en la capacidad de los seres humanos para comunicarse por medio de diferentes signos lingüísticos. Estos signos pueden ser caracterizados como secuencias sonoras, gestos y señas, signos gráficos.
Este lenguaje tiene un gran poder expresivo lo que permite analizar situaciones complejas y razonar en detalle sobre ellas. La riqueza del componente semántico y la relación con los contextos en los cuales son usados da a los lenguajes naturales su gran poder expresivo.
Se puede decir que las reglas y combinaciones del lenguaje natural fueron utilizadas desde un principio sin contar con una definición previa sino que fueron construidas una vez que los humanos ya eran capaces de comunicarse.

3.2.2 - Lenguaje Formal

En matemáticas, lógica, y ciencias de la computación, un lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados. Al conjunto de los símbolos primitivos se le llama el alfabeto o vocabulario del lenguaje, y al conjunto de las reglas se lo llama la gramática formal o sintaxis.
A diferencia de los lenguajes naturales un lenguaje formal es debidamente definido antes de su utilización.

3.2.3 - Diferencias entre Lenguaje Natural y Lenguaje Formal

Principalmente podemos decir que si bien ambos tipos de lenguajes definen el significado de una cadena por su forma, es decir la manera en que los componentes son combinados, los lenguajes naturales tienen componentes extras que entran en juego para determinar la validez de esta cadena; estos son el componente semántico y componente pragmático.
La Semántica es el estudio del significado de los signos lingüísticos, esto es, palabras, expresiones y oraciones.
La pragmática afirma que una gran parte de la interpretación de un enunciado es la información sobre el mundo y el contexto que los hablantes traen consigo; es decir, no toda la información necesaria se encuentra en las palabras mismas.

Los lenguajes naturales poseen una semántica ambigua debido a la flexibilidad de sus construcciones y a la ambigüedad referencial que se da cuando una palabra o frase puede referenciar a dos o más propiedades o cosas, teniendo en cuenta el contexto.

3.3 - Gramática

La gramática es el estudio de las reglas y principios que regulan el uso de las lenguas y la organización de las palabras dentro de una oración. También se denomina así al conjunto de reglas y principios que gobiernan el uso de un lenguaje determinado; de esta forma, cada lengua tiene su propia gramática. Entonces, podemos decir que la gramática de un lenguaje es el conjunto de reglas capaces de generar todas las posibles combinaciones de ese lenguaje, ya sea éste un lenguaje formal o un lenguaje natural.
Dado que una gramática es un conjunto de reglas que aplica un lenguaje para obtener las estructuras de palabras válidas, se producen cadenas o palabras que pertenecen a algún lenguaje, y la producción de estas palabras está determinada por reglas de sustitución bien definidas. A estas reglas también se les llama producciones.
Entonces, un lenguaje es descrito por una gramática, G, la cual puede definirse como una tupla de 4, G= (N, S, T, P), donde:

- **N**. Es un conjunto finito de símbolos *no terminales*, que funcionan como variables a través de las reglas de producción. Son sustituidas, en cada paso, por una secuencia de símbolos que pueden ser terminales, no terminales o una combinación de ambos.
- **S** $\in$ **N**. Es el símbolo *inicial* del cual derivan las demás producciones por medio de las reglas de producción.
- **T**. Es un conjunto finito denominado *alfabeto*, sobre el cual se generará el lenguaje. A los símbolos de este alfabeto se les llama *terminales*, ya que una vez que forman parte de la palabra producida, no pueden ser reemplazados por algún otro símbolo.
- **P**. Es un conjunto finito de reglas de *producción*, con el cual se generarán las posibles cadenas que integren el lenguaje.

Se conoce como fórmulas bien formadas, a las cadenas formadas a partir de las reglas de la gramática formal existente y al lenguaje formal como el conjunto de todas las fórmulas bien formadas. Vale aclarar que una gramática formal no especifica el significado de las fórmulas bien formadas, sino que solo es capaz de especificar su forma.

3.4 - Jerarquía de Chomsky

En 1956 y 1959, el lingüista norteamericano Noam Chomsky publicó dos trabajos sobre los lenguajes naturales que aplicados al área de los lenguajes formales produjeron lo que se conoce como *Jerarquía de Chomsky*. Esta jerarquía establece una clasificación de cuatro tipos de gramática formales que, a su vez, generan cuatro tipos diferentes de lenguajes formales. A continuación se describen brevemente los tipos o clases de gramática definidos por Chomsky.

Gramáticas de tipo 0 o sin restricciones. Este tipo incluye a todas las gramáticas formales y generan todos los lenguajes capaces de ser reconocidos por una máquina de Turing. Los lenguajes son conocidos como lenguajes recursivamente enumerables. Estos lenguajes son diferentes de los lenguajes recursivos, cuya decisión puede ser realizada por una máquina de Turing que se detenga.

Gramáticas de tipo 1 o gramáticas sensibles al contexto. Son las que generan los lenguajes sensibles al contexto. Estas gramáticas tienen reglas de la forma αAβ→αYβ con A un no terminal y α, β e Y cadenas de terminales y no terminales. Las cadenas α y β pueden ser vacías, pero Y no puede serlo. La regla S → ε, siendo ε una cadena vacía, está permitida si S no aparece en la parte derecha de ninguna regla. Los lenguajes descritos por estas gramáticas son exactamente todos aquellos lenguajes reconocidos por una máquina de Turing determinista cuya cinta de memoria está acotada por un cierto número entero de veces sobre la longitud de entrada, también conocidas como autómatas linealmente acotados.

Gramáticas de tipo 2 o gramáticas libres de contexto. Estas gramáticas generan los lenguajes independientemente del contexto. Las reglas son de la forma A → Y con A un no terminal e Y una cadena de terminales y no terminales. Estos lenguajes son aquellos que pueden ser reconocidos por un autómata con pila.

Gramáticas de tipo 3 o gramáticas regulares. Las cuales generan los lenguajes regulares. Estas gramáticas se restringen a aquellas reglas que tienen en la parte izquierda un no terminal, y en la parte derecha un solo terminal, posiblemente seguido de un no terminal. La regla S → ε también está permitida si S no aparece en la parte derecha de ninguna regla. Estos lenguajes son aquellos que pueden ser aceptados por un autómata finito. También esta familia de lenguajes puede ser obtenida por medio de expresiones regulares.

Chomsky demostró que se pueden construir modelos matemáticos donde las propiedades del mismo reflejan directamente el grado de complejidad estructural de los lenguajes que se ajustan a los modelos; por lo que si dos lenguajes pueden ser representados a través del mismo modelo matemático, ambos lenguajes tienen la misma complejidad estructural. Podemos caracterizar los modelos matemáticos a través de los cuales se pueden caracterizar los lenguajes, estos son: ***autómatas*** y ***gramáticas***. Si bien ambos parten de principios diferentes está demostrado que son equivalentes; por lo que sí podemos caracterizar un lenguaje mediante un autómata propio de una determinada complejidad, entonces existe un procedimiento automático con el cual se puede caracterizar el lenguaje mediante la gramática correspondiente, lo mismo ocurre en el sentido contrario.

3.5 - Representación de gramáticas libres de contexto

3.5.1 - BNF

La notación Backus-Naur (BNF, *Backus-Naur form*) es una forma matemática formal de describir un lenguaje. Fue desarrollada por John Backus, para describir la sintaxis del lenguaje de programación Algol 60.
Esta notación se utiliza para definir formalmente una gramática libre de contexto de un lenguaje de forma que no existan desacuerdos o ambigüedades con respecto a lo que está permitido y lo que no en dicho lenguaje. BNF es inequívoco, es decir que existe una gran cantidad de teoría matemática en torno a este tipo de gramáticas y se puede construir mecánicamente un parser para un lenguaje dada una gramática BNF definida para él. Usando esta notación un programador o compilador puede determinar si un programa es sintácticamente correcto, es decir si se ajusta a la gramática y las reglas de puntuación del lenguaje de programación.
Los meta símbolos de BNF son:

- ::= o : = que se consideran equivalentes y su significado es "se define como".
- | que significa "o", se utiliza para representar opciones.
- <> los símbolos de mayor y menor son utilizados para rodear los nombres de las reglas o símbolos no terminales.
- Los símbolos no definidos entre <> se consideran símbolos terminales.

BNF al definir una gramática sigue sus mismas reglas, comienza con un símbolo llamado símbolo de inicio y se definen reglas para determinar lo que se puede reemplazar con este símbolo. El lenguaje definido por la gramática BNF, es el conjunto de todas las cadenas que se pueden producir siguiendo estas reglas. Estas reglas son conocidas como reglas de producción e indican que el símbolo que se encuentra del lado izquierdo de := debe reemplazarse por una de las posibles alternativas que se encuentran del lado derecho. El lado derecho de la producción puede estar compuesto por una o más alternativas separadas por | y estas alternativas puede ser tanto un símbolo terminal o no terminal. Algunas variantes son:

```
<Simbolo> :=   <NoTerminal>
<Simbolo> :=   Terminal
<Simbolo> :=   <NoTerminal> | Terminal
```

Otras versiones de BNF definen qué se debe encerrar los terminales entre comillas para distinguirlos de los símbolos no terminales o requieren que explícitamente se indique donde se permite el espacio en blanco mediante el uso de un símbolo especial.

Ejemplo 1. BNF no recursivo, oración

```
<oración>      ::= <sujeto> <predicado>
<sujeto>       ::= Juan | Julia
<predicado>    ::= <verbo> <adverbio>
<verbo>        ::= maneja | corre
<adverbio>     ::= descuidadamente | rápido | frecuentemente
```

Ejemplo 2. BNF recursivo, números decimales

```
<expr>    := '-' <num>| <num>
<num>     := <digits> | <digits>'.' <digits>
<digits>  := <digit>  | <digit> <digits>
<digit>   := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

En el ejemplo 2 puede observarse una regla de producción de producción recursiva,

```
<digits> := <digit>| <digit> <digits>
```

donde el término izquierdo de la misma aparece también del lado derecho generando la recursividad.

3.5.2 - EBNF

EBNF (*Extended Backus–Naur form*) es una notación que sirve para describir formalmente la sintaxis de un lenguaje al igual que BNF ya que está notación es una extensión del mismo. EBNF no es más poderoso que BNF, ambos poseen el mismo poder expresivo. Es decir, una gramática representada en EBNF puede ser equivalentemente representada en BNF. Está extensión de BNF sólo simplifica la definición de las gramáticas.

3.5.2.1 - Reglas en EBNF

Las reglas en EBNF siguiendo la misma estructura que las reglas reglas BNF. Cada regla está compuesta por tres partes: un lado izquierdo (LHS, *Left Hand Side*), un lado derecho (RHS, *Right Hand Side*) y el carácter := que separa estos dos lados.
Existen diferentes versiones de EBNF donde los conceptos son los mismos lo que cambia son los meta símbolos para representarlos.
LHS es un símbolo o no terminal que comúnmente se encuentra representado entre <>.
El RHS proporciona una descripción para el símbolo definido del en el LHS y que puede contener: un terminal o una concatenación de terminales y no terminales o un conjunto de strings separados por el símbolo | que se utiliza para determinar alternativas. El RHS también puede contener los siguientes meta símbolos y operadores.

- [] significa "opcional", el elemento o secuencia de elementos entre corchetes puede ser incluido o no.
- {} representa una "repetición", el contenido entre llaves se puede repetir cero o más veces.
- () se utiliza para agrupar elementos.
- ? indica que el elemento es opcional, es decir puede aparecer cero o una vez.
- * el elemento se puede repetir 0 o más veces.
- + el elemento se puede repetir al menos una vez, 1 o más.

Los operadores ?, * y + pueden ser aplicados sobre grupos.

Ejemplo 1, números decimales
Basado en el ejemplo 2 de BNF recursivo presentado en la sección 3.5.1 con el objetivo de ilustrar cómo EBNF simplifica la definición.

```
<expr>  := '-'? <digit>+ ('.' <digit>+)?
<digit> := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Ejemplo 2, números enteros

```
Dígito ⇐ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Número entero ⇐ [+ | -] dígito {dígito}
```

Esto se lee como:

- Un *dígito* se define como uno de los diez caracteres alternativos de 0 a 9.
- Un *entero entero* se define como una secuencia de tres elementos: un signo opcional (el cual puede ser + o -), seguido de cualquier dígito, seguido por una repetición de cero o más dígitos, donde cada dígito es elegido independientemente de la lista de alternativas en la regla de dígitos.

El RHS de las reglas *número entero* y *dígito* combinan casi todas las formas de control en EBNF: secuencia, opción, selección, repetición. Si bien existen descripciones de EBNF más largas y complicadas, sus reglas siempre usan solo estas cuatro formas de control.

3.5.2.2 - Recursión en EBNF

La recursión es más poderosa que la repetición y se deben utilizar reglas de EBNF recursivas para especificar la estructura de ciertos símbolos complicados.

Una descripción EBNF recursiva contiene reglas que utilizan su nombre de una manera especial, a estas reglas se las llama directamente recursivas.
Una regla *directamente recursiva* es aquella que en su definición al lado derecho hace referencia a su propio nombre o lado izquierdo. Por ejemplo, r ⇐ Ar que describe una secuencias de caracteres As de manera recursiva.

3.5.2.3 - Validación de reglas EBNF

Para probar que una cadena es válida de acuerdo con una regla, se deben igualar todos sus caracteres con todos los elementos contenidos en la descripción de dicha regla. Si hay una coincidencia exacta, se clasifica a la cadena como válida. Hay diferentes métodos para probar la validez de una cadena con respecto a una gramática: Prueba verbal, Pruebas tabulares y árboles de derivación.

Prueba verbal
Se utiliza el idioma (en nuestro caso castellano) para describir el proceso de producción de la cadena mediante la enumeración de las reglas aplicadas y las opciones seleccionadas.
Supongamos que se quiere probar la validez de la cadena +142 es una cadena válida de la gramática definida en el Ejemplo 2 de números enteros presentado en la sección 3.5.2.1. Iniciamos la validación por el signo, que es el primero de los 3 ítems en la parte derecha de la regla *Número entero,* elegimos incluirlo ya que es opcional y seleccionamos el signo +. En este punto ya hemos validado el primer carácter de la cadena. Procedemos con el siguiente elemento de la cadena, el carácter 1 que puede ser rápidamente reconocido como un *dígito* ya que existe entre las alternativas posibles en el lado derecho de la regla *dígito.* Finalmente debemos repetir dígito cero o más veces; en este caso particular utilizamos 2 repeticiones: para la primera repetición elegimos la alternativa 4 y para la segunda la alternativa 2.
Como todos los caracteres de la cadena +142 han sido igualados con su correspondiente regla o elemento en la gramática definida, hemos probado que la cadena es válida.

Pruebas tabulares
Son pruebas más formales que las verbales utilizadas para demostrar la validez de una cadena. La primera fila de la tabla siempre es la regla contra la que estamos tratando de validar la cadena y la última fila es la cadena que estamos tratando de validar. Cada fila de la tabla deriva de la anterior de acuerdo con una de las siguientes reglas:

1. Reemplazar el nombre de una regla por su definición.
2. Elegir una alternativa.
3. Determinar si incluir o no una opción.
4. Determinar el número de repeticiones.

Siguiendo otra vez el ejemplo de los números enteros para validar la cadena +142 obtenemos la siguiente tabla:

Estado	Razón (nro regla aplicada)
Número entero	Regla definida
[+|-] dígito {dígito}	Reemplazamos Número entero por su definición (1)
[+] dígito {dígito }	Elegimos la alternativa + (2)
+ dígito {dígito }	Incluimos la opción (3)
+1 {dígito}	Reemplazamos el primer dígito por la alternativa 1 (1)(2)
+1 dígito dígito	Definimos numero repeticiones en 2 (4)
+14 dígito	Reemplazamos el primer dígito por la alternativa 4 (1)(2)
+142	Reemplazamos el primer dígito por la alternativa 4 (1)(2)

Figura 3-1. Ejemplo Pruebas Tabulares

Árboles de derivación

Un árbol de derivación es una forma de mostrar gráficamente cómo se puede derivar una cadena para un lenguaje dado a partir del símbolo inicial definido en su gramática.

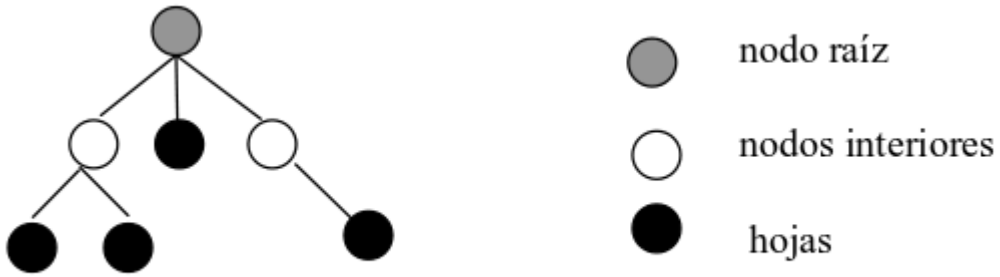


Figura 3-2. Estructura Árbol

Las propiedades de un árbol de derivación, dada una gramática libre de contexto representada como G=(N, S, T, P) son:

- Todos los nodos tienen una etiqueta.
- La etiqueta de la raíz es el símbolo inicial S.
- Las hojas contienen símbolos terminales.
- Los nodos interiores contienen símbolos no terminales.
- Dado un nodo n que tiene etiqueta A y los nodos $n_1,n_2,...,n_k$ sus hijos ordenados de izquierda a derecha, con etiquetas $a_1,a_2,..,a_k$ respectivamente, entonces: $A \rightarrow a_1,a_2,...,a_k$ debe ser una producción en P.

Siguiendo el ejemplo del Número entero, +142 obtenemos el siguiente árbol de derivación:

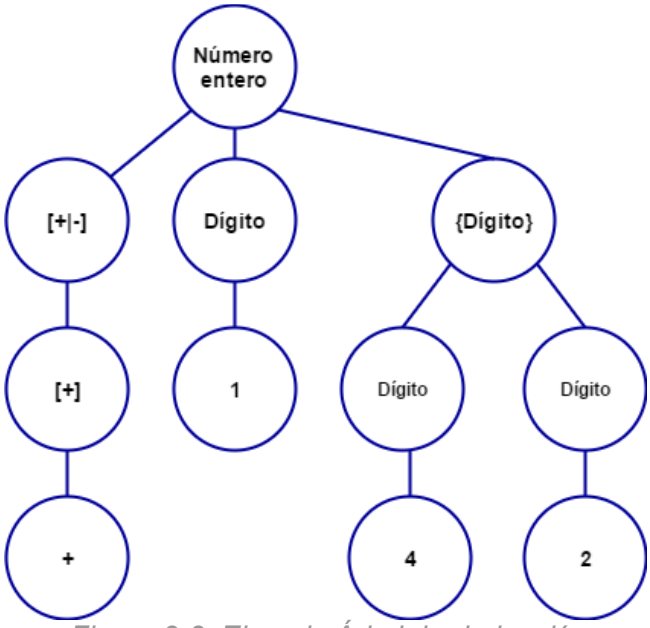


Figura 3-3. Ejemplo Árbol de derivación

3.6 - Lenguaje Natural

3.6.1 - Castellano

El lenguaje castellano se puede definir como el conjunto de todas las oraciones en castellano. La mayoría de los lenguajes son recursivos, a partir de una oración existen procedimientos que permiten formar otras mayores y más complejas; es por esto que es necesario encontrar propiedades o conjuntos de propiedades que permitan definirlos unívocamente. Esto es, dada la oración castellana "*el coche es gris*" es posible construir otras como:

- "*mi amigo dice que el coche es gris*".
- *"si mi amigo dice que el coche es gris, es que el coche es gris".*
- "*si me contaron que mi amigo dice: "el coche es gris", mi amigo dice que el coche es gris*".

Una oración en castellano es una secuencia finita de palabras del castellano donde sabemos que el conjunto de esas palabras es finito y aunque no todas las combinaciones de palabras son permitidas es necesario que las combinaciones permitidas sean correctas. Es decir, no solo se busca que respeten la sintaxis (reglas que determinan las cadenas válidas) sino que tengan sentido (semántica). Esa sintaxis y esa semántica constituyen un orden en la teoría del lenguaje castellano, aquel que permite la definición de todas las oraciones del lenguaje castellano.
Supongamos que el conjunto de palabras pertenecientes al diccionario del castellano contiene: {el, hombre, tomó, compró, balón}
Entonces, con este diccionario existirán frases creadas que serán correctas tanto sintácticamente como semánticamente, como:

- "*el hombre tomó el balón*".
- "*el hombre compró el balón*".

Otras frases sean correctas sintácticamente pero no semánticamente:

- "*el balón compró el hombre*".
- "*el balón tomó el hombre*".

O pueden ser invalidadas tanto semánticamente como sintácticamente:

- "*tomó compró balón el*".
- "*él tomó hombre balón el*".

Podemos decir que en un lenguaje natural, como el castellano, su formalización mediante una gramática o teoría es posterior a la formación de oraciones; es por esta razón, que se denomina a estos lenguajes naturales porque no son construidos o artificiales.
Por otro lado, en los lenguajes naturales las palabras en una oración tienen un significado y poseen su significante. Lo que esto significa es que independientemente del significado de cada palabra debemos tener en cuenta el sentido correcto que éstas adquieren según el contexto. Está variación en el significado dependiendo del contexto se conoce como polisemia. Uno de los objetivos de la informática es encontrar una forma de especificar rigurosamente estos significados por medio del uso de los métodos de interpretación de los sistemas formales.

Un ejemplo en el uso de la palabra *banco* donde dependiendo del contexto variará su significado:

- *"El banco de madera es blanco"*
- *"Tengo que ir al banco a depositar plata"*

El carácter polisémico de un lenguaje es lo que lleva a incrementar la riqueza de su componente semántico. Este hecho hace la formalización del lenguaje imposible. Se considera que la polisemia es una propiedad de los lenguajes naturales adquirida recientemente y que estos lenguajes habrían sido semejantes a los lenguajes formales, donde la polisemia es el resultado de un enriquecimiento progresivo.
Podemos resumir las propiedades de los lenguajes naturales en que:

- Fueron desarrollados por enriquecimiento progresivo antes de ser o intentar ser formalizados mediante una teoría.
- Su componente semántico es el que le da importancia a su carácter expresivo.
- Es muy difícil o imposible generar una formalización completa.

3.6.2 - Lenguaje natural limitado o reducido

Si limitamos el uso del lenguaje a un dominio particular y acotado, podríamos definir con mayor precisión la sintaxis y semántica del subconjunto del lenguaje natural que se aplica en dicho dominio.
En la presente tesis nos limitaremos al uso del lenguaje natural para expresar invariantes y expresiones que se han especificado en el lenguaje formal OCL.

Capítulo 4: Lenguaje Formal, OCL

El lenguaje formal OCL es un estándar conocido por la gran mayoría de los diseñadores que utilizan el lenguaje UML para modelar ya que está integrado a éste. OCL es un lenguaje orientado a objetos que permite especificar restricciones semánticas que no se pueden expresar a partir de una notación gráfica. En el presente trabajo son estas restricciones definidas en el lenguaje formal OCL las que se buscan transformar a lenguaje natural; es decir, a un lenguaje no técnico. De esta forma logramos que las restricciones puedan ser leídas e interpretadas por cualquier persona independientemente de sus conocimientos técnicos.

4.1 - Definición

Cuando se busca documentar y diseñar un sistema de software, una forma conocida y eficiente es a través del lenguaje unificado de modelado. Los llamados diagramas UML permiten crear, entre otras cosas, diagramas de clases; pero estos diagramas no permiten definir aspectos del sistema que son importantes de fijar en la especificación. Estos aspectos son por ejemplo, restricciones. Si bien las mismas podrían ser escritas en lenguaje natural, esto podría llevar a que no sean lo suficientemente claras, o que resulten en ambigüedades, lo que más tarde traerá problemas. Es por esto que se utilizan los llamados lenguajes formales; estos lenguajes son manejados por personas con un gran conocimiento matemático, lo que dificulta su entendimiento por parte de diseñadores, programadores y personas sin conocimientos técnicos. OCL se ha desarrollado para solucionar dicha deficiencia. El lenguaje de especificación de modelado OCL, es un lenguaje formal orientado a objetos. Es fácil de entender y de escribir para los diseñadores del sistema y permite introducir restricciones al modelo UML de forma de evitar ambigüedades.

4.2 - Descripción de OCL

OCL es un lenguaje de especificación de modelado, diseñado y mantenido por el Object Management Group (OMG) el cual crea y mantiene el estándar UML. OCL está diseñado como un complemento para UML.

Es un lenguaje formal que se utiliza para describir expresiones en modelos UML. Estas expresiones describen condiciones invariantes que tienen los sistemas modelados y/o consultas sobre los objetos allí especificados. No generan efectos colaterales al ser evaluadas; es decir, una evaluación no afecta el estado de un objeto. Sin embargo una expresión OCL puede ser utilizada para representar un cambio de estado (por ejemplo una post-condición).

Los diseñadores pueden realizar consultas escritas en un metalenguaje independizándose así de la plataforma de implementación del sistema.

OCL no es un lenguaje de programación por lo que no se pueden realizar lógica de programa o flujos de control, tampoco es posible invocar procesos y solo se pueden realizar operaciones de consulta. Todas las expresiones tienen un tipo, ya que es un lenguaje tipado. Además, por ser un lenguaje de especificación, no se pueden expresar sobre el mismo cuestiones relacionados con la implementación.

Entonces, ¿para qué puede ser utilizado OCL? Puede ser utilizado para los siguientes propósitos:

- para especificar los valores iniciales de las propiedades.
- para especificar invariantes en clases y tipos en un modelo de clases.
- para describir pre y post condiciones en Operaciones y Métodos.
- para especificar restricciones en operaciones.

- para describir Guardas.
- para especificar reglas de derivación para una propiedad.
- para especificar invariantes de tipos para Estereotipos.
- como lenguaje de consulta.

OCL es independiente del lenguaje de programación que se utilice. Las restricciones no pueden ejecutarse directamente y ser controladas en tiempo de ejecución, por lo que podría ocurrir que existan violaciones a las restricciones y las mismas no sean detectadas.

4.3 - Restricciones en OCL

OCL permite especificar diferentes tipos de restricciones sobre el modelo, de forma de obtener modelos más precisos y libres de ambigüedades. A continuación se analizan las principales construcciones sintácticas en OCL.
Se utilizará el siguiente diagrama de clases en la presentación de los ejemplos en OCL a lo largo de este capítulo.

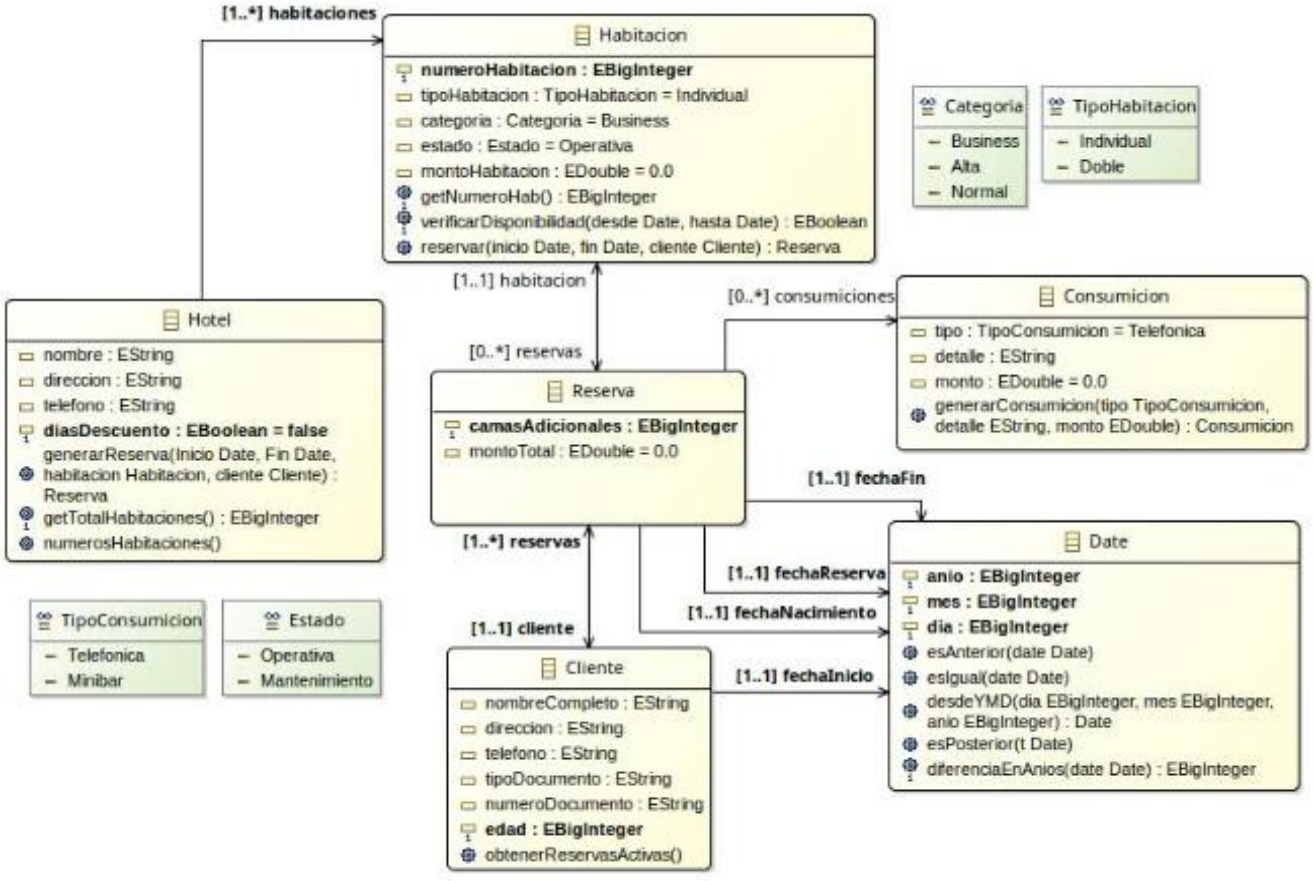


Figura 4-1. Ejemplo Reservas en Hotel

4.3.1 - Invariante

Una invariante es una expresión OCL que define una restricción sobre un elemento del modelo. Dicha restricción, debe cumplirse para todas las instancias del elemento. La sintaxis para su definición es:

```
context [nombreClase:]
     inv: <Expresión OCL>
```

La palabra clave *context*, define sobre el elemento que aplica la restricción, *inv* es la etiqueta que determina que se está definiendo una invariante.
Como ejemplo y basándonos en el modelo anterior, podríamos definir una restricción que determine que el nombre del hotel no puede ser vacío:

```
context Hotel
  inv: self.nombre<>''
```

En el ejemplo, Hotel corresponde al contexto sobre el que aplica la expresión; luego, para todas las instancias del tipo la restricción definida debe ser válida.
En la expresión OCL, la palabra reservada *self* hace referencia a la instancia contextual; en este caso, a una instancia de Hotel.

4.3.1 - Precondición

Una precondición, es una condición que debe ser válida antes de que se ejecute la operación. Es decir, se asegura que se cumple dicha condición antes de la ejecución de la operación.
A diferencia de lo que ocurre con las invariantes, las cuales deben cumplirse en todo momento, una precondición, solo debe ser válida antes de ejecutar la operación. La siguiente es la sintaxis para la definición de una operación con una pre condición:

```
context Typename::operationName(parameter1:Type1,...):ReturnType
  pre: <OclExpression>
```

Un ejemplo, podría ser que al momento de reservar una habitación se considera como precondición que la habitación está 'Operativa', es decir que es una habitación que se puede reservar (podría ocurrir que la habitación esté en mantenimiento) y que no existen reservas activas para la habitación.

```
context Habitacion::reservar(inicio:Date, fin:Date,cliente:Cliente):Reserva
    pre: self.estado='Operativa' and
        self.reservas->select(r|
                r.fechaInicio.esAnterior(inicio)
                        and r.fechaFin.esPosterior(inicio))->isEmpty()
```

4.3.2 - Postcondición

Una postcondición, es una condición que se espera se cumpla luego de ejecutar la operación.
Al igual que las precondiciones y a diferencia de lo que ocurre con las invariantes una postcondición solo debe ser válida después de ejecutar la operación. Su sintaxis es:

```
context Typename::operationName(parameter1:Type1,...):ReturnType
     post: <OclExpression>
```

Como ejemplo de postcondición, se espera que luego de generarse una reserva exista para la habitación involucrada, una reserva activa, en las fechas y para el cliente informados.

```
context Habitacion::reservar(inicio:Date, fin:Date,cliente:Cliente):Reserva
    post: self.reservas->select(r|r.fechaInicio.esIgual(inicio)
                                and r.fechaFin.esIgual(inicio)
                                    and r.cliente=cliente)->size()=1
```

Para las postcondiciones existe el prefijo *@pre* para las propiedades, con el cual se puede hacer referencia al valor de una propiedad de un objeto al comienzo de la ejecución de la operación.

4.3.3 - Body

Body o expresión de consulta, es una expresión OCL que se liga a una operación con el objetivo de realizar una consulta sobre propiedades y su ejecución que no cambia el estado del modelo. Al igual que ocurre con las pre y post condiciones, los parámetros de la operación pueden ser utilizados en la expresión. Además, la expresión debe ser acorde al tipo de la operación.

```
context Typename :: operationName(p:Type1,...):ReturnType
     body: -- alguna expresión
```

Siguiendo con el ejemplo utilizado a los largo de este capítulo, vamos a definir *getNumeroHab()* de la clase Habitación:

```
context Habitacion::getNumeroHab():Integer
    body: self.numeroHabitacion
```

Se pueden utilizar precondiciones, postcondiciones y expresiones de consultas, de forma combinada luego de que se ha especificado el contexto de la operación en cuestión. Por ejemplo:

```
context Hotel::getTotalHabitaciones():Integer
    pre:self.habitaciones<>null
    body:self.habitaciones->size()
```

4.3.4 - Definición

Una definición es una restricción que tiene como objetivo definir expresiones OCL reusables. La sintaxis genérica es:

```
context [VariableName:] TypeName
     def:[VariableName] | [OperationName(ParameterName1:Type,..)]:
          ReturnType = <OclExpression>
```

En el siguiente ejemplo la expresión OCL define una variable llamada get*HabitacionDisponible()*, que retorna si la habitación está disponible (con disponible nos referimos que no está en mantenimiento).

```
context Habitacion
    def: getHabitacionDisponible(): Boolean = self.estado='Operativa'
```

La variable *getHabitacionDisponible()* es conocida solo en el contexto donde es definida, Habitación, por lo que solo se puede hacer referencia a la misma dentro de ese contexto. El siguiente ejemplo muestra su uso:

```
context Habitacion::reservar(inicio:Date, fin:Date,cliente:Cliente):Reserva
    pre:getHabitacionDisponible() and
        self.reservas->select(r|
                r.fechaInicio.esAnterior(inicio)
                        and r.fechaFin.esPosterior(inicio))->isEmpty()
```

4.4 - Expresión de valor inicial

Una expresión de valor inicial es una expresión OCL a través de la cual se determina el valor inicial de una propiedad. Se debe conocer el tipo de la propiedad para la que se define el valor inicial y se debe considerar la multiplicidad, cuando hay una multiplicidad mayor que uno el tipo será un Set/Ordered Set del tipo de la propiedad conocido. La sintaxis:

```
context Typename :: propertyName: Type
     init: -- alguna expresión representando el valor inicial
```

Como ejemplo podemos definir, basándonos en el modelo anterior, que las habitaciones como valor inicial estarán todas disponibles:

```
context Habitacion:: estado: String
    init:'Operativa'
```

4.5 - Valores básicos y tipos

OCL cuenta con un conjunto de tipos básicos que son independientes del modelo y ya vienen predefinidos. Además, cada uno de estos tipos tiene un conjunto de operaciones básicas asociadas. Estos tipos predefinidos se muestran en la Figura 4-2.

Tipo	Valores	Coherente con las definiciones de implementación
OclInvalid	invalid	
OclVoid	null, invalid	
Boolean	true, false	(MOF) http://www.w3.org/TR/xmlschema-2/#boolean
Integer	1, -5, 2, 34, 26524,...	(MOF) http://www.w3.org/TR/xmlschema-2/#integer
Real	1.5, 3.14,...	http://www.w3.org/TR/xmlschema-2/#double
String	Es un string'	(MOF) http://www.w3.org/TR/xmlschema-2/#string
UnlimitedNatural	0, 1, 2, 42, ..., *	http://www.w3.org/TR/xmlschema-2/#nonNegativeInteger

Figura 4-2. OCL Tipos y valores

Algunos ejemplos de operaciones predefinidas para tipos predefinidos pueden verse en la Figura 4-3

Tipo	Operaciones
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()
Boolean	and, or, xor, not, implies, if-then-else
String	concat(), size(), substring()
UnlimitedNatural	*, +, /

Figura 4-3. Ejemplo Operación por tipos básicos

4.5.1 - Valores indefinidos

Los valores indefinidos en una expresión OCL pueden generarse cuando la expresión está siendo evaluada. Existen dos excepciones a esta regla para las cuales el valor será definido.

- Sí False AND independientemente de la próxima expresión, ya sea que es true o false, toda la expresión se evalúa a FALSE.
- Si True OR independientemente de la próxima expresión, ya sea que es true o false, toda la expresión será TRUE.

4.5.2 - Ajustes de tipos

OCL es un lenguaje tipado y, al igual que otros lenguajes tipados, posee un conjunto de tipos básicos. Dichos tipos están organizados en una jerarquía a través de la cual se puede determinar la conformidad entre tipos. De esto surge, que no se puede comparar un Booleano con un Integer o con un String, dado que sus tipos no son compatibles. En la Figura 4-4, se definen las reglas de ajuste.
Para que una expresión OCL sea válida, es necesario que todos los tipos se ajusten correctamente, en caso contrario la expresión se considera inválida. Entonces, dado un tipo A que se ajusta a un tipo B, cuando una instancia de tipo A puede ser sustituida por una instancia de tipo B, las reglas de ajuste entre tipos están dadas por:

- Cada tipo se ajusta a su supertipo.
- El ajuste de tipos es transitivo. Es decir, dado un tipo A que se ajusta a un tipo B y dado que el tipo B se ajusta al tipo C entonces el tipo A se ajusta al tipo C.

Tipo	Se ajusta a/Es subtipo de	Condición
Set(T1)	Collection(T2)	if T1 conforms to T2
Sequence(T1)	Collection(T2)	if T1 conforms to T2
Bag(T1)	Collection(T2)	if T1 conforms to T2
OrderedSet(T1)	Collection(T2)	if T1 conforms to T2
Integer	Real	
UnlimitedNatural	Integer	* es un entero inválido

Figura 4-4. Reglas ajustes de tipos

Para las colecciones, se mantiene la correspondencia de ajuste si los tipos de los elementos se ajustan entre sí. En la Figura 4-5, se muestran algunos ejemplos de expresiones validas e invalidas.

Expresión OCL	Válida	Explicación
1 + 2 * 34	SI	
1 + 'motocicleta'	NO	el tipo String no se ajusta al tipo Integer
23 * false	NO	el tipo Boolean no se ajusta al tipo Integer
12 + 13.5	SI	

Figura 4-5. Expresiones válidas e inválidas

4.5.3 - Uso de operadores infijos

En el lenguaje OCL está permitido el uso de operadores infijos. Los operadores que se utilizan como infijos son: "+", "-", "*", "/", "<", ">","<>", "<=", ">=". Entonces, si un tipo define uno de estos operadores correctamente, pueden usarse como operador infijo.
La expresión: x + y es conceptualmente igual a la expresión: x.+(y). Esto ocurre, invocando a la operación "+" con y como parámetro. Los operadores infijos definidos para un tipo deben ser exactamente uno por parámetro. El tipo de retorno debe ser booleano para los operadores infijos: "<",">", "<>", "<=", ">=", "and", "or" y "xor".

4.5.4 - Expresiones Let

La expresión let permite definir un atributo derivado o una operación para ser usada luego en otras expresiones OCL.
Por ejemplo, pensando en el modelo de hoteles anterior, se define un atributo indicando si un Cliente tiene reservas a su nombre, entonces debe tener al menos 18 años:

```
context Cliente inv:
    let tieneReserva : Boolean =
            self.reservas-> notEmpty()
                        in tieneReserva implies edad>=18
```

4.6 - Colecciones

EL lenguaje OCL también maneja colecciones. Las mismas son identificadas por un tipo llamado **Collection**. *Collection* es un tipo abstracto, el cual posee un conjunto de operaciones asociadas. Este tipo tiene asociado un conjunto de subtipos concretos, estos son:

- *Set:* este tipo es un conjunto matemático que no contiene repetidos. Por ejemplo `Set {1, 8, 4, 9}`.
- *Bag:* este tipo es un conjunto con la diferencia que puede contener elementos repetidos, una o más veces. `Bag {1, 3, 4, 3, 5 }`.
- *Sequence:* es un Bag pero donde sus elementos están ordenados. Sequence `{1, 3, 3, 45}`.

En las colecciones se pueden definir secuencias de valores enteros consecutivos, utilizando dos puntos seguidos. Por ejemplo:

`Sequence{ 1..(6 + 4) }` y `Sequence{ 1..10 }`
son equivalentes a
`Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }`

4.6.1 - Operaciones en colecciones

En el lenguaje OCL existe un conjunto de operaciones definidas sobre los tipos colecciones vistos anteriormente.

Select y Reject Con la operación select se obtiene un subconjunto de una colección y su sintaxis es:

```
collection->select(c:Tipo | expresion-logica-con-c)
```

Donde collection es la colección sobre la que se aplica el select, c es un iterador con el que se va pasar por cada uno de los elementos de la colección y se evaluara cada elemento de la colección con expresion-logica-con-c. Esto dará como resultado todos los elementos de la colección que cumplen la expresión evaluada. De esta forma se obtiene como resultado una colección que puede ser: vacía, un subconjunto de la colección original evaluada (cuando algunos elementos cumplen con la condición evaluada), o todos los elementos de la colección evaluada (en el caso de que todos los elementos cumplan la condición evaluada).
El tipo de la variable iterador es opcional, por lo que la siguiente sintaxis también es válida:

```
collection->select( c | expresion-logica-con-c)
```

Se lo puede abreviar aún más:

```
collection->select(expresion-logica)
```

Las tres formas de escribir un select se presentan en el siguiente ejemplo donde se obtienen todas las habitaciones cuyo tipo es Individual.

```
context Hotel
    inv:self.habitaciones->
        select(h: Habitacion | h.tipoHabitacion='Individual')->size()>0
    inv:self.habitaciones->
        select(h | h.tipoHabitacion='Individual')->size()>0
    inv:self.habitaciones->
        select(tipoHabitacion='Individual')->size()>0
```

La operación *reject* es equivalente a una operación *select* con su condición negada, es decir se recuperan todos los elementos de la colección que no cumplen la condición. Entonces, la expresión puede escribirse como:

```
collection-> reject (a:Tipo | expresion-logica-con-a)
collection-> select(a:Tipo | not expresion-logica-con-c)
```

Collect La operación *collect* se utiliza cuando queremos especificar una colección que deriva de otra colección pero que contiene objetos diferentes a la colección original. Por ejemplo, obtener una colección que solo contendrá los números de las habitaciones del hotel.

```
context Hotel:: numerosHabitaciones():Collection
    body:self.habitaciones->collect(h|h.numeroHabitacion)->asSet()

context Hotel:: numerosHabitaciones():Collection
    body:self.habitaciones->collect(numeroHabitacion)->asSet()

context Hotel:: numerosHabitaciones():Collection
    body:self.habitaciones.numeroHabitacion
```

Como resultado se obtiene un Bag o un Set, dependiendo de los elementos de la colección original y la cantidad de elementos resultantes será igual al tamaño que la colección original.

ForAll - Exists La operación *ForAll* especifica una expresión booleana que debe valer para todos los elementos de una colección. Su sintaxis es:

```
collection->forAll (c: Tipo | expresion-logica-con-c)
```

En este caso, el resultado será verdadero si la condición booleana es verdadera sobre todos los elementos de la colección. Si la expresión no es válida al menos para uno de los elementos de la colección entonces el resultado será falso.
Por ejemplo, supongamos que queremos verificar que todas las habitaciones del hotel tienen asignado un tipo de habitación. Esto es:

```
context Hotel
    inv:self.habitaciones->
        forAll(h:Habitacion | h.tipoHabitacion<>null)
```

Esta invariante es verdadera si todas las habitaciones tienen tipo.
Es posible definir más de un iterador, esto es un forAll entre el producto cartesiano de la colección y ella misma. Por ejemplo:

```
context Hotel
    inv:self.habitaciones->
        forAll(h1,h2 | h1<>h2 implies
                h1.numeroHabitacion<>h2.numeroHabitacion)
```

La operación *exists* permite verificar una condición booleana sobre una colección donde dicha condición debe ser válida al menos para un elemento de la colección para que esta operación retorne un valor verdadero.

```
collection->exists (e:Tipo|expresion-logica-con-e)
```

Por ejemplo, verificamos que existe al menos una habitación con categoría Business.

```
context Hotel
    inv:self.habitaciones->
        exists(h:Habitacion | h.categoria='Business')
```

Iterate es una operación más compleja que las anteriores ya que es muy genérica. Las operaciones vistas anteriormente se pueden describir en términos de un iterate.

En un iterate existe un acumulador que va construyendo un valor iterando sobre una colección. La sintaxis:

```
collection->iterate(elem:Type;acc:Type=<expression>
                                    |expression-with-elem-and-acc)
```

La variable elem es el iterador. La variable acc es el acumulador. El acumulador toma un valor inicial <expresión>.
Cuando iterate es evaluado, elem itera sobre la colección evaluando la expresión sobre cada elemento de la colección. Luego de evaluar el elemento de la colección contra la expresión definida, el valor se asigna a acc. Es por esto, que el valor de acc se construye durante la iteración de la colección.
Iterate() es una operación de bajo nivel muy poco declarativa cuyo uso es desaconsejado en las especificaciones OCL. Puede ser reemplazada por un uso adecuado del *select()*, *collect()*, r*eject()* y por ese motivo no la incluiremos en nuestra traducción a lenguaje natural.

Iteradores en operaciones de colecciones estas operaciones toman un OCLExpression como parámetro y pueden tener una declaración de iterador (opcional). Para cada operación (op) la sintaxis es:

```
collection->op(iter : Type | OclExpression)
collection->op(iter | OclExpression)
collection->op(OclExpression)
```

4.7 - Tipos predefinidos en OCL

Como ya se presentó en una sección previa el conjunto de tipos básicos son *Integer*, *Real*, *String* y *Boolean*. Estos tipos básicos se complementan con *OClExpression*, *OclType* y *OclAny*.

OclType, los tipos definidos en un modelo tienen un tipo definido. Este tipo es una instancia del tipo OCL llamado OclType.
OclAny, este tipo es el supertipo de los tipos básicos predefinidos en OCL y de los tipos definidos en el modelo. Los tipos Collection predefinidos en OCL son subtipos de OclAny. Las propiedades de OclAny se encuentran presentes en todos los objetos, en todas las expresiones OCL.
Todas las propiedades heredadas de OclAny empiezan con el prefijo ocl. Se puede utilizar la operación oclAsType() para referir a las propiedades OclAny.
OclState, este tipo se utiliza como parámetro para la operación oclInState. No existen propiedades definidas para OclState. Se especifica usando el nombre del estado como ocurre en una máquina de estados.
OclExpression, representa el tipo de las expresiones en OCL. Cada expresión OCL es un objeto en el contexto OCL. Este tipo se utiliza para definir la semántica de las propiedades toman una expresión como un de sus parámetros: *select*, *collect*, *forAll*, etc. Estas expresiones incluyen un iterador y un acumulador; el iterador es opcional.
Real, este tipo representa el concepto matemático real.
Integer, el tipo Integer de OCL representa el concepto matemático integer.
String, el tipo String de OCL representa las cadenas ASCII.
Boolean, este tipo representa los valores true/false.
Enumeration, representa las enumeraciones definidas en el modelo.

4.8 - Propiedades y Objetos

Una expresión OCL puede referirse a Classifiers (interfaces, clases, tipos, asociaciones que actúan como tipos y tipos de datos). Todos los atributos, asociaciones, operaciones y métodos que no produzcan efectos laterales que están definidas en estos tipos pueden utilizarse.
En un modelo de clases, un método u operación se definen como libres de efectos laterales si el atributo isQuery de la operación o método es verdadero.
Una propiedad puede ser:

- Un atributo.
- Una asociación.
- Una operación con el atributo isQuery verdadero.
- Un método con isQuery verdadero.

4.8.1 - Propiedades

Una propiedad de un objeto que se define en un diagrama de clases se especifica por un punto seguido por el nombre de la propiedad.

```
context Tipo inv:self.property
```

Si *self* es una referencia a un objeto entonces *self.property* es el valor de la propiedad property en el objeto self.

4.8.2 - Atributos

Un atributo es una especificación a través de la cual se determina la propiedad de un objeto.

```
context Habitacion
    inv:self.numeroHabitacion>0
```

En este ejemplo el valor de la subexpresión *self.numeroHabitacion* es el valor del atributo *numeroHabitacion* de una instancia particular de la clase *Habitación*, la cual se identifica a través de self. El tipo de la subexpresión es el tipo que corresponde al atributo en cuestión que en este caso es Integer.
Utilizando los atributos y las operaciones definidas en los tipos básicos se pueden expresar cálculos sobre el modelo de clases. En el ejemplo anterior el requerimiento seria "*el número de Habitación debe ser siempre mayor a cero*".

4.8.3 - Operaciones

Las operaciones en OCL pueden tener parámetros o no. En el ejemplo presentado a continuación, la operación *reservaEnCurso* recibe como parámetros de entrada la fecha actual y un cliente. El resultado de la operacion sera un valor del tipo definido como tipo de retorno, en este caso será un objeto de tipo *Boolean*.
La operación podría definirse como una postcondición donde el objeto retornado puede ser referenciado usando la palabra clave *Result.* A continuación convertimos el body de la operación *reservaEnCurso* en una postcondición:

```
context Habitacion::reservaEnCurso(fechaActual:Date, cliente:Cliente):Boolean
    post:result = self.reservas->exists(r|r.fechaInicio.esAnterior(fechaActual)
                  and r.fechaFin.esPosterior(fechaActual) and r.cliente=cliente)
```

Una operación que no recibe parámetros, debe contener los paréntesis vacíos.

4.8.4 - Asociaciones y navegación

Las asociaciones establecen relaciones entre clases, son generalmente bidireccionales (pueden ser unidireccional), poseen un rol a cada extremo que sirve para identificar la direccionalidad y tiene multiplicidad a ambos lados que indica la cantidad de objetos con los que se relaciona en el otro extremo.
OCL permite navegar una asociación del diagrama de clases para referirse a otros objetos y sus propiedades a partir de un objeto específico mediante el uso de estas asociaciones. Para esto se utilizan los nombres de los roles:

```
object.rolename
```

El valor de la expresión estará definido por el conjunto de objetos que se encuentran del otro lado de la asociación. Es decir, si la multiplicidad de la asociación tiene un máximo de uno entonces el valor de la expresión es un objeto.

```
context Habitacion
    inv: self.categoria<>''

context Hotel
    inv: self.habitaciones->notEmpty()
```

En el primer caso donde el contexto es Habitación, *self.categoria* es de tipo Categoría ya que la multiplicidad es de 1 y en el ejemplo del contexto Hotel es un set ya que el hotel tiene más de una habitación (es una colección de habitaciones).
Las colecciones son tipos predefinidos en OCL que tienen un conjunto de operaciones predefinidas y se acceden usando '->' seguido del nombre de la propiedad.

4.8.5 - Características de clases

Todas las propiedades vistas son propiedades de instancias de clases. Los tipos son predefinidos en OCL o son definidos en el modelo de clases. El método *allInstances* es la forma de recuperar todas las instancias de un tipo determinado y se obtiene un Set con todas las instancias del tipo en un tiempo específico cuando la expresión es evaluada.
Supongamos que se quiere verificar que todos los clientes tienen número de documento diferente; es decir, que no se puede registrar dos veces el mismo cliente de forma de evitar duplicidad:

```
context Hotel
    inv: Cliente.allInstances()->
        forAll(c1, c2| c1<>c2
            implies c1.numeroDocumento<>c2.numeroDocumento)
```

En este ejemplo *Cliente.allInstances* devuelve el conjunto de todos los Clientes y es de tipo Set (Cliente)

4.8.6 - Propiedades predefinidas en todos los objetos

Existen propiedades que aplican a todos los objetos y estas son:

```
oclIsTypeOf(t:OclType):Boolean
oclIsKindOf(t:OclType):Boolean
oclInState(s:OclState):Boolean
oclIsNew():Boolean
oclAsType(t:OclType):instance of OclType : instance of OclType
```

La operación *oclTypeOf* es verdadera si el tipo de self y t son el mismo. Por ejemplo:

```
context Habitacion
    inv: self.oclIsTypeOf(Habitacion) --is true
    inv: self.oclIsTypeOf(Hotel) --is false
```

La propiedad *oclIsTypeOf* trata con el tipo directo del objeto, en cambio la propiedad *oclIsKindOf* determina si t es tipo directo o uno de los supertipos del objeto.
La operación *oclInState(s)* es verdadera si el objeto está en el estado s. Los posibles valores de s son los nombres de los estados en la máquina de estado. La operación *oclIsNew* será verdadera si, usada es una postcondición, el objeto es creado durante la operación; es decir, no existía al momento de evaluar la precondición. Por último, *oclAsType*(t) convierte el objeto origen en un objeto de tipo t que debe ser de un subtipo o supertipo del tipo origen.

4.9 - Package Context

Se puede especificar explícitamente el paquete al que pertenece una invariante, pre o post condición. Esto se realiza agrupando las restricciones entre las palabras claves *package* y *endpackage*:

```
package Package::SubPackage
     context X inv:
          ... definimos invariantes ...
     context X::operationName(..)
          pre: ... definimos precondición ...
endpackage
```

Dentro de un archivo OCL pueden existir varias declaraciones package permitiendo a todas las restricciones ser escritas y almacenadas en un solo archivo. Hay que tener en cuenta que este archivo debe coexistir con el modelo como una entidad separada.

Capítulo 5: Transformación de modelos

5.1 - Transformaciones de modelos

El proceso de convertir un modelo en otro modelo es lo que se denomina transformación entre modelos. Para esto es necesario definir un conjunto de reglas de transformación para especificar cómo transformar de un modelo origen en un modelo destino. También, es necesario contar con información de los metamodelos que describen la representación de los elementos y restricciones de los modelos a transformar.
Para definir una transformación entre modelos es necesario:

- Definir el tipo de transformación y el lenguaje de definición de transformaciones a utilizar.
- Seleccionar la herramienta que nos permita implementar los modelos y las transformaciones de forma automática.

En la figura 5-1 se pueden observar los participantes involucrados en una transformación de modelos. Se muestra un escenario de transformación con un modelo de entrada y uno de salida, los cuales deben respetar sus respectivos metamodelos. Como se vio anteriormente, un metamodelo define una sintaxis abstracta y los modelos que la respetan se dice que son conformes a ella. Cuando se define una transformación entre modelos se hace respecto a sus metamodelos. Posteriormente se utilizará un motor de transformación para ejecutarla sobre modelos concretos. En general, una transformación puede tener varios modelos origen y destino (source model y target model, en inglés, respectivamente), siendo posible además que el metamodelo origen y destino sean el mismo. Un ejemplo de transformación donde coinciden el modelo origen y destino la encontramos, por ejemplo, cuando se proporcionan transformaciones que describen casos de estudio de un sistema.

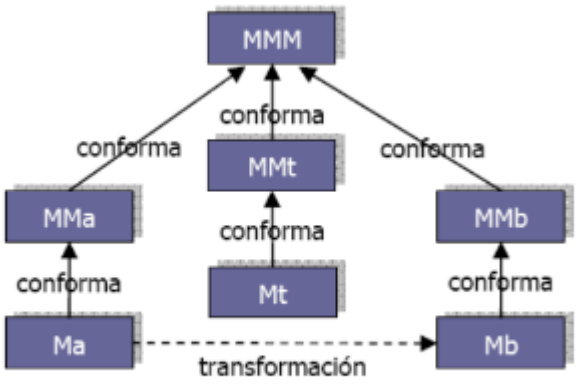


Figura 5-1. Transformación entre modelos

Un modelo *Ma* conforme a un metamodelo *MMa* es transformado al modelo *Mb* conforme a un metamodelo *MMb*. La transformación es definida por el modelo de transformación de modelos *Mt*, el cual a su vez es conforme al metamodelo de transformación *MMt*. Éste último metamodelo, *MMt*, junto con los metamodelos *MMa* y *MMb* deben conformar a un meta-metamodelo *MMM*.

5.2 - Tipos de transformaciones

En la Ingeniería de Software dirigida por Modelos (MDE, *Model-Driven Engineering*) se describen diferentes tipos de transformaciones entre modelos dependiendo de varios criterios. Estos criterios se analizan a continuación.

5.2.1 - Nivel de abstracción de los modelos de entrada y salida

***Transformaciones verticales*.** Relacionan modelos del sistema que se encuentran en distintos niveles de abstracción y pueden aplicarse tanto en sentido descendente como en sentido ascendente, siendo estos últimos el resultado de aplicar un proceso de ingeniería inversa.

Transformaciones horizontales. Relacionan los modelos que describen un sistema desde un nivel de abstracción similar. Se utilizan también para mantener la consistencia entre distintos modelos de un sistema; es decir, garantizan que la información modelada sobre una entidad en un modelo es consistente con lo que se dice sobre dicha entidad en cualquier otra especificación situada al mismo nivel de abstracción.

5.2.2 - Tipo de lenguaje que se utiliza para especificar las reglas

Lenguajes declarativos. Son un paradigma de programación basado más en las matemáticas y en la lógica. Estos lenguajes son más cercanos al razonamiento humano ya que no definen cómo se debe hacer (no se provee la solución) sino que definen qué se debe hacer (se describe el problema).

Lenguajes imperativos o híbridos. La programación imperativa es un paradigma de programación de los más generales. Permite describir la programación en términos del estado del programa y las sentencias que afectan sobre el estado. Son programas compuestos por un conjunto de instrucciones que indican a la computadora cómo realizar la tarea. La programación híbrida es un paradigma de programación que permite traducir programas escritos en un lenguaje de alto nivel a un lenguaje intermedio diseñado para facilitar la interpretación. En este caso, ambos tipos pueden combinarse.

5.2.3 - Direccionalidad en las transformaciones

Transformaciones unidireccionales*.* Las reglas se ejecutan en una sola dirección.

Transformaciones bidireccionales*.* En este caso, las transformaciones se pueden aplicar en ambas direcciones.

5.2.4 - Dependiendo de los modelos origen y destino

Transformaciones exógenas. Diferentes metamodelos son utilizados como origen y destino. Este es el escenario más habitual en el que se cuenta con un modelo de entrada y mediante la aplicación de reglas de transformación se obtiene el modelo de salida.

Transformaciones endógenas. El mismo metamodelo es utilizado como origen y destino; a estas transformaciones se las denomina transformaciones *in-place*. En este tipo de transformaciones las reglas se aplican sobre el modelo de entrada hasta que no es posible aplicar más reglas y el modelo se convierte en el modelo de salida.

5.2.5 - Tipo de modelo destino

Transformaciones modelo a modelo (M2M). Generan modelos de salida a partir de otros modelos. Estas transformaciones pueden ser clasificadas en subgrupos, como se verá en la sección 5.3

Transformaciones modelo a texto (M2T). Son las que generan cadenas de texto a partir de modelos. Si se desea generar código o documentos este es el tipo de transformación que se debe utilizar. Se pueden dividir en diferentes subgrupos que serán presentados en la sección 5.4.

5.3 - M2M

Las transformaciones modelo a modelo convierten un modelo de entrada (o múltiples) en un modelo de salida (o múltiples). Los modelos tanto de entrada como de salidas pueden ser instancias del mismo o de diferente metamodelo.

Este tipo de transformaciones son necesarias en ciertos casos donde es necesario contar con un modelo intermedio para disminuir la complejidad de una transformación directa entre PIMs y PSMs cuando los niveles de abstracción son muy distintos. Por ejemplo, cuando se intenta transformar desde un diagrama de clases a una implementación de EJB (*Enterprise Java Beans*) algunas herramientas, como OptimalJ, generan un modelo de componentes EJB intermedio que contiene toda la información necesaria para producir el código Java.

5.3.1 - Manipulación Directa

Este enfoque ofrece una representación del modelo más una API para su manipulación. La implementación es comúnmente llevada a cabo como un framework orientado a objetos que provee una pequeña infraestructura para organizar las transformaciones. Sin embargo, son los usuarios los encargados de implementar las reglas de transformación, trazas y otras características desde cero en algún lenguaje de programación como por ejemplo, Java. Un ejemplo de esto es Jamba [18].

5.3.2 - Relacional

Este enfoque agrupa lenguajes declarativos donde el concepto principal son las relaciones matemáticas. Se establecen relaciones entre elementos del modelo origen y elementos del modelo destino mediante la utilización de restricciones. En su formato original las especificaciones de restricciones no son ejecutables pero se les puede asignar semántica ejecutable mediante el uso de programación lógica usando matching basado en unificación, búsquedas, backtracking, etcétera.

Todos los enfoques relacionales y libres de efectos secundarios, usualmente soportan backtracking y crean los elementos del modelo destino implícitamente. Las especificaciones relacionales pueden ser interpretadas bidireccionalmente. En algunos casos las transformaciones in-place no están permitidas. Un ejemplo de esto es **QVT** (Query/ View/ Transformation) que es el lenguaje estándar para describir transformaciones de modelos definido por el OMG. La especificación del lenguaje QVT tiene una naturaleza híbrida, declarativa/imperativa, con la parte declarativa dividida en una arquitectura de dos niveles.

5.3.3 - Grafos

Este enfoque se basa en el uso de lenguajes que permitan operar sobre grafos que son tipados, que contienen atributos y etiquetas. Este tipo de transformación puede verse como una representación en forma de grafo de modelos de clases simplificados.

Las reglas de transformación de grafos son unidireccionales e in-place; están formadas por dos patrones, el derecho y el izquierdo, y la aplicación de la regla consiste en que se localice el patrón izquierdo en el modelo y este sea reemplazado por el patrón derecho que corresponda.
Algunos ejemplos de sistemas que implementan la propuesta teórica para grafos con atributos y sus transformaciones son AGG, AToM3, UMLX, MOLA. AGG y AToM3.

5.3.4 - Enfoques basados en la estructura

Para las transformaciones basadas en el modelo podemos diferenciar dos fases; en la primera fase se crea la estructura jerárquica del modelo y en la segunda fase se definen los atributos y las referencias en el modelo.
El framework se encarga de determinar el scheduling y la estrategia de aplicación por lo que los usuarios sólo deben encargarse de definir las reglas de transformación.
OptimalJ, es un ejemplo de framework de transformación modelo a modelo basado en este enfoque y está implementado en Java.

5.3.5 - Híbridos

Los enfoques híbridos utilizan las diferentes técnicas presentadas anteriormente de forma combinada y está combinación puede ser a nivel de reglas individuales, como un modo granularmente más fino, o como componentes separados.
QVT es una propuesta de naturaleza híbrida, con tres componentes separadas, llamadas Relations, Operational mappings, y Core; ejemplos de combinación a nivel fino son ATL y YATL [1].

5.3.6 - Herramientas

5.3.6.1 - ATL

ATLAS Transformation Language (ATL), es un lenguaje para la transformación de modelos desarrollado como parte del framework ATLAS Model Management Architecture (AMMA). Este lenguaje viene acompañado por un conjunto de herramientas desarrolladas sobre la plataforma Eclipse conocido como ATL *Development Tools* (ADT). ADT está compuesto por el motor de transformación ATL y el Ambiente de Desarrollo Integrado (IDE, *Integrated Development Environment*) ATL que provee un editor, un builder y un debugger [5].
ATL es un lenguaje híbrido ya que el mismo permite realizar tanto construcciones imperativas como declarativas. Entonces los mapeos entre objetos de un modelo origen y un modelo destino se realizan de forma declarativa pero de ser necesario, y debido a la complejidad de la transformación, se pueden utilizar declaraciones imperativas. Este lenguaje solo soporta transformaciones unidireccionales operando el modelo origen como de solo lectura y generando un modelo destino de solo escritura.

Arquitectura
De manera abstracta podemos definirla como que se encuentra compuesta por 4 componentes: Un núcleo (Core en inglés), un compilador y un parser, máquinas virtuales (Virtual Machines o VMs), y un IDE.
En la siguiente figura, se pueden observar los roles que estos componentes de ATL desempeñan durante la ejecución de una transformación.

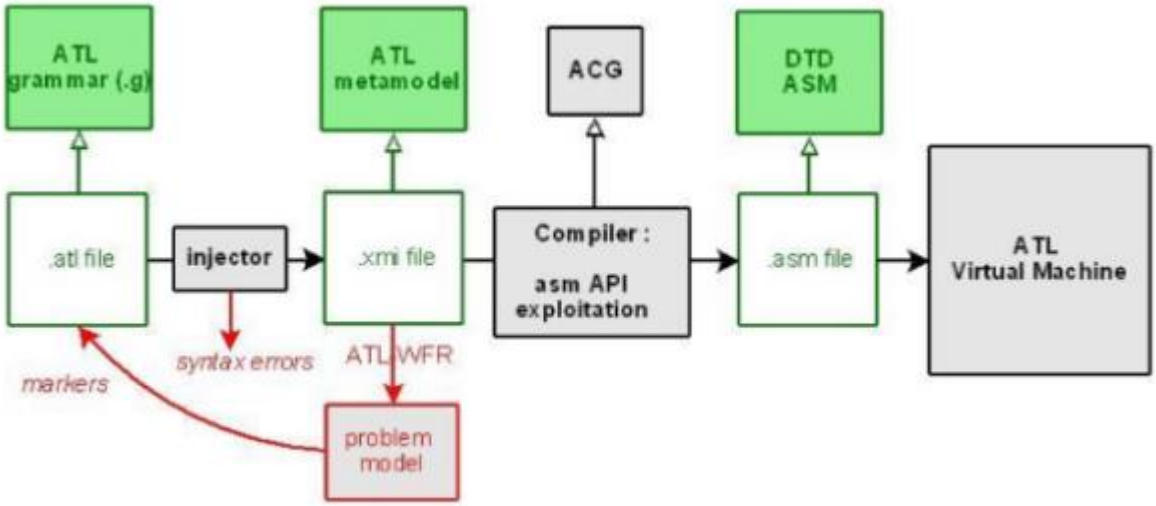


Figura 5-2. ATL Componentes

Estos componentes son presentados brevemente a continuación con el fin de dar una visión general del funcionamiento de ATL.

ATL Core. En la figura 5-3 se puede observar la API que se encuentra presente en el núcleo de ATL y cómo ésta interactúa con otras herramientas como Ant tasks. Los conceptos más importantes mostrados en la representación del núcleo son:

- **IModel:** que es una representación del modelo apta para ser utilizada en transformaciones y que provee un conjunto de métodos que permiten entre otras cosas localizar y crear nuevos elementos.
- **IRefeferenceModel:** esta interfaz extiende *IModel*, y es una versión específica que simboliza metamodelos. Define operaciones específicas de los metamodelos que son útiles en las transformaciones.
- **ModelFactory:** se dedica a la creación de modelos y modelos de referencia.
- **IInjector**, **IExtractor:** permiten guardar y cargar modelos que fueron creados previamente por la *modelFactory*.
- **ILauncher:** está interface debe ser implementada por las VMs ya que define los métodos utilizados para parametrizar y ejecutar una transformación.

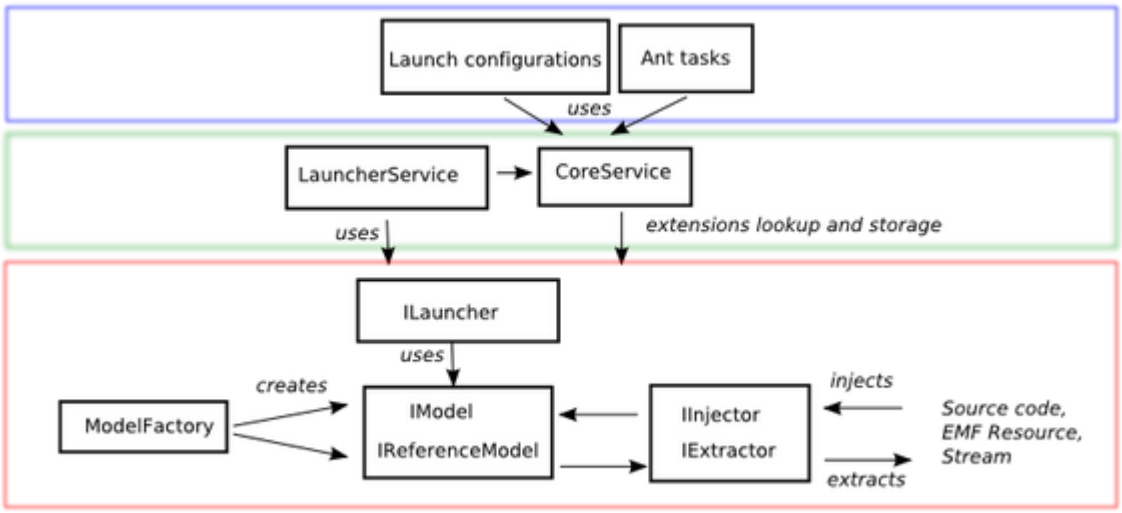


Figura 5-3. API en el núcleo ATL

Todo los visto anteriormente tiene un nivel de complejidad alto, es por esto que para poder simplificar el uso del núcleo ATL y reducir la duplicación de código se definen dos servicios: *CoreService* y *LauncherService*. *CoreService* permite buscar entre las extensiones disponibles dentro de Eclipse diferentes implementaciones para el *Core* y registrarlas para utilizar ATL de manera autónoma (sin depender de Eclipse). Por otro lado, *LauncherService* permite ejecutar una transformación a partir de un conjunto de parámetros; de esta forma se puede crear configuraciones personalizadas para ejecutar transformaciones.

Modos de ejecución. En ATL la ejecución del módulo puede realizarse de dos formas diferentes. En el modo de ejecución por defecto, modo normal, el desarrollador debe especificar explícitamente la forma en que los elementos del modelo destino se generan a partir de los elementos de un modelo origen.
Si el objetivo de la transformación apunta a copiar el modelo origen a un modelo destino sin grandes cambios, puede resultar en algo tedioso y complejo utilizar el modo normal. En este caso, resulta más conveniente utilizar el modo de ejecución refinado, el cual fue diseñado para permitir al desarrollador especificar sólo aquellas reglas que generan una modificación entre la transformación origen y los modelos destino.
Estos modos de ejecución son explicados en más detalle a continuación.

- **Modo normal**: Este modo es el modo por defecto. Está asociado a la palabra clave *from* en la sección cabecera (header) del módulo. Este modo de ejecución se utiliza en aquellas transformaciones donde el modelo origen es distinto del modelo destino.
- **Modo refinado**: este modo de ejecución, se creó para facilitar la programación de transformaciones donde el modelo origen es similar al modelo destino. Este modo está asociado con la palabra clave *refine* que se define en la sección cabecera. Es decir, en este modo el desarrollador se concentra solo en el código dedicado a la generación de elementos del modelo destino que son distintos de los definidos en el modelo origen. Para el caso de los elementos que no varían entre el modelo origen y el modelo destino estos son copiados implícitamente por el motor ATL. Este modo sólo puede utilizarse en aquellos casos en que la transformación posee un único modelo destino y tanto el modelo origen como destino son definidos conforme al mismo metamodelo.

Tipos de Unidades. Existen tres tipos de unidades sobre las que se describen operaciones sobre el modelo:

- **Library:** ATL permite desarrollar bibliotecas independientes que contienen helpers y que pueden importarse desde otras unidades ATL.
- **Query:** Es una operación que calcula un valor primitivo a partir de un conjunto de modelos origen. Estas operaciones permiten obtener valores de tipo primitivo Boolean, String, Real, Integer.
- **Module:** aquí se especifican las transformaciones como un conjunto de reglas definidas sobre los modelos origen y destino.

Todos los tipos se definen en archivos cuya extensión es *.atl*.

5.3.6.2 - *QVT*

QVT (Query, View, Transformation), es un estándar del OMG que permite escribir transformaciones; las especificaciones tienen una naturaleza híbrida y está formado por tres lenguajes M2M: dos de estos lenguajes llamados Relations y Core son declarativos y el tercer lenguaje de naturaleza imperativa llamado Operation Mappings. Estos lenguajes y sus relaciones puede observarse en la figura 5-4. A continuación describimos los 3 lenguajes.

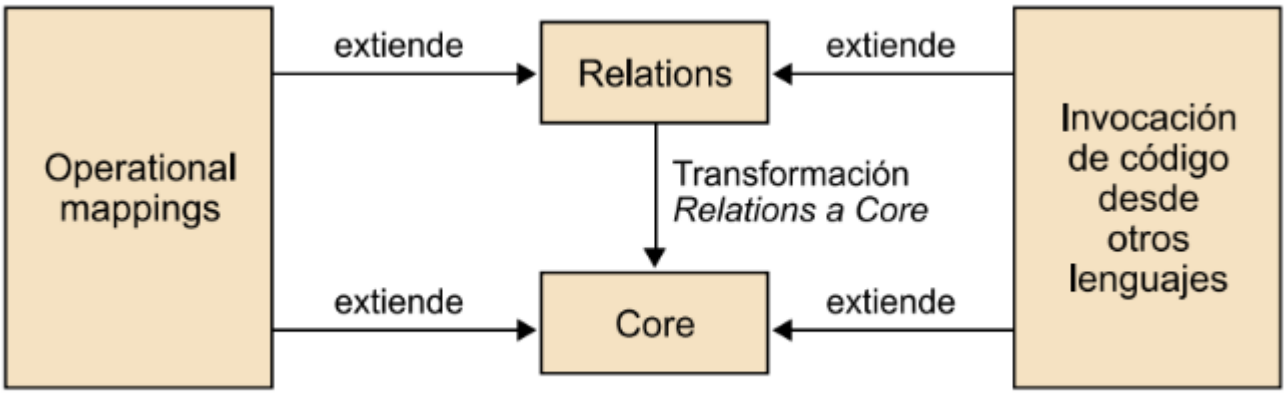


Figura 5-4. Relación metamodelos

Relations. Es el lenguaje que utilizan los desarrolladores para definir las transformaciones como un conjunto de relaciones entre modelos, lo que les permite crear elementos en el modelo destino a partir de elementos del modelo origen y también aplicar cambios sobre modelos existentes. Este lenguaje manipula automáticamente las trazas, que son estructuras de datos donde se guardan las relaciones entre los conceptos de los modelos origen y los modelos destino, haciendo su uso transparente al desarrollador.

Core. Es más simple que Relations por lo que las transformaciones escritas en este lenguaje suelen ser más extensas que sus equivalentes en Relations. Las trazas son tratadas como elementos del modelo por lo que el desarrollador es el responsable de su manipulación.
Una de las razones más importante para la existencia de este lenguaje es proporcionar una base sobre la que especificar la semántica del lenguaje Relations. Dicha semántica se da como una transformación desde Relations a Core, la cual es escrita en el lenguaje Relations.

Operational Mappings. Extiende al lenguaje Relations con constructores imperativos y constructores OCL. Además, incorpora elementos para operar (crear, modificar y eliminar) sobre el contenido de los modelos. Se estructura básicamente en procedimientos u operaciones, denominados mapping operations y helpers o queries.

Hay varios productos, comerciales o de código abierto, que requieren conformidad al estándar QVT. QVT define una manera estándar para transformar modelos de entrada en modelos de salida.

5.3.6.3 - Viatra

Visual automated model transformations, es una herramienta de transformación que forma parte del framework VIATRA2. Este framework está implementado en lenguaje Java e integrado en Eclipse.

Esta herramienta permite realizar transformaciones entre modelos y describir modelos y metamodelos a través de los lenguajes textuales que provee conocidos como *VIATRA Textual Command Language (VTCL)* y *VIATRA Textual Metamodeling Language (VTML)*. Utiliza un lenguaje de naturaleza declarativo y se basa en técnicas de descripción de patrones aunque se puede utilizar código imperativo. Utiliza métodos formales como las transformaciones de grafos (GT) y máquina de estados abstractos (ASM) de forma de poder manipular modelos basados en grafos y realizar tareas de verificación, seguridad y validación, también permite realizar evaluaciones de características no funcionales como la fiabilidad, disponibilidad y productividad del sistema bajo diseño [1]. El proyecto está compuesto por:

- Un Motor de consultas incrementales junto con un lenguaje basado en patrones de grafos que permite especificar y ejecutar consultas sobre el modelo de forma eficiente.
- Un DSL interno implementado sobre el lenguaje Xtend de forma de especificar transformaciones reactivas tanto batch como event-driven.
- Un motor complejo de procesamiento de eventos sobre modelos en EMF para especificar reacciones tras la detección de secuencias complejas de eventos.
- Un framework para la exploración del espacio diseño basado en reglas para explorar candidatos de diseño como modelos que satisfacen diferentes criterios.
- Un ofuscador de modelos de forma de poder eliminar información que es confidencial en un modelo.

En la última versión de VIATRA, la cual es una reescritura completa de VIATRA se incorpora compatibilidad con los modelos EMF que no era provista por las versiones anteriores.

5.3.6.4 - Epsilon

Epsilon es una familia de lenguajes y herramientas para la generación de código, transformaciones modelo a modelo, validación de modelos, comparación, migración y refactorización, que funcionan sin configuración extra con EMF y otros tipos de modelos [23]. Esta herramienta es una plataforma desarrollada sobre Eclipse como un conjunto de plugins. En el núcleo de Epsilon se encuentra Epsilon Object Language (EOL) que combina el estilo procedural de JavaScript con las poderosas capacidades de consulta de OCL. Este lenguaje puede ser utilizado para la modificación, consulta y creación de modelos EMF. Epsilon es una plataforma para la construcción de lenguajes específicos de tareas que se basan en EOL y pueden ser utilizados para tareas de gestión de modelos como transformación,

comparación, fusión, refactorización y validación de modelos, etc [23]. Estos lenguajes pueden verse como extensiones de EOL, ya sea sintácticamente o semánticamente, y entre los lenguajes provistos se encuentran los siguientes:

- *Epsilon Transformation Language (ETL)*, es un lenguaje de transformación M2M basado en reglas que soporta transformaciones muchos a muchos, reglas greedy y lazy, herencia y la posibilidad de consultar y modificar tanto el modelo origen como el modelo destino.
- *Epsilon Validation Language (EVL),* es el que brinda soporte para poder utilizar modelos EMF sin la necesidad de configuración y permite definir restricciones similares a las restricciones en OCL con la diferencia de que puede evaluarlas desde otros modelos (restricciones entre modelos).
- *Epsilon Generation Language (EGL),* es un lenguaje basado en templates para la generación de código, documentación y otros artefactos textuales. Provee regiones protegidas de forma de poder mezclar texto generado con fragmentos agregados manualmente.
- *Epsilon Wizard Language (EWL),* es un lenguaje diseñado para la realización de transformaciones in-place de manera interactiva a partir de elementos del modelo seleccionados por el usuario. Está integrado con EMF/GMF lo que permite ejecutar los wizards usando sus editores.
- *Epsilon Comparison Language (ECL),* es un lenguaje para descubrir correspondencias entre elementos de modelos de diversos metamodelos.
- *Epsilon Merging Language (EML)*, es un lenguaje para la combinación de modelos de diversos metamodelos. Se utiliza después de descubrir las correspondencias entre elementos utilizando ECL o algún otro lenguaje.
- *Epsilon flock,* este lenguaje se utiliza para actualizar un modelo en respuesta cambios en el metamodelo.

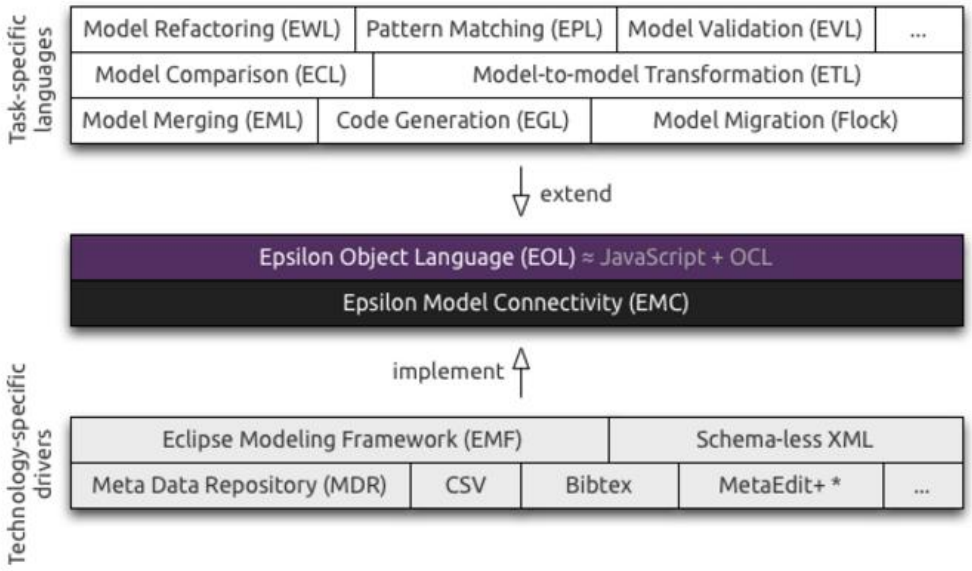


Figura 5-5. Arquitectura Epsilon

Epsilon proporciona un conjunto de herramientas y utilidades, basadas en la plataforma Eclipse, para cada uno de estos lenguajes. También provee un intérprete por cada uno de estos lenguajes de forma de poder ejecutar los programas escritos en ellos.

Como se mencionó anteriormente el objetivo principal de EOL es poder brindar un conjunto de servicios reutilizables que sean comunes a todos los lenguajes. Sin embargo, EOL también puede usarse como un lenguaje de propósito general independiente para la automatización de tareas que no caen en los patrones de los lenguajes específicos de tareas existentes.

5.4 - M2T

Las transformaciones de Modelo a Texto (M2T, *Model-To-Text*) tienen por objetivo la generación de artefactos de tipo texto a partir de un modelo. Estas transformaciones son utilizadas en el manejo de operaciones de modelos, para la generación de código, para la documentación y serialización.
También son conocidas como transformaciones de modelo a código y pueden ser vistas como un caso especial de las transformaciones modelo a modelo en donde lo único que debemos proveer es el metamodelo del lenguaje de salida.

5.4.1 - Visitante

Este enfoque consiste en la generación de código que provea una clase de mecanismo para recorrer la representación interna del modelo y escribir el texto en un archivo de salida. Por ejemplo Jamda, un framework orientado a objetos, provee lo que denomina CodeWriters que proveen un mecanismo para la implementación de este tipo de transformaciones.

5.4.2 - Plantilla

De todas las herramientas MDA disponibles actualmente la mayoría soporta el uso de plantillas para la transformación modelo a texto.
Una plantilla es un archivo de texto que representa el formato de salida de la transformación pero contiene referencias a variables que serán reemplazadas en la ejecución por valores tomados del modelo de entrada. Es decir, que puede verse como una mezcla de elementos de texto estáticos que son compartidos por todos los artefactos y por partes dinámicas que serán completadas en cada caso particular.
Este método de transformación requiere de un motor que es el encargado de procesar las plantillas y consultar modelos para completar las partes dinámicas.
Cuando las plantillas operan sobre el texto, los patrones que contienen no están tipificados y pueden representar fragmentos de código sintácticamente o semánticamente incorrectos.
El proceso de creación de plantillas resulta simple ya que el proceso de abstracción desde un ejemplo concreto de código a una plantilla es directo, solo se deben identificar las partes dinámicas. Si comparamos las plantillas con las transformaciones basadas en visitante, la estructura de una plantilla está más relacionada al formato del código que se va a generar lo que hace muy sencillo revisar los efectos de la transformación.

5.4.3 - Herramientas

5.4.3.1 - MOFScript

Fue desarrollada por SINTEF ICT para la generación de código y documentación, es una herramienta muy utilizada para transformaciones de modelo a texto. La implementación de esta herramienta se basa en la especificación M2T de la OMG, se alinea con el estándar QVT y utiliza EMF y Ecore como frameworks para los metamodelos. Provee un lenguaje que es independiente del metamodelo lo que permite utilizar cualquier metamodelo y sus instancias para la generación de texto.

MOFScript se distribuye como un plugin que incorpora las herramientas necesarias para que el usuario pueda interactuar con los servicios de reconocimiento y verificación de los programas, ejecución de las transformaciones, etc.
En la imagen 5-5 se muestra la arquitectura MOFScript.

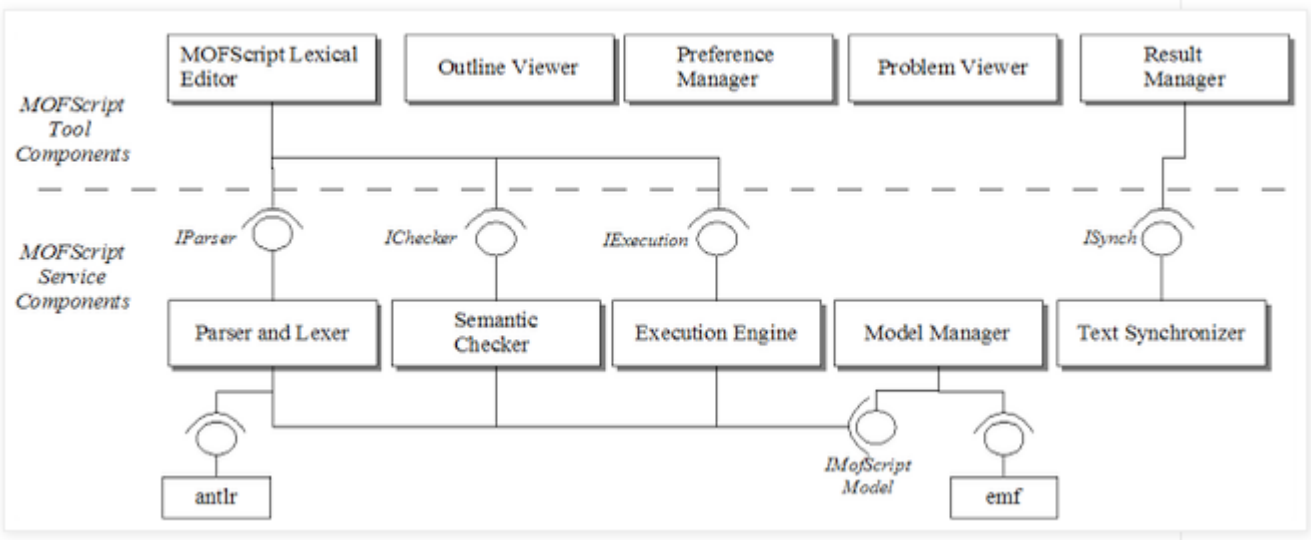


Figura 5-6. Arquitectura MOFScript

La arquitectura de MOFScript puede dividirse en dos partes, herramientas (Tool Components) y servicios (Service Components). Las herramientas permiten al usuario la edición e interacción con los servicios y los servicios proveen parsing, chequeo y ejecución del lenguaje de transformación.
Esta herramienta permite generar múltiples archivos de salida y la integración con código Java externo (una transformación puede ejecutar métodos de una clase Java). MOFScript es un lenguaje imperativo que es sencillo, con pocos constructores, esto es similar a los lenguajes de scripting. Para poder invocar las reglas es necesario hacerlo explícitamente, excepto la regla inicial main y se permite la herencia de transformaciones.

5.4.3.2 - Acceleo

Acceleo es un plugin Eclipse que se integra con el IDE desarrollado en Java y construido sobre EMF y OCL, también tecnologías Eclipse. El lenguaje que se utiliza para definir un generador de código es una implementación del estándar MOFM2T. Este lenguaje utiliza el enfoque basado en plantillas, donde la plantilla contiene una parte dedicada donde se procesan objetos del modelo de entrada para generar el texto de salida.
Acceleo proporciona un asistente para la generación automática de las transformaciones conocidas como módulos, cuya extensión es .mtl, en el cual es necesario indicar previamente el metamodelo y el modelo de entrada mediante una cadena a través de la cual se indica la ubicación y el nombre; luego se deben indicar las plantillas a ser utilizadas en el proceso de generación.

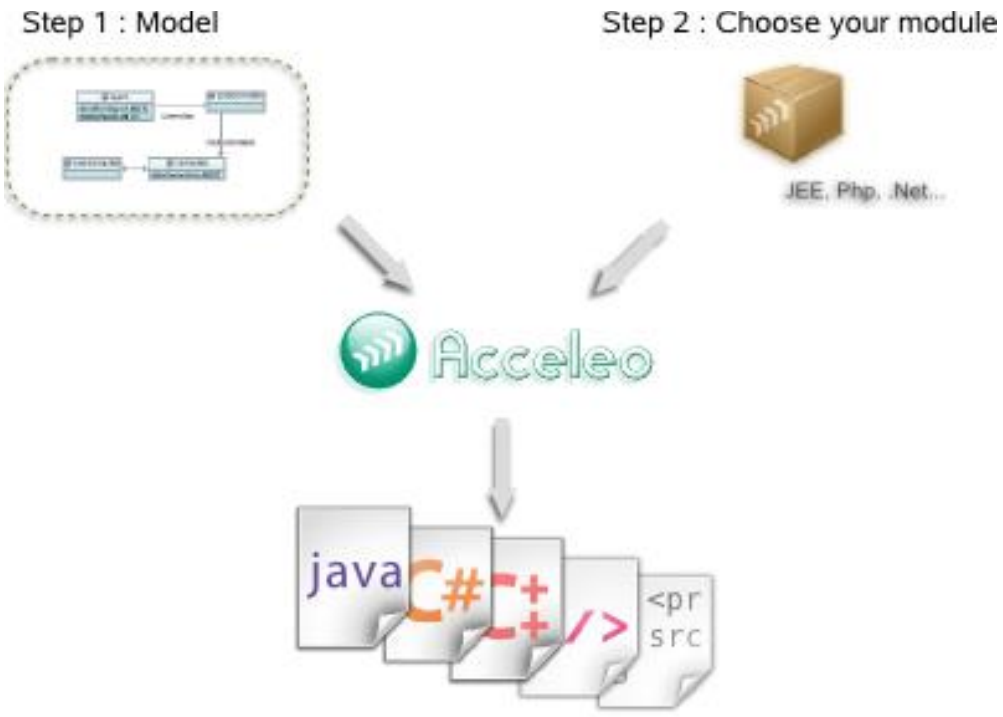


Figura 5-7. Funcionamiento de ACCELEO

Gracias a que Acceleo tiene un enfoque basado en prototipos facilita el trabajo de los desarrolladores permitiéndoles crear, fácil y rápido, un generador o plantilla desde el código fuente de los prototipos predefinidos con los que cuenta. Una vez que la plantilla es creada y mediante el uso de las herramientas provistas por Acceleo como las de refactoring es muy fácil mejorar la plantilla y convertirla en una plantilla completa.
En la imagen 5.14 se muestra un ejemplo de plantilla.

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/uml2/3.0.0/UML')/]

[template public generate(aClass : Class)]

    [comment @main /]
    [file (aClass.name, false, 'UTF-8')]
    [aClass.name/]
    [/file]

[/template]
```

Figura 5-8. Ejemplo Plantilla ACCELEO

Estas generaciones o módulos definidos en Acceleo además de plantillas pueden contener consultas definidas usando la palabra clave Query que se utilizan para extraer información de los modelos procesados usando OCL o importar otros módulos mediante el uso de la palabra clave *import*. A continuación la sintaxis de ambos:

```
[query private
    getterPrefix(feature : EStructuralFeature) :
    String = if feature.eType.name = 'EBoolean' then 'is'
  else 'get' endif/]

  [import qualified::name::of::imported::module/]
```

Una vez que se define el modelo de entrada y las reglas en la planilla de transformación, la herramienta analiza de forma automática el modelo y le aplica las transformaciones correspondientes a cada elemento de entrada en base a las reglas contenidas en la plantilla, obteniendo así el texto plano de salida.
Acceleo proporciona el siguiente conjunto de herramientas:

- *Editor* que resalta sintaxis, provee autocompletado, opciones de refactoring y detecta errores en tiempo real.
- *Debugger* que permite seguir el progreso de la transformación mediante el uso de breakpoints, realizar chequeos del estado de variables y ejecutar paso a paso para encontrar errores.
- *Profiler,* que permite la visualización de las instrucciones ejecutadas durante la transformación, el número de veces que esas instrucciones fueron ejecutadas y el tiempo que requirió la ejecución.
- *Traceability*, el motor de Acceleo puede procesar todas las trazas de todos los elementos involucrados en la generación de un archivo. Permite por ejemplo determinar qué elementos del modelo de entrada han sido utilizados para la generación del texto de salida y que parte del generador de código se utilizó.

Como ya se mencionó anteriormente, Acceleo está basado en los principales estándares lo que garantiza su compatibilidad e interoperabilidad con otras aplicaciones, como GMF. Además, es compatible con XMI y permite extender las funcionalidades a través de la importación de las bibliotecas de Java que permiten agregar funcionalidades a las plantillas y la generación de código.

5.4.3.3 - XPAND

Es un framework para la generación de transformaciones M2T que está integrado en Eclipse. Para lograr esta integración y poder utilizar el framework es necesario contar con dos herramientas complementarias: *XTEXT* y *XTEND* que permiten realizar el análisis del modelo mediante la definición de una gramática y definir las transformaciones sobre el mismo respectivamente. Tanto XPAND como XTEND se encuentran construidos sobre la misma base de expresiones y tipos lo que hace su uso más simple ya que no se necesita aprender dos lenguajes y la forma en la que ambos interactúan con los modelos, metamodelos y meta-metamodelos es exactamente la misma.
Este framework provee un potente lenguaje de expresiones estáticas y brinda una capa de abstracción que permite interactuar con los diferentes metamodelos compatibles entre los que se encuentran Ecore, UML, XML, XMI. La capa de abstracción que provee es comúnmente llamada sistema de tipos y provee acceso a los tipos de datos que están registrados y que se corresponden con los tipos de datos del modelo o modelos concretos que se intentan analizar. Los elementos de un modelo tienen un tipo asignado y cada tipo tiene sus propias operaciones y propiedades. Los tipos a su vez pueden heredar de otros. Cada tipo definido en el modelo debe estar registrado en el sistema de tipos para poder ser iterado utilizando lenguaje XPAND.

El lenguaje XPAND utiliza plantillas de forma de poder controlar y gestionar la generación de una salida a partir de un modelo de entrada. Las plantillas se guardan en ficheros con extensión '.xpt'. La sintaxis de este lenguaje define que cada expresión debe estar definida entre los símbolos << y >> y permite realizar importaciones, bucles, entre otras operaciones comunes. Entonces, partiendo de un modelo base que sea reconocido por la herramienta se itera utilizando las expresiones entre comillas para generar la salida en el texto que se necesite. En la imagen 5-15 puede verse un ejemplo de las plantillas XPAND.
Al utilizar plantillas XPAND, un problema que puede surgir es que el sistema de tipos no tenga registrado alguno de los elementos que se desea transformar y que se deban realizar transformaciones más complejas. Si esto sucede XTEND es la solución. XTEND es un lenguaje que permite al usuario definir sus propias bibliotecas de operaciones independientes y extensiones del metamodelo que no sean invasivas. Dichas bibliotecas pueden cargarse y ser invocadas desde cualquiera de los demás lenguajes de expresiones dentro del framework. Los archivos en este lenguaje se definen con una extensión *.ext*.

```
«IMPORT paginaweb»

«DEFINE main FOR Pagina»
«FILE nombre+".html"»
    «FOREACH secciones AS seccion»
    <div style="border:1px">
        <h1>«seccion.nombre»</h1>
        <table border="1">
        «FOREACH seccion.articulos AS articulo»
            <tr>
                <td colspan="2">«articulo.urlImagen»</td>
            </tr>
            <tr>
                <td><img src="«articulo.urlImagen»"/></td>
                <td>«articulo.contenido»</td>
            </tr>
        «ENDFOREACH»
        </table>
    </div>
    «ENDFOREACH»
«ENDFILE»
«ENDDEFINE»
```

Figura 5-9. Ejemplo Plantilla de transformación XPAND

XPAND se considera obsoleto y los tiempos de ejecución empeoran a medida que la cantidad de transformaciones se incrementa. Una versión más eficiente de este framework es XTEND2 que combina Xtext y Xbase.

5.5 - Conclusiones

En las secciones anteriores se realizó una presentación de los posibles tipos de transformaciones, M2M y M2T, y algunas de las herramientas disponibles de cada tipo.
Para el desarrollo de la herramienta propuesta en la presente tesina se tomaron decisiones en dos niveles, que tipo de transformación y que herramienta se adapta mejor a los objetivos del desarrollo. El primer nivel analizado fue el tipo de transformación a utilizar, M2M o M2T. Se decidió que como debíamos transformar las restricciones definidas representadas en su correspondiente metamodelo e implementar el metamodelo para lenguaje natural de manera de reducirlo al dominio de OCL para disminuir su complejidad y evitar la ambigüedad lo que se necesitaba eran transformaciones M2M. Otra razón en la que se basó está elección es la

de proveer la opción de implementar las transformación a la inversa, de lenguaje natural a OCL, que no sería posible en el caso de usar una transformación M2T.
En el segundo nivel, una vez decidido que la transformación se realizaría utilizando M2M, se analizaron las herramientas investigadas y se pudo observar que la herramienta que más se adapta a las necesidades era ATL.
Un punto de comparación es el sentido de la transformación, bidireccional o unidireccional, QTV permite transformaciones bidireccionales mientras que en ATL y EOL las transformaciones definidas son unidireccionales.
También podemos comparar estos lenguajes basándose en si son declarativos, imperativos o híbridos, en el caso de los 3 lenguajes vistos todos son híbridos. Sin embargo, QVT operational que es la implementación de Operation Mappings de QVT no provee una implementación completa. Los lenguajes híbridos proveen la ventaja de que incrementan la expresividad permitiendo crear transformaciones complejas, QVT al no proveer una implementación completa queda descartado como opción a utilizar; dejando solamente a ATL y EOL en carrera.
La facilidad de uso y aprendizaje del lenguaje fue otro punto importante en la decisión, EOL y en general el framework Epsilon proveen mucha más flexibilidad que la necesaria para el desarrollo de nuestro prototipo. Se intentó realizar ejemplos en ambos lenguajes, ATL y EOL, resultando en ATL siendo más fácil de entender y demostrando que existe más soporte disponible ya que la comunidad que lo utiliza es mayor.

Capítulo 6: Herramientas

6.1 - Transformación de Modelos en ATL

En el capítulo anterior presentamos diversas herramientas para cada tipo de transformación, M2M y M2T, y se realizó un análisis de las mismas concluyendo que las transformaciones M2M y ATL en particular eran las opciones que más se adaptaban a las necesidades de este proyecto.
El lenguaje ATL se basa en la definición de reglas de transformación de un modelo de entrada a un modelo de salida. En esta sección se revisa de manera más profunda la sintaxis y semántica de la definición de reglas de transformación ATL y las características imperativas del lenguaje.

6.1.1 - Visión general de las transformaciones ATL

ATL sigue el proceso de transformación modelo a modelo donde se aplica un patrón de transformaciones el cual puede verse en la Figura 6-1. Esto es, un modelo origen *Ma* se transforma a un modelo destino *Mb*. Para lograr esto, se define un programa de transformación, *mma2mmb.atl,* escrito en ATL. Tanto el modelo *Ma* como el modelo *Mb* se ajustan a un metamodelo, *MMa* y *MMb* respectivamente. Estos metamodelos a su vez se ajustan al meta-metamodelo MOF.

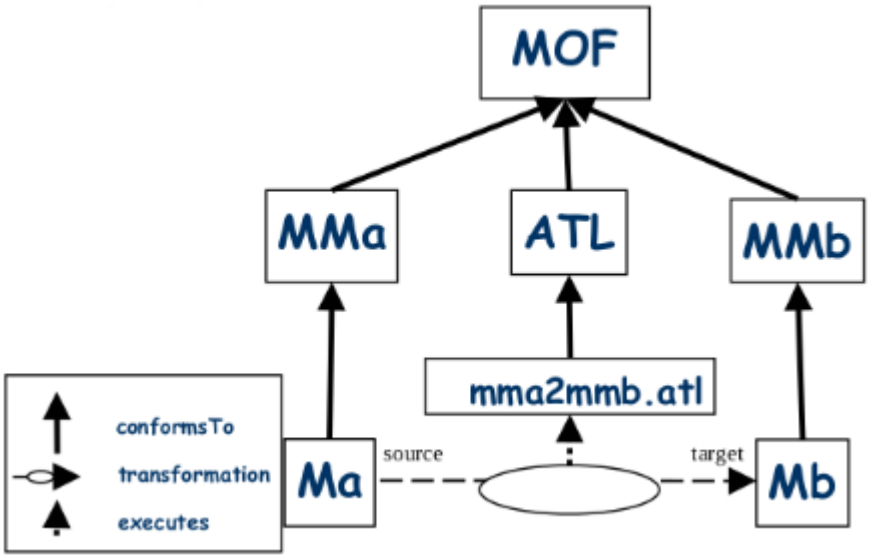


Figura 6-1. Visión general transformación ATL

Como se mencionó anteriormente, ATL es un lenguaje de transformación híbrido ya que permite construcciones declarativas e imperativas. Se recomienda utilizar sentencias declarativas pero pueden utilizarse sentencias imperativas en caso de que la regla a definir sea muy compleja. Las transformaciones ATL son unidireccionales, es decir el modelo origen es de solo lectura y el modelo destino siendo generado es de solo escritura. Esto significa que durante la ejecución de una transformación se puede navegar el modelo origen pero no se pueden realizar cambios sobre el mismo y se pueden realizar cambios en el modelo destino pero no se lo puede navegar. Para lograr transformaciones bidireccionales es necesario definir dos transformaciones, cada una con un conjunto de reglas en cada dirección.

6.1.2 - Estructura de las transformaciones ATL, Modules

Las transformaciones ATL son también llamadas módulos y son estos los que contienen la configuración de la transformación conocida como header, una sección que permite importar bibliotecas y las reglas y funciones conocidas como helpers. Las reglas y helpers no se definen en una sección específica por lo que pueden ser declarados en cualquier orden, siempre y cuando respeten ciertas condiciones. Veremos en detalle todos estos componentes estructurales a continuación con la excepción de las reglas que por ser uno de los componentes más importantes se explicarán en profundidad en su propia sección.

6.1.2.1 - Header

En esta sección se define el nombre del módulo de transformación y el nombre de las variables que corresponden al modelo de entrada y al modelo de salida. También incluye el tipo de ejecución a ser utilizado. La sintaxis utilizada es la siguiente:

```
module module_name;
create output_models [from|refines] input_models;
```

6.1.2.2 - Import

Está sección es opcional pero cuando está definida se utiliza para declarar que bibliotecas ATL deben ser incluidas en el módulo. Se utiliza la siguiente sintaxis para su declaración:

```
uses extensionless_library_file_name;
```

Es posible incluir múltiples bibliotecas mediante la declaración sucesiva de sentencias *uses*.

6.1.2.2 - Helpers

Los helpers se utilizan para definir atributos o funciones. Permiten la modularización del código ATL y pueden ser invocados desde otros helpers o reglas. La definición de helpers es a través de un nombre, un contexto de aplicación (opcional), parámetros (opcionales) y un tipo de retorno. La sintaxis:

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

Mediante la palabra reservada *context* se indica el tipo de elementos a los cuales el helper se aplica; es decir, el tipo de elementos desde los cuales se podrá invocar al helper. En caso que no se especifique el contexto, el helper se asocia al contexto global que es el módulo ATL; esto significa que las referencias a *self* en el cuerpo del helper refieren al módulo. Con la palabra reservada *def* se define el nombre del helper. Los helpers se identifican por su nombre, contexto, parámetros y tipo de retorno por lo que es posible tener 2 helpers con el mismo nombre y que realicen la misma operación siempre y cuando el contexto sea diferente.
La definición de un parámetro incluye el nombre y tipo del mismo. Los parámetros se indican entre paréntesis a continuación del nombre del helper y en caso de ser más de uno se separan por comas. La sintaxis para la definición de un parámetro es:

```
parameter_name : parameter_type
```

Cualquiera de los tipos soportados por ATL puede utilizarse en la definición de los tipos del contexto, parámetros o retorno. El cuerpo de un helper se especifica con una expresión OCL.

Como mencionamos al inicio de esta sección los helpers pueden utilizarse para definir atributos que, comparados con las funciones helpers pueden verse como constantes definidas en un contexto en particular. La principal diferencia entre una función y un atributo es que los helper atributos no permiten la definición de parámetros. La sintaxis es similar y se define como sigue:

```
helper [context context_type]?
     def: attribute_name: return_type = exp;
```

La declaración de una función sin parámetros y la de un atributo pueden parecer equivalentes pero no lo son ya que existe una diferencia en la semántica de su ejecución. El código de las funciones es ejecutado cada vez que la misma es invocada mientras que un atributo siempre retorna el mismo valor para un mismo contexto de ejecución por lo que la evaluación se realiza solo una vez.

6.1.2.3 - Tipo de dato, ATL Module

Este tipo de dato es específico del lenguaje y se utiliza internamente para representar una unidad, que puede ser un módulo o una consulta, que está siendo ejecutada por el motor ATL. Existe una única instancia de este tipo de datos y se puede hacer referencia a ella desde el código a través de la variable *thisModule*. Esta variable hace posible tener acceso a los helpers y a los atributos que se hayan declarado en el contexto del módulo.

El tipo de dato ATL module también proporciona la operación *resolveTemp* que permite apuntar, desde una regla ATL, a cualquiera de los elementos del modelo destino que serán generados a partir de un elemento del modelo origen mediante la ejecución de una regla. La sintaxis de esta operacion es:

```
resolveTemp(var, target_pattern_name)
```

El parámetro *var* corresponde a una variable ATL que contiene el elemento del modelo origen desde el que se realiza la transformación al elemento del modelo destino. El parámetro *target_pattern_name* es un string que corresponde al nombre del patrón del destino que mapea el elemento del modelo origen que se indicó en *var*. Como esta operación se define en el ámbito del módulo ATL debe ser invocada mediante el uso de *thisModule*.

La operación *resolveTemp* no puede ser invocada antes de que se complete la fase de matching de la regla. Esto significa que esta operación puede ser invocada desde:

- Las secciones target pattern y *do* de cualquier matched rule.
- Las secciones target pattern y *do* de una called rule siempre y cuando está regla es ejecutada después de la fase de matching; es decir no es invocada desde la regla de entrada de la transformación.

Estos conceptos serán clarificados en la siguiente sección cuando se explique la semántica de las reglas y los bloques imperativos en ATL.

6.1.3 - Reglas de transformación

Las reglas definidas en ATL permiten definir cómo se realizarán las transformaciones; es decir, cómo será la correspondencia entre los elementos del modelo de entrada con los elementos del modelo de salida. Las reglas pueden contener condiciones, variables, asignaciones y otro tipo de sentencias.
Como se ha mencionado con anterioridad, ATL al ser un lenguaje híbrido permite definir sentencias declarativas y sentencias imperativas. A continuación presentamos los diferentes tipos de reglas y cómo estas se relacionan con la definición de los diferentes tipos de sentencias.

6.1.3.1 - Tipos de reglas

Matched Rules. Permiten definir la forma en que los elementos en el modelo destino son creados a partir de los elementos en el modelo origen. Estas reglas se dividen en cuatro partes:

1. *From,* determina el elemento del modelo origen.
2. *Using* (opcional), dónde se declaran las variables a ser utilizadas en el resto de la regla.
3. *To,* es donde se especifica la cantidad y el tipo de los elementos a ser generados en el modelo destino
4. *Do* (opcional), se utiliza para definir código imperativo, en caso de ser necesario.

La sintaxis para la especificación de este tipo de reglas es:

```
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
     }]?
}
```

Cada regla se identifica mediante el rule_name y este debe ser único entre las reglas definidas en el módulo de transformación.
El patrón origen definido con la palabra clave from contiene in_var que es el nombre que se le dara a la variable de entrada e in_type define el tipo de elemento de entrada que será transformado por la regla. Se puede definir condiciones booleanas para filtrar qué elementos del tipo definido son transformados.
El patrón destino se define usando la palabra clave *to*, que debe contener al menos un patrón definido y si se define más de uno estos son separados por comas. *out_vari* define el nombre de la variable que contiene el elemento de salida que será del tipo *out_typei.*
La sección imperativa se define mediante el uso de la palabra clave *do* y es ejecutada una vez que el resto de la regla es completada (la fase de matching y la inicialización de elementos destino). Veremos en la siguiente sección el tipo de sentencias imperativas que pueden utilizarse.

Lazy Rules. Son reglas declarativas como las matched rules pero son invocadas explícitamente desde otras reglas. La sintaxis sólo varía con respecto a las matched rules en que la palabra clave *lazy* es incluida en la definición de la regla:

```
lazy rule rule_name {…}
```

Estas reglas también pueden ser declaradas como *unique* cambiando el comportamiento de la regla. La regla retorna siempre el mismo elemento destino para un mismo elemento de origen.

```
unique lazy rule {…}
```

Called Rules. Estas reglas permiten a partir de código imperativo generar un elemento del modelo destino. La forma de invocarlas es desde la parte imperativa de otras reglas con excepción de las definidas como *entrypoint* or *endpoint*. Si una regla es definida como *entrypoint* esto significa que la regla será ejecutada implícitamente al inicio del módulo. Por el contrario, si se la define como *endpoint* la regla será implícitamente invocada como última regla antes de la finalización del módulo.

```
[entrypoint | endpoint]? 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
    }]?}
```

Una called rule puede recibir, o no, parámetros y estos se definen de la misma forma en que se hace con los helpers. Estas reglas se componen de 3 secciones opcionales: *to*, *using* y *do* que tienen la misma semántica que la definida para las matched rules. Es importante destacar que no poseen una sección *from* ya que no es necesario.
Las transformaciones definidas en el módulo soportan herencia simple que puede utilizarse como un mecanismo de reutilización de código y también como un mecanismo para especificar reglas polimórficas. ATL provee las palabras claves *abstract* y *extends* para definir la relación de herencia.

6.1.3.2 - Semántica de la ejecución de reglas

Las reglas se ejecutan sobre un patrón de origen y para una coincidencia encontrada entre un elemento del modelo de origen y el patrón de origen definido en la regla, se crean los elementos destino de los tipos especificados en el modelo destino y son inicializados usando enlaces. La ejecución de una regla crea además una traza de las estructuras internas del motor de transformación relacionando la regla, el elemento origen que disparó la ejecución y los elementos de destino que recién se crearon.
El algoritmo aplicado para la inicialización de los elementos destino se conoce como algoritmo de resolución ATL. Este algoritmo evalúa la expresión definida y el resultado se resuelve antes de ser asignado a la entidad destino. La resolución depende del tipo del valor, por lo que si el tipo es un tipo primitivo entonces el valor se asigna directamente y si el tipo es un tipo del metamodelo hay dos posibilidades:

- Cuando el valor es un elemento destino se asigna simplemente a la entidad.
- Cuando el valor es un elemento origen, primero se resuelve en un elemento de destino mediante los links de trazabilidad. La resolución da como resultado un elemento del modelo destino que se asigna a la entidad. Este algoritmo utiliza los links de trazabilidad para identificar los elementos del destino creados a partir de elementos del origen como resultado de la aplicación de una regla de transformación.

Si nos enfocamos en los tipos de reglas más utilizadas podemos decir que:

- *Matched rules*, son aplicadas cada vez que se encuentra un elemento en el modelo origen que coincide con el patrón origen definido.
- *Lazy rules*, se aplican tantas veces como estas sean invocadas. Pueden ser aplicadas varias veces para una misma coincidencia.
- *Unique Lazy rules*, son invocadas por otras reglas pero solo se aplican una vez por coincidencia. Es decir, si para el mismo mapeo entre un elemento del modelo de entrada está regla es invocada múltiples veces solo un elemento destino será generado y este será reutilizado cada vez.

6.1.3.3 - Características imperativas de ATL

El estilo declarativo en la definición de transformaciones tiene como ventaja que se basa en especificar las relaciones entre patrones de origen y destino, entonces tiende a estar más cerca de la forma en que los desarrolladores intuitivamente perciben la transformación. Este estilo se apoya en la codificación de estas relaciones y oculta los detalles relacionados con la selección de los elementos origen, el desencadenamiento y ordenación de las reglas, etc. Sin embargo, en algunos casos puede ser necesario definir algoritmos más complejos y

puede ser difícil especificar una solución declarativa pura para ellos por lo que ATL provee los bloques imperativos que consisten de una secuencia de sentencias imperativas.
Los bloques imperativos pueden utilizarse para inicializar elementos destino que no han sido inicializados de manera declarativa o para modificar elementos ya inicializados. La sintaxis de las sentencias dentro de estos bloques es diferente que la utilizada en la definición de sentencias declarativas. Cada sentencia debe terminar con un punto y coma (;) y solo se permiten tres tipos de sentencias: sentencias de asignación, sentencias *if* y sentencias *for*.

Asignación. Permiten asignar un valor a una variable definida dentro del contexto del módulo ATL o a un atributo de un elemento destino. La sintaxis de esta sentencia es:

```
target <- exp;
```

If. Permite expresar alternativas, donde la condición debe ser una expresión OCL que retorna un valor booleano. La declaración de una alternativa *else* es opcional como se puede ver en la especificación de su sintaxis:

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

for. Permite realizar iteraciones sobre una colección. La sintaxis de está sentencia se define como:

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

Este tipo de bloque posee la limitación de que no permite la declaración de variables. Las únicas variables que pueden ser utilizadas en estos bloques son:

- Los elementos origen y destino declarados en la regla actual.
- Las variables declaradas localmente en la regla.
- Atributos declarados como globales. Es decir, en el contexto del módulo.

Es importante notar que las variables declaradas en la regla pueden ser utilizadas pero no modificadas dentro del bloque imperativo, por lo que si se necesita contar con variables que puedan ser modificadas en estos bloque se deben declarar como atributos del módulo.

6.2 - Xtext

6.2.1 - Introducción

A través de Xtext se puede desarrollar gramáticas de lenguaje usando una sintaxis similar a EBNF. Xtext nos permite crear un metamodelo basado en Ecore, un editor de texto para

Eclipse y su parser. La idea de Xtext es, en lugar de ir de un modelo existente y derivar a una gramática que lo soporte, comenzar con la gramática y producir el modelo Ecore.
De todas formas, este lenguaje permite realizar transformaciones desde y hacia modelos Ecore manejando interoperabilidad con otras herramientas basadas en EMF (por ejemplo MOFScript, ATL, etc). En caso de ser necesario, se puede especificar la gramática usando un metamodelo existente por medio de un mecanismo de importación.
Resulta sencillo trabajar con el lenguaje generado ya que como se mencionó anteriormente se genera un editor de texto y este resalta sintaxis, autocompleta el código, detecta errores, entre otras cosas.
Xtext es un proyecto de código abierto de Eclipse.org por lo que está completamente integrado a Eclipse.

6.2.2 - Estructura Xtext

Al crear un nuevo proyecto Xtext en Eclipse podemos utilizar el asistente Xtext integrado. En este asistente se define el nombre del proyecto (el cual se espera sea significativo), el nombre del lenguaje y un nombre para la extensión de los archivos generados en el lenguaje. Una vez que se ha creado el nuevo proyecto se encontrará que hay 5 nuevos proyectos no solo uno. Estos 5 proyectos poseen diferentes nombres y componentes:

- Un proyecto donde se definirá la gramática y que incluye todos los componentes específicos del lenguaje (parser, linker, lexer, validator, etc).
- Un proyecto con sufijo .tests, donde se definen los test de unidad del lenguaje.
- Un proyecto con sufijo .ide, donde se encuentran todas las funcionalidades IDE independientes de la plataforma como servicios de ayuda para el desarrollo de contenido o código.
- Un proyecto .UI que contiene el editor Eclipse, vistas y perspectivas, entre otros.
- Un proyecto .UI.test, donde se definen los test de unidad del editor Eclipse.

Todos los proyectos tienen dos carpetas source. *src* que contiene código generado por el desarrollador y la carpeta *src-gen* que contiene código autogenerado por lo que no debe modificarse su contenido manualmente ya que su contenido es sobrescrito cada vez que se genere el lenguaje a partir de la gramática.
Xtext provee generación de código que por defecto viene configurada para generar algunos componentes automáticamente (como el linker, serializer, algunas clases útiles, etc). Está configuración puede ser fácilmente modificada para agregar código autogenerado, a las configuraciones o fragmentos ya existentes o crear nuevos.

6.2.3 - Gramática

Vamos a analizar la estructura del archivo correspondiente a la gramática del lenguaje natural reducido definida como parte del desarrollo de la herramienta de la presente tesina. Se creó un proyecto llamado ***LenguajeNaturalReducido*** y dentro del proyecto se creó automáticamente un archivo con el mismo nombre con extensión .xtext donde se debe definir la gramática. Una gramática tiene dos propósitos:

- Describir la sintaxis concreta de un lenguaje.
- Contener información sobre cómo el parser debe crear un modelo durante el análisis.

En las primeras líneas de la gramática definimos:

```
grammar org.xtext.tesina.LenguajeNaturalReducido
            with org.eclipse.xtext.common.Terminals
```

El nombre de la gramática se corresponde con el nombre del archivo de la gramática que se encuentra en org/xtext/tesina/LenguajeNaturalReducido y cuya extensión es .xtext; entonces el nombre de la gramática es org.xtext.tesina.LenguajeNaturalReducido. En la segunda parte de la declaración, *with* org.eclipse.xtext.common.Terminals determina que la gramática reutiliza las reglas de la gramática especificada. El uso de *with* es opcional. org.eclipse.xtext.common.Terminals es una gramática proveída con Xtext que define las reglas terminales más comunes como ID, String o Int. También contiene reglas para comentarios de una o varias líneas, reglas para espacios en blanco.
La siguiente declaración, *generate*, le indica al framework Xtext que debe inferir el modelo Ecore de la gramática. Es decir, le indica que debe generar un paquete Ecore con el nombre indicado (lenguajeNaturalReducido).

```
generate lenguajeNaturalReducido
        "http://www.xtext.org/tesina/LenguajeNaturalReducido"
```

Xtext usa modelos EMF para representar en memoria cualquier archivo de texto parseado. En lugar del modelo también podemos referirnos a él como modelo semántico o Abstract Syntax Tree (AST). En este punto, hay que tener en cuenta que Xtext es en realidad un grafo en lugar de un árbol ya que también contiene enlaces cruzados.
Luego, en el cuerpo de la gramática se definen las reglas gramaticales. En estas reglas definen expresiones que pueden tener diferentes cardinalidades; las cuales pueden verse en la figura 6-5.

(no operator)	exactly one
?	zero or one
*	zero or more
+	one or more

Figura 6-5. Cardinalidad elementos

Nuestra gramática comienza definiendo una regla Documento, que es la regla inicial:

```
Documento:
    encabezado=Literal FinOracion
    oraciones+=Oracion*;
```

Esta regla determina que un Documento está compuesto por un encabezado (que es un objeto Literal y un objeto FinOracion) y por un conjunto de oraciones (que contendrá cero o muchos objetos Oracion).
Como se muestra en el ejemplo cada regla termina con un punto y coma. Una gramática Xtext no solo describe reglas para el parser sino que también definen la estructura del AST resultante. Normalmente cada regla del parser crea un nuevo objeto en el árbol y el tipo de ese elemento se puede especificar después del nombre de la regla usando la palabra clave *returns*. Si consideramos que el nombre del tipo es el mismo que el nombre de la regla:

```
Documento returns Documento:
    encabezado=Literal FinOracion
    oraciones+=Oracion*;
```

La palabra clave *returns* es opcional pero si no se la especifica el tipo del objeto es inferido del nombre de la regla convirtiendo a la regla definida antes y a la siguiente equivalentes:

```
Documento:
    encabezado=Literal FinOracion
    oraciones+=Oracion*;
```

Además de definir las reglas es necesario conectar los diferentes objetos; es decir, asignar los elementos devueltos por las reglas llamadas en el lado derecho de la misma a alguna característica del objeto de la regla actual. Esto se logra a través de las asignaciones de las cuales existen 3 tipos:

1. = es la asignación directa que asigna un único valor al atributo.
2. += es una operación que no asigna sino que espera un atributo multivaluado al que puede agregar el elemento del lado derecho. Está asignación es la utilizada en la regla Documento presentada como ejemplo y lo que hace es agregar elementos del tipo Oracion a la lista de oraciones.
3. ?= es una asignación booleana que espera un atributo booleano que será verdadero si el lado derecho consumió algún valor independientemente del valor específico.

También existen reglas terminales que son aquellas que devuelven tipos de datos simples como String, Date o Integer, etc. Un ejemplo de estas reglas, que reconoce las palabras definidas en las alternativas y devuelve el valor como String es la siguiente:

```
Nexo:
    nexo=('y' | 'o' | 'entonces');
```

6.2.3.1 - Generar artefactos del lenguaje

Una vez definida la gramática es necesario correr el generador que deriva los diversos componentes del lenguaje. Para esto es necesario buscar el archivo llamado *GenerateLenguajeNaturalReducido.mwe2* y al ejecutarlo se activa el generador del lenguaje Xtext. Este generador crea el parser, el serializador y algún código de infraestructura adicional. Los mensajes de registro pueden verse a través de la consola.

6.2.3.2 - Ejecutar el plug-in generado

Si la generación de código fue exitosa podemos testear el plugin haciendo click derecho sobre el proyecto y eligiendo la opción *Run As-> Eclipse Application*. Al ejecutarla se abrirá un nuevo espacio de trabajo Eclipse en el que se puede generar un nuevo proyecto y dentro de este un nuevo archivo con la extensión correspondiente a la gramática generada (la extensión se define al momento de la creación del proyecto Xtext). Al abrir este nuevo archivo el editor de nuestro lenguaje debería estar funcionando proveyendo un asistente para completar código, resaltando sintaxis y mostrando errores de sintaxis.

6.2.4 - El generador

Xtext proporciona muchos componentes genéricos como parte de la infraestructura para generar lenguajes (intérpretes de gramáticas, meta-metamodelos, etc), aunque también se usa generación de código para crear algunos componentes extra. Estos componentes generados son entre otros el parser, el serializador, el modelo Ecore inferido (en caso que exista) y un par de clases base convenientes para la asistencia de contenido. El generador también aporta recursos de proyectos compartidos como los módulos plugin.xml, MANIFEST.MF, etc.

6.2.4.1 - Arquitectura General

Un generador en Xtext está compuesto de configuraciones de lenguajes y por cada una de ellas se debe proporcionar una URI para determinar cuál es la gramática y proveer la extensión asignada para los archivos del lenguaje. Entonces un lenguaje se configura con una lista de *IGeneratorFragments*.
El generador en sí está compuesto por fragmentos que generan diferentes componentes: parser, serializador, código EMF, etc. En la figura 6-7 puede verse la arquitectura general de un generador.

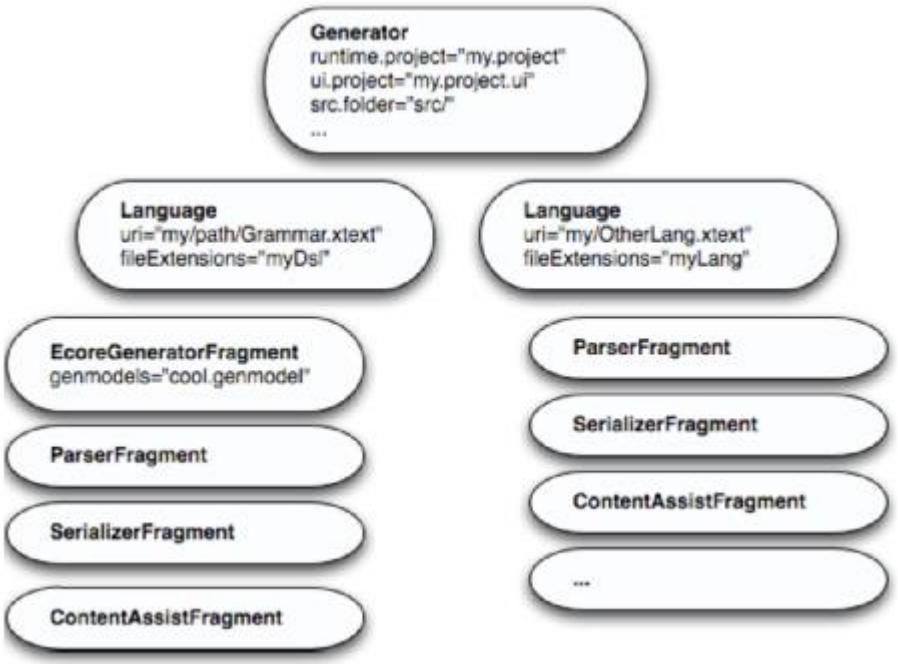


Figura 6-6. Arquitectura general

Fragmentos del generador. Cada fragmento obtiene la gramática del lenguaje como un modelo EMF. Un fragmento puede generar código en una de las ubicaciones configuradas y contribuir con varios artefactos compartidos. *IGeneratorFragment* es una interfaz, compatible con la clase abstracta *AbstractGeneratorFragment*, que por defecto delega a una plantilla Xpand que tenga el mismo nombre que la clase y delega algunas de las invocaciones a las definiciones de plantillas.
Existen un conjunto de fragmentos estándar del generador y en la figura 6-7 se muestran los más comunes.

Configuración. Se utiliza Modelling Workflow Engine 2 (MWE2) de *Eclipse Modeling Framework Technology* (EMFT) para instanciar, configurar y ejecutar la estructura de

componentes. MWE2 permite componer grafos de objetos de forma declarativa de una manera muy compacta. Mediante el uso de una sintaxis simple y concisa los usuarios pueden declarar instancias de objetos, valores de atributos y referencias. Una ventaja es que solo instancia las clases Java y la configuración se hace a través de métodos públicos (setter y adder). MWE2 puede utilizarse para instanciar modelos de objetos Java arbitrarios, sin ninguna dependencia o limitación a implementaciones específicas de MWE2.

Class	Generated Artifacts	Related Documentation
EcoreGeneratorFragment	EMF code for generated models	Model inference
XtextAntlrGeneratorFragment	ANTLR grammar, parser, lexer and related services	
GrammarAccessFragment	Access to the grammar	
ResourceFactoryFragment	EMF resource factory	Xtext Resource
ParseTreeConstructorFragment	Model-to-text serialization	Serialization
ImportNamespacesScopingFragment	Index-based scoping	Index-based namespace scoping
JavaValidatorFragment	Model validation	Model validation
FormatterFragment	Code formatter	Declarative formatter
LabelProviderFragment	Label provider	Label provider
OutlineNodeAdapterFactoryFragment	Outline view configuration	Outline
TransformerFragment	Outline view configuration	Outline
JavaBasedContentAssistFragment	Java-based content assist	Content assist
XtextAntlrUiGeneratorFragment	Content assist helper based on ANTLR	Content assist
SimpleProjectWizardFragment	New project wizard	Project wizard

Figura 6-7. Algunos de los fragmento generadores

6.2.5 - Serialización

La serialización es el proceso de transformar un modelo EMF en su correspondiente representación textual. Entonces podemos decir que la serialización complementa el parsing y el lexing. El proceso de serialización en Xtext sigue los siguientes pasos:

- Validación del modelo semántico. Es opcional y se encuentra activado por defecto, es realizado por el validador de sintaxis concreto. Puede deshabilitarse al pasar como parámetro las opciones llamadas *SaveOption* al objeto *Serializer*.
- Hacer coincidir los elementos del modelo con las reglas gramaticales y crear un flujo de tokens. De esto se encarga el constructor del árbol de parse.
- Asociar los comentarios con sus objetos semánticos.
- Asociar los nodos existentes en el modelo de nodos con tokens del flujo de tokens.
- Combinar espacios en blanco existentes y ajustes automáticos de línea en el flujo de tokens.
- Agregar espacios en blanco de ser necesario o sustituir todos los espacios en blanco mediante el uso de un formatter.

La serialización se invoca al llamar a *XtextResource.save*. Además, la clase *Serializer* proporciona soporte independiente del recurso para la serialización. La serialización no es invocada cuando el contenido de editores textuales del lenguaje se guarda a disco. Otra situación donde se utiliza la serialización es en la aplicación de arreglos rápidos que modifican la semántica.

6.2.5.1 - Parse Tree Constructor

En la mayoría de los casos no es necesario personalizarlo ya que se genera automáticamente de la gramática Xtext pero puede ser útil mirar su codificación para entender mejor sus mensajes de error y su rendimiento en tiempo de ejecución.
Para que la serialización pueda llevarse a cabo con éxito este constructor debe ser capaz de consumir todos los elementos del modelo de EMF a ser serializado. Consumir, cuando nos referimos a gramáticas, significa escribir el elemento en la representación textual del modelo. En caso de que un modelo no pueda ser serializado se lanza una excepción del tipo *XtextSerializationException*. Esto puede ocurrir por diferentes razones:

- Un elemento del modelo no puede ser consumido, las razones pueden ser:
  - El elemento no debe almacenarse en el modelo.
  - La gramática necesita una asignación que consuma el elemento del modelo.
  - Se puede indicar que este elemento del modelo no debe consumirse ya que se trata de un elemento no persistido.
- Una asignación en la gramática no tiene ningún elemento del modelo como valor. El constructor puede serializar valores predeterminados si esto es requerido por una restricción gramatical para poder serializar otro elemento del modelo. Posibles soluciones son:
  - Agregar un elemento del modelo.
- La asignación definida en la gramática debe ser opcional.
- El tipo de elemento del modelo difiere del tipo definido en la gramática. El tipo del elemento del modelo debe ser idéntico al tipo de retorno de la regla gramatical o el tipo de la acción. Los subtipos no están permitidos.
- Se produce un error en la conversión de valores. Es decir, el valor no se puede serializar por lo que se lanza una *ValueConveterException*.
- Un Enum Literal no está permitido en esta posición. Esto puede suceder si la regla enum referenciada sólo enumera un subconjunto de los literales de la enumeración real.

Para entender los mensajes de error y los problemas de rendimiento de este constructor se debe entender que el mismo implementa un algoritmo de backtracking. Esto significa que la gramática es utilizada para especificar la estructura de un árbol en el que un camino desde el nodo raíz a un nodo hoja es una serialización válida de un modelo específico. El objetivo de este constructor es encontrar este camino cumpliendo con la condición de que todos los elementos del modelo son consumidos a medida que se recorre este camino. La estrategia utilizada es seleccionar la rama que consumirá la mayor cantidad de elementos del modelo y si la rama conduce a un callejón sin salida (por ejemplo, un elemento del modelo que debe ser consumido no está definido en el modelo) el constructor regresa por el camino hasta que se pueda tomar una rama diferente. Este tipo de estrategia tiene dos consecuencias:

- Si se produce un error el constructor ha encontrado callejones sin salida sin llegar a ninguna hoja. Entonces, el mensaje de error enumera los caminos más largos que derivaron en un callejón sin salida, un fragmento de la serialización y el motivo por el cual no se puede continuar por ese camino. El desarrollador tiene que darse cuenta por sí mismo cuál es la razón del error.
- Por motivos de rendimiento es importante que el constructor seleccione el camino que consumirá más elementos al comenzar y que detecte errores en los caminos de forma temprana. Para lograr esto se debe evitar tener muchas reglas que devuelvan el mismo tipo y que sean invocadas desde dentro de la misma alternativa en la gramática.

6.3 - Plugin OCL

Eclipse OCL es la implementación del estándar OMG de OCL para modelos basados en EMF. El soporte para la definición de expresiones sobre modelos es brindado por el lenguaje conocido como *Essential OCL* que por sí solo es bastante limitado por lo que es extendido de diferentes maneras. El lenguaje *Complete OCL* permite definir en un documento separado de invariantes que complementan un metamodelo existente mientras que el lenguaje *OCLinEcore* se utiliza para embeber OCL en las anotaciones de un modelo Ecore. También se provee un lenguaje que permite la definición de bibliotecas de OCL estándar que se lo conoce como *OCLstdlib*.
En este momento el proyecto OCL se encuentra en un proceso de transición a una nueva infraestructura lo que hace que existan 2 proyectos OCL paralelos, el conocido como *Classic OCL* y el nuevo denominado *Unified o Pivot OCL*.

6.3.1 - Classic OCL

Fue inicialmente desarrollado como una utilidad para programadores Java para definir restricciones OCL sobre metamodelos Ecore, pero fue evolucionando hasta incluir soporte para modelos UML. Debido a los cambios requeridos para incluir UML el código Java resultante es complicado de entender.
El código definido intenta seguir la especificación estándar de OCL desde su versión original hasta la actual pero todavía hay ciertas áreas que son difíciles de implementar.
Existen un conjunto de plugins a través de los cuales se define está implementación de Classic OCL:

- ***org.eclipse.ocl:*** Contiene al núcleo donde se encuentran los servicios de parsing, evaluación y asistente de contenidos. Define una API para la configuración del ambiente y el modelo de sintaxis abstracta de OCL. Estas APIs son genéricas; es decir que no dependen de ningún metamodelo.
- ***org.eclipse.ocl.uml:*** Es la implementación del metamodelo UML que une al ambiente genérico con las APIs definidas para el lenguaje UML. Permite trabajar con restricciones OCL y consultas sobre modelos UML.
- ***org.eclipse.ocl.ecore:*** Corresponde a la implementación del entorno del metamodelo Ecore que se utiliza para conectar al ambiente genérico con las APIs definidas para Ecore, permitiendo así trabajar con restricciones OCL y consultas sobre modelos Ecore.

6.3.2 - Complete OCL

Por si mismo OCL no es muy útil; si no existe un modelo sobre el cual aplicar las restricciones, las mismas no tendrían sentido. OCL como mencionamos anteriormente provee 2 formas de asociar restricciones con un modelo, embebiendo las expresiones OCL dentro de un modelo para enriquecerlo (OCLinEcore) o definiendo estas restricciones en un documento separado (Complete OCL) que complementa al modelo.
Para la presente tesina se definió un modelo de ejemplo, Biblioteca, y sobre dicho modelo un conjunto de restricciones usando Complete OCL. En la figura 6-8 puede verse el documento de restricciones definido.

```
import '../model/Biblioteca.ecore'

package Biblioteca

  context Biblioteca
     inv NotEmpty: self.libros->notEmpty()
     inv exist: self.empleados->exists(e| e.rol = 'Bibliotecario/a')

  context Empleado
      inv Simple: self.edad >= 18

  context Libro
      inv Size: self.copias->size() >= 1
      inv Select_notEmpty: self.copias->select(c| c.estadoDeLaCopia = 'Bueno')->notEmpty()


  context Autor
    inv Compuesta: self.nombre <> '' and self.apellido <> '' and self.nacionalidad <> ''
    inv forAll: self.obras->forAll(o| o.fechaDeEdicion >= self.fechaDeLaPrimerPublicacion)


endpackage
```

Figura 6-8. Documento Complete OCL

La sentencia *import* no está definida por la especificación OMG OCL 2.4, lo que hace imposible proveer la funcionalidad de relacionar restricciones definidas independientemente con el modelo sobre el cual se las debe aplicar. Por lo tanto, Eclipse OCL provee una extensión que probablemente sea incluida en el estándar OCL en el futuro, que permite importar modelos mediante su URIs y asignarles un alias si se requiere. En el ejemplo de la figura anterior no se define un alias para el modelo importado pero en caso de requerir un alias la sentencia quedaría como:

```
import ecore : '../model/Biblioteca.ecore'
```

La sintaxis definida para un documento CompleteOCL define que la extensión por defecto es *.ocl.

6.3.3 - Metamodelo Unificado o Pivot

El metamodelo Unificado o Pivot es un prototipo para la resolución de una serie de problemas fundamentales con la especificación OCL 2.4. Cuando se utiliza el metamodelo Pivot para los metamodelos Ecore o UML, se crea una instancia del metamodelo Pivot sobre la marcha de forma de poder proporcionar una funcionalidad unificada combinada con OCL para las instancias del metamodelo Ecore y UML. Este metamodelo Pivot generado

desacopla el modelo de los detalles específicos de implementación del tipo de metamodelo utilizado.
Analizado desde la perspectiva de su especificación podemos decir que el metamodelo Pivot:

- Está alineado con UML.
- Soporta el modelado de la biblioteca estándar OCL.
- Permite la inclusión de definiciones Complete OCL adicionales.
- Soporte la representación XMI facilitando la interacción entre módulos.
- Soporta un *oclType* completamente reflectante.

También podemos analizarlo desde la perspectiva de Eclipse por lo que el metamodelo Pivot:

- Oculta las diferencias de Ecore con respecto a EMOF.
- Oculta diferencias de MDT/UML2 con respecto a UML.
- Permite que gran parte de la semántica sea definida en una sola biblioteca del modelo.
- Permite al usuario extender y remplazar las bibliotecas del modelo.
- Permite el cumplimiento exacto de la especificación OMG.

El código del metamodelo unificado es proporcionado por el plugin *org.eclipse.ocl.pivot* y se puede agregar soporte adicional para UML si se incluye el plugin *org.eclipse.ocl.pivot.uml*.

Capítulo 7: Traducción

En este capítulo se presentarán diferentes tipos de invariantes sobre el modelo de ejemplo definido, comenzando con los más simples y hasta los más complejos. Mostrando los invariantes, la oración que se espera generar como salida en Lenguaje Natural, realizando una revisión más detallada de la gramática creada para su representación y de la transformación realizada mediante la utilización de reglas en ATL.

7.1 - Modelo Ecore - Biblioteca

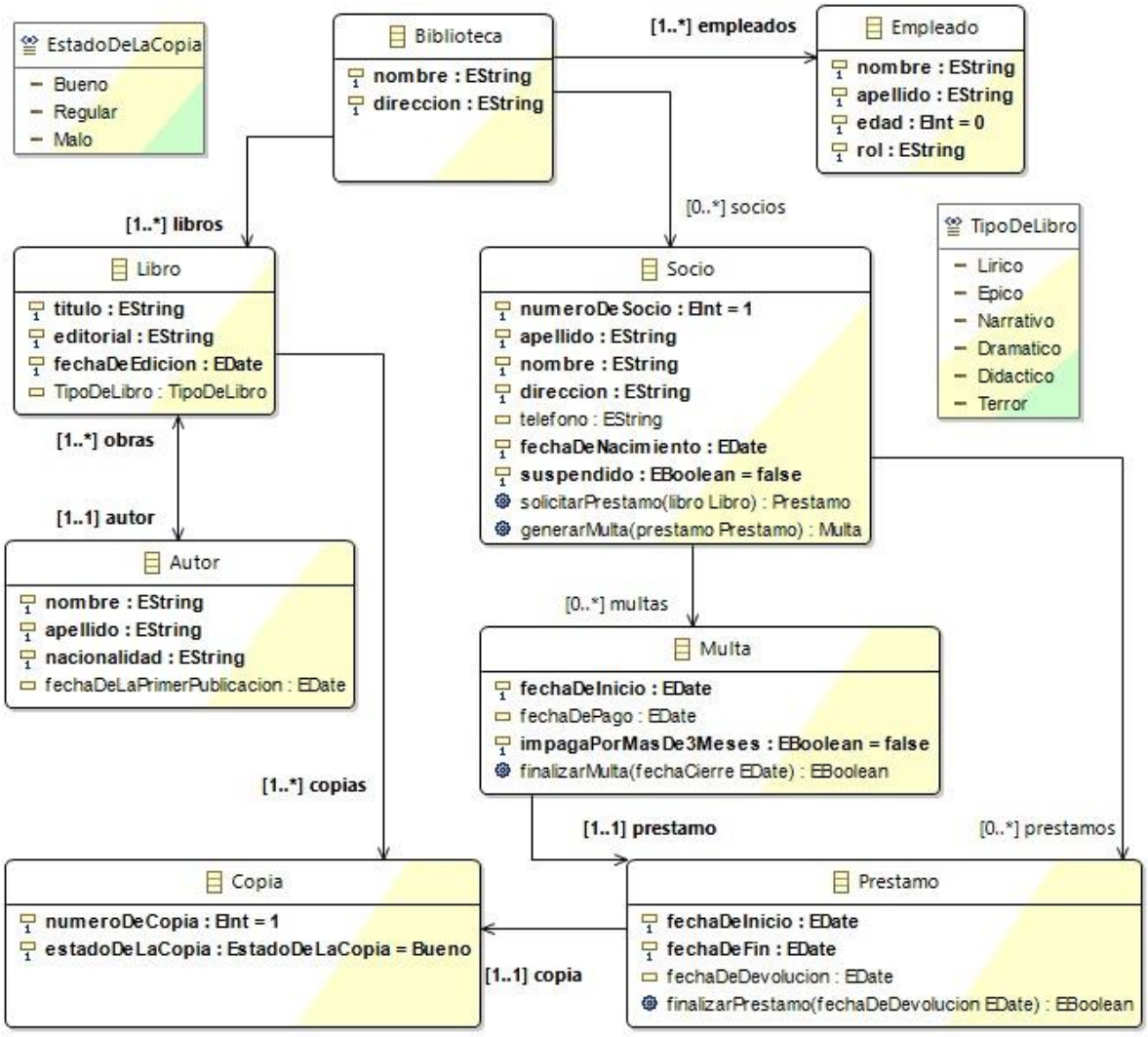


Figura 7-1. Metamodelo Biblioteca

7.2 - Reglas de traducción - ATL

Se definieron diferentes tipos de reglas para la transformación de objetos del modelo Pivot (o Complete OCL) al modelo de Lenguaje Natural Reducido que presentaremos en la siguiente sección. Estas reglas serán presentadas (sólo las principales reglas serán descriptas) cuando analicemos los diferentes invariantes y analicemos como estos son transformados para lograr su representación en Lenguaje Natural.

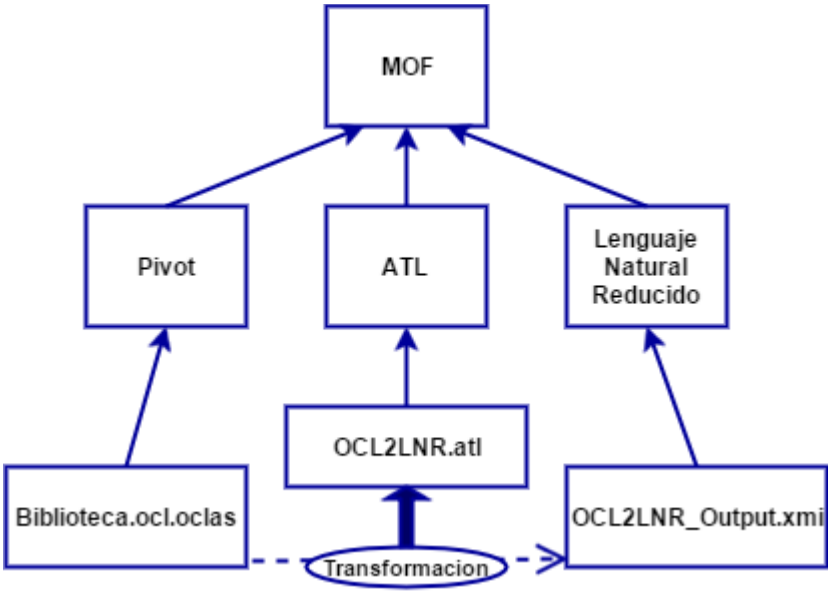
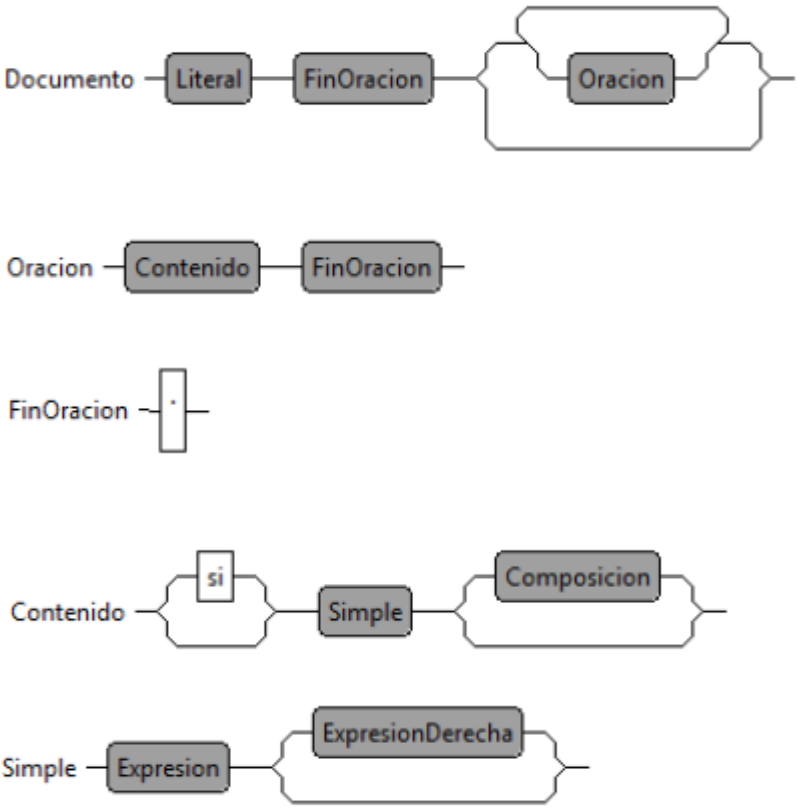


Figura 7-1. Visión específica de la transformación ATL

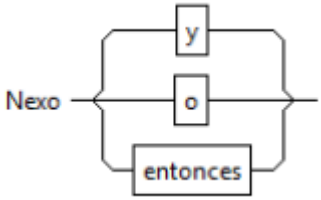
7.3 - Gramática - Lenguaje Natural Reducido

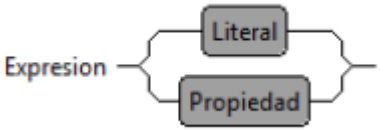
A continuación se presenta la versión gráfica de la gramática definida en Xtext para la versión reducida de Lenguaje Natural.
Los rectángulos sin sombreado de fondo representan símbolos terminales y los rectángulos con las esquinas redondeadas y sombreado gris representan los símbolos no terminales.

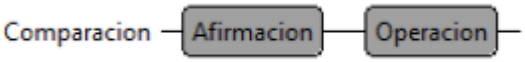


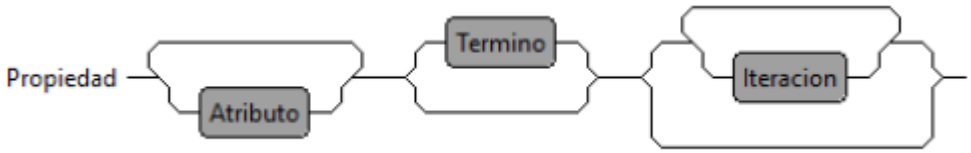

ExpresionDerecha
Comparacion
Expresion

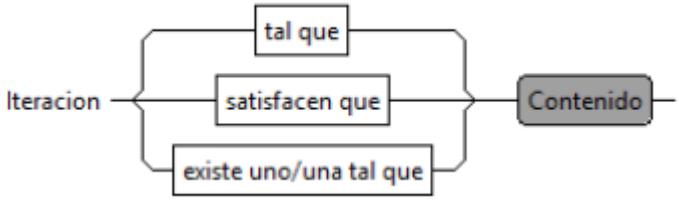

Composicion
Nexo
Contenido

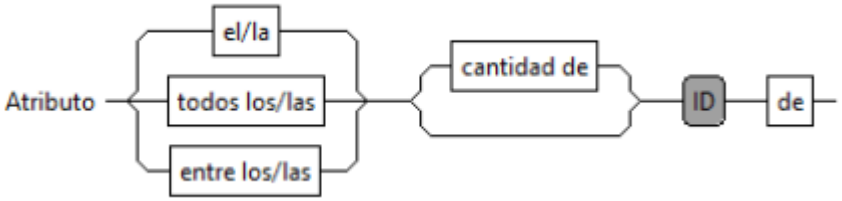

Nexo
y
o
entonces


Expresion
Literal
Propiedad

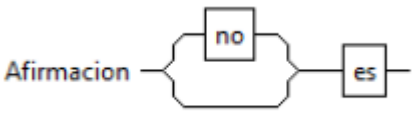

Comparacion
Afirmacion
Operacion

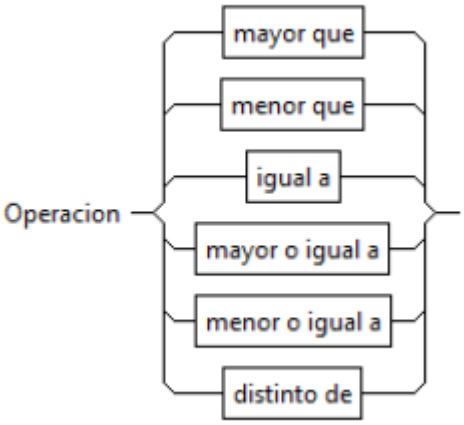

Propiedad
Atributo
Termino
Iteracion

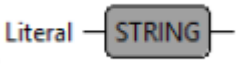

Iteracion
tal que
satisfacen que
existe uno/una tal que
Contenido


Atributo
el/la
todos los/las
entre los/las
cantidad de
ID
de


Termino
un/una
ID







7.4 - Invariante simple

Teniendo en cuenta el siguiente invariante que valida que la edad de un Empleado sea mayor o igual que 18 (que sea mayor de edad), podemos decir que este es el tipo más básico de invariante que puede definirse ya que realiza una validación sobre una propiedad simple de un objeto.

```
context Empleado
    inv Simple: self.edad >= 18
```

Cuando se analiza el invariante anterior para intentar traducirlo al lenguaje natural se encuentran diferentes formas de hacerlo. Siendo la primera opción la traducción directa o literal:

Contexto Empleado **inv**(ariante) Simple **yo mismo** edad mayor igual 18

La traducción anterior, si bien se encuentra en lenguaje natural, no sigue las formas sintácticas o semánticas como para ser interpretado correctamente. La siguiente opción representa el significado del invariante correctamente en un lenguaje natural simple que puede ser interpretado fácilmente por el usuario. Por su naturaleza ambigua el lenguaje natural permite representar este invariante de diferentes formas pero hemos optado solo por una de ellas y es la siguiente:

```
el/la edad de un/una Empleado es mayor o igual a "18".
```

Una vez definida la oración a utilizar para traducir este tipo de invariantes, se definieron los componentes abstractos con el objetivo de generar una gramática que permita su representación.

'el/la' <Identificador> **'de'** **'un/una'** <Identificador> **'es'** <Operacion> **'18'.**

La utilización de *'el/la'* y *'un/una'* se debe a que determinar género no es algo sencillo ya que requiere de un análisis profundo del sujeto de la oración. Por lo tanto, decidimos implementar una solución genérica que no requiere ningún procesamiento extra para la determinación del género. Entonces, cuando se traducen propiedades como en este caso siempre utilizamos *'el/la'*; y al definir el contexto en el que estamos aplicando la invariante, que sabemos siempre es único, utilizamos *'un/una'*. De esta manera, no tenemos que preocuparnos por determinar el género.
Podemos combinar los componentes abstractos definidos anteriormente para formar otros y lograr un nivel de abstracción más elevado de la siguiente manera:

<Atributo> <Término> <Comparación> <Literal>

Nuestra gramática posee niveles de abstracción aún más elevados que sirven para separar las representaciones de oraciones simples, compuestas e iteraciones. Estos niveles utilizan la representación de oraciones simples (con mínimas alteraciones) presentada en esta sección como parte de su definición y esto se verá más adelante cuando se presentan ejemplos específicos.
Como mencionamos anteriormente, utilizamos la sintaxis abstracta de complete OCL para representar los invariantes como entrada para la transformación en ATL. La sintaxis abstracta, representada en XMI, para el invariante presentado en esta sección es la siguiente:

```
<ownedClasses xmi:id="T.Biblioteca.Empleado" name="Empleado">
    <ownedInvariants xmi:id="ciBiblioteca.Empleado.Simple" name="Simple">
      <ownedSpecification xsi:type="pivot:ExpressionInOCL"
          type="pivot:PrimitiveType
              http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
          body="self.edad >= 18&#xA;&#x9;&#x9;&#xA;&#x9;">
        <ownedBody xsi:type="pivot:OperationCallExp" name=">="
            type="pivot:PrimitiveType
                http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
            referredOperation="http://www.eclipse.org/ocl/2015/
                        Library.oclas#o.ocl.OclComparable.%62%%61%..ocl.OclSelf">
          <ownedSource xsi:type="pivot:PropertyCallExp"
              type="pivot:DataType http://www.eclipse.org/emf/2002/Ecore.oclas#T.ecore.EInt"
              referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Empleado.edad">
            <ownedSource xsi:type="pivot:VariableExp"
                type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Empleado"
                referredVariable="#//@ownedPackages.0/@ownedClasses.2/@ownedInvariants.0/
                            @ownedSpecification/@ownedContext"/>
          </ownedSource>
          <ownedArguments xsi:type="pivot:IntegerLiteralExp"
              type="pivot:PrimitiveType
                  http://www.eclipse.org/ocl/2015/Library.oclas#Integer"
              integerSymbol="18"/>
        </ownedBody>
        <ownedContext name="self"
            type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Empleado"/>
      </ownedSpecification>
    </ownedInvariants>
</ownedClasses>
```

La transformación ATL procesa el XMI que contiene el modelo a transformar y aplica las reglas definidas para generar los objetos que correspondan en el modelo de salida.
En este caso en particular, si observamos el ownedBody en la representación XMI de este invariante podemos ver que está representado como una OperationCallExp por lo que implementamos la transformación desde OperationCallExp (Complete OCL) a Contenido Simple (gramática de lenguaje natural reducido, LNR).
La regla que contiene todo el procesamiento se presenta a continuación aunque existen otras reglas anteriores en la cadena pero que no son explicadas ya que solo son utilizadas para iterar sobre los elementos del modelo de entrada y crear los objetos contenedores, por ejemplo la regla Constraint2Oracion genera un objeto Oración y su Contenido por cada invariante.

```
1  lazy rule OperationCallExp2Simple {
2    from operation: OCL!OperationCallExp
3    to   simple: LNR!Simple (
4      expresion_izq <- if (operation.name = 'not') then
5            thisModule.OCLExpression2Expresion(operation.ownedSource.ownedSource)
6          else
7            if (thisModule.removePrefix(operation.referredOperation.toString())
8                =  'isEmpty')
9            or (thisModule.removePrefix(operation.referredOperation.toString())
10               = 'notEmpty') then
11             thisModule.OCLExpression2Expresion(operation)
12           else
13             thisModule.OCLExpression2Expresion(operation.ownedSource)
14           endif
15         endif,
16       expresion_der <- expDer
17     ),
18     expDer: LNR!ExpresionDerecha (
19       comparacion <- comparacion,
20       expresion <- if (operation.name = 'not') then
21           thisModule.OCLExpression2Expresion(
22             operation.ownedSource.ownedArguments.first())
23         else
24           if (thisModule.removePrefix(operation.referredOperation.toString())
25               =  'isEmpty')
26           or (thisModule.removePrefix(operation.referredOperation.toString())
27               = 'notEmpty') then
28             thisModule.String2LiteralExp('0')
29           else
30             thisModule.OCLExpression2Expresion(operation.ownedArguments.first())
31           endif
32         endif
33     ),
34     comparacion: LNR!Comparacion (
35       afirmacion <- afirmacion,
36       operacion <- operacion
37     ),
38     afirmacion: LNR!Afirmacion (
39       negacion <- if (operation.name = 'not') then
40           'no'
41         else
42           OclUndefined
43         endif,
```

```
44        afirmacion <- 'es'
45      ),
46      operacion: LNR!Operacion (
47        operacion <- if (operation.name = 'not') then
48            thisModule.operationName(operation.ownedSource.name)
49          else
50            if (operation.name.oclIsUndefined()) then
51              thisModule.operationName(thisModule.removePrefix(
52                operation.referredOperation.toString()))
53            else
54              thisModule.operationName(operation.name)
55            endif
56          endif
57      )
58  }
```

Este código se basa en la estructura definida para el invariante en el XMI. En la línea 4 se define la generación de la expresión a izquierda para lo que se invoca a la regla OCLExpression2Expresion en la línea 13 con la rama izquierda del AST para este invariante que es el OwnedSource. OCLExpression2Expresion es la regla que se encarga de traducir el subárbol izquierdo (que contiene la parte de la oración a la izquierda de la operación, *"el/la edad de un/una Empleado"*) basándose en el tipo de OCLExpression al que corresponde OwnedSource; estos tipos son: PropertyCallExp (propiedad de un objeto), PrimitiveLiteralExpression (literal), OperationCallExp (otra operación, comparación o colecciones).
La expresión a derecha en línea 18, contiene la comparación (línea 19) y la expresión a derecha en sí misma (línea 20). La comparación (línea 34) hace explícito que nos referimos a un invariante (condición que es siempre verdadera en el sistema) mediante la adición del literal *"es"* (línea 44) y la operación en línea 46 representada en lenguaje natural que se obtiene como resultado de llamar al helper operationName en línea 54 con *">="* como parámetro. La expresión (línea 20) que genera la expresión a derecha del operador invoca nuevamente la regla que mencionamos anteriormente para la creación de la expresión a izquierda, OCLExpression2Expresion (línea 30) pero esta vez con el subárbol derecho del AST representado en OCL por ownedArguments.first() que en este caso particular es una PrimitiveLiteralExpression que solo retornara *"18"*.

Una vez que estas reglas finalizan su ejecución, ATL genera un XMI de salida que se corresponde con el formato definido en el metamodelo/gramática de Lenguaje Natural Reducido, y que representa la conversión del invariante original a una oración en lenguaje natural.

```
<oraciones>
    <contenido>
      <simple>
        <expresion_izq>
          <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
            <atributo determinante="el/la" nombre="edad" enlace="de"/>
            <termino indeterminante="un/una" contexto="Empleado"/>
          </expresion>
        </expresion_izq>
        <expresion_der>
          <comparacion>
            <afirmacion afirmacion="es"/>
            <operacion operacion="mayor o igual a"/>
          </comparacion>
          <expresion>
            <expresion xsi:type="lenguajeNaturalReducido:Literal" literal="18"/>
          </expresion>
        </expresion_der>
      </simple>
    </contenido>
</oraciones>
```

7.5 - Invariante compuesta

Una invariante compuesta es aquella que incluye operadores lógicos para conectar 2 o más invariantes simples. Analizaremos el ejemplo que se presenta a continuación:

```
context Socio
    inv Compuesta:
         self.direccion <> '' and self.apellido <> '' and self.nombre <> ''
```

La misma definición puede aplicarse en lenguaje natural para una oración compuesta; es una composición de oraciones simples unidas por un conector. Por lo tanto, la traducción de este invariante es:

```
el/la nombre de un/una Socio es distinto de ""
y
el/la apellido de un/una Socio es distinto de ""
y
el/la direccion de un/una Socio es distinto de "".
```

La representación de este invariante en Complete OCL usando el metamodelo Pivot es la siguiente:

```
<ownedClasses xmi:id="T.Biblioteca.Socio" name="Socio">
    <ownedInvariants xmi:id="ciBiblioteca.Socio.Compuesta" name="Compuesta">
      <ownedSpecification xsi:type="pivot:ExpressionInOCL" isRequired="false"
          type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
              /Library.oclas#Boolean"
          body="self.direccion &lt;> '' and self.apellido &lt;> ''
              and self.nombre &lt;> ''&#xA;&#x9;&#x9;&#xA;">
        <ownedBody xsi:type="pivot:OperationCallExp" name="and" isRequired="false"
            type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
              /Library.oclas#Boolean"
            referredOperation="http://www.eclipse.org/ocl/2015
                    /Library.oclas#o.ocl.Boolean.and..ocl.Boolean">
          <ownedSource xsi:type="pivot:OperationCallExp" name="and" isRequired="false"
              type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                /Library.oclas#Boolean"
              referredOperation="http://www.eclipse.org/ocl/2015
                      /Library.oclas#o.ocl.Boolean.and..ocl.Boolean">
            <ownedSource xsi:type="pivot:OperationCallExp" name="&lt;>"
                type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                  /Library.oclas#Boolean"
                referredOperation="http://www.eclipse.org/ocl/2015
                        /Library.oclas#o.ocl.String.%60%62%..ocl.OclSelf">
              <ownedSource xsi:type="pivot:PropertyCallExp"
                  type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                    /Library.oclas#String"
                  referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Socio.direccion">
                <ownedSource xsi:type="pivot:VariableExp"
                    type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Socio"
                    referredVariable="#//@ownedPackages.0/@ownedClasses.5
                        /@ownedInvariants.0/@ownedSpecification/@ownedContext"/>
              </ownedSource>
              <ownedArguments xsi:type="pivot:StringLiteralExp"
                  type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                    /Library.oclas#String"
                  stringSymbol=""/>
            </ownedSource>
            <ownedArguments xsi:type="pivot:OperationCallExp" name="&lt;>"
                type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                  /Library.oclas#Boolean"
                referredOperation="http://www.eclipse.org/ocl/2015
                        /Library.oclas#o.ocl.String.%60%62%..ocl.OclSelf">
              <ownedSource xsi:type="pivot:PropertyCallExp"
                  type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                    /Library.oclas#String"
                  referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Socio.apellido">
                <ownedSource xsi:type="pivot:VariableExp"
                    type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Socio"
                    referredVariable="#//@ownedPackages.0/@ownedClasses.5
                        /@ownedInvariants.0/@ownedSpecification/@ownedContext"/>
              </ownedSource>
              <ownedArguments xsi:type="pivot:StringLiteralExp"
                  type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                    /Library.oclas#String"
                  stringSymbol=""/>
            </ownedArguments>
          </ownedSource>
```

```
        <ownedArguments xsi:type="pivot:OperationCallExp" name="&lt;>"
            type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                /Library.oclas#Boolean"
            referredOperation="http://www.eclipse.org/ocl/2015
                                /Library.oclas#o.ocl.String.%60%%62%..ocl.OclSelf">
          <ownedSource xsi:type="pivot:PropertyCallExp"
              type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                  /Library.oclas#String"
              referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Socio.nombre">
            <ownedSource xsi:type="pivot:VariableExp"
                type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Socio"
                referredVariable="#//@ownedPackages.0/@ownedClasses.5
                        /@ownedInvariants.0/@ownedSpecification/@ownedContext"/>
          </ownedSource>
          <ownedArguments xsi:type="pivot:StringLiteralExp"
              type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015
                  /Library.oclas#String"
              stringSymbol=""/>
        </ownedArguments>
      </ownedBody>
      <ownedContext name="self"
          type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Socio"/>
    </ownedSpecification>
  </ownedInvariants>
</ownedClasses>
```

La oración anterior puede ser descrita en un primer intento como <*simple*> 'y' <*simple*> 'y' <*simple*> pero esta definición en la gramática no permitiría una oración compuesta por n oraciones simples. Entonces definimos una gramática que no solo permite la definición de composiciones múltiples mediante el uso de la recursión sino que permite el uso indistinto de los conectores 'y' u 'o'.
La gramática definida en conjunto con la siguiente regla en ATL hace posible la traducción de invariantes compuestas interconectadas por los operadores lógicos, *and* y *or*.

```
1  rule OperationCallExp2ContenidoAndOr {
2    from
3      operation: OCL!OperationCallExp (
4        operation.name = 'and' or operation.name = 'or'
5        or operation.name = 'implies'
6      )
7    to
8      contenido: LNR!Contenido (
9        condicional <- if (operation.name = 'implies') then
10           'si'
11         else
12           OclUndefined
13         endif,
14       simple <- if (operation.name = 'implies') then
15           thisModule.OperationCallExp2Simple(operation.ownedSource)
16         else
17           if (operation.ownedSource.oclIsTypeOf(OCL!IteratorExp)) then
18             thisModule.Iterator2Simple(operation.ownedSource)
19           else
20             thisModule.OperationCallExp2Simple(operation.ownedArguments.first())
21           endif
22         endif,
```

```
23        composicion <- composicion
24      ),
25      composicion: LNR!Composicion (
26        nexo <- nexo,
27        contenido <- if (operation.name = 'implies') then
28            thisModule.resolveTemp(operation.ownedArguments.first(), 'contenido')
29          else
30            if (operation.ownedArguments.first().oclIsTypeOf(OCL!IteratorExp))
31                then
32              thisModule.IteratorExp2Contenido(operation.ownedArguments.first())
33            else
34              thisModule.resolveTemp(operation.ownedSource, 'contenido')
35            endif
36          endif
37      ),
38      nexo: LNR!Nexo (
39        nexo <- thisModule.operationName(operation.name)
40      )
41  }
```

Esta regla genera una operación simple y lo que denominamos composiciones, *nexo* más *contenido*. Estas composiciones son las que hacen posible la recursión para poder traducir n operaciones lógicas.
En este caso en particular, se crea un objeto *contenido*, *simple* con *composición*. La sentencia *simple* será la que represente *"el/la apellido de un/una Socio es distinto de ' '"* mientras que la *composición* tendrá el *nexo* "y" y otro objeto *contenido simple* para representar la próxima *oración simple* y *composición*, esto se repite 2 veces para poder transformar el invariante completo al metamodelo de LNR. A continuación se muestra el resultado de está transformación.

```
<oraciones>
  <contenido>
    <simple>
      <expresion_izq>
        <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
          <atributo determinante="el/la" nombre="nombre" enlace="de"/>
          <termino indeterminante="un/una" contexto="Socio"/>
        </expresion>
      </expresion_izq>
      <expresion_der>
        <comparacion>
          <afirmacion afirmacion="es"/>
          <operacion operacion="distinto de"/>
        </comparacion>
        <expresion>
          <expresion xsi:type="lenguajeNaturalReducido:Literal" literal=""/>
        </expresion>
      </expresion_der>
    </simple>
    <composicion>
      <nexo nexo="y"/>
      <contenido>
```

```
        <simple>
          <expresion_izq>
            <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
              <atributo determinante="el/la" nombre="apellido" enlace="de"/>
              <termino indeterminante="un/una" contexto="Socio"/>
            </expresion>
          </expresion_izq>
          <expresion_der>
            <comparacion>
              <afirmacion afirmacion="es"/>
              <operacion operacion="distinto de"/>
            </comparacion>
            <expresion>
              <expresion xsi:type="lenguajeNaturalReducido:Literal" literal=""/>
            </expresion>
          </expresion_der>
        </simple>
        <composicion>
          <nexo nexo="y"/>
          <contenido>
            <simple>
              <expresion_izq>
                <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
                  <atributo determinante="el/la" nombre="direccion" enlace="de"/>
                  <termino indeterminante="un/una" contexto="Socio"/>
                </expresion>
              </expresion_izq>
              <expresion_der>
                <comparacion>
                  <afirmacion afirmacion="es"/>
                  <operacion operacion="distinto de"/>
                </comparacion>
                <expresion>
                  <expresion xsi:type="lenguajeNaturalReducido:Literal" literal=""/>
                </expresion>
              </expresion_der>
            </simple>
          </contenido>
        </composicion>
      </contenido>
    </composicion>
  </contenido>
</oraciones>
```

Cabe mencionar que aunque tanto el estándar OCL, como nuestra herramienta permiten definir y traducir invariantes compuestos, es altamente recomendado escribir invariantes en su forma más simple. Por lo tanto, se recomienda separar las condiciones unidas por el operador lógico “and” en múltiples invariantes.

7.6 - Invariante operación de colección (size, isEmpty, notEmpty)

La operación size siempre que es utilizada en un invariante es definida en conjunto con una comparación ya que un invariante debe retornar una operación booleana, y size() retorna el número de elementos contenidos en una colección.

Si tomamos como ejemplo el siguiente invariante:

```
context Libro
    inv Size: self.copias->size() >= 1
```

El formato de traducción elegido en lenguaje natural se definió de forma tal de reutilizar la definición de oración simple. Por lo que el invariante anterior es traducido como:

```
el/la cantidad de copias de un/una Libro es mayor o igual a "1".
```

Notar que la representación anterior es muy similar a la de una oración *simple*, por lo que con un cambio en el componente *atributo* en la gramática para incluir *"cantidad de"* podemos representar oraciones derivadas de una operación size().

```
<ownedInvariants xmi:id="ciBiblioteca.Libro.Size" name="Size">
  <ownedSpecification xsi:type="pivot:ExpressionInOCL"
      type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
      body="self.copias->size() >= 1 &#xA;&#x9;   &#x9;">
    <ownedBody xsi:type="pivot:OperationCallExp" name=">="
        type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
        referredOperation="http://www.eclipse.org/ocl/2015/
                           Library.oclas#o.ocl.OclComparable.%62%%61%..ocl.OclSelf">
      <ownedSource xsi:type="pivot:OperationCallExp"
          type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Integer"
          referredOperation="http://www.eclipse.org/ocl/2015/Library.oclas#o.ocl.Collection.size">
        <ownedSource xsi:type="pivot:PropertyCallExp"
            type="#//@ownedPackages.1/@ownedClasses.2"
            referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Libro.copias">
          <ownedSource xsi:type="pivot:VariableExp"
              type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Libro"
              referredVariable="#//@ownedPackages.0/@ownedClasses.3/@ownedInvariants.0/
                                @ownedSpecification/@ownedContext"/>
        </ownedSource>
      </ownedSource>
      <ownedArguments xsi:type="pivot:IntegerLiteralExp"
          type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Integer"
          integerSymbol="1"/>
    </ownedBody>
    <ownedContext name="self" type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Libro"/>
  </ownedSpecification>
</ownedInvariants>
```

Como mencionamos anteriormente representamos las operaciones size como una oración simple con un prefijo agregado a la traducción de la propiedad. Por lo tanto, se ejecuta la misma regla que para la traducción de invariantes simples. La única diferencia es que al invocar a la regla OCL2Expresion y convertir la propiedad si se trata de una operación size se agrega el prefijo *'cantidad de'*.

El modelo de salida generado basado en el metamodelo de LNR es el siguiente:

```
<oraciones>
  <contenido>
    <simple>
      <expresion_izq>
        <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
          <atributo determinante="el/la" cuantitativo="cantidad de" nombre="copias" enlace="de"/>
          <termino indeterminante="un/una" contexto="Libro"/>
        </expresion>
      </expresion_izq>
      <expresion_der>
        <comparacion>
          <afirmacion afirmacion="es"/>
          <operacion operacion="mayor o igual a"/>
        </comparacion>
        <expresion>
          <expresion xsi:type="lenguajeNaturalReducido:Literal" literal="1"/>
        </expresion>
      </expresion_der>
    </simple>
  </contenido>
</oraciones>
```

La operación de colecciones Empty posee dos variantes, isEmpty y notEmpty, que retornan un valor booleano dependiendo si la colección está vacía o no.
De nuestro ejemplo podemos utilizar el siguiente invariante:

```
context Biblioteca
    inv NotEmpty: self.libros->notEmpty()
```

Este invariante puede ser traducido como *"la colección de libros de una biblioteca no está vacía"* pero utilizar el término *"colección"* puede resultar confuso para personas sin conocimientos técnicos de lógica u orientación a objetos. Decir que una colección no está vacía puede también ser expresado como "la cantidad de elementos que contiene una colección es mayor que cero". Por lo tanto, podemos decir que el invariante anterior es equivalente al invariante presentado anteriormente utilizando size, pudiendo ser traducido como:

```
el/la cantidad de copias de un/una Libro es mayor que "0".
```

En caso de utilizar la operación isEmpty la operación de comparación utilizada es *"es igual a"*.
La representación de este invariante en el XMI generado es la siguiente:

```
<ownedInvariants xmi:id="ciBiblioteca.Biblioteca.NotEmpty" name="NotEmpty">
  <ownedSpecification xsi:type="pivot:ExpressionInOCL"
      type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
      body="self.libros->notEmpty()&#xA;&#x9; &#x9;">
    <ownedBody xsi:type="pivot:OperationCallExp"
        type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
        referredOperation="http://www.eclipse.org/ocl/2015/
                           Library.oclas#o.ocl.Collection.notEmpty">
      <ownedSource xsi:type="pivot:PropertyCallExp"
          type="#//@ownedPackages.1/@ownedClasses.0"
          referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Biblioteca.libros">
        <ownedSource xsi:type="pivot:VariableExp"
            type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Biblioteca"
            referredVariable="#//@ownedPackages.0/@ownedClasses.1/@ownedInvariants.0/
                              @ownedSpecification/@ownedContext"/>
      </ownedSource>
    </ownedBody>
    <ownedContext name="self"
        type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Biblioteca"/>
  </ownedSpecification>
</ownedInvariants>
```

La gramática no requiere ningún cambio, ya que es la traducción en sí la que al detectar una operación isEmpty o notEmpty la convertirá siguiendo la estructura gramatical de una operación size. La diferencia es que tanto la comparación como la expresión de la derecha son creadas forzosamente ya que el invariante de entrada no los posee; generando un formato de salida exactamente igual al de la operación size.

```
<oraciones>
  <contenido>
    <simple>
      <expresion_izq>
        <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
          <atributo determinante="el/la" cuantitativo="cantidad de"
                    nombre="libros" enlace="de"/>
          <termino indeterminante="un/una" contexto="Biblioteca"/>
        </expresion>
      </expresion_izq>
      <expresion_der>
        <comparacion>
          <afirmacion afirmacion="es"/>
          <operacion operacion="mayor que"/>
        </comparacion>
        <expresion>
          <expresion xsi:type="lenguajeNaturalReducido:Literal"
                     literal="0"/>
        </expresion>
      </expresion_der>
    </simple>
  </contenido>
</oraciones>
```

7.7 - Invariante con iterador

Los iteradores son operaciones sobre colecciones donde las condiciones definidas se aplican en cada iteración o a cada elemento de la colección, y dependiendo de que iterador estemos utilizando obtenemos los ítems correspondientes en una nueva colección de salida o un booleano indicando si alguno o todos los elementos de la colección cumplen con la condición definida.
OCL posee varios iteradores, como mencionamos anteriormente estos pueden clasificarse en 2 categorías; aquellos que retornan una nueva colección de salida y los que retornan un valor booleano. Presentamos algunos ejemplos en las siguientes dos secciones.

7.7.1 - Valor booleano como salida

Exists (condiciones): Retorna verdadero si al menos uno de los elementos que forman parte de la colección resulta verdadero al evaluar las condiciones. Analicemos el siguiente invariante de nuestro ejemplo:

```
context Biblioteca
    inv exist: self.empleados->exists(e| e.rol = 'Bibliotecario/a')
```

El resultado que esperamos obtener al transformar este invariante a su representación en Lenguaje Natural Reducido es:

```
entre los/las empleados de un/una Biblioteca existe uno/una tal que el/la
rol de un/una Empleado es igual a "Bibliotecario/a".
```

Si observamos en detalle la representación en XML del invariante siguiendo el metamodelo Pivot OCL presentada a continuación, podemos observar que la expresión en OCL dejó de ser una OperationCallExp, como lo era para los casos presentados anteriormente, para pasar a ser un IteratorExp.

```
<ownedInvariants xmi:id="ciBiblioteca.Biblioteca.exist" name="exist">
  <ownedSpecification xsi:type="pivot:ExpressionInOCL" isRequired="false"
      type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
      body="self.empleados->exists(e| e.rol = 'Bibliotecario/a')    ">
    <ownedBody xsi:type="pivot:IteratorExp" isRequired="false"
        type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
        referredIteration="http://www.eclipse.org/ocl/2015/
                           Library.oclas#i.ocl.Collection.exists..T">
      <ownedSource xsi:type="pivot:PropertyCallExp"
          type="#//@ownedPackages.1/@ownedClasses.1"
          referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Biblioteca.empleados">
        <ownedSource xsi:type="pivot:VariableExp"
            type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Biblioteca"
            referredVariable="#//@ownedPackages.0/@ownedClasses.1/@ownedInvariants.1/
                              @ownedSpecification/@ownedContext"/>
      </ownedSource>
      <ownedBody xsi:type="pivot:OperationCallExp" name="="
          type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
          referredOperation="http://www.eclipse.org/ocl/2015/
                             Library.oclas#o.ocl.String.%61%..ocl.OclSelf">
        <ownedSource xsi:type="pivot:PropertyCallExp"
            type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#String"
            referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Empleado.rol">
          <ownedSource xsi:type="pivot:VariableExp"
              type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Empleado"
              referredVariable="#//@ownedPackages.0/@ownedClasses.1/@ownedInvariants.1/
                                @ownedSpecification/@ownedBody/@ownedIterators.0"/>
        </ownedSource>
        <ownedArguments xsi:type="pivot:StringLiteralExp"
            type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#String"
            stringSymbol="Bibliotecario/a"/>
      </ownedBody>
      <ownedIterators name="e"
          type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Empleado"
          representedParameter="http://www.eclipse.org/ocl/2015/
                                Library.oclas#i0i.ocl.Collection.exists..T"/>
    </ownedBody>
    <ownedContext name="self"
        type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Biblioteca"/>
  </ownedSpecification>
</ownedInvariants>
```

El código de transformación en ATL cuenta con una regla principal que es la encargada de procesar todos los invariantes definidos en un documento de Complete OCL y que determina que en caso de estar procesando un IteratorExp se debe invocar la siguiente regla (en caso contrario se ejecutará automáticamente una de las reglas presentadas anteriormente OperationCallExp2ContenidoSimple o OperationCallExp2ContenidoAndOr):

```
1  lazy rule IteratorExp2Contenido {
2    from
3      iterador: OCL!IteratorExp
4    to
5      contenido: LNR!Contenido (
6        simple <- thisModule.Iterator2Simple(iterador),
7        composicion <- OclUndefined
8      )
9  }
10
```

```
lazy rule Iterator2Simple {
  from
    iterador: OCL!IteratorExp
  to
    simple: LNR!Simple (
      expresion_izq <- exp_izq,
      expresion_der <- OclUndefined
    ),
    exp_izq: LNR!Expresion (
      expresion <- thisModule.IteratorExp2Propiedad(iterador)
    )
}

lazy rule IteratorExp2Propiedad {
  from
    iterator: OCL!IteratorExp
  using {
    children: Sequence(OCL!PropertyCallExp) = iterator.ownedSource.
        getPropertyCallExpCollection();
    iterators: Sequence(OCL!IteratorExp) = iterator.getIteratorExpCollection();
  }
  to
    propiedad: LNR!Propiedad (
      atributo <- children -> collect(e | thisModule.
          PropertyCallExp2Atributo(e)),
      termino <- termino,
      iteracion <- iterators -> collect(e | thisModule.IteratorExp2Iteracion(e))
    ),
    termino: LNR!Termino (
      indeterminante <- 'un/una',
      contexto <- if (iterator.oclIsUndefined()) then
          OclUndefined
        else
          iterator.getExpressionInOCL().owningConstraint.context.name
        endif
    )
  do {

    if (iterators.first().referredIteration.name = 'select' or iterators.
    first().referredIteration.name = 'exists')
    {
      if(iterators -> exists(it | it.referredIteration.name = 'forAll'))
      {
        propiedad.atributo.first().determinante <- 'todos los/las';
      }
      else {
          propiedad.atributo.first().determinante <- 'entre los/las';
      }
    }
    else {
        propiedad.atributo.first().determinante <- 'todos los/las';
      }
    }
}

lazy rule IteratorExp2Iteracion {
  from
    iterator: OCL!IteratorExp
```

```
69   to
70     iteracion: LNR!Iteracion (
71       condicion <- if (iterator.referredIteration.name = 'forAll') then
72           'satisfacen que'
73         else
74           if (iterator.referredIteration.name = 'exists') then
75             'existe uno/una tal que'
76           else
77             'tal que'
78           endif
79         endif,
80       contenido <- iterator.ownedBody
81     )
```

Cuando escribimos invariantes que contienen iteraciones se pueden presentar diferentes casos:

- un único iterador que retorna un resultado booleano,
- un iterador que toma como entrada la salida del iterador previo y
- un iterador que no retorna un resultado booleano requiere ir acompañado de una comparación.

El invariante presentado en esta sección posee un único iterador que retorna un resultado booleano por lo que no es necesaria una operación de comparación. Al observar la regla IteratorExp2Contenido, se ve que un iterador es representado como una oración simple por lo que no se define composición (Línea 7) y que solo posee una expresión a izquierda (Líneas 16 y 19) ya que como mencionamos anteriormente este iterador retorna un resultado booleano por lo que la expresión derecha no es necesaria (Línea 17).
Cuando procesamos iteradores en un invariante que contiene más de uno, la recursión sucede al nivel donde para ejemplos anteriores generamos “el/la [cantidad de] atributo de un/una <contexto>” a los que denominamos OperationCallExp2Propiedad y PropertyCallExp2Propiedad ya que es necesario en este nivel introducir las condiciones que deben ser cumplidas. La regla utilizada para la transformación de la expresión izquierda es IteratorExp2Propiedad (línea 24) que dependiendo de que iterador estamos procesando actualmente o de si el iterador es aplicado a la salida de otro iterador determina el comienzo de la oración. En este caso particular, el inicio de la oración *“el/la”* es reemplazado por *“entre los/las”* y se invoca la regla IteratorExp2Iteracion para poder determinar en base al iterador como se presentan las condiciones, ya que para forAll será *“satisfacen que”*, para exists será *“existe al menos uno/una tal que”* y *“tal que”* para todos los demás.

Después de transformar el invariante usando las reglas descritas anteriormente obtenemos el siguiente modelo de Lenguaje Natural Reducido:

```
<oraciones>
  <contenido>
    <simple>
      <expresion_izq>
        <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
          <atributo determinante="entre los/las" nombre="empleados" enlace="de"/>
          <termino indeterminante="un/una" contexto="Biblioteca"/>
          <iteracion condicion="existe uno/una tal que">
            <contenido>
              <simple>
                <expresion_izq>
                  <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
                    <atributo determinante="el/la" nombre="rol" enlace="de"/>
                    <termino indeterminante="un/una" contexto="Empleado"/>
                  </expresion>
                </expresion_izq>
                <expresion_der>
                  <comparacion>
                    <afirmacion afirmacion="es"/>
                    <operacion operacion="igual a"/>
                  </comparacion>
                  <expresion>
                    <expresion xsi:type="lenguajeNaturalReducido:Literal" literal="Bibliotecario/a"/>
                  </expresion>
                </expresion_der>
              </simple>
            </contenido>
          </iteracion>
        </expresion>
      </expresion_izq>
    </simple>
  </contenido>
</oraciones>
```

forAll (condición/es): Retorna verdadero si todos los elementos de la colección retornan verdadero al evaluar las condiciones definidas.

```
context Autor
    inv forAll: self.obras->
                forAll(o| o.fechaDeEdicion >= self.fechaDeLaPrimerPublicacion)
```

La traducción en Lenguaje Natural Reducido de este invariante es la siguiente:

```
todos los/las obras de un/una Autor satisfacen que el/la fechaDeEdicion
de un/una Libro es mayor que el/la fechaDeLaPrimeraPublicacion de un/una Autor.
```

La representación del invariante en la sintaxis abstracta de complete OCL es la siguiente:

```
<ownedInvariants xmi:id="ciBiblioteca.Autor.forAll" name="forAll">
<ownedSpecification xsi:type="pivot:ExpressionInOCL" isRequired="false"
    type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
    body="self.obras->forAll(o| o.fechaDeEdicion >= self.fechaDeLaPrimerPublicacion)
        &#xA;&#xA;&#x9;&#x9;&#xA;&#x9;">
  <ownedBody xsi:type="pivot:IteratorExp" isRequired="false"
      type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
      referredIteration="http://www.eclipse.org/ocl/2015/
                         Library.oclas#i.ocl.Collection.forAll..T">
    <ownedSource xsi:type="pivot:PropertyCallExp"
        type="#//@ownedPackages.1/@ownedClasses.0"
        referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Autor.obras">
      <ownedSource xsi:type="pivot:VariableExp"
          type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Autor"
          referredVariable="#//@ownedPackages.0/@ownedClasses.0/@ownedInvariants.1/
                            @ownedSpecification/@ownedContext"/>
    </ownedSource>
    <ownedBody xsi:type="pivot:OperationCallExp" name=">="
        type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
        referredOperation="http://www.eclipse.org/ocl/2015/
                           Library.oclas#o.ocl.OclComparable.%62%%61%..ocl.OclSelf">
      <ownedSource xsi:type="pivot:PropertyCallExp"
          type="pivot:DataType http://www.eclipse.org/emf/2002/Ecore.oclas#T.ecore.EDate"
          referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Libro.fechaDeEdicion">
        <ownedSource xsi:type="pivot:VariableExp"
            type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Libro"
            referredVariable="#//@ownedPackages.0/@ownedClasses.0/@ownedInvariants.1/
                              @ownedSpecification/@ownedBody/@ownedIterators.0"/>
      </ownedSource>
      <ownedArguments xsi:type="pivot:PropertyCallExp" isRequired="false"
          type="pivot:DataType http://www.eclipse.org/emf/2002/Ecore.oclas#T.ecore.EDate"
          referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Autor.fechaDeLaPrimerPublicacion">
        <ownedSource xsi:type="pivot:VariableExp"
            type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Autor"
            referredVariable="#//@ownedPackages.0/@ownedClasses.0/@ownedInvariants.1/
                              @ownedSpecification/@ownedContext"/>
      </ownedArguments>
    </ownedBody>
    <ownedIterators name="o"
        type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Libro"
        representedParameter="http://www.eclipse.org/ocl/2015/
                              Library.oclas#i0i.ocl.Collection.forAll..T"/>
  </ownedBody>
  <ownedContext name="self" type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Autor"/>
</ownedSpecification>
</ownedInvariants>
```

Las reglas de transformación involucradas en está traducción son las presentadas anteriormente para el iterador exists, solo que tanto el inicio de la oración como el prefijo de las condiciones del iterador es basado en el hecho de que el iterador siendo procesado es un “forAll”. Está transformación genera la siguiente salida:

```
<oraciones>
  <contenido>
    <simple>
      <expresion_izq>
        <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
          <atributo determinante="todos los/las" nombre="obras" enlace="de"/>
          <termino indeterminante="un/una" contexto="Autor"/>
          <iteracion condicion="satisfacen que">
            <contenido>
              <simple>
                <expresion_izq>
                  <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
                    <atributo determinante="el/la" nombre="fechaDeEdicion" enlace="de"/>
                    <termino indeterminante="un/una" contexto="Libro"/>
                  </expresion>
                </expresion_izq>
                <expresion_der>
                  <comparacion>
                    <afirmacion afirmacion="es"/>
                    <operacion operacion="mayor que"/>
                  </comparacion>
                  <expresion>
                    <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
                      <atributo determinante="el/la" nombre="fechaDeLaPrimerPublicacion" enlace="de"/>
                      <termino indeterminante="un/una" contexto="Autor"/>
                    </expresion>
                  </expresion>
                </expresion_der>
              </simple>
            </contenido>
          </iteracion>
        </expresion>
      </expresion_izq>
    </simple>
  </contenido>
</oraciones>
```

7.7.2 - Nueva Colección como salida

Select (condiciones): Retorna un subconjunto de los elementos contenidos en la colección como una nueva colección. Los elementos contenidos en esta nueva colección son aquellos para los que la evaluación de las condiciones retorna verdadero. Al referirnos a invariantes sucede como en el caso de la operación size, como no retorna un resultado booleano siempre ira acompañado de alguna operación o comparación que resulte en un booleano.

```
context Libro
    inv Select_notEmpty: self.copias->
                              select(c| c.estadoDeLaCopia = 'Bueno')->notEmpty()
```

Como se presentó anteriormente notEmpty es traducido como una comparación pero este invariante representa una combinación de iteradores, que es traducida como:

```
el/la cantidad de copias de un/una Libro tal que el/la estadoDeLaCopia de
un/una Copia es igual a "Bueno" es mayor que "0".
```

```
<ownedInvariants xmi:id="ciBiblioteca.Libro.Select_notEmpty" name="Select_notEmpty">
  <ownedSpecification xsi:type="pivot:ExpressionInOCL"
      type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
      body="self.copias->select(c| c.estadoDeLaCopia = 'Bueno')->notEmpty()  ">
    <ownedBody xsi:type="pivot:OperationCallExp"
        type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
        referredOperation="http://www.eclipse.org/ocl/2015/Library.oclas#o.ocl.Collection.notEmpty">
      <ownedSource xsi:type="pivot:IteratorExp"
          type="#//@ownedPackages.1/@ownedClasses.2"
          referredIteration="http://www.eclipse.org/ocl/2015/
                    Library.oclas#i.ocl.OrderedSet.select..T">
        <ownedSource xsi:type="pivot:PropertyCallExp"
            type="#//@ownedPackages.1/@ownedClasses.2"
            referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Libro.copias">
          <ownedSource xsi:type="pivot:VariableExp"
              type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Libro"
              referredVariable="#//@ownedPackages.0/@ownedClasses.3/@ownedInvariants.1/
                        @ownedSpecification/@ownedContext"/>
        </ownedSource>
        <ownedBody xsi:type="pivot:OperationCallExp" name="="
            type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#Boolean"
            referredOperation="http://www.eclipse.org/ocl/2015/
                      Library.oclas#o.ocl.OclAny.%61%..ocl.OclSelf">
          <ownedSource xsi:type="pivot:PropertyCallExp"
              type="pivot:Enumeration Biblioteca.ecore.oclas#T.Biblioteca.EstadoDeLaCopia"
              referredProperty="Biblioteca.ecore.oclas#p.Biblioteca.Copia.estadoDeLaCopia">
            <ownedSource xsi:type="pivot:VariableExp"
                type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Copia"
                referredVariable="#//@ownedPackages.0/@ownedClasses.3/@ownedInvariants.1/
                          @ownedSpecification/@ownedBody/@ownedSource/@ownedIterators.0"/>
          </ownedSource>
          <ownedArguments xsi:type="pivot:StringLiteralExp"
              type="pivot:PrimitiveType http://www.eclipse.org/ocl/2015/Library.oclas#String"
              stringSymbol="Bueno"/>
        </ownedBody>
        <ownedIterators name="c" type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Copia"
            representedParameter="http://www.eclipse.org/ocl/2015/
                      Library.oclas#i0i.ocl.OrderedSet.select..T"/>
      </ownedSource>
    </ownedBody>
    <ownedContext name="self" type="pivot:Class Biblioteca.ecore.oclas#T.Biblioteca.Libro"/>
  </ownedSpecification>
</ownedInvariants>
```

Las operaciones *Empty* and *notEmpty* son traducidas como comparaciones, de esta forma tendremos una expresión a izquierda que será la que contenga la traducción necesaria para incluir el iterador *Select* y una expresión fija a derecha que será definida como *"mayor que 0"*.
La expresión izquierda es definida de manera muy similar a los iteradores presentados anteriormente. Es una expresión que contiene un atributo (colección) y una iteración compuesta por las condiciones definidas en el iterador. Estas condiciones son representadas como oraciones simples o compuestas (si se definen múltiples condiciones conectadas por un operador lógico).

La estructura en lenguaje natural para representar el invariante analizado en esta sección puede ser observada a continuación:

```
<oraciones>
  <contenido>
    <simple>
      <expresion_izq>
        <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
          <atributo determinante="el/la" cuantitativo="cantidad de" nombre="copias" enlace="de"/>
          <termino indeterminante="un/una" contexto="Libro"/>
          <iteracion condicion="tal que">
            <contenido>
              <simple>
                <expresion_izq>
                  <expresion xsi:type="lenguajeNaturalReducido:Propiedad">
                    <atributo determinante="el/la" nombre="estadoDeLaCopia" enlace="de"/>
                    <termino indeterminante="un/una" contexto="Copia"/>
                  </expresion>
                </expresion_izq>
                <expresion_der>
                  <comparacion>
                    <afirmacion afirmacion="es"/>
                    <operacion operacion="igual a"/>
                  </comparacion>
                  <expresion>
                    <expresion xsi:type="lenguajeNaturalReducido:Literal" literal="Bueno"/>
                  </expresion>
                </expresion_der>
              </simple>
            </contenido>
          </iteracion>
        </expresion>
      </expresion_izq>
      <expresion_der>
        <comparacion>
          <afirmacion afirmacion="es"/>
          <operacion operacion="mayor que"/>
        </comparacion>
        <expresion>
          <expresion xsi:type="lenguajeNaturalReducido:Literal" literal="0"/>
        </expresion>
      </expresion_der>
    </simple>
  </contenido>
</oraciones>
```

Capítulo 8: Herramienta Desarrollada

El objetivo de la herramienta prototipada es permitir la traducción de invariantes definidas usando el lenguaje formal OCL a una representación de dominio reducido del lenguaje natural, permitiendo la fácil interpretación de las mismas por parte de usuarios con mínimos o nulos conocimientos técnicos sobre orientación a objetos o lógica de primer orden.

8.1 - Diseño

La herramienta diseñada es un plugin de Eclipse que contiene diferentes módulos que se interconectan para poder realizar la traducción. Estos módulos se definen a continuación.

Procesamiento de OCL: Después de investigar todas las posibles variantes que el plugin OCL ofrece para definir restricciones sobre un modelo, decidimos que el uso de Complete OCL es el más adecuado ya que permite definir restricciones en un archivo independiente del modelo subyacente por lo que la implementación no se liga a un modelo específico, Ecore o UML sino que utiliza la implementación genérica conocida como Pivot.

Representación de Lenguaje Natural: Para permitir la traducción se definió un metamodelo que permitiera representar sentencias u oraciones en lenguaje natural. Debido a la amplia extensión del lenguaje natural se procedió a realizar un modelo reducido al dominio que se intenta traducir, en este caso particular OCL.
Para la creación de este metamodelo se eligió Xtext por su facilidad para la definición de la gramática del lenguaje, basada en EBNF, y por su poder de generar todas las herramientas necesarias para el procesamiento del lenguaje definido.

Transformación de modelos: Este módulo es donde se realiza la traducción en sí misma mediante la definición de reglas en el lenguaje ATL. Debido a que la correlación entre la sintaxis de OCL y Lenguaje natural no es de uno a uno, es decir que no se puede traducir de un lenguaje al otro directamente sin realizar diferente tipos de transformaciones, tanto las reglas de traducción como la gramática de LN debieron ser adaptadas para poder representar las restricciones de una manera simple y legible para el usuario.

8.2 - Implementación

La idea es permitir al usuario una forma rápida y sencilla de traducir invariantes OCL definidas en un archivo independiente del modelo, esto es posible gracias a la existencia de Complete OCL que permite que la definición de invariantes sea independiente del lenguaje de modelado utilizado (Ecore o UML). El plugin OCL que contiene la implementación provee la funcionalidad de poder generar un archivo con la sintaxis abstracta de las invariantes desde el archivo previamente definido usando Complete OCL. Este archivo contenedor de la sintaxis abstracta de las invariantes es la información de entrada utilizada para transformar mediante el uso de reglas definidas en ATL los invariantes a Lenguaje Natural Reducido.
Para poder probar la herramienta desarrollada se implementó un modelo Ecore de ejemplo, este modelo fue presentado en el capítulo anterior, y se definió un conjunto de invariantes para presentar algunos de todos los tipos de invariantes que la herramienta es capaz de traducir. Para este desarrollo se utilizaron los plugins, EMF y OCL. En la figura 8-1 se puede observar la estructura de este proyecto llamado ModeloEcoreTesina.

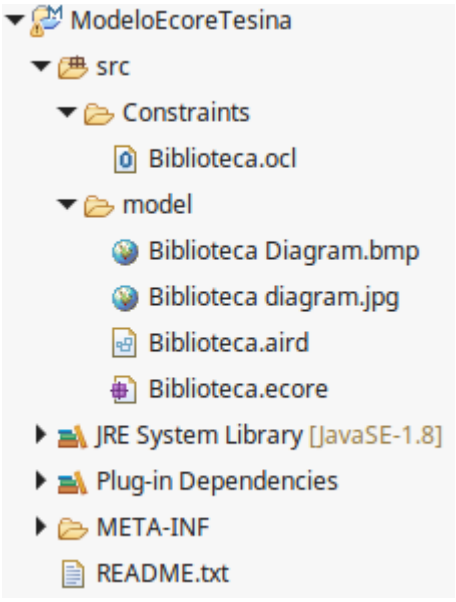


Figura 8-1. Proyecto Modelo Ecore

La implementación cuenta con otros tres proyectos interconectados y necesarios para realizar las traducciones.

Proyecto transformador es un proyecto ATL que contiene las reglas necesarias para la traducción OCL a LNR y una configuración predefinida para poder ejecutar la traducción. Este es el proyecto donde la parte más importante del proceso de traducción sucede; realiza la transformación de un modelo a otro.
La clase OCL2LNR contiene los métodos necesarios para ejecutar las reglas ATL definidas en OCL2LNR.atl desde el plugin de traducción.
OCL2LNR.properties contiene las referencias a los metamodelos utilizados, Pivot para OCL y Lenguaje Natural Reducido para la salida en lenguaje natural.

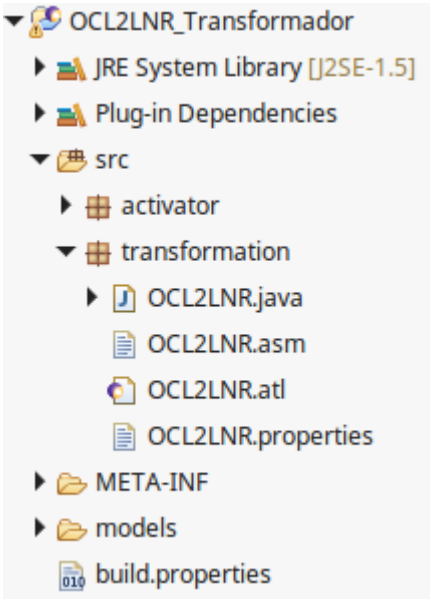


Figura 8-2. Proyecto Transformador OCL a LN

org.xtext.tesina.LenguajeNaturalReducido es el proyecto Xtext, o un conjunto de proyectos, que contienen la gramática definida para nuestra versión acotada de lenguaje

natural y todas las herramientas autogeneradas necesarias para procesamiento, edición, serialización, etc. La figura 8-3 muestra la estructura de este Proyecto.

TesinaPlugin es el proyecto que contiene el plugin prototipado para mostrar que es posible la traducción de OCL a LNR de una manera simple. Este es el proyecto principal ya que es la definición de nuestro plugin y el que provee la interfaz de interacción con el usuario. La estructura de este Proyecto puede verse en la figura 8-4.

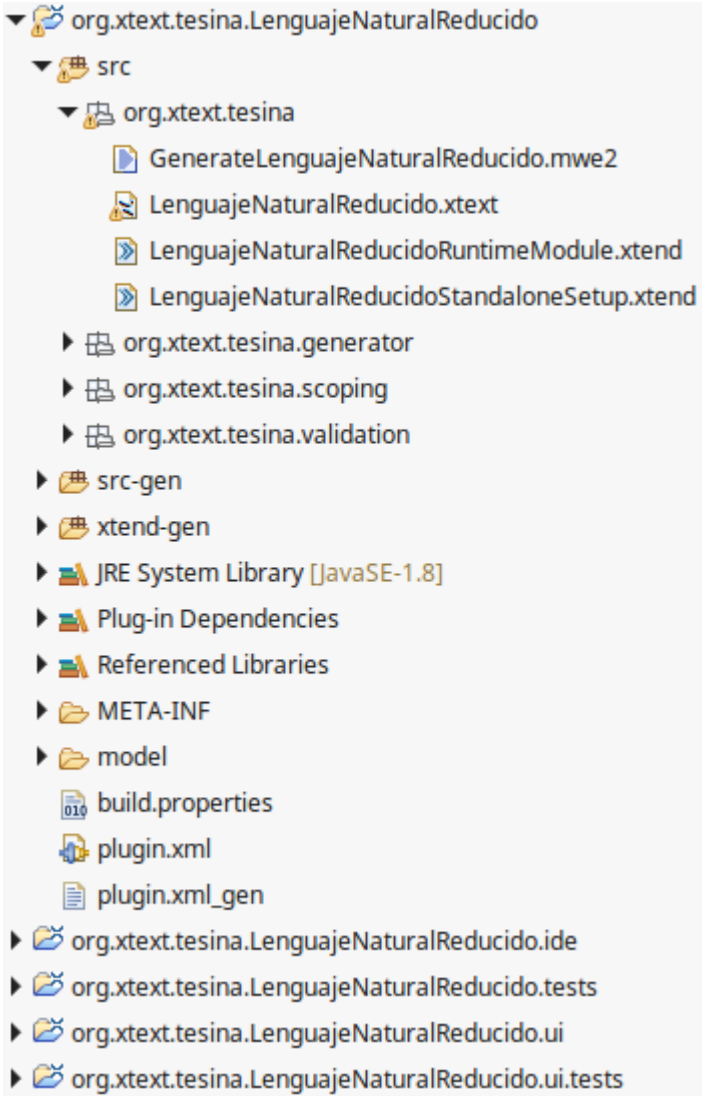


Figura 8-3. Proyecto Lenguaje Natural Reducido en Xtext

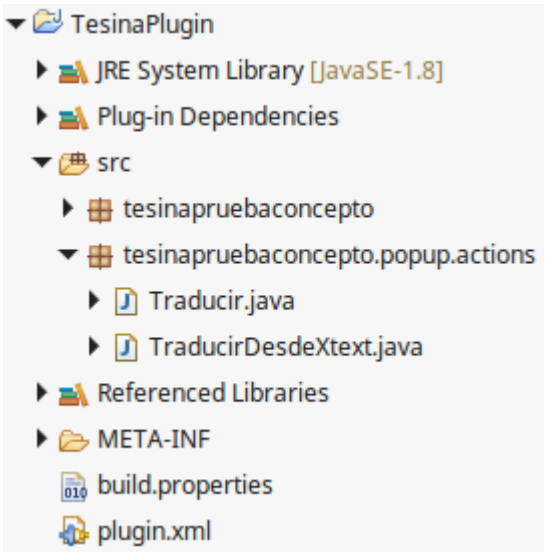


Figura 8-4. Proyecto Plugin Prototipado

8.3 - Manual de uso

Supongamos que contamos con el metamodelo Ecore y un archivo CompleteOCL conteniendo los invariantes definidos, para este manual usaremos el ejemplo previamente presentado de la Biblioteca.
El primer paso es generar un archivo con la sintaxis abstracta (cuya extensión es .ocl.oclas) correspondiente al archivo que contiene las invariantes, en este caso particular Biblioteca.ocl. Para generar la sintaxis abstracta se debe contar con el plugin de OCL instalado. Este plugin es el que habilita un menú contextual al hacer click derecho sobre un archivo con extensión .ocl (figura 8-5).
Se abre una ventana de diálogo donde se debe ingresar el nombre que se le quiere dar al archivo de sintaxis abstracta, por defecto utiliza el mismo nombre que el del archivo fuente, por lo que Biblioteca.ocl.oclas es el nombre predefinido. El plugin de traducción OCL2LNR requiere que el archivo .ecore, .ocl y ocl.oclas posean el mismo nombre, así que es necesario mantener el nombre predefinido.
Una vez creado el archivo Biblioteca.ocl.oclas, abrirlo ya que el plugin de traducción habilita un menú contextual solamente sobre archivos de extensión ocl.oclas para evitar que el usuario trate de traducir otro tipo de archivos por error.
Una vez que el usuario selecciona la opción Traducir OCL a LN (figura 8-6), el plugin realiza una serie de operaciones y pasos en el background haciendo el proceso de traducción transparente al usuario. Las operaciones y pasos que la herramienta realiza son:

1. Inicializar la transformación ATL, generando el archivo de configuración correspondiente con las referencias al archivo .ocl.oclas definido por el usuario y al respectivo modelo Ecore.
2. Una vez inicializado ATL, se ejecuta la traducción en sí, que no es más que aplicar las reglas definidas en ATL sobre el modelo de entrada, Biblioteca.ocl.oclas, para generar un modelo de salida conforme con la gramática de Lenguaje Natural que definimos en Xtext.
3. El modelo de salida no es visible por el usuario pero si es serializado mediante el uso del serializador que Xtext provee para generar un archivo de salida con el resultado de la transformación realizada que muestra de manera textual al usuario las invariantes traducidas al lenguaje natural.

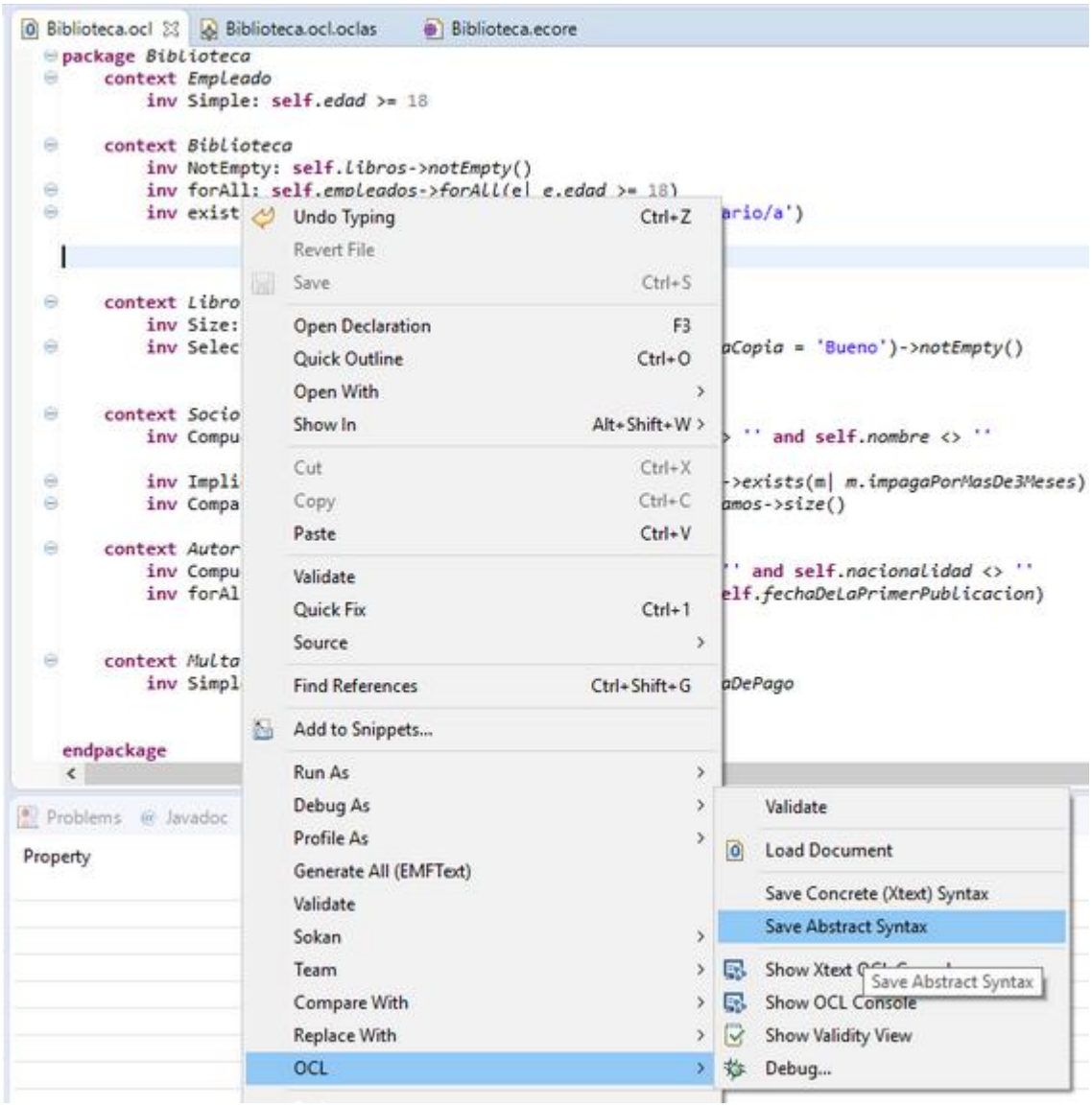


Figura 8-5. Generar archivo de sintaxis abstracta

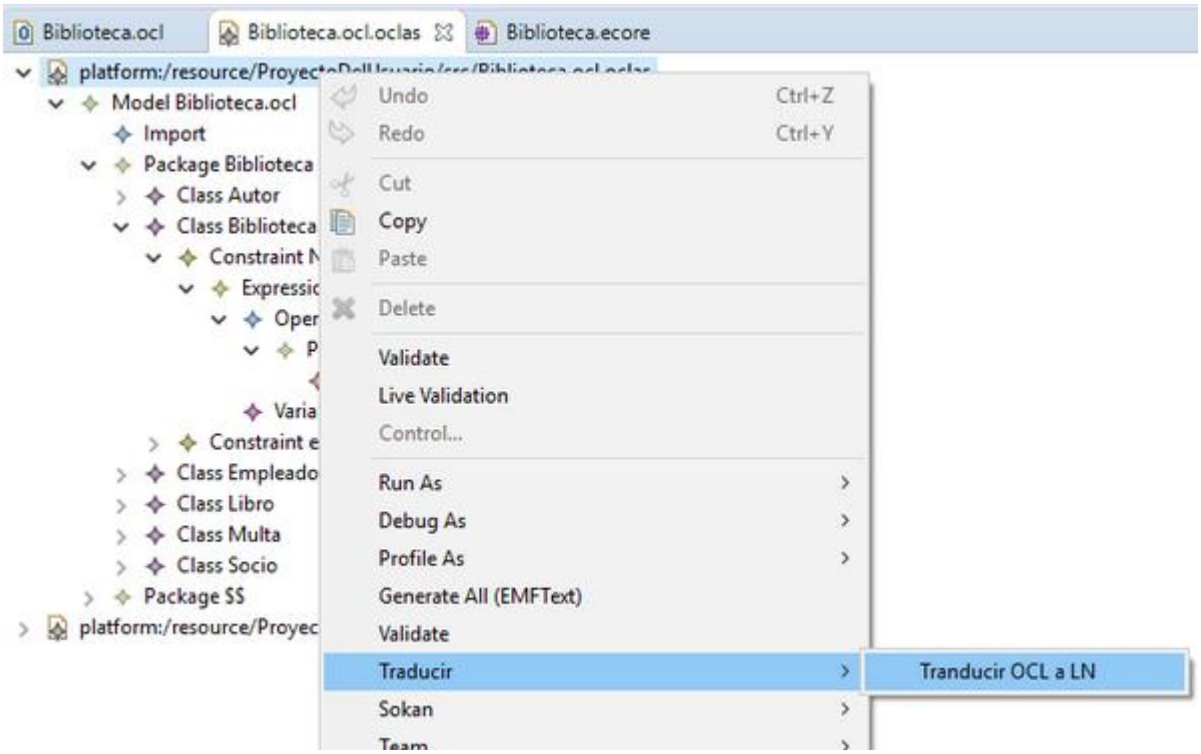


Figura 8-6. Traducir OCL a LN

El archivo de salida tendrá el mismo nombre que el archivo .ocl.oclas de entrada más el timestamp de cuando fue creado y una extensión .lnr (lenguaje natural reducido, podemos usar el editor Xtext que resalta la sintaxis). Este archivo se almacenará en una carpeta que el plugin crea en el proyecto del usuario, llamada “Traducciones”. La estructura del proyecto del usuario se puede observar a continuación en la figura 8-7 y el contenido del archivo conteniendo la traducción en la figura 8-8.

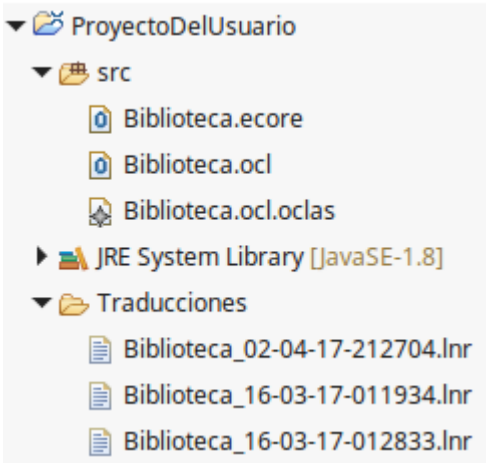


Figura 8-7. Archivos generados salida traducción

Biblioteca_02-04-17-212704.lnr

```
"Las siguientes afirmaciones deben ser verdaderas en el sistema:" .
el/la nacionalidad de un/una Autor es distinto de "" y
el/la apellido de un/una Autor es distinto de "" y
el/la nombre de un/una Autor es distinto de "" .
todos los/las obras de un/una Autpr satisfacen que el/la fechaDeEdicion de un/una Libro
es mayor o igual a el/la fechaDeLaPrimerPublicacion de un/una Autor .
el/la cantidad de libros de un/una Biblioteca es mayor que "0" .
entre los/las empleados de un/una Biblioteca existe uno/una tal que
el/la rol de un/una Empleado es igual a "Bibliotecario/a" .
el/la edad de un/una Empleado es mayor o igual a "18" .
el/la cantidad de copias de un/una Libro es mayor o igual a "1" .
el/la cantidad de copias de un/una Libro tal que
el/la estadoDeLaCopia de un/una Copia es igual a "Bueno" es mayor que "0" .
el/la fechaDeInicio de un/una Multa es menor o igual a el/la fechaDePago de un/una Multa .
el/la nombre de un/una Socio es distinto de "" y
el/la apellido de un/una Socio es distinto de "" y
el/la direccion de un/una Socio es distinto de "" .
```

Figura 8-8. Contenido archivo traducción lenguaje natural

Capítulo 9: Trabajos Relacionados

9.1 - UML/OCL a especificaciones SBVR: Transformación desafiante

En [39] los autores presentan una forma de realizar transformaciones entre UML/OCL y SBVR de forma automática.
UML es uno de los lenguajes de modelado más ampliamente utilizado actualmente para la especificación de esquemas conceptuales (Conceptual Schemas, CS) de un sistema de información (Information System, IS). Sin embargo, UML se queda corto cuando se trata de permitir a personas del negocio definir utilizando su propio lenguaje (usar sus propios términos en lenguaje natural) las políticas y reglas en las cuales se basa el negocio. Para este propósito, se propuso una especificación del metamodelo de la semántica del vocabulario y reglas del negocio (Semantics of Business vocabulary and Rules, SBVR). SBVR está conceptualizado óptimamente para personas del negocio y diseñado para ser usado con propósitos de negocios independientemente de los diseños de los sistemas informáticos. Claramente, SBVR y UML no pueden considerarse como lenguajes aislados; muchas de las reglas del negocio especificadas por personas del negocio deben ser ejecutadas automáticamente por el sistema de información subyacente, y por lo tanto deben aparecer en el esquema conceptual UML. En este sentido, el objetivo principal del trabajo es reducir la brecha entre UML y SBVR proporcionando una transformación automática de especificaciones desde UML a SBVR.

9.2 - Parafraseando expresiones OCL con SBVR

Los autores en [40] muestran cómo se pueden parafrasear expresiones OCL mediante el uso del estándar SBVR.
Un esquema conceptual (CS) debería ser explicado a los stakeholders para validar que representa apropiadamente todo el conocimiento del dominio. Una de las mejores maneras de explicar el CS es describirlo mediante el uso de expresiones en lenguaje natural (Parafraseando o su equivalente en inglés, Paraphrasing). Incluso cuando el parafraseo de los elementos más típicos de un CS han sido estudiados, los métodos actuales son, en general, incapaces de soportar las reglas textuales del negocio que complementan el CS. El trabajo apunta a cubrir esta brecha, mediante la presentación de un método que genera explicaciones en lenguaje natural para reglas del negocio expresadas en OCL, el lenguaje estándar para la especificación de reglas del negocio sobre CSs basados en UML. Como un paso intermedio, el método traduce las expresiones definidas en OCL a una representación en SBVR.

9.3 - Verbalización de reglas: Aplicación a restricciones OCL en el dominio Utility

Los autores en [41] exponen una forma de traducir reglas expresadas en un lenguaje de diseño a un lenguaje semi-natural y lo hacen mediante la aplicación sobre el dominio Utility.
Las reglas de negocio son definidas, especificadas y validadas por expertos del negocio pero son diseñadas e implementadas por implementadores técnicos. Cada uno de ellos utiliza lenguajes adaptados a su actividad y habilidad. La verbalización de las reglas de negocio permite a los expertos del negocio obtener una expresión semi-natural de las reglas diseñadas por los implementadores técnicos facilitando así su tarea de validación. La

herramienta es capaz de realizar transformaciones automatizando la verbalización y aplicando esto a restricciones OCL en el dominio Utility.

9.4 - De especificaciones de software de lenguaje natural a modelos de clase UML

En [42] los autores introducen un enfoque basado en SBVR para generar una especificación controlada de una especificación en lenguaje natural.
Las especificaciones de software son capturadas típicamente en lenguajes naturales y luego los analistas de software las analizan y producen manualmente los modelos de software como los modelos de clase. Varios enfoques, frameworks y herramientas han sido presentados para la traducción automática de modelos de software como CM-Builder, Re-Builder, NL-OOML, GOOAL, etc. Sin embargo, los experimentos con estas herramientas muestran que no proporcionan una alta precisión en la traducción. La razón principal para la menor precisión reportada en la literatura es la naturaleza ambigua e informal de los lenguajes naturales. Este artículo, apunta a tratar este problema y a presentar un enfoque mejor para el procesamiento de lenguajes naturales y producir modelos de software UML más precisos. El enfoque presentado está basado en la Semántica del Vocabulario y Reglas de Negocio (SBVR) un estándar recientemente adoptado por el OMG. El enfoque funciona ya que las especificaciones de software en lenguaje natural se asignan primero a la representación de reglas en SBVR. Las reglas SBVR son fáciles de traducir a otras representaciones formales como OCL y UML ya que SBVR está basada en lógica de orden superior. El caso de estudio que viene a resolver la herramienta ***NL2UMLviaSBVR*** es presentado y el análisis comparativo de herramientas de investigación con otras herramientas disponibles muestra que el uso de SBVR en la traducción NL a UML ayuda a mejorar la precisión.

9.5 - Generación de restricciones OCL a partir de una especificación en LN

En [43] los autores exhiben como generar a partir de una especificación en lenguaje natural restricciones en OCL.
El Lenguaje de Restricción de Objetos (OCL) juega un papel clave en el Lenguaje de Modelado Unificado (UML). En los estándares UML, OCL se utiliza para expresar restricciones tales como criterios que especifican cuando algo está bien definido. Además, OCL se puede utilizar para especificar restricciones en los modelos y pre/post condiciones en las operaciones, mejorando la precisión de la especificación. Como resultado, OCL ha recibido una considerable atención por parte de la comunidad investigadora. Sin embargo, a pesar de su papel clave, existe un consenso común en qué OCL es el menos adoptado entre todos los lenguajes en UML. A menudo se argumenta que, los profesionales de software se apartan de OCL debido a su sintaxis desconocida. Para garantizar una mejor adopción de OCL, los problemas de usabilidad relacionados con la producción de sentencias en OCL deben ser abordados. Para abordar este problema, el trabajo pretende establecer un método involucrando el uso de expresiones NaturalLanguage y tecnología de Transformación de Modelos. El objetivo del método es producir un framework para que el usuario de la herramienta UML pueda escribir restricciones y pre/post condiciones en inglés y el framework convierta dichas expresiones en lenguaje natural a su sentencia equivalente en OCL. Como resultado, el enfoque tiene como objetivo la simplificación del proceso de generación de sentencias OCL, permitiendo al usuario beneficiarse de las ventajas proporcionadas por las herramientas UML que soportan OCL. El enfoque sugerido depende

de la Semántica del Vocabulario y Reglas del Negocio (SBVR) para apoyar la formulación de expresiones del lenguaje natural y sus transformaciones a OCL. El documento también presenta el esquema de una herramienta prototipo que implementa el método.

9.6 - Usabilidad de OCL: un gran desafío en la adopción de UML

En [44] los autores presentan un enfoque para abordar el problema de usabilidad de OCL produciendo restricciones OCL automáticamente a partir de texto en inglés.
Los aspectos principales del problema de usabilidad OCL se atribuyen a la sintaxis dura del lenguaje, la naturaleza ambigua de las expresiones OCL y la interpretación dificultosa de expresiones OCL largas. La contribución del presente artículo, es un enfoque novedoso que pretende presentar un método que implica el uso de expresiones de lenguaje natural y la tecnología de transformación de modelos para mejorar la usabilidad de OCL. El objetivo del método es producir un framework para que el usuario de herramientas UML pueda escribir restricciones y pre/post condiciones en inglés y el framework convierte dichas expresiones en inglés a las sentencias OCL equivalentes. El enfoque propuesto es implementado en una herramienta de software ***NL2OCLviaSBVR*** que genera restricciones OCL a partir de texto en inglés a través de SBVR. La herramienta permite a los modeladores y desarrolladores de software generar expresiones OCL bien formadas que resultan en modelos válidos y precisos. Una evaluación empírica de las restricciones OCL revela que el enfoque basado en el lenguaje natural para generar restricciones OCL supera significativamente a la técnica más estrechamente relacionada en términos de esfuerzo y efectividad.

9.7 - Conclusiones

Los trabajos relacionados presentados en las secciones anteriores todos tienen el mismo objetivo, incrementar mediante nuevos métodos y herramientas la ***usabilidad*** del lenguaje OCL. Para esto todos los trabajos, incluido el nuestro, se enfocan en proveer un medio para la traducción de restricciones OCL a un lenguaje natural y/o viceversa.
Todos los métodos o desarrollos de herramientas presentados por estos trabajos se basan en el uso de ***MDD*** y ***transformación de modelos***, al igual que esta tesina. El uso de estas metodologías ayuda a reducir la complejidad del desarrollo, permitiendo enfocarse en los metamodelos a ser transformados y la definición de las reglas de transformación sin necesidad de implementar dichas transformaciones a bajo nivel (el lenguaje de transformación se encarga de procesar los metamodelos, mantener las trazas y generar elementos del modelo de salida).
SBVR es un metamodelo para la representación de la semántica del vocabulario y reglas del negocio. Este metamodelo permite que las reglas representadas pueden ser verbalizadas fácilmente por cualquier persona. Es por esto que muchos de los trabajos relacionados utilizan este metamodelo como paso intermedio en la traducción y para lo que se requieren 2 transformaciones, una M2M UML/OCL a SBVR y una M2T de SBVR a lenguaje natural. Nuestro enfoque requiere la implementación de una única transformación M2M.
Algunos trabajos requieren ***extender el metamodelo UML*** de forma de poder diferenciar los diferentes tipos de reglas modales de la misma forma en que SBVR lo hace. La herramienta desarrollada en está tesina no requiere ninguna extensión de los metamodelos utilizados.
Es importante destacar que todos los trabajos traducen a lenguaje o desde el lenguaje ***Ingles*** mientras que nuestro trabajo se basa en la traducción a lenguaje ***castellano*** haciéndola una herramienta única.

Capítulo 10: Conclusiones y Trabajos Futuros

10.1 - Conclusiones

Se desarrolló una herramienta que permite la traducción de invariantes definidas en el lenguaje OCL sobre un modelo que puede ser tanto UML como Ecore ya que se utilizó la forma genérica de representación de estas invariantes conocida como Complete OCL. Este desarrollo fue implementado siguiendo los principios del desarrollo de software dirigido por modelos, de esta forma se utilizó el metamodelo existente de OCL (Pivot) y se definió un metamodelo para el Lenguaje Natural Reducido (limitado al dominio de invariantes OCL) con el fin de aplicar la transformación de modelos para conseguir el resultado esperado, una traducción de invariantes OCL a Lenguaje Natural. El lenguaje natural utilizado es castellano lo que hace a esta herramienta una de las primeras en realizar traducciones de OCL a este lenguaje.
Como parte del proceso de transformación de modelos se analizaron las diferentes opciones disponibles, Modelo a Modelo o Modelo a Texto y se decidió implementar una transformación Modelo a Modelo mediante el uso del lenguaje ATL. Esta elección se realizó de forma tal de proveer la posibilidad de que la traducción pueda ser realizada en ambos sentidos aunque en esta tesis solo se implementó la traducción de OCL a Lenguaje Natural.
El objetivo de esta herramienta es demostrar que es posible realizar una traducción desde OCL a Lenguaje Natural de una manera sencilla mediante el uso de MDD y ATL en particular como lenguaje de transformación de modelos. Otro objetivo importante es proveer la capacidad de obtener rápidamente un documento con las invariantes expresadas en lenguaje natural que pueden ser interpretadas por cualquier persona, especialmente por aquellas relacionadas al proceso de desarrollo de software que no poseen los conocimientos técnicos suficientes para interpretar OCL. De esta forma lo que se busca es incrementar la usabilidad de OCL.

10.2 - Trabajos Futuros

Extender la implementación de las invariantes para poder incluir las operaciones de colecciones que no fueron incluidas, por ejemplo definition, let, reject(), allInstances(), entre otras.
Permitir la traducción de métodos junto con sus precondiciones, postcondiciones y body.
Mejorar la gramática de Lenguaje Natural Reducido para eliminar la repetición del contexto en cada oración simple cuando las oraciones simples conectadas se refieren al mismo contexto. Por ejemplo:

```
context Socio
    inv DatosCompletos: self.nombre<>'' and self.direccion<>''
```

La traducción incluye el contexto socio por cada propiedad pero podría simplificarse agregando una descripción del contexto antes de traducir sus invariantes como se muestra a continuación.

```
"Para un socio las siguientes afirmaciones son verdaderas: "
el/la nombre es distinto de "" y el/la direccion es distinto de "".
```

Otra mejora que se podría agregar a la herramienta es la capacidad de reconocer géneros, para evitar tener que escribir “el/la”, y la capacidad de determinar el número, para poder usar

“las”/“los” cuando corresponda. Usando el ejemplo definido para las mejorar propuestas anteriormente, la oración después de los cambios queda:

```
el nombre es distinto de "" y la direccion es distinto de "".
```

Generar la traducción inversa en ATL, Lenguaje Natural Reducido a OCL, para completar la herramienta y que sea posible realizar traducciones bidireccionales.

Referencias Bibliográficas

[1] Pons, C., Giandini, R., Pérez, G., 2010, Desarrollo de Software Dirigido Por Modelos, Conceptos Teóricos y su aplicación práctica; Universidad Nacional de La Plata.

[2] Eclipse Modeling Framework Project (EMF) - https://eclipse.org/modeling/emf/

[3] OCL - https://projects.eclipse.org/projects/modeling.mdt.ocl

[4] Xtext - https://eclipse.org/Xtext/

[5] ATL - https://eclipse.org/atl/

[6] Model to Text (M2T) - https://eclipse.org/modeling/m2t/

[7] Model to Model (M2M) - https://projects.eclipse.org/projects/modeling.mmt

[8] OCL by Example. Dr Birgit Demuth, Department of Computer Science, Institute for Software and Multimedia Technology, Technische Universitat Dresden - https://st.inf.tu-dresden.de/files/general/OCLByExampleLecture.pdf

[9] Introduction to OCL - Jordi Cabot - http://es.slideshare.net/jcabot/ocl-tutorial

[10] Desarrollo de software dirigido por modelos. Francisco Durán Muñoz, Javier Troya Castilla, Antonio Vallecillo Moreno. Universitat Oberta de Catalunya - https://eva.fing.edu.uy/pluginfile.php/123820/mod_resource/content/1/MDSDVallecillo.pdf

[11] OCL Documentation, Christian Damus, Adolfo Sánchez-Barbudo Herrera, Axel Uhl, Edward Willink and contributors - Copyright 2002 - 2014 - http://download.eclipse.org/ocl/doc/5.0.0/ocl.pdf

[12] Documents Associated With Object Constraint Language™ (OCL™), Version 2.4 - http://www.omg.org/spec/OCL/2.4/

[13] Samin Salemi, Ali Selamat, Marek Penhaker, A model transformation framework to increase OCL usability, Journal of King Saud University - Computer and Information Sciences,
http://www.sciencedirect.com/science/article/pii/S1319157815000853

[14] The Stanford Natural Language Processing Group - http://nlp.stanford.edu/

[15] Imran Sarwar Bajwa, Behzad Bordbar, Mark Lee. (2010). "OCL Constraints Generation from Natural Language Specification", in IEEE/ACM 14th International EDOC Conference 2010, Vitoria, Brazil, October 2010. http://www.cs.bham.ac.uk/~bxb/Papres/1005.pdf

[16] The Eclipse Project. Home Page. Copyright IBM Corp. and others, 2000-2004. http://www.eclipse.org/

[17] Documents associated with Object Constraint Language (OCL), Version 2.4. http://www.omg.org/spec/OCL/2.4/

[18] The Jamda Project http://jamda.sourceforge.net/

[19] Acceleo http://www.eclipse.org/acceleo/

[20] MOFScript https://eclipse.org/gmt/mofscript/

[21] TCS http://www.eclipse.org/gmt/tcs/

[22] Xtext User Guide

[23] Epsilon Documentación http://www.eclipse.org/epsilon/

[24] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, ATL: A model transformation tool, Science of Computer Programming, Volume 72, Issues 1–2, 1 June 2008, Pages 31-39, ISSN 0167-6423, http://doi.org/10.1016/j.scico.2007.08.002.

(http://www.sciencedirect.com/science/article/pii/S0167642308000439)

Keywords: Model engineering; Model transformation; M2M (model-to-model transformation)

[25] Dr Markus Kuhn

ISO/IEC 14977: 1996(E) - Extended EBNF Syntax

http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

[26] Lars M. Garshol, BNF and EBNF: What are they and how do they work?, Last update 2008-08-22

http://www.garshol.priv.no/download/text/bnf.html#id1

[27] BNF and EBNF http://gnindia.dronacharya.info/itDept/Downloads/Questionpapers/5th_sem/Principle-Programming-Language/Unit-1/EBNF.pdf

[28] Fundamentos de la Teoría de Gramáticas Formales http://www.uhu.es/francisco.moreno/talf/docs/tema3.pdf

[29] Jianan Yue, Transition from EBNF to Xtext

http://ceur-ws.org/Vol-1258/src5.pdf

[30] Alberto Pacheco, Notación BNF: Última actualización Mzo 1, 1999

http://www.socrates.itch.edu.mx/~apacheco/teoria/bnf.htm

[31] José Manuel Silva, Árboles De Derivación: 23 June 2015

https://prezi.com/y4jd_5-0-9ip/arboles-de-derivacion/

[32] Jean Bézivin, Frédéric Jouault, David Touzet, An introduction to the ATLAS Model Management Architecture

http://se-pubs.dbs.uni-leipzig.de/files/Bezivin2005AnIntroductiontotheATLAS.pdf

[33] The epsilon book: January 18, 2017

https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/plain/doc/org.eclipse.epsilon.book/EpsilonBook.pdf

[34] Frédéric Jouaulta, Freddy Allilairea, Jean Bézivina, Ivan Kurtevb, ATL: A model transformation tool: 1 June 2008, Pages 31–39

http://ac.els-cdn.com/S0167642308000439/1-s2.0-S0167642308000439-main.pdf

[35] The Viatra-I Model Transformation Framework Pattern Language Specification http://www.eclipse.org/viatra/doc/ViatraSpecification.pdf

[36] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi

, Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework: 12 May 2016.

https://link.springer.com/article/10.1007/s10270-016-0530-4

[37] Alejandro Tovar Moreno, Ingeniería dirigida a modelos, sistemas de transformación modelo a texto, complemento de refactorización de código C++ a C++11 para eclipse: 2013-06-25. http://e-archivo.uc3m.es/bitstream/handle/10016/19209/TFG_Alejandro_Tovar_Moreno.pdf

[38] Arnaud Dieumegard, Andres Toom, Marc Pantel, Comparing transformation languages for the implementation of certified model transformations. 2012.

[39]Jordi Cabot, Raquel Pau, Ruth Raventós, From UML/OCL to SBVR specifications: A challenging transformation, Information Systems, Volume 35, Issue 4, June 2010, Pages 417-440, ISSN 0306-4379, http://doi.org/10.1016/j.is.2008.12.002. (http://www.sciencedirect.com/science/article/pii/S030643790800094X)
Keywords: UML; OCL; SBVR; Model transformation

[40] Raquel Pau and Jordi Cabot. 2008. Paraphrasing OCL Expressions with SBVR. In *Proceedings of the 13th international conference on Natural Language and Information Systems: Applications of Natural Language to Information Systems* (NLDB '08), Epaminondas Kapetanios, Vijayan Sugumaran, and Myra Spiliopoulou (Eds.). Springer-Verlag, Berlin, Heidelberg, 311-316. DOI=http://dx.doi.org/10.1007/978-3-540-69858-6_30

[41] Baghli, Rayhana and Bruno Traverson. “Verbalization of business rules: Application to OCL constraints in the utility domain.” *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)* (2014).

[42] R. Sharma, P. K. Srivastava and K. K. Biswas, "From natural language requirements to UML class diagrams," *2015 IEEE Second International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, Ottawa, ON, 2015, pp. 1-8.

http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7337625&isnumber=7337617

[43] I. S. Bajwa, B. Bordbar and M. G. Lee, "OCL Constraints Generation from Natural Language Specification," *2010 14th IEEE International Enterprise Distributed Object Computing Conference*, Vitoria, 2010, pp. 204-213. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5630350&isnumber=5629535

[44] Imran Sarwar Bajwa, Behzad Bordbar, and Mark Lee. 2014. OCL usability: a major challenge in adopting UML. In *Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering* (RAISE 2014). ACM, New York, NY, USA, 32-37. DOI=http://dx.doi.org/10.1145/2593801.2593807

Glosario

ACG - Atl Code Generation
AST - Abstract syntax tree
ADT - ATL Development Tools
AMMA - ATLAS Model Management Architecture
ATL - ATLAS Transformation Language
BNF - Backus-Naur form
CIM - Computational Independent Model
CS - Conceptual Schema
DOM - Document Object Model
DSL - Domain Specific Language
EMF - Eclipse Modelling framework
EMFT - Eclipse Modelling framework Technology
EJB - Enterprise Java Beans
ECL - Epsilon Comparison Language
EGL - Epsilon Generation Language
EML - Epsilon Merging Language
EOL - Epsilon Object Language
ETL - Epsilon Transformation Language
EVL - Epsilon Validation Language
EWL - Epsilon Wizard Language
EBNF - Extended Backus–Naur form
IDE - Integrated Development Environment
IM - Implementation Model
IS - Information System
MDR - Meta Data repository
MBD - Model Based Development
MDA - Model Driven Architecture
MDD - Model Driven Development
MDE - Model-Driven Engineering
M2T - Model-To-Text
MWE2 - Modelling Workflow Engine 2
OCL - Object Constraint Language
OMG - Object Management Group
PIM - Platform Independent Model
PSM - Platform Specific Model
SBVR - Semantics of Business Vocabulary and Rules
TCS - Textual Concrete Syntax
UML - Unified Modeling Language
VTCL - VIATRA Textual Command Language
VTML - VIATRA Textual Metamodeling Language

XMI - XML Metadata Interchange
XSD - XML Schema Definition