Universidad ORT Uruguay
Facultad de Ingeniería

Interpretación de KM3/ATL en Teoría de Tipos

Entregado como requisito para la obtención del título
de
Master en Ingeniería

Carlos Fornaro
101143

Tutor: Álvaro Tasistro

2014

Declaración de Autoría

Yo, Carlos Nicols Fornaro Rosado, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Aseguro, bajo mi entera responsabilidad, que:

- La obra fue producida en su totalidad mientras realizaba la tesis final del postgrado Master en Ingeniería;
- Cuando he consultado trabajos publicados por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros autores, he indicado las fuentes. Con excepción de dichas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué parte fue contribuida por dichos terceros, y qué parte fue contribuida por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto los casos en que se han realizado las aclaraciones correspondientes.

Carlos Fornaro

Agradecimientos

Deseo agradecer a mi esposa Esther e hijos, Juan y Sofía, por la paciencia y aliento en todo momento. Gran parte del tiempo invertido en este trabajo les pertenece. También a mi madre, quien a lo largo de toda mi vida, y hasta el día de hoy, me alentó y motivó a estudiar, siempre con un apoyo incondicional.

Agradezco muy especialmente a Álvaro Tasistro la oportunidad y confianza brindadas, y sobre todo su infinita paciencia y cordial amistad. Sus enseñanzas, contribuciones y correcciones fueron indispensables para este trabajo.

Finalmente agradezco a la Universidad ORT Uruguay por la beca otorgada para la realización de esta Maestría.

Abstract

El término *Ingeniería Dirigida por Modelos* (MDE) se utiliza para describir enfoques de desarrollo de software en los cuales se crean modelos abstractos de sistemas de software que son transformados de manera sistemática para obtener implementaciones concretas de los sistemas que representan. En este contexto, se han desarrollado lenguajes para describir transformaciones ejecutables por máquina. ATL aparece como uno de los lenguajes más utilizados para especificar este tipo de transformaciones, en conjunto con KM3 para expresar la estructura de los modelos. Es de nuestro interés la verificación formal de los modelos y transformaciones especificadas utilizando ATL y KM3, para mejorar la calidad y confiabilidad de los sistemas desarrollados.

La principal dificutad para razonar sobre transformaciones y modelos especificados en estos lenguajes es la falta de una semántica formal. Es buscando eliminar esta dificultad que presentamos una interpretación de ATL y KM3, dando una semántica formal, precisa, concisa y tratable por máquina. La misma se formula en la Teoría Constructiva de Tipos, un formalismo diseñado para el desarrollo formal de la Matemática Constructiva.

En particular, la semántica se implementa como un traductor de ATL/KM3 a Agda, un lenguaje funcional con tipos dependientes. Al basar esta implementación en Agda es posible aprovechar algunas características del lenguaje para garantizar propiedades de manera común a todas las transformaciones, como el tipado correcto, la terminación y la confluencia. Se habilita además el uso de un ambiente con un alto grado de expresividad, en el que es posible codificar y demostrar propiedades para los modelos y transformaciones.

Finalmente, presentamos la implementación del traductor automático que permite representar meta-modelos, modelos y transformaciones existentes en la nueva interpretación. Esto permite utilizar transformaciones definidas para la implementación actual, sin necesidad de codificarlas bajo una nueva sintaxis. Durante el desarrollo de esta tesis describimos el proceso de traducción diseñado, e ilustramos su funcionamiento, uso y las ventajas que ofrece mediante su aplicación a un caso de estudio.

Palabras Clave

Transformacion de modelos, tipos dependientes, semántica formal.

Índice general

Capítulo 1

Introducción

Este capítulo describe la motivación y el contexto en el que se enmarca esta tesis. Se presenta la Ingeniería Dirigida por Modelos, algunos de los desafíos que presenta y nuestra propuesta para afrontarlos. Finalmente se incluye una descripción de la organización general de esta tesis.

1.1. Ingeniería dirigida por Modelos

Durante las últimas cinco décadas, los investigadores y desarrolladores de software han creado abstracciones que los ayudan a programar en función de su intención de diseño, en lugar del ambiente de computación subyacente (CPU, memoria, dispositivos de red, etc.), ayudándolos a aislarse de la complejidad de estos ambientes [1].

El término Ingeniería Dirigida por Modelos (MDE) se utiliza para describir enfoques de desarrollo de software en los cuales se crean modelos abstractos de sistemas de software, que son transformados de manera sistemática para obtener implementaciones concretas de los sistemas que representan.

Este enfoque se presenta como un enfoque prometedor para resolver la complejidad de las plataformas tecnológicas y expresar conceptos de dominio de manera efectiva. Propone el uso sistemático de modelos, considerados elementos de primera clase, durante el ciclo de desarrollo de software. Concretamente consiste en la definición de modelos de alto nivel e independientes de la tecnología, los que mediante la aplicación de sucesivas transformaciones se convierten en modelos dependientes de la tecnología, y finalmente, en forma ideal, en la implementación del sistema a construir.

El enfoque con base en los modelos cambia el centro de atención desde los lenguajes de programación hacia modelos expresados en lenguajes de dominio específico, con el objetivo de incrementar la productividad y reducir el tiempo de desarrollo. Esto se logra mediante la definición de conceptos menos atados a

la implementación tecnológica y más cercanos al dominio del problema, lo que hace a los modelos más fáciles de especificar, entender y matener [2], facilitando la comprensión de problemas complejos y sus soluciones mediante abstracciones. MDE promete una mejora en la productividad y la calidad del software generado debido a que se reduce el salto semántico entre el dominio del problema y el de la solución, se reducen los tiempos de desarrollo y las herramientas de generación pueden aplicar frameworks, patrones y técnicas con resultados conocidos.

El concepto de MDE surge como una generalización de la arquitectura dirigida por modelos (MDA), propuesta por el OMG (Object Management Group [3]) en el 2001 [4]. Kent [5] define MDE basado en MDA, agregando la noción de proceso de desarrollo de software y espacios de modelos. Favre [6] popone una visión de MDE donde MDA es simplemente una posible instancia de MDE, implementada en el conjunto de tecnologías del OMG. En términos generales, la principal diferencia es que MDA se basa en modelos independientes de la plataforma (PIM), transformándolos en modelos dependientes de la plataforma (PSM), mientras que MDE hace más foco en el proceso de desarrollo.

En MDE los modelos no son considerados solo documentación, sino artefactos precisos que pueden ser entendidos y manipulados por programas. En este contexto se vuelve clave el concepto de meta-modelado, entendido como una técnica para definir la sintaxis abstacta de los modelos y las relaciones entre sus elementos. El meta-modelado puede ser visto como la construcción de un conjunto de conceptos. Un modelo es la abstracción de un elemento de la realidad, al tiempo que un meta-modelo es otra abstracción que describe las propiedades del modelo. Se define la relación de conformidad entre un modelo y su meta-modelo en forma análoga a la conformidad entre un programa válido y la gramática del lenguaje de programación en el que fue escrito [7].

Además del meta-modelado, las transformaciones de modelos juegan un rol central en MDE. La guía de MDA [4] define una transformación de modelos como el proceso de convertir un modelo en otro modelo del mismo sistema. Kleppe et al. [8] definen una transformación como la generación automática de un modelo de salida a partir de un modelo de entrada, de acuerdo a una definición de transformación. La definición de una transformación consiste en un conjunto de reglas que definen la forma de construir un modelo de salida, a conformar con un meta-modelo de salida, a partir de un modelo de entrada, conformante con el mismo u otro meta-modelo. Una regla de transformación es la descripción de cómo una o más construcciones del meta-modelo de entrada se convierten en una o más construcciones en el meta-modelo de salida.

Existen diversas propuestas para especificar meta-modelos y modelos, y para especificar y ejecutar transformaciones entre modelos. Diversos trabajos se han encargado de catalogar y categorizar estas propuestas y sus características, siendo [9, 10, 11, 12] los más completos. Entre los lenguajes más utilizados para transformaciones de modelos aparecen: QVT [13], Tefkat [14] y ATLAS Transformation Language (ATL) [15]. Para este trabajo seleccionamos ATL, ya que aparece como el más utilizado en la bibliografía y el más popular en la industria. Como es usual para los lenguajes de transformación, ATL está definido de manera intuitiva e infomal, dando descripciones para sus características principales

en lenguaje natural.

Como se menciona en [16], a medida que la complejidad de las transformaciones de modelos crece, la necesidad de contar con semánticas formales para los lenguajes de transformación se vuelve un asunto crítico. La verificación de los modelos y sus transformaciones es crucial para poder mejorar la calidad y confiabilidad de los productos desarrollados utilizando el paradigma propuesto por MDE. Si bien las técnicas de verificación formal pueden ser costosas, la verificación tanto de modelos como de transformaciones puede ser clave en ciertos escenarios, en especial cuando la construcción de software se basa en su uso sistemático.

La implementación de referencia de ATL se encuentra disponible en la forma de meta-modelos para la descripción del lenguaje y su máquina virtual, implementaciones para un compilador entre el lenguaje y el código para la máquina virtual y un intérprete para esta última. El problema con este tipo de implementación es que no provee una semántica abstracta y significativa independiente de la implementación. Esto dificulta en extremo la utilización de métodos formales para razonar sobre las transformaciones.

1.2. Teoría Constructiva de Tipos

En este trabajo utilizamos la Teoría Constructiva de Tipos como lenguaje matemático para expresar la semántica de KM3 y ATL. La Teoría Constructiva de Tipos es un sistema fundacional para la matemática constructiva desarrollada por el lógico sueco Per Martin-Löf.

Siguiendo el isomorfismo de Curry-Howard [17], un teorema es representado mediante un tipo de datos y una prueba de este teorema es un objeto de ese tipo. Así, una prueba de un teorema es en general una función que, dadas pruebas de las hipótesis, computa una prueba de la tesis del teorema. Este sistema puede también ser usado como un lenguaje de programación, donde las especificaciones son representadas como tipos de datos y los programas como objetos de esos tipos.

El lenguaje es formal y el control de tipos decidible, por lo que tenemos una forma automática de chequear la corrección de las demostraciones y de los programas. Basados en esta característica, definimos una forma de codificar modelos y sus transformaciones, para en base al poder expresivo de la Teoría Constructiva de Tipos, razonar formalmente acerca de propiedades de modelos y las transformaciones.

Para la implementación de nuestro trabajo elegimos utilizar Agda [18, 19], un lenguaje de programación con tipos funcionales dependientes basado en la Teoría Constructiva de Tipos.

1.3. Propuesta y Aportes

Varios trabajos identifican la necesidad de contar con una semántica apropiada del lenguaje de transformación para poder demostrar propiedades de las transformaciones [12, 20, 16]. En el caso particular de ATL, la semántica se presenta en [21] de forma narrativa e informal, utilizando ejemplos y en forma incompleta.

En este trabajo investigamos el uso de la Teoría Constructiva de Tipos para dar una semántica formal a ATL. Esta teoría ofrece un marco potente para la formalización, con implementaciones probadas y usables. Agda en particular puede ser usado como lenguaje de programación o como asistente de demostración.

Como aportes principales de este trabajo identificamos:

- La construcción de una interpretación de ATL y KM3 dando una semántica formal, precisa, concisa y tratable por máquina, basada en el formalismo de la Teoría de Tipos. Creemos que esta interpretación se ajusta razonablemente bien a las descripciones informales existentes tanto para ATL como para KM3.
- La implementación de un traductor automático que permite modelar meta-modelos, modelos y transformaciones existentes en la nueva interpretación. Esto permite utilizar transformaciones definidas para la implementación actual, sin necesidad de codificarlas bajo una nueva sintaxis.
- Como resultado final se obtiene una implementación de ATL alternativa a la oficial. Al embeber esta implementación en Agda es posible aprovechar el *type checker* de este lenguaje para garantizar algunas características comunes a todas las transformaciones, como el tipado, la terminación y la confluencia.
- La interpretación generada habilita el uso un ambiente con un alto grado de expresividad, en el que es posible codificar y demostrar propiedades de los modelos y transformaciones en forma particular.
- La demostración de algunas propiedades generales, a modo de ejemplo, para esta implementación. Estas propiedades quedan certificadas independientemente de las transformaciones a ejecutar y los meta-modelos utilizados.
- Un ejemplo de uso concreto y válido a nivel industrial para la programación con tipos dependientes.

Si bien existen otros trabajos de formalización de la semántica para lenguajes de transformación de modelos con otros formalismos, no conocemos otra implementación funcional que utilice Teoría de Tipos como formalización subyacente.

1.4. Descripción del Desarrollo del Trabajo

Los próximos capítulos de este trabajo se estructuran de la siguiente manera:

- En el Capítulo 2 presentamos una descripción de los trabajos sobre semánticas formales para lenguajes de transformación encontrados en la bibliografía. Se analizan los formalismos propuestos y los resultados obtenidos, tanto para ATL como para otros lenguajes de transformación. Finalmente se realiza una búsqueda de revisiones sobre el estado del arte en la verificación de transformaciones de modelos, con el fin de identificar qué propiedades se verifican en trabajos existentes y que técnicas se utilizan.
- El Capítulo 3 presenta una introducción a KM3 y ATL, describiendo sus características principales y sintaxis. Se describe además en forma narrativa la semántica de ATL, extraída de su documentacioón.
- En el Capítulo 4 brindamos una introducción a la programación funcional con tipos dependientes. Presentamos aquí algunos detalles de Agda, el lenguaje de programación utilizado.
- En el Capítulo 5 describimos en detalle nuestra interpretación de ATL y KM3 en Agda. Presentamos el conjunto de reglas que permiten generar programas en Agda a partir de especificaciones KM3 y ATL. Se muestra además cómo instanciar modelos e incluso cómo ejecutar transformaciones.
- En el Capítulo 6 se presenta la interpretación aplicada a un caso concreto, y se analizan las posibilidades que ofrece la formalización de ATL y KM3 en Agda. El ejemplo, llamado *Class to Relational*, describe la transformación simplificada de un modelo de esquema de objetos (en el contexto de la programación orientada a objetos) a un modelo de esquema de base de datos relacional. Aquí se repasan los distintos tipos de propiedades a verificar encontradas en la revisión de trabajos existentes y se muestra cómo pueden ser verificadas siguiendo nuestra propuesta.
- El Capítulo 7 presenta algunas conclusiones y posibilidades de trabajo a futuro.
- El apéndice A se presenta una definición detallada de KM3.
- El apéndice B se presenta una definición detallada de ATL.
- El apéndice C presenta en forma completa la transformación del ejemplo utilizada a lo largo de la tesis.
- El apéndice D contiene el resultado completo de la traducción del ejemplo *Class to Relational* a nuestra interpretación de ATL.
- El apéndice E muestra, a modo de ejemplo, el código del traductor de archivos KM3 a la interpretación propuesta en Agda.

Capítulo 2

Trabajos Relacionados

En este capítulo presentamos una descripción de los trabajos sobre semánticas formales para lenguajes de transformación encontrados en la bibliografía. Se analizan los formalismos propuestos y en base a ellos, las capacidades de verificación obtenidas.

2.1. Semánticas Formales

Como se menciona en [20], debido a la falta de semánticas formales para los lenguajes de dominio específico para transformación de modelos, el significado real de estos lenguajes sólo se encuentra disponible en la implementación de sus distintos intérpretes. Como consecuencia de ello, no es posible verificar de manera formal distintos aspectos de los lenguajes y las transformaciones implementadas sobre ellos. Si bien existe una variada gama de lenguajes de transformación, y algunos han tenido un considerable éxito inicial, los mismos no ofrecen una especificación formal de sus semánticas o validación de transformaciones, lo que ha abierto la puerta a una serie de trabajos que ofrecen definiciones de semánticas formales, tanto para ATL como para otros lenguajes.

Una forma de definir la semántica de un lenguaje es establecer una traducción de sus expresiones en otro lenguaje con semántica bien definida [22]. Estas traducciones entre dominios semánticos son muy útiles, ya que además de proveer semántica, también permiten simular, analizar y razonar sobre los lenguajes utilizando el marco lógico y la semántica disponibles en el lenguaje de destino [23].

En [24] se propone una extensión a AMMA (ATLAS Model Management Architecture) para especificar la semántica dinámica de un amplio rango de DSLs mediante el uso de Máquinas de Estado Abstractas (ASMs), y se presenta un caso de estudio donde se define la semántica de un fragmento de ATL ("matched rules"). Si bien las ASMs presentan un alto poder de expresión, la naturaleza

declarativa de ATL no ayuda para lograr una semántica formal completa para el lenguaje con este formalismo, habiendo quedado de lado aspectos importantes del mismo. Aunque dicho trabajo propone una posible formalización de la semántica, no describe cómo la misma podría ser utilizada para analizar, razonar o verificar propiedades de las transformaciones, ya sea en forma general o particular.

Existen varios trabajos relacionados con gramáticas, definiciones lógicas y formalizaciones de transformaciones de grafos. Por ejemplo [25] propone una combinación de gramáticas de grafos con lógica monádica de segundo orden para estudiar las propiedades de los grafos y sus transformaciones. Siguiendo un enfoque similar, en [26] se estudia una formalización de la semántica de transformaciones de grafos en el lenguaje PROGRES. Si bien estos trabajos presentan formalizaciones adecuadas y distintas herramientas para la verificación de propiedades, los mismos se alejan bastante del lenguaje especifico de transformaciones, presentando en todos los casos procesos extremadamente complejos y no automatizados para representar el lenguaje de transformación, las transformaciones y los modelos mediante grafos.

Según [23], la formalización más común para las transformaciones de grafos es el llamado enfoque algebraico, el cual utiliza la teoría de categorías para expresar reescrituras. Este enfoque permite la utilización de ciertas técnicas interesantes de análisis, como la detección de dependencias entre reglas [27] o el cálculo de pares críticos (contexto mínimo de pares de reglas conflictivas) [28]. Sin embargo esta formalización se presenta limitada para otros tipos de análisis, como el de cobertura, chequeo de modelos, etc.

Para resolver dichas limitaciones, algunos autores definen un mapeo entre las transformaciones de grafos y otras formalizaciones, traduciendo desde y hacia cada una de ellas en función de la propiedad a validar o el aspecto a analizar. Se presentan como ejemplo de esta técnica los trabajos [29, 30, 31]. La principal desventaja de estos enfoques es que requieren, desde el punto de vista del constructor de la transformación, un conocimiento profundo de cada uno de los lenguajes de destino, sólo para poder implementar la transformación. Este problema puede ser solucionado (al menos en forma parcial) si las traducciones entre formalizaciones pueden ser automatizadas, tal como proponemos en este trabajo.

Otros trabajos ([32, 33]) proponen la utilización de Alloy para formalizar y analizar sistemas de transformaciones de grafos, y en base a ellos, ATL en particular. Alloy como lenguaje de programación presenta un sistema de tipos extremadamente simple, sólo con enteros como tipos primitivos, por lo que no es posible utilizar este enfoque para analizar propiedades que involucren otros tipos, como texto (Strings) o números reales. Los modelos en Alloy son además estáticos, por lo que el enfoque que se presenta en [33] sólo puede ser utilizado para razonar sobre propiedades estáticas de las transformaciones.

En [34] se define un lenguaje similar a QVT [13] reutilizando los conceptos principales de los sistemas de transformación de grafos. Las transformaciones se formalizan como teorías en lógica de reescritura, de forma de poder utilizar

las características de análisis y chequeo de modelos de Maude. Con este método sólo es posible representar una parte reducida de las posibles relaciones a especificar en QVT, y presenta además la desventaja de definir un nuevo lenguaje de transformaciones.

Siguiendo esta misma línea, Maude ha sido propuesto como notación formal para especificar y analizar modelos y meta-modelos [23, 35]. En particular, en [16] se presenta una implementación completa de ATL versión 3.0, siendo éste el trabajo más completo encontrado sobre el tema. Si bien en [16] se persiguen y obtienen los mismos resultados que en el presente trabajo, la técnica utilizada para la formalización es distinta, y el proceso de pasaje de ATL a Maude no es automático, por lo que la formalización debe ser construida en forma manual.

Finalmente, es posible encontrar trabajos que proveen semántica formal a lenguajes de transformación de modelos utilizando tipos. Este es el caso de [36] que utiliza Teoría Constructiva de Tipos, para formalizar transformaciones y verificar su corrección en relación a pre y post condiciones. Si bien la propuesta presentada en [36] presenta puntos de contacto con este trabajo, como el modelo matemático elegido para la formalización o incluso algunos aspectos de la implementación, el mismo se centra sólo en resolver los aspectos estáticos de las transformaciones, en particular la forma de representar los modelos y meta-modelos, y no en cómo codificar las transformaciones, ejecutarlas o razonar sobre ellas. En dicho trabajo la semántica del lenguaje de transformación sigue quedando librada a la implementación particular de las reglas para una transformación, por lo que no presenta una semántica general.

2.2. Verificación de Propiedades

Además de la definición de una semántica formal para los lenguajes de transformación, aparece en la literatura un amplio conjunto de trabajos orientados a la verificación, más o menos formal, de propiedades de transformaciones de modelos. Parece entonces interesante realizar una búsqueda de revisiones sobre el estado del arte en la verificación de transformaciones de modelos, con el fin de identificar qué propiedades se verifican en trabajos existentes y qué técnicas se utilizan.

Este tema es analizado en detalle en [37], dividiendo el problema en tres dimensiones: el lenguaje de transformaciones utilizado, las propiedades de interés a verificar, y finalmente las técnicas formales utilizadas para la verificación. Dicho trabajo es luego extendido por [38] tomando en cuenta algunos aspectos no cubiertos, en particular relacionados a las propiedades a verificar y las técnicas a utilizar, y concluyendo finalmente la necesidad de un ambiente integrado y heterogéneo para la verificación de transformaciones de modelos.

La primera de las dimensiones analizadas queda fijada para nuestro trabajo, ya que éste se basa en la formalización de la semántica de ATL, un lenguaje particular para especificar transformaciones de modelos. De todas formas se analizaron trabajos con formalizaciones relativas a otros lenguajes de transfor-

mación, con el objetivo de conocer su alcance y los formalismos utilizados.

2.3. Propiedades a Verificar

Independientemente del lenguaje utilizado para describir las transformaciones, es necesario analizar qué propiedades de las mismas son de interés para la verificación.

Existen varios trabajos que describen el problema de la verificación de transformaciones desde la perspectiva de las propiedades a verificar, como [39, 40] o el ya mencionado [37], aunque el contenido de estas propuestas se encuentra muy bien descripto y resumido en [38], por lo que a continuación se describe la categorización de propiedades allí presentada.

Esta clasificación categoriza las propiedades a verificar en dos grupos: por un lado aquellas dependientes del lenguaje de transformación utilizado y por el otro las que dependen de la transformación particular a ejecutar.

2.3.1. Independientes de la Transformación

El primer grupo identificado corresponde a aquellas propiedades que no dependen de una transformación particular, y por tanto pueden ser verificadas a nivel del lenguaje de transformación, lo que garantiza su cumplimiento para cualquier transformación implementada dentro del mismo. Por ejemplo, ciertos lenguajes imponen que las transformaciones necesariamente terminen. En el caso de ATL, si bien no se garantiza en su descripción, toda transformación debe terminar, ya que se ejecuta un conjunto finito de reglas, sobre un conjunto finito de elementos de entrada y solo es posible iterar sobre este último.

Consistentemente con ello, en el caso de nuestro trabajo, y dado que la implementación completa del lenguaje se ecuentra embebida en un lenguaje funcional con tipos dependientes, es posible garantizar las propiedades de terminación y confluencia para cualquier transformación válida codificada dentro de la implementación.

Otra de las propiedades identificadas es el tipado, que asegura que la especificación de la transformación se encuentra bien formada en función de los meta-modelos de entrada y salida. Esta propiedad es de especial valor para ATL, ya que el mismo no cuenta con chequeo de tipos en tiempo de diseño, siendo los errores de tipado sólo detectados en tiempo de ejecución. Nuevamente, en nuestro trabajo, el tipo de implementación del lenguaje de transformaciones (funcional con tipos dependientes) impone también reglas de tipado estrictas, las que se trasladan a las transformaciones codificadas utilizando el lenguaje, por lo que esta propiedad también queda garantizada.

2.3.2. Dependientes de la Transformación

El segundo grupo de propiedades analizadas está constituido por aquéllas dependientes de una transformación específica.

La primer propiedad a verificar en función de la transformación tiene que ver directamente con los modelos de entrada y salida. Como se especifica en [40], como mínimo se debe asegurar la correctitud sintáctica, que asegure que el modelo de salida conforme el meta-modelo de destino, validando condiciones estructurales y no estructurales.

Además de la conformidad, generalmente existen relaciones que involucran elementos de los meta-modelos de entrada y salida, asegurando que ciertos elementos o estructuras del modelo de entrada se transforman en otros elementos del modelo de salida. Surge un problema cuando este tipo de relaciones no se puede asegurar simplemente mirando las reglas de la transformación, o cuando los lenguajes de transformación no permiten especificar estas relaciones, por lo que algún otro lenguaje de restricciones debe ser utilizado.

Mas allá de relaciones estructurales entre los modelos de entrada y salida, es posible encontrar propiedades semánticas que se deben preservar, conocidas como "Model Semantics Relations" [40]. Estas propiedades generalmente dependen de la semántica de los meta-modelos o del tipo de transformación. Algunas propiedades de interés pueden ser la equivalencia semántica, la similitud débil y la preservación de propiedades generales o temporales.

Finalmente, en [38] se agrega una nueva categoría, no mencionada en [37], llamada Comportamiento Funcional, que refiere a la identificación de la transformación como una función matemática. De acuerdo a este trabajo, las transformaciones pueden ser inyectivas, suyectivas, biyectivas, o al menos ejecutables (existe un par de modelos de entrada y salida que satisfacen la transformación). También es posible analizar estas propiedades considerando reglas individuales de una transformación, un enfoque que se presenta en [30]. Mas aún, existe una propiedad específica, conocida como Completitud Sintáctica que refiere a la necesidad (en algunos casos) de cubrir completamente los modelos de entrada y salida con la transformación. Esta propiedad también se presenta en [39] bajo el nombre de Cobertura del Meta-modelo, presentando la problemática de que si la transformación no cubre totalmente el meta-modelo de entrada, entonces existen algunos modelos de entrada que no pueden ser transformados. Desde un punto de vista funcional, la completitud sintáctica establece que la transformación es una función total.

Considerado para una función específica, el determinismo es también una propiedad deterministica, lo que se introduce en [30], llamando transformación funcional a aquella que es total y deterministica.

Habiendo analizado las propiedades a verificar de interés encontradas en la literatura, es posible realizar el resumen que se presenta en la siguiente tabla:

Propiedades dependientes del Lenguaje	
Terminación	[41][42][43][44] [45][46][47]
Determinismo	[41][30][48][49] [44][50][45][47]
Tipado	[51][52][53]
Semántica Ejecución	[54][30][55] [56][57][47]
Propiedades dependientes de la Transformación	
Conformidad	[54][58][59] [60][61][62][63] [56][64][45][65][36] [66][67][47]
Rel. Sintácticas	[58][59][60][61] [62][68][55][69] [56][64][45][70] [71][36][66][72]
Rel. Semánticas	[73][74][34][75] [60][76][61][77] [78][79][80][45][65] [81][82][83][84][36] [85][66][67][40][68]
Comp. Funcional	[30][68][55][39] [45][57][86]

2.4. Métodos de Verificación

Una vez identificadas las propiedades de interés para verificar, es importante conocer qué métodos aparecen en la literatura para verificarlas, particularmente de manera formal. Para ello se toma como base la tercera de las dimensiones mencionadas en [37]: las técnicas de verificación formal utilizadas. Una revisión similar puede ser encontrada en [87], aunque planteada desde la perspectiva del testing y no de la verificación.

Ambos trabajos clasifican las técnicas utilizadas en cuatro categorías:

- el tipo de técnica formal
- el nivel de abstracción en relación a los elementos de la transformación
- el nivel de abstracción en relación a la implementación de la transformación
- la dependencia o independencia de la transformación específica

Dentro de la primera categoría, la que refiere al tipo de técnica utilizada para la verificación, aparece la inferencia lógica, que consiste en la utilización de una representación matemática del sistema y las propiedades a verificar, así

como una lógica que permite el razonamiento sobre dicha representación, yendo de premisas a conclusiones. Este proceso generalmente se lleva a cabo utilizando software para la prueba de teoremas, siendo generalmente automatizado en forma parcial. Dentro de esta categoría se identificaron los siguientes trabajos: [54], [58], [59], [73], [42], [88], [61], [89], [78], [56], [64], [45], [83], [36], [66] y [67].

Otra técnica ampliamente utilizada es el chequeo de modelos (*Model Checking*), el que también consiste en la utilización de una representación matemática de un sistema, y en el cual las pruebas consisten de una exploración sistemática y exhaustiva del modelo.

Finalmente es posible encontrar estrategias basadas en el análisis estático, las que se basan generalmente en técnicas de semi-decisión. En este sentido, las mismas son eficientes pero no pueden asegurar la correctitud total del diseño.

El presente trabajo se enmarca dentro del primero de los grupos identificados, ya que se basa en el uso de la formalización provista por la Teoría Constructiva de Tipos.

La segunda categoría hace referencia al nivel de abstracción en relación a los elementos involucrados en la transformación, siendo también referida como verificación en y fuera de línea [59], o dependiente e independiente de la entrada [37]. Es posible utilizar información del meta-modelo para verificar propiedades válidas para cualquier modelo que conforme el meta-modelo, así como utilizar modelos de entrada arbitrarios y demostrar propiedades que sólo se cumplen para ellos. Como se señala en [VP03], trabajar a nivel de meta-modelos requiere el uso de técnicas de prueba de teoremas y herramientas con un alto costo de verificación. Es por esta razón que el segundo enfoque, trabajando a nivel de modelos específicos puede ser una ayuda de valor en ciertos escenarios, aunque no puede verificar la ausencia total de errores, ya que sólo analiza un número finito de casos. Por otro lado, trabajando a nivel de modelos, con un nivel de abstracción más bajo, el rango de propiedades que pueden ser verificadas es mucho mayor que al trabajar a nivel de meta-modelos.

Nuestra propuesta permite razonar a nivel del lenguaje de manera independiente de los meta-modelos o de la transformación específica, ya que contiene una implementación del lenguaje. Es posible además razonar a nivel de meta-modelos y transformaciones puntuales, ya que las mismas son codificadas utilizando elementos del lenguaje huésped. Al mismo tiempo, y utilizando los elementos generados en el lenguaje huésped, es posible codificar modelos particulares y razonar sobre ellos. Es por estas tres características que consideramos que nuestra propuesta permite al desarrollador trabajar en cualquiera de los niveles identificados, seleccionando el que mejor se ajuste a las necesidades de verificación.

Finalmente, y como se describe en [77], la verificación puede ser realizada a nivel de modelo (especificación) o a nivel de implementación del motor de transformaciones. La verificación a nivel de la especificación involucra sólo la definición de la transformación en un lenguaje particular, y en consecuencia, la semántica definida para dicho lenguaje. Por otro lado, la verificación a nivel de

implementación requiere considerar también la forma en que la transformación será ejecutada por el motor de transformaciones. Luego de analizar la literatura disponible, la gran mayoría de las técnicas de verificación encontradas se encuentran en el primer grupo, ya que se asume que los motores de transformación son conformantes con la semántica del lenguaje de transformación, y por tanto las propiedades no dependen de cómo se ejecuta la transformación, incluyendo el caso del detererminismo, la terminación y la preservación de propiedades semánticas de ejecución.

En el caso de ATL esta premisa no parece ser válida, ya que como fue mencionado, el lenguaje no cuenta con una especificación formal de su semántica, la que está definida directamente en su implementación. La presentación de una semántica formal es uno de los objetivos de este trabajo.

Capítulo 3

ATL y KM3

Este capítulo presenta las conceptos utlizados para la formalización de los modelos y transformaciones en nuestro trabajo, así como descripciones detalladas de ATL y KM3.

3.1. Modelos y Meta-Modelos

Un modelo es una representación simplificada de un sistema que facilita su comprensión. La sintaxis abstracta para definir un modelo se define mediante un meta-modelo. Un meta-modelo es entonces un modelo que sirve para definir otros modelos, y puede ser visto como un análogo a las gramáticas en el diseño de lenguajes. Se define la relación de conformancia o *conformsTo* entre un modelo y un meta-modelo para indicar que el primero está correctamente definido en función del segundo. Es común la utilización del término "modelo de referencia" para referirse al meta-modelo de un modelo.

La sintáxis abstracta de los meta-modelos se define mediante un meta-meta-modelo. Un meta-meta-modelo es un modelo que es su propia referencia, es decir, conforma consigo mismo. Es usual en la bibliografía encontrar referencias al conjunto de los modelos como *M1*, al de los meta-modelos como *M2* y al de los meta-meta-modelos como *M3*.

KM3 es un ejemplo de meta-meta-modelo, ya que permite definir meta-modelos. Como se verá más adelante, uno de los conceptos básicos en KM3 es el concepto de clase, similar al concepto del mismo nombre en el paradigma de orientación a objetos. Un ejemplo de meta-modelo puede ser el meta-modelo relacional, que permite especificar tablas, columnas, relaciones y los demás elementos del dominio de las bases de datos relacionales. En la definición del meta-modelo relacional, es posible utilizar clases de KM3 para describir los conceptos de tabla, columna o relación. Luego, y en función del meta-modelo relacional, es posible instanciar un modelo particular, que partiendo de las definiciones de

tabla y columna podría definir una tabla de nombre “Persons” con columnas de nombre “name” y “lastName” La figura 3.1 ilustra el ejemplo.

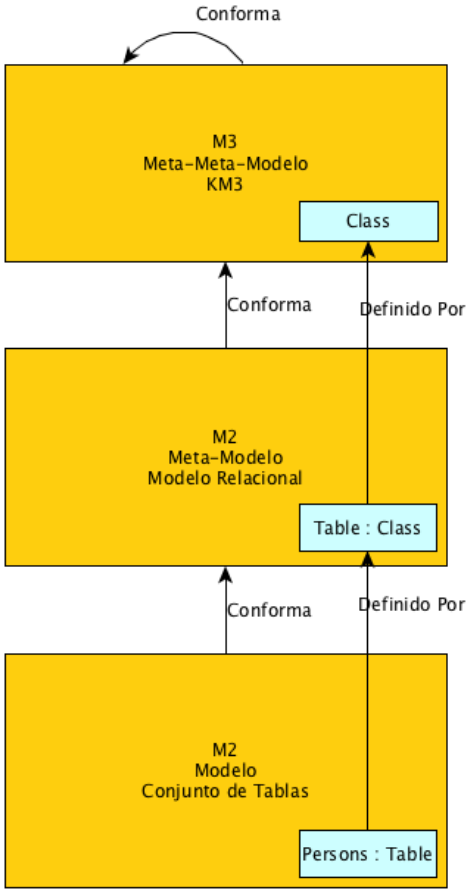

Figura 3.1: Relaciones de conformidad entre modelos

Las transformaciones de modelos se escriben en lenguajes específicos. La mayoría de estos lenguajes están basados en reglas, por lo que una transformación se especifica mediante un conjunto de reglas que describen cómo se transforma uno o más modelos de entrada en uno o más modelos de salida. Se suele nombrar los modelos de entrada para una transformación y sus meta-modelos de referencia como modelos y meta-modelos de origen y los de salida como modelos y meta-modelos de destino.

3.2. KM3

KM3, o Kernel MetaMetaModel, provee una sintaxis textual y concreta que facilita la definición de meta-modelos. Tomando como referencia las definiciones

presentadas en la sección anterior, KM3 es un meta-meta-modelo, conformante consigo mismo, por lo que se encuentra en el conjunto *M3*.

La definición de los meta-modelos se construye en base a las nociones de paquete, clase, atributo, referencia y tipos primitivos de datos, y sus características están fuertemente basadas en el paradigma de orientación a objetos. KM3 define las características de dichos elementos y las relaciones entre ellos mediante una estructura jerárquica. Además de la estructura, define una lista de restricciones no estructurales que deben ser cumplidas para que un meta-modelo se considere correctamente definido en KM3. En [90] se presenta una definición (incompleta) basada en predicados de lógica de primer orden con su correspondiente implementación en Prolog para demostrar que KM3 conforma consigo mismo.

La figura 3.2 provee una descripción gráfica de KM3 como meta-meta-modelo, describiendo los posibles tipos de elementos a definir y las relaciones estructurales entre ellos. Su corresponiente descripción textual y la definición de su sintaxis se encuentran disponibles en el Apéndice A.

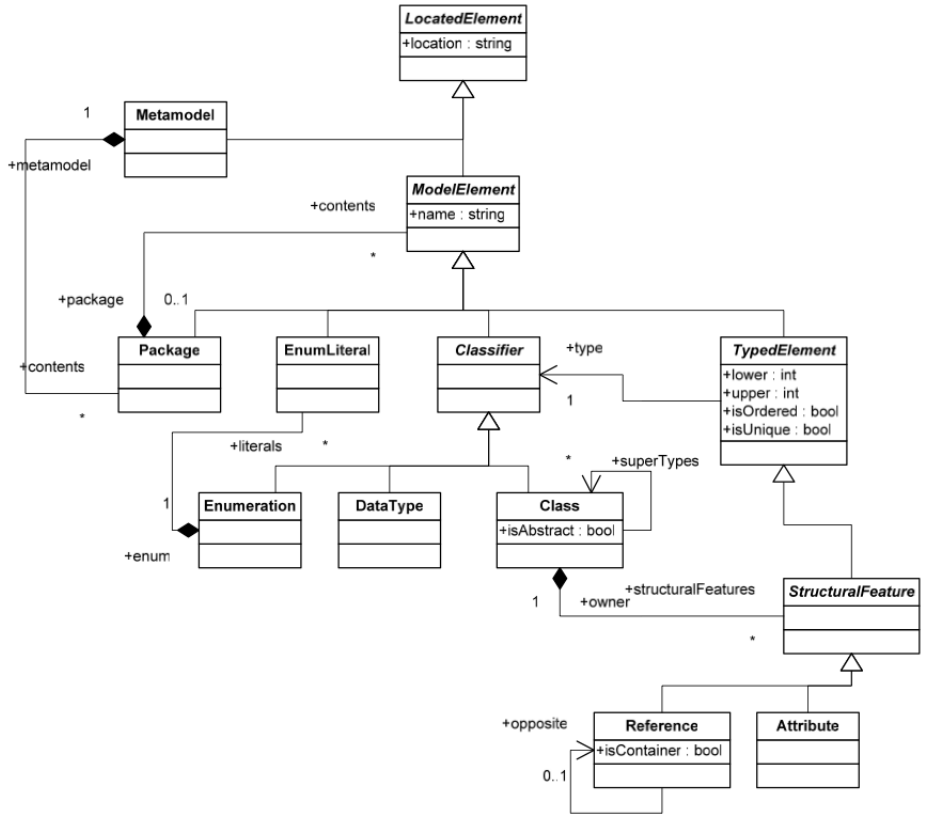

Figura 3.2: Descripción gráfica de KM3 [91].

Como se puede observar, los conceptos utilizados están inspirados en el paradigma de orientación a objetos, y su notación en UML. Se definen entidades para representar tipos de elementos de la realidad (*Class*, *DataType* y *Enu-*

meration), características para estas entidades (*Reference*, *Attribute*) y posibles relaciones (*supertype*, *owner*, *type*). También se definen elementos de agrupación y organización como *Model* y *Package*.

KM3 no provee un mecanismo adicional para expresar restricciones más allá de las definidas por su estructura en la definición del meta-modelo. Sin embargo, es posible especificar e implementar restricciones OCL en ATL, como se presentará más adelante. Además de la descripción estructural, el manual de KM3 [91] presenta el siguiente conjunto de restricciones, las que deben ser respetadas para considerar un modelo KM3 correcto:

- El nombre de un *Package* debe ser único dentro del *Package* que lo contiene.
- Todo *Classifier* debe pertenecer a un *Package*.
- Todo *EnumLiteral* debe pertenecer a un *Package*.
- El nombre de un *Classifier* debe ser único dentro del *Package* que lo contiene.
- Un elemento del tipo *Class* no puede estar contenido entre sus referencias *supertype* en forma directa o indirecta. Esto implica que no se permiten ciclos en las jerarquías de herencia.
- El nombre de todo *StructuralFeature* debe ser único dentro del elemento *Class* al que pertenece, así como en todos aquellos incluidos entre las referencias *supertypes* de ese elemento *Class*.
- La referencia *opposite* de un tipo *Reference* debe a su vez tener una referencia *opposite*, siendo ésta la primera de las referencias. En otras palabras, si una referencia A tiene opuesto, el opuesto del opuesto de la referencia A debe ser siempre la propia referencia A.
- El *type* del *opposite* de un elemento del tipo *Reference* debe ser el *owner* del propio *Reference*.
- El atributo *lower* de un *TypedElement* no puede ser menor que 0.
- El atributo *upper* de un *TypedElement* debe ser *unbounded* o mayor o igual a 1.
- El atributo *upper* de un *TypedElement* no puede ser menor que el atributo *lower*.
- El atributo *isOrdered* de un *TypedElement* no puede ser *true* si el atributo *upper* tiene valor 1.
- El atributo *type* de un *Reference* debe ser un *Class*.

3.3. ATL

ATL (ATLAS Transformation Language) es un lenguaje de dominio específico para describir transformaciones entre modelos. Es parte de la plataforma AMMA (ATLAS Model Management Architecture) y surge en respuesta a la solicitud de propuestas del OMG titulada MOF/QVT RFP [13], enfocándose específicamente en las transformaciones entre modelos y basado fundamentalmente en el formalismo de OCL (Object Constraint Language) [92]. La utilización de OCL se basa en su adopción alta en la comunidad MDE y el hecho de ser un lenguaje estándar soportado por el OMG y la mayoría de proveedores de herramientas de desarrollo.

Como característica principal, aparece su naturaleza híbrida, permitiendo la mezcla de construcciones imperativas y declarativas. Sus desarrolladores alientan el estilo declarativo de especificación de transformaciones [93], ya que al basarse en la especificación de relaciones entre los modelos de entrada y salida tiende a estar más cercana a la forma en la que los desarrolladores perciben en forma intuitiva las transformaciones. Más allá de la sugerencia, existen oportunidades en las que es realmente difícil ofrecer una solución completamente declarativa para un problema de transformaciones, casos en los cuales es posible ayudarse mediante las características imperativas del lenguaje.

Todas las transformaciones en ATL son unidireccionales, operando sobre un modelo de origen (entrada) que es de sólo lectura, y produciendo un modelo de destino (salida) que es de sólo escritura. Esto implica que durante la ejecución de una transformación el modelo de entrada solo puede ser navegado, pero no modificado. El modelo de salida solo puede ser modificado, pero no recorrido, navegado ni consultado. ATL no soporta en forma nativa transformaciones bi-direccionales, las que siempre deben ser implementadas como dos transformaciones, una para cada sentido, independientes.

El lenguaje permite la definición de tres tipos de unidades básicas: los módulos de transformación (*modules*), las consultas ATL (*queries*) y las bibliotecas (*libraries*), cada una de ellas pensada para una utilidad diferente, pero compuestas en general por los mismos elementos.

Los módulos permiten especificar transformaciones propiamente dichas, mediante la declaración de un encabezado mandatorio, sección de importaciones y cierto número de funciones (*helpers*) y reglas. El encabezado define el nombre del módulo y declara los modelos de entrada y salida mediante variables tipadas por meta-modelos:

```
module Class2Relational;
    create OUT : Relational from IN : Class;
```

Las *queries* consisten de expresiones declarativas ATL que permiten generar un valor a partir de un modelo de entrada, mientras que las librerías son conjuntos de elementos de ATL factorizados en un archivo. Ambos elementos tienen como finalidad facilitar el trabajo de los desarrolladores, permitiendo reutilizar

definiciones en distintas transformaciones.

A continuación se presentan los posibles componentes para cada una de estas unidades.

3.3.1. Tipos de Datos

El esquema de tipos de ATL está inspirado en el definido para OCL, aunque presenta algunas diferencias. La figura 3.3 presenta el esquema de tipos con sus relaciones de generalización.

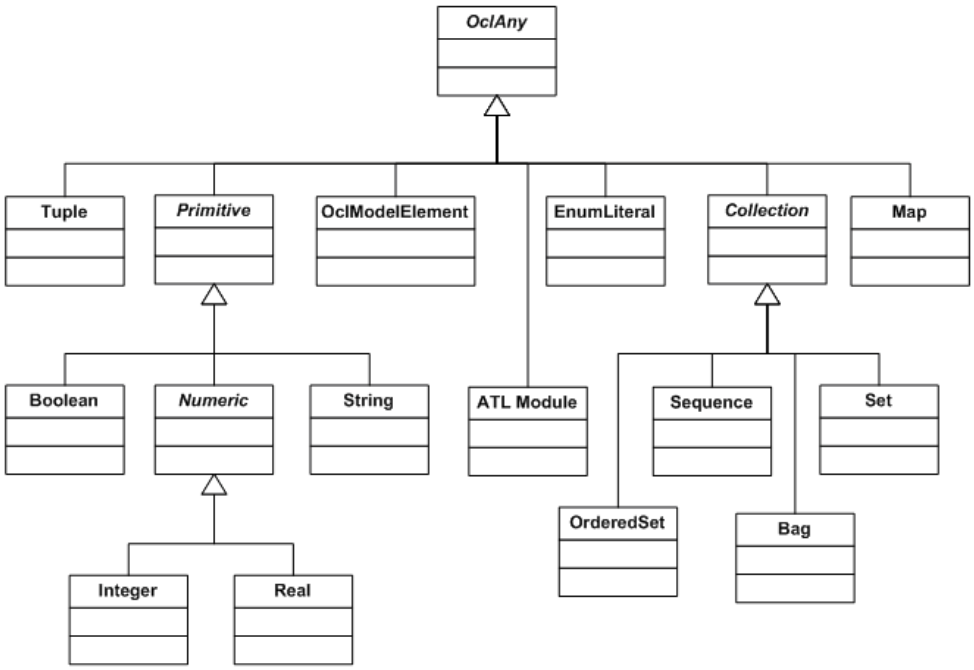

Figura 3.3: Tipos de ATL [21]

ATL presenta seis grupos principales de tipos de datos: primitivos, colecciones, tuplas, mapas, enumeraciones y elementos de modelo. Los mapas son una herramienta provista por ATL pero no surgen de la especificación de OCL siendo ésta una de las diferencias.

Toda expresión OCL, incluyendo aquellas asociadas a cada tipo de datos (y que se presentan más adelante junto al tipo de datos) se define en el contexto de un valor para un tipo de datos específico. Esto es, para poder ejecutar cualquier expresión es necesario invocarla sobre el valor que actuará como receptor del mensaje, y por tanto, como contexto. En ATL, de la misma manera que en OCL, la palabra reservada *self* se utiliza para referirse a dicho valor contextual. La única excepcion a esta regla es la operación allInstances(), aplicable no a un valor de un tipo sino directamente a un tipo, retornando un conjunto con todas las instancias (valores) existentes para dicho tipo.

La sintaxis utilizada para invocar una operación sobre una variable sigue la notación clásica del punto (.).

```
self.operation_name(parameters)
```

El anexo B presenta una descripción detallada de cada uno de los tipos de datos soportados por ATL y sus operaciones disponibles.

3.3.2. Expresiones Declarativas

Además de las expresiones declarativas correspondientes a las instancias de los tipos de datos y la invocación de operaciones sobre ellos, OCL define expresiones declarativas adicionales para permitir a los desarrolladores estructurar código.

La expresión *if* permite implementar la estructura *if-then-else* clásica. Como expresión, un *if* debe evaluar en cualquier caso, lo que significa que la sección *else* es obligatoria y no puede ser omitida. Todas las expresiones *if* deben cumplir con la siguiente sintaxis:

```
if condicion
then
    exp1
else
    exp2
endif
```

La condición debe ser una expresión booleana, y el resultado general de la expresión *if* corresponderá al resultado de *exp1* en caso de que la condición se evalúe como *true* y *exp2* en otro caso. Algunos ejemplos:

```
if 3 > 2
then
    'three is greater than two'
else
    'this case should never occur'
endif
```

```
if mySequence->notEmpty()
then
    if mySequence->includes(myElement)
    then
        'the element is at position '
        + mySequence->indexOf(myElement).toString()
```

```
    else
        'the sequence does not contain the element'
    endif
else
    'the sequence is empty'
endif
```

La expresión *let* permite la definición de variables de acuerdo a la siguiente sintaxis:

```
let var_name : var_type = var_init_exp in exp
```

El identificador *var_name* corresponde al nombre de la variable a declarar, mientras que *var_type* identifica su tipo. Una variable declarada mediante una expresión *let* debe ser inicializada mediante una expresión *var_init_exp* (expresión OCL de cualquier tipo), incluyendo sus expresiones *let* anidadas. Finalmente, la palabra clave *in* define la expresión a la que se ata (el alcance) de la variable definida. A modo de ejemplo:

```
let a : Integer = 1 in a + 1
```

Es posible combinar varias expresiones *let*, como lo muestra el siguiente ejemplo:

```
let x : Real =
    if aNumber > 0
    then
        aNumber.sqrt()
    else
        aNumber.square()
    endif

in let y : Real = 2 in x/y
```

A diferencia de la mayoría de los lenguajes de programación, la semántica de las expresiones compuestas tal y como se define en OCL, implica que las expresiones siempre son evaluadas en forma completa. Como consecuencia, algunas expresiones comunmente correctas generan errores en ATL, como lo ilustra el siguiente ejemplo:

```
not person.oclIsUndefined() and person.name = 'Isabel'
```

El que generará un error en el caso de que *person* sea indefinido.

3.3.3. Helpers

ATL permite a los desarrolladores definir funciones, llamadas *helpers*, en sus diferentes unidades. Permiten definir código ATL que puede ser luego utilizado desde distintos puntos de un programa. Los *helpers* se definen siguiendo el siguiente esquema:

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

Cada *helper* queda definido por su contexto (*context_type*), nombre (*helper_name*), conjunto de argumentos (*parameters*) y tipo de retorno (*return_type*). El contexto define el tipo de elementos sobre los que aplica, es decir, el tipo de elementos desde el que podrá ser invocado (tipo de *self*). El contexto puede ser omitido, en cuyo caso se asumirá el módulo ATL donde se encuentra definido como contexto, siendo *AtlModule* el tipo de *self*. La expresión *exp* es una expresión declarativa y constituye el cuerpo del *helper*, es decir, define el resultado del mismo en tiempo de ejecución.

El siguiente es un ejemplo de *helper* válido:

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

Este *helper*, llamado *averageLowerThan*, es definido en el contexto de un módulo ATL (ya que no se especifica contexto), y tiene como objetivo calcular un valor booleano indicando si el promedio de los valores incluidos en una secuencia es menor que un valor dado.

3.3.4. Reglas

En ATL, la generación de elementos del modelo de destino se logra mediante la espacificación de reglas. Estas son por lo tanto un elemento central del lenguaje. ATL define dos tipos distintos de reglas de transformación: las reglas *matched* y *called*. Una regla *matched* permite filtrar o seleccionar elementos del modelo de origen, y generar a partir de ellos, un determinado número de elementos del modelo de salida.

Las reglas *called* en cambio deben ser invocadas desde un bloque de código imperativo de ATL para poder ser ejecutadas. Un bloque iterativo puede ser definido tanto dentro del bloque de acción de una regla *matched* como en el cuerpo de una regla *called*.

Código Imperativo

Es posible utilizar código imperativo incluído en bloques dedicados, tanto desde las reglas *matched* como desde las *called*. Un bloque imperativo está compuesto por una secuencia de instrucciones imperativas, que al igual que en lenguajes como Java, C o C++ deben terminar con un punto y coma (";").

ATL provee tres tipos de sentencias iterativas: las asignaciones, los *if* y los *for*. A diferencia de las expresiones declarativas, estas instrucciones no devuelven ningún valor, por lo que no pueden ser utilizadas en expresiones declarativas.

Las asignaciones permiten definir valores, ya sea para atributos definidos dentro de un módulo o para caraterísticas del modelo de salida de una transformación. Su sintaxis es la siguiente:

```
target <- exp;
```

La sentencia *if* permite la ejecución condicional de bloques imperativos, siguiendo la sintaxis clásica:

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

A modo de ejemplo:

```
if(aPerson.gender = #male) {
    thisModule.menNb <- thisModule.menNb + 1;
    thisModule.men->including(aPerson);
}
```

Otro ejemplo:

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

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

La sentencia *for* permite definir secciones imperativas iterativas siguiendo la siguiente sintaxis:

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

A modo de ejemplo:

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

Reglas *Matched*

Estas reglas proveen a los desarrolladores un mecanismo conveniente para especificar la forma en que los elementos del modelo de destino deben ser generados a partir de los elementos del modelo de origen. Para ello una regla *matched* debe especificar qué elemento del modelo de origen debe ser utilizado, la cantidad de elementos de salida a generar, y la forma en que éstos deben ser inicializados en función del elemento de entrada identificado. La especificación de estas reglas sigue la siguiente sintaxis:

```
rule rule_name {
    from
        in_var : in_type [in model_name]? [(
            condition
        )]?
    [using {
        var1 : var_type1 = init_exp1;
        ...
        varn : var_typen = init_expn;
    }]?
    to
        out_var1 : out_type1 [in model_name]? (
            bindings1
```

```
        ),
        out_var2 : distinct out_type2 foreach(e in collection)(
            bindings2
        ),
        ...
        out_varn : out_typen [in model_name]? (
            bindingsn
        )
    [do {
        statements
    }]?
}
```

Cada regla de este tipo se identifica por su nombre (*rule_name*), que debe ser único dentro de la transformación. Cada regla está compuesta de dos partes obligatorias (*from* y *to*) y dos opcionales (*using* y *do*).

La sección *from* corresponde al patrón de origen, y especifica el tipo de los elementos del modelo de origen que serán tenidos en cuenta para esta regla. Además del tipo puede contener una expresión booleana (condición) que permita especificar un subconjunto de los elementos de dicho tipo a tener en cuenta. Esta condición recibe el nombre de guarda. Si la sección no contiene una condición explícita, todos los elementos del tipo *in_type* actuarán como elementos de entrada para la regla.

La sección opcional *using* permite declarar variables locales a la transformación. A modo de ejemplo:

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

La sección *to* describe el patrón de salida de la regla, mediante la declaración de un número predeterminado de elementos del meta-modelo de salida, los que pueden ser de dos tipos: simples o iterativos. Cada uno, independientemente del tipo, corresponde a una variable caracterizada por su nombre (*out_vari*) y su tipo (*out_typei*). Un elemento simple se especifica mediante un conjunto de enlaces que definen la forma en que se inicializarán sus caracerísticas (atributos y referencias), cada una de la forma:

```
feature_name <- exp
```

El nombre de la característica a inicializar (*feature_name*) debe referir a una característica definida para el tipo de elemento especificado. Se puede tomar como ejemplo la siguiente regla, con un patrón simple de salida:

```
rule Journal2Book {
    from
        j : Biblio!Journal
    to
        b : Biblio!Book (
            title <- j.title + '_(' + j.vol + '):' + j.num,
            authors <- j.articles
                    ->collect(e | e.authors)->flatten()->asSet()
            chapters <- j.articles,
            pagesNb <- j.articles->collect(e | e.pagesNb)->sum()
        )
}
```

A diferencia de las reglas con patrones de destino simples, los patrones de salida iterativos permiten generar un conjunto de elementos del modelo de salida, todos del mismo tipo. Los patrones de salida iterativos se definen mediante la palabra clave *distinct* y producen un elemento del meta-modelo de salida para cada elemento perteneciente a una colección ordenada (Sequence u OrderedSet) dada. Esta colección, junto con su iterador, es definida mediante la palabra clave *foreach*. Podemos tomar como ejemplo la siguiente regla:

```
using {
    coll : Sequence(String) = Sequence{'a', 'b', 'c'};
}
to
    cells : distinct Table!Cell foreach(e in coll)(
        content <- e,
        id <- coll->indexOf(e)
    )
```

La última sección de una regla *matched* es la sección opcional *do*, la que permite especificar una secuencia de instrucciones imperativas a ejecutar luego de inicializados los elementos del modelo de destino. Como ejemplo:

```
helper def : id : Integer = 0;
...
rule Journal2Book {
    from
        j : Biblio!Journal
    to
        b : Biblio!Book (
            ...
        )
    do {
        thisModule.id <- thisModule.id + 1;
        b.id <- thisModule.id;
    }
}
```

La semántica para las reglas *matched* implica que se ejecutan en función de su elemento de origen. Para cada coincidencia con el elemento de entrada en el modelo de entrada, los elementos especificados para el elemento de salida son creados y sus características inicializadas mediante los enlaces. Esta inicialización utiliza un algoritmo de resolución con la siguientes reglas:

- si el valor es un elemento del meta-modelo de salida, simplemente se asigna
- si el valor es un elemento del meta-modelo de entrada, primero se resuelve a un elemento de salida, siguiendo enlaces de trazabilidad con elementos creados por otra regla.

Esta resolución resulta en elementos del modelo de destino creados a partir del modelo de entrada por alguna regla dada, y permite que los elementos de destino se relacionen a partir de la navegación del modelo de entrada.

Esta es una caraterística bastante particular de ATL, y permite que al momento de escribir reglas sea posible asignar a propiedades de elementos de salida, valores de elementos de entrada. Esto es posible siempre y cuando exista una regla que convierta los elementos del tipo de entrada en elementos del tipo de salida.

ATL se encarga, durante la ejecución de una transformación, de mantener *links* entre los objetos generados (llamados links de trazabilidad), y es capaz de sustituir en una asignación un elemento de entrada por el elemento de salida generado para el.

Es posible identificar distintos tipos de reglas *matched*, en función de la forma en la que se disparan:

- Las reglas estándar son ejecutadas una vez por cada elemento coincidente en el modelo de entrada.
- Las reglas *lazy* son solo disparadas por otras reglas, y se identifican mediante la palabra clave *lazy*.
- Las reglas *lazy unique* también son disparadas por otras reglas, solo que se ejecutan una sola vez, y si vuelven a ser invocadas retornan siempre el mismo valor para el mismo elemento de origen.

```
lazy rule getCross {
    from
        i: ecore!EObject
    to
        rel: metamodel!Relationship (
    )
}
```

Cuando se ejecuta una regla *lazy unique*, siempre retorna el mismo elemento de salida para el mismo elemento de entrada, el que se consigue navegando los links de trazabilidad, del mismo modo que las reglas estándar.

Las reglas lazy no únicas no utilizan los links de trazabilidad, y crean un nuevo elemento de salida en cada ejecución.

Reglas *Called*

Además de las reglas *matched*, ATL define un tipo adicional de reglas que habilitan la generación explícita de elementos del modelo de salida desde código imperativo. Excepto por una única regla llamada *entrypoint*, este tipo de regla debe ser llamado en forma explícita desde una sentencia imperativa. Su especificación corresponde a la siguiente sintaxis:

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

3.3.5. Semántica de ATL

ATL presenta dos modos de ejecución, el modo por defecto y el de refinamiento.

En el modo de ejecución por defecto, toda transformación se ejecuta en tres fases:

1. Fase de inicialización del módulo.

2. Fase de coincidencia entre el modelo de entrada y las reglas.

3. Fase de inicialización de los elementos del modelo de salida.

Durante la fase de inicialización, se inicializan los atributos definidos a nivel de módulo, pudiendo para ello utilizarse atributos de elementos del modelo de entrada. Si se definió una *called rule* del tipo *entry point* en el módulo ATL, el código de esta regla (incluyendo la generación de elementos del módulo de salida) se ejecuta luego de la inicialización de los atributos del módulo.

Es durante la fase de coincidencia, que se evalúan las condiciones de coincidencia de las reglas contra los elementos del modelo de entrada. Cuando se verifica la coincidencia de un elemento con la condición de una regla, el motor ATL crea los elementos correspondientes al patrón de salida de la regla, pero no inicializa aún sus atributos.

La última fase corresponde a la inicialización de los elementos del modelo de salida creados durante la fase anterior. En esta fase, cada uno de los elementos de salida es inicializado mediante la ejecución del código de los enlaces definidos en la regla correspondiente. El código imperativo que puede ser definido en el alcance de una *matched rule* es ejecutado una vez que el paso de inicialización se completa. Este código imperativo puede disparar la ejecución de algunas *called rules* definidas dentro del módulo.

ATL define otro modo de ejecución, pensado para los casos en los que es necesario modificar el modelo de entrada, llamado modo de refinamiento. En este modo se debe especificar mediante reglas qué elementos del modelo de entrada se modifican y cómo. El resto de los elementos se copian directamente del modelo de entrada al modelo de salida sin modificaciones.

Nuestro trabajo ofece una implementación completa del modo por defecto.

Capítulo 4

Programación con Tipos Dependientes en Agda

Agda es el último desarrollo de una serie de implementaciones de la Teoría Constructiva de Tipos hechos en Gotemburgo que comenzó con ALF [94] en 1990. La versión actual (Agda 2) ha sido implementada por Ulf Norell.

Agda es un asistente de pruebas basado en la Teoría Constructiva de Tipos de Martin-Löf, presentada muy brevemente en el capítulo de introducción de esta tesis, y extendido con *pattern matching* [95]. Es un sistema interactivo para escribir y verificar pruebas, y a su vez, un lenguaje de programación con tipos dependientes. En la Teoría Constructiva de Tipos de Martin-Löf, las demostraciones de los teoremas son representadas como funciones que toman pruebas de las hipótesis y devuelven una prueba de la tesis. Agda asegura que los objetos construidos estén bien formados y bien tipados. Dado que las pruebas son objetos, el chequeo de tipos de objetos asegura así la correctitud de la demostración. Para una introducción más profunda sobre Agda sugerimos ver [18, 19].

4.1. Tipos y Funciones en Agda

Todo archivo Agda está compuesto por un módulo raiz que debe tener el mismo nombre que el archivo. Por ejemplo, el archivo *EjemploAgda.agda* debería comenzar con:

```
module EjemploAgda where
```

Los tipos Agda se definen mediante la declaración *data*, dando un nombre para el tipo y un conjunto de constructores con sus tipos, por ejemplo:

```
data Bool : Set where
```

```
  true  : Bool
  false : Bool
```

Agda presenta una jerarquía de tipos crecientes, en la que el tipo *Set* es el tipo de los tipos denominados pequeños, siendo *Set* del tipo *Set1*, *Set1* del tipo *Set2* y así sucesivamente. El tipo de *Bool* es entonces *Set*.

Las funciones pueden ser definidas utilizando *pattern matching*:

```
not : Bool -> Bool
not true  = false
not false = true
```

Los tipos son siempre definidos en forma inductiva (por definición de sus constructores). Un ejemplo claro es la definición del tipo *Nat*, para representar los números naturales.

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

Las funciones en Agda no pueden fallar, por lo que las definiciones deben cubrir todos los casos posibles. El *type checker* de agda controla que las mismas sean completas y genera un error si falta cubrir casos. El siguiente ejemplo es la suma para *Nat*, la que puede ser definida como una función recursiva:

```
_+_ : Nat -> Nat -> Nat
zero + m  = m
suc n + m = suc (n + m)
```

Al igual que están obligadas a ser completas, las funciones deben terminar. Para asegurar la terminación, las llamadas recursivas deben ser hechas sobre argumentos estructuralmente más chicos. En el ejemplo de $\_+\_$, el primer argumento en la llamada recursiva es de estructura menor, lo que permite que pase el chequeo de terminación.

Agda está basado en el cálculo lambda con tipos. La abstracción lambda puede ser escrita a la Curry:

```
\x -> e
```

sin etiquetas de tipo en el argumento x, o a la Church:

```
\(x : A) -> e
```

con etiqueta de tipo.

Es posible parametrizar los tipos en base a otros tipos. A modo de ejemplo, el tipo *List* para representar listas, parametrizado sobre el tipo de elementos a contener (A):

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A -> List A -> List A
```

En la definición anterior, el tipo A se convierte en un parámetro de la definición, por aparecer a la izquierda de los dos puntos. Esto lo convierte en un argumento en los constructores. Vemos ahora la definición de una función para calcular el largo para una lista formada por elementos de un tipo A cualquiera.

```
length : (A : Set) -> List A -> N
length A [] = 0
length A (x :: xs) = 1 + (length xs)
```

Es interesante notar que el parámetro A no tiene utilidad alguna en la implementación de la función, mientras que sí es necesario en la declaración de tipos de esta función. Por este motivo Agda define un mecanismo para omitir argumentos de este tipo, siempre y cuando estos puedan ser inferidos durante el chequeo de tipos.

Este mecanismo es conocido como *argumentos implicitos* y permite definir parámetros (entre llaves) que pueden ser luego omitidos, evitando la necesidad de abstraerlos con un término lambda. La definición de la función queda entonces:

```
length : {A : Set} -> List A -> N
length [] = 0
length (x :: xs) = 1 + (length xs)
```

Otro ejemplo mediante la función identidad:

```
id : {A : Set} -> A -> A
id = \x -> x
```

Tipos Dependientes: Los tipos dependientes son aquellos que dependen de valores de otros tipos. Un ejemplo es el tipo *Vec(A,n)* de listas de tipo A y largo n.

```
data Vec (A : Set) : Nat -> Set where
  []   : Vec A zero
  _::_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)
```

Es importante notar que el tipo de *Vec A* es $Nat \rightarrow Set$, lo que significa que *Vec A* es una familia de tipos indexados por los naturales. Par cada natural n, *Vec A n* es un tipo. Los constructores pueden crear elementos en cualquier tipo de la familia. en el ejemplo, *[]* construye elementos en *Vec A zero*, mientras que *_::_* crea elementos en *Vec A (suc n)* para un n dado.

Se diferencia la parametrización e indexación de los tipos, considerando que *Vec* está parametrizado en A (depende de un tipo) e indexado sobre los naturales (depende de un valor).

4.2. Programas como Pruebas

El sistema de tipos de agda es lo suficientemente expresivo para representar proposiciones como tipos, cuyos valores corresponden a pruebas de dichas proposiciones. Por ejemplo, dados dos naturales m y n, podemos definir la familia de pruebas de que m es menor o igual que n de la siguiente manera:

```
data _<=_ : Nat -> Nat -> Set where
  leq-zero : {n : Nat} -> zero <= n
  leq-suc : {m n : Nat} -> m <= n -> suc m <= suc n
```

El constructor *leq-zero* permite construir una demostración de $0 \leq n$ para cualquier n, mientras que el constructor *leq-suc* permite construir una demostración para dos naturales dados distintos de cero, partiendo de la demostración de sus predecesores. Esta representación permite por ejemplo escribir funciones haciendo *pattern match* sobre pruebas de proposiciones, lo que facilita la demostración de propiedades. El siguiente ejemplo presenta una demostración de la transitividad de la relación $\leq$:

```
leq-trans : {l m n : Nat} -> l <= m -> m <= n -> l <= n
leq-trans leq-zero _ = leq-zero
leq-trans (leq-suc p) (leq-suc q) = leq-suc (leq-trans p q)
```

La función *leq-trans* intenta construir una demostración de $l \leq n$ recibiendo como argumentos demostraciones de $l \leq m$ y $m \leq n$. Dadas las características del *type checker* de Agda, si la función tipa entonces la demostración es correcta.

4.3. Otras Características

A continuación se describen algunas otras características de Agda ampliamente utilizadas durante el desarrollo de esta tesis, con el fin de facilitar al lector la comprensión del código Agda que se presentará más adelante.

Maybe: El tipo *Maybe* es una familia de tipos indexada en *Set*, la cual para un tipo A cualquiera devuelve el tipo formado por los elementos de este tipo, a los que se les yuxtapone el constructor *just*, y el nuevo elemento *nothing*. Este último se utiliza para modelar en un conjunto cualquiera un valor indefinido.

Records: Los *records* son tipos que permiten agrupar valores, dando (opcionalmente) nombres a los mismos. La forma general para definir un *record* es:

```
record <name> <parameters> : Set <level> where
  constructor <operator>
  field
     <fname> : <ty>
     <fname> : <ty>
      ...
  <declaration>
  ...
```

Todos los componentes son opcionales y se pueden especificar en cualquier orden. Un ejemplo concreto de *record*:

```
record Table : Set where
  constructor newTable
  field name : String
        col : List Column
        key : List Column
```

Los mismos pueden ser creados utilizando un constructor, seguido de cada uno de los *fields*:

```
  newTable "hola" [] []
```

o en forma explícita:

```
 record { name = "hola"; col = []; key = []}
```

También ofrecen una sintáxis para actualizar sus valores, siendo esta simplemente una abreviación para una función que genera un nuevo valor con todos los campos iguales salvo los que se especifique:

```
changeName : Table -> String -> Table
changeName t s = record t { name = s }
```

Standard Library: Está disponible para Agda como una descarga por separado la biblioteca estándar. Esta biblioteca contiene tipos comunes, como String, Bool, List que son ampliamente utilizados a lo largo de este trabajo.

Capítulo 5

ATL y KM3 en Agda

Con el objetivo de verificar propiedades de los modelos y sus transformaciones en teoría de tipos, proponemos representaciones para los elementos de KM3 y las construcciones y semántica de ATL implementadas en Agda. Estas representaciones pueden ser construidas en forma automática por un compilador. Esto permitiría al desarrollador, partiendo de las definiciones originales, obtener una representación en teoría de tipos con la cual trabajar, pudiendo razonar sobre los elementos e incluso ejecutar la transformación.

Este capítulo describe el enfoque utilizado para representar meta-modelos expresados en KM3 (y sus modelos instanciados) y transformaciones expresadas en ATL mediante programas Adga, siguiendo un conjunto específico de reglas. Estas reglas fueron automatizadas en un compilador y el detalle de su implementación se puede ver en el Apéndice E

5.1. KM3

Para poder tipar modelos proponemos representar los meta-modelos expresados en KM3 como un conjunto de tipos en Agda. Los modelos pueden ser luego representados por conjuntos de valores de dichos tipos.

Las reglas para definir los tipos que representan un meta-modelo en Agda son las siguientes:

1. Para cada *package* en KM3 se genera un *module* en Agda con el mismo nombre, agregando el prefijo **Pkg**. El objetivo del prefijo es evitar nombres repetidos en Agda, ya que KM3 permite repetir nombres entre elementos de los tipos *Package* y *Classifier*, pero en Agda el nombre de cada elemento debe ser único. Los módulos en Agda se generan dentro de un módulo raíz con el nombre del meta-modelo. Para la siguiente definición de paquetes incluida en el meta-modelo Class.km3 en ATL:

```
    package Class {
      ...
    }

    package PrimitiveTypes {
      ...
    }
}
```

se genera la siguiente estructura de módulos Agda:

```
module RelationalKm3 where

  module PkgRelational where
  ...

  module PgkPrimitiveTypes where
   ...
```

Es importante notar que la gramática de KM3 (ver Apéndice A) no permite la anidación de paquetes.

2. Cada *Enumeration* del modelo KM3 se traduce en un *Set* (tipo) de Agda, definiendo un constructor para cada uno de los *EnumLiteral* que lo compone. Dada la siguiente enumeración en ATL:

```
enumeration Color {
    literal green;
    literal blue;
    literal orange;
}
```

se genera el siguiente tipo dentro del módulo correspondiente en Agda:

```
data Color : Set where
     green : Color
     blue : Color
     orange : Color
```

3. ATL no permite la definición de *DataTypes* más allá de los incluidos en su implementación, por lo que esta caraterística de KM3 no es utilizada. El mapeo entre los tipos de datos de ATL y la implementación en Agda es directo, utilizando los tipos definidos en la librería estándar de Agda. Para poder utilizarlos en cualquier modelo, se incluye una sentencia *open import* para cada tipo mapeado al comienzo de cada módulo:

```
module Class2Relational where
```

```
open import Data.Nat
open import Data.String
open import Data.Bool
open import Data.List
...
```

4. Las clases del meta-modelo KM3 se representan como registros (*records*) en Agda solo si son concretas. Para cada clase concreta se genera un *record* que lleva su nombre y un constructor que comienza con el prefijo *new*. Por ejemplo, para las clases del meta-modelo Relacional:

```
abstract class Named {
  ...
}

class Table extends Named {
  ...
}

class Column extends Named {
  ...
}

class Type extends Named {
  ...
}
```

se genera en Agda:

```
record Table : Set where
  constructor newTable

record Column : Set where
  constructor newColumn

record Type : Set where
  constructor newType
```

En el caso del ejemplo, no se genera una representación directa para la clase *Named* ya que al ser abstracta no es posible crear valores o instancias.

5. Un aspecto importante para la semántica de la ejecución de las reglas es la identidad o referencia de cada elemento. El termino identidad puede ser engañoso en este contexto, y refiere a la forma de saber si un valor ocupa la misma dirección de memoria que otro. A la hora de ejecutar transformaciones, es necesario conocer esta identidad, ya que dos elementos del modelo de destino pueden referenciar a un mismo tercer elemento, y es necesario saber si ya está generado o no a partir del elemento de origen.

Dado que los elementos de origen pueden tener todas sus características iguales, es clave poder decidir si dos elementos del modelo de entrada son iguales o son el mismo. Para simular esa identidad se agrega a todos los *records* que representan clases del modelo de origen un *field* de tipo *String* llamado *id*, que no debe repetirse entre elementos del mismo tipo y permite diferenciar elementos del modelo de entrada.

```
record Table : Set where
  constructor newTable
  field id : String

record Column : Set where
  constructor newColumn
  field id : String

record Type : Set where
  constructor newType
  field id : String
```

6. El mecanismo de herencia en KM3 presenta dos utilidades bien definidas: (1) permitir el reuso de atributos y características definidas en super clases y (2) tipar elementos en función de las super clases, de manera que en tiempo de ejecución sea posible utilizar un elemento de cualquier sub clase (polimorfismo en el contexto de orientación a objetos). La sobre escritura de características heredadas no está permitida.

   Nuestro enfoque para modelar la primera de las características (1) es incluir en la generación del *record* correspondiente a la sub clase todas las características (atributos y referencias) heredadas de todas sus super clases. Para modelar la segunda de las características (2) es necesario modelar el sub tipado en el sentido de la orientación a objetos. En la actualidad Agda no soporta sub tipos, por lo que es necesario modelarlo de manera indirecta.

   La forma elegida para modelarlo se basa en la definición de nuevos tipos representando los super tipos, cuyos constructores implementan en forma explícita la operación de upcast. Estos nuevos super tipos pueden ser luego utilizados para definir parámetros, atributos o referencias a valores de tipos que pueden ser reemplazados por valores de sub tipos, utilizando el constructor que representa al *upcast* específico.

   Esto implica que para cada clase en el modelo KM3 que tenga sub clases, es necesario generar un nuevo tipo, con constructores para crear valores del super tipo a partir de valores de los sub tipos. Esta operación es conocida en el contexto de orientación a objetos como *upcast*.

   Siguiendo el ejemplo, la clase *Named* tiene tres sub tipos (*Table*,*Column* y *Type*), lo que implica que cualquier referencia a valores de *Named* puede contener valores de sus sub tipos. Para modelarlo se genera el siguiente tipo:

```
data -Named : Set where
```

```
Named-From-Table : Table -> -Named
Named-From-Column : Column -> -Named
Named-From-Type : Type -> -Named
```

Al tratar un valor de una sub clase como un valor de la super clase, solo se tiene acceso a las características definidas en la super clase. Para poder acceder a esas características es necesario generar funciones en Agda que, a partir de un valor de una super clase (especificado por el constructor correspondiente), accedan a los valores. En el ejemplo, la clase *Named* define el atributo *name*, por lo que es necesaria la siguiente función:

```
Named_name : -Named -> String
Named_name (Named-From-Table t)  =  Table.name t
Named_name (Named-From-Column c) =  Column.name c
Named_name (Named-From-Type t)   = Type.name t
```

7. Los atributos y referencias de una clase se representan mediante *fields* cuyos tipos dependerán del tipo del atributo o referencia y de su multiplicidad (valores para *lower* y *upper*).

   Los atributos y referencias con multiplicidad 1 se representan como *fields*, cuyo tipo corresponde con la representación en Agda del tipo del atributo o referencia. Para tipos con sub tipos se utilza el *Set* que permite *upcasts*, mientras que para tipos sin sub tipos se utiliza el *record* generado. Los atributos y referencias con multiplicidad distinta de 1 se representan mediante valores del tipo *List* de la librería estándar de Agda.

   Se genera un *field* para cada uno de los atributos y referencias definidas en la clase y en cada una de sus super clases. Podemos tomar como ejemplo la clase *Table*:

```
class Table extends Named {
    reference col[*] ordered container : Column oppositeOf owner;
    reference key[*] : Column oppositeOf keyOf;
}
```

   para la que se genera en Agda:

```
record Table : Set where
  constructor newTable
  field field id : String
        ...
        name : String
        col : List Column
        key : List Column
```

   El atributo *name* en el ejemplo es heredado de la clase *Named*. Otro ejemplo interesante es la clase *Attribute*, que define un atributo *type* del tipo abstracto *Classifier*. Su definición en KM3 es:

```
class Attribute extends NamedElt {
    attribute multiValued : Boolean;
    reference type : Classifier;
    -- +: opposite of attr
    reference owner : Class oppositeOf attr;
}
```

y su representación en Agda:

```
record Attribute : Set where
  constructor newAttribute
  field id : String
        name : String
        multiValued : Bool
        type : -Classifier
        owner : Class
```

8. Dadas las características de Agda, y la posibilidad de que existan relaciones bidireccionales entre las clases que se modelan, es necesario incluir las definiciones de clases dentro de una instrucción *mutual*.

9. Finalmente, y para poder crear modelos (instancias del meta-modelo), es necesario contar con un tipo que represente al modelo en su conjunto, y que permita contener todos los elementos de un modelo. Para ello se genera en Agda un nuevo *record* con *fields* para contener los valores de cada una de las clases no abstractas del meta-modelo. En el caso del meta-modelo *Relacional*, el tipo que lo representa es:

   ```
   record RelationalModel : Set where
     constructor newRelationalModel
     field _Table : List Table
           _Column : List Column
           _Type : List Type
   ```

   También se agregan algunas funciones que implementan funcionalidad ofrecida por OCL. Un ejemplo claro es la función *allInstances()* de OCL, que retorna todas las instancias de un tipo dado. Por ejemplo:

   ```
   DataType_allInstances : ClassModel -> List DataType
   DataType_allInstances m = ClassModel._DataType m

   DataType_allInstances : ClassModel -> List DataType
   DataType_allInstances m = ClassModel._DataType m
   ```

La representación para el meta-modelo *Relacional* es entonces:

```
module RelationalKm3 where
```

```
module PkgRelational where
mutual

  data -Named : Set where
    Named-From-Table : Table -> -Named
    Named-From-Column : Column -> -Named
    Named-From-Type : Type -> -Named

  record Table : Set where
    constructor newTable
    field name : String
          col : List Column
          key : List Column

  record Column : Set where
    constructor newColumn
    field name : String
          owner : Maybe Table
          keyOf : List Table
          type : Maybe Type

  record Type : Set where
    constructor newType
    field name : String

  record RelationalModel : Set where
    constructor newRelationalModel
    field _Table : List Table
          _Column : List Column
          _Type : List Type

Named_name : -Named -> String
Named_name (Named-From-Table t)  =  Table.name t
Named_name (Named-From-Column c) =  Column.name c
Named_name (Named-From-Type t)   = Type.name t
```

Vale la pena aclarar que el paquete *PrimitiveTypes* del ejemplo no genera ninguna representación en Agda, ya que solo contiene *DataTypes*, y los mismos, como se menciona en el punto 3, son mapeados directamente a tipos de la librería estándar de Agda. Los paquetes sin contenido no se representan en el programa Agda.

5.2. ATL

Además de generar tipos para poder instanciar los meta-modelos, es uno de los objetivos principales de este trabajo implementar las transformaciones en

Agda, de forma de poder ejecutarlas o razonar sobre ellas, aprovechando así las capacidades del lenguaje con tipos dependientes. En esta sección se enumeran cada una de las construcciones válidas de ATL, detallando las reglas necesarias para representarlas en Agda. Finalmente, se describe la interpretación sugerida para la semántica de ATL y cómo se traduce a código Agda.

Esta traducción es realizada en forma automática por un compilador, según se muestra en la figura 5.1.

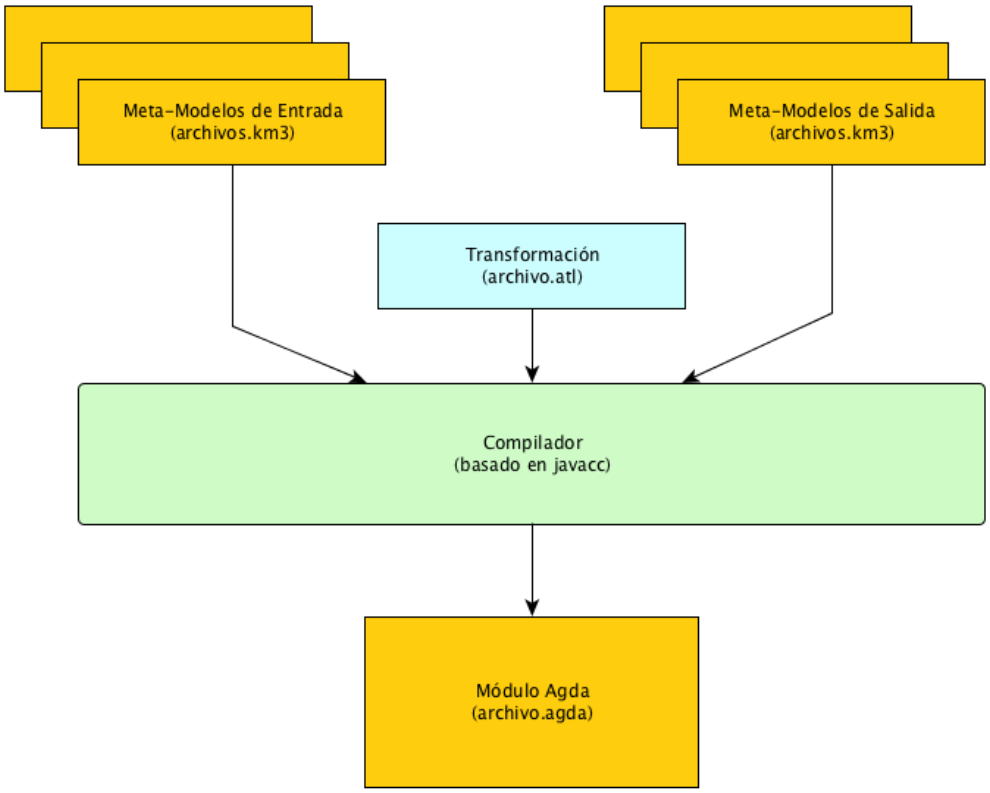

Figura 5.1: Generación Automática de código Agda.

Cada archivo conteniendo una transformación ATL (por ejemplo Class2Relational.atl) es transformado en un módulo Agda incluyendo:

- La representación el o los meta-modelos de entrada, siguiendo las reglas que se detallan en la sección anterior.
- La representación de el o los meta-modelos de salida.
- Un conjunto de funciones y tipos que representan funcionalidad de ATL y OCL y que no dependen de la transformación. Idealmente estas funciones y tipos deberían estar en una librería separada. En la implementación actual se copian a cada transformación.

- La traducción de todas las reglas definidas en la transformación.
- Un conjunto de tipos y funciones que implementan la semántica de ATL tal como la interpretamos para este trabajo. Estos tipos y funciones aparecen como muy útiles a la hora de razonar sobre la transtormación o demostrar propiedades.

A continuación presentamos una descripción detallada de cada uno de estos componentes y las reglas que se siguen para generarlos.

5.2.1. Tipos de Datos

Como se menciona en el Capítulo 3 y se detalla en el Apéndice B, ATL presenta un conjunto limitado de tipos primitivos. Estos tipos se representan en Agda mediante un mapeo a los tipos de la librería estándar de Agda. Para ello, es necesario que al comienzo de cada archivo que representa una transformación se incluya una secuencia de sentencias *open import*.

```
module Class2Relational where

open import Data.Nat
open import Data.String
open import Data.Bool
open import Data.List
open import Data.Maybe
...
```

5.2.2. Expresiones Declarativas

ATL se basa fuertemente en expresiones OCL y define un amplio conjunto de operaciones disponibles para los tipos de datos. Un buen número de estas operaciones fueron implementadas de manera funcional en Agda, y se insertan al comienzo de cada transformación como nuestra implementación de OCL. Si bien para este trabajo no se implmentó la totalidad de las expresiones y funciones soportadas por ATL, se incluyó un número considerable, entre las que se encuentran todas aquellas que se utilizan en los ejemplos probados. A modo de ejemplo, todo archivo Agda generado agrega al comienzo del módulo, y luego de la importación de los tipos primitivos, la implementación de OCL embebida en Agda, que comienza así:

```
------------------------------
--  ATL & helper functions
------------------------------
```

```
-- Operations on List

_->first : { A : Set } -> List A -> Maybe A
_->first []  = nothing
_->first (x   xs) = just x

_->firstOrDefault_ : { A : Set } -> List A -> A ->  A
_->firstOrDefault_ [] a = a
_->firstOrDefault_ (x   xs)  a = x


_->select_ : {A : Set} -> List A -> ( A -> Bool) -> List A
_->select_ l p = Data.List.filter ( \e -> p e) l

_->union_ : {A : Set} -> List A -> List A -> List A
_->union_ l1 l2 with l1
... | [] = l2
... | (x   xs) = (x   (xs ->union l2))

Sequence : {A : Set} -> A -> List A
Sequence a = (a   [])

...

-----------------------------
-- Operations on Strings
-----------------------------

_+'_ : String -> String -> String
_+'_ a b = Data.String._++_ a b

...

...
```

Esta implementación de OCL permite luego construir las expresiones declarativas traduciendo directamente desde ATL. A modo de ejemplo, la expresión:

```
if mySequence->notEmpty()
then
    'the secuence is NOT empty'
else
    'the sequence is empty'
endif
```

Se traduce a Agda de la siguiente manera:

```
if mySequence ->notEmpty() then
    "the secuence is NOT empty"
else
    "the sequence is empty"
endif
```

gracias a las funciones tomadas de la implementación de OCL. En este caso el *if then else* se mapea directo con la función definida en *Data.Bool* de la biblioteca estándar de Agda, y la función *notEmpty* tiene la siguiente implementación:

```
_->notEmpty : {A : Set} -> List A -> Bool
_->notEmpty [] = false
_->notEmpty _  = true
```

5.2.3. Helpers

Cada *helper* se traduce mediante la creación de una función. El primer parámetro de la función es el contexto (*self*) sobre el que aplica el *helper*. Luego se agrega el resto de los parámetros, y el tipo de retorno. El contenido del *helper* se traduce siguiendo las reglas para las expresiones.

Para el *helper*:

```
helper def: objectIdType : Relational!Type =
    Class!DataType.allInstances()->select(e | e.name = 'Integer')->first();
```

se genera:

```
objectIdType : Memory ->  ClassModel -> Maybe Type
objectIdType mem cm =   (DataType_allInstances cm)
                        ->select( \e -> e.name == "Integer")->first();
```

Si bien el *helper* original y su función correspondiente parecen tener diferencias, es importante tener en cuenta varios aspectos. Primero, el *helper* no define contexto, por lo que su contexto es el módulo entero de transformación. Por esta razón la implementación en Agda recibe como parámetro una memoria *Memory* (tipo que describe en detalle más adelante), y un ClassModel, representando el modelo de origen. Vale la pena recordar que el modelo de destino en ATL es de solo escritura, por lo que los *helpers* y relgas no tienen acceso a él. Segundo, la función *DataType_allInstances* devuelve todos los elementos del tipo *DataType* en el *record* que representa al modelo. Al generar los tipos se genera una función igual para cada uno.

La función *ClassModel._DataType_allInstances* se encuentra definida dentro del módulo que representa al modelo de entrada, y devuelve todas las instancias de dicho tipo contenidas en el modelo:

```
DataType_allInstances : ClassModel -> List DataType
DataType_allInstances m = ClassModel._DataType m
```

5.2.4. Código Imperativo

ATL provee tres tipos de instrucciones imperativas: las asignaciones, los *ifs* y los *for*. Estas instrucciones pueden ser agrupadas en bloques y ejecutadas como parte de una regla. La implementación de cada una de estas instrucciones es la siguiente:

- **Asignaciones:** Las asignaciones permiten definir valores, ya sea para atributos definidos dentro de un módulo o para caraterísticas del modelo de salida de una transformación. Los atributos definidos dentro de un módulo son la "memoria" de ATL mientras ejecuta una transformación. Con el objetivo de representar esa memoria, al generar cada módulo se genera un *record* llamado *Memory*, que entre otras cosas contiene *fields* para cada uno de los atributos del módulo, y provee operaciones para cambiar su valor.

  En el contexto de nuestro programa funcional, cambiar un valor de memoria significa generar un nuevo valor del tipo *Memory* con todos sus valores internos iguales, salvo el que se desea modificar.

  Por ejemplo, si el módulo de la transformación define una variable *Integer* llamada *contador*, para contar cuántas veces se ejecuta una regla, el *record Memory* será:

  ```
  record Memory : Set where
    constructor newMemory
    field contador :
          ...
  ```

  y la función para modificar la variable:

  ```
  _<-contador_ : Memory ->  -> Memory
  m <-contador c = record m { contador = c }
  ```

- **if:** La sentencia *if* permite la ejecución condicional de bloques imperativos. Se modela en Agda de la siguiente manera:

  ```
  _if_then_ : Memory -> Bool -> Memoy -> Memory
  m if c then m1 with c
  ... | true = m1
  ... | false = m

  _if_then_else_ : Memory -> Bool -> Memoy -> Memory -> Memory
  m if c then m1 else m2 with c
  ```

```
... | true = m1
... | false = m2
```

Un ejemplo interesante de bloque imperativo es:

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

y genera:

```
...
m if ((Person.gender aPerson) == "male") then
      (m <-fullName "Mr. "" +' (Person.name aPerson) +'
         " "  +' (Person.suname aPerson))
  else (
      (if ((Person.isSingle aPerson) then
        ( m <-fullName (" Miss "" +' Person.name) aPerson)
          <-surname (Person.surname aPerson)
      else
        ( m <-fullName (" Mrs "" +' Person.name) aPerson)
          <-surname (Person.surname (Person.marriedTo aPerson))  )
      <-fullName ( (Memory.fullName m) +' " " + (Memory.surname m))
    )
```

- **for:** Esta sentencia es aplicable solo en el contexto de colecciones:

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

y se interpreta en Agda como:

```
for : {A : Set} -> Memory -> List A  -> (Memory -> A -> Memory) -> Memory
for m [] f = m
for m (x  xs) f = for m1 xs f
                  where m1 = f m x
```

5.2.5. Reglas y Semántica de ATL

Todas las reglas ATL se definen mediante dos componentes obligatorios, el *from*, que identifica los elementos del modelo de origen y el *to*, que especifica los elementos a generar en el modelo de destino. Las reglas se interpretan en Agda mediante una función, teniendo como argumentos y tipo los tipos generados para los elementos del meta-modelo. Por ejemplo, para la regla:

```
rule DataType2Type {
    from
        dt : Class!DataType
    to
        out : Relational!Type (
            name <- dt.name
        )
}
```

se genera una función tipada de la siguiente manera:

```
DataTypeAttribute2Column :  Memory -> Attribute -> Column
```

Todas las funciones que representan reglas toman como primer argumento un valor del tipo *Memory*, lo que les da acceso a los atributos definidos a nivel de módulo. Esto permite además, en caso de que la regla especifique la sección opcional *do*, ejectuar código imperativo que acceda a la memoria. En el caso del ejemplo, el cuerpo de la regla tiene una traducción muy directa, ya que simplemente crea un valor del tipo de destino, inicializado con valores tomados del valor de entrada.

```
DataType2Type : Memory -> DataType -> Type
DataType2Type m d =  record { name = DataType.name d }
```

ATL presenta dos opciones al momento de definir los valores para los elementos de salida de una regla:

- La opción más sencilla es inicializar los valores de salida con valores de tipos del modelo de destino. Podemos tomar como ejemplo la regla presentada anteriormente, donde el resultado de la regla es un valor de un tipo del meta-modelo de destino (*Type*):

  ```
  DataType2Type : Memory -> DataType -> Type
  DataType2Type m d =  record { name = DataType.name d }
  ```

- Otra opción es asociar al resultado valores de elementos de entrada, como en el siguiente ejemplo:

```
rule DataTypeAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!DataType) and not a.multiValued
        )
    to
        out : Relational!Column (
            name <- a.name,
            type <- a.type
        )
}
```

En este caso, al atributo *type* del *Column* que se está creando, que es del tipo *Type*, se le asigna el valor del atributo *type* de un *Attribute*, que es del tipo *DataType*. Esto parecería una inconsistencia, ya que se intenta asignar a una variable de un tipo perteneciente al meta-modelo de salida, un valor de un tipo perteneciente al meta-modelo de entrada. Esto es aceptado por ATL, y como los tipos no coinciden, primero se debe resolver el valor de *a.type* a un elemento del tipo *Type*, siguiendo enlaces de trazabilidad con elementos creados por otra regla.

Esto implica que las reglas deben ser ejecutadas en orden, ya que existen dependencias entre ellas. Además agrega la necesidad de registrar enlaces de trazabilidad. Para el caso del ejemplo, antes de ejecutar la regla *DataTypeAttribute2Column* se debe ejecutar *DataType2Type*. Además se deben guardar links que relacionen cada *DataType* con el *Type* que se generó a partir de él. Es aquí donde entra en juego el concepto de "identidad" definido anteriormente, y donde además se utiliza la memoria de la transformación.

Para almacenar los links de trazabilidad, el tipo que representa la memoria contiene listas de links que relacionan la identidad el elemento de origen con el valor del elemento de destino generado. Cada *link* es representado por un valor del siguiente tipo, parametrizado en el tipo de elemento a generar (la identidad siempre es un *String*):

```
record Link(A : Set) : Set where
  constructor [_,_]
  field from : String
        to   : A
```

Además es necesario que la memoria contenga una lista para cada par de tipos de origen y destino posible:

```
record Memory : Set where
  constructor newMemory
  field DataType_Type : List (Link Type)
        Attribute_Column :  List (Link Column)
        Attribute_Table : List (Link Table)
        Class_Table : List (Link Table)
```

Se generan además funciones para navegar estos links, por ejemplo:

```
lookUpDataTypeType : Memory -> String -> Maybe Type
lookUpDataTypeType (newMemory [] _ _ _) id = nothing
lookUpDataTypeType (newMemory (x   xs) _ _ _)  id
  with (Link.from x) == id
... | true = just (Link.to x)
... | false = lookUpDataTypeType (newMemory xs [] [] []) id
```

Finalmente, la regla presentada anteriormente termina siendo traducida como:

```
DataTypeAttribute2Column :  Memory -> Attribute -> Column
DataTypeAttribute2Column m a = record {
                  name = Attribute.name a;
                  owner = nothing;
                  keyOf = [];
                  type = lookUpDataTypeType
                              m
                              (Classifier_id (Attribute.type a))
                }
```

- Independientemente del tipo de asociación, y en caso de que la regla especifique una guarda (condición o filtro para los elementos de entrada), la misma es implementada en ATL con la forma de una función

  ```
  gRuleName : A -> Bool
  ```

  donde A es el tipo de origen declarado para la regla.

La forma general para ejecutar las reglas es la siguiente:

```
r : { A B : Set } -> (m : Memory) -> List A ->
                     ( A -> String) -> ( Memory -> A -> B)
                     -> List (Link B)
r  _ [] _ _ = []
r  m (x   xs) id rule = [ id x , rule m x ]  (r m xs id rule)
```

Todas las reglas se ejecutan utilizando la función *r*, que recibe la memoria en su estado actual, una lista de elementos (aquellos que cumplen las condiciones de entrada a la regla), una función que permite a partir de un elemento de la lista obtener su identidad *id* y la función que especifica la regla de la transformación *rule*.

Para cada elemento en la lista se ejecuta la función correspondiente a la regla y se genera un *Link*, que no es más que una tupla que asocia la identidad del elemento de entrada con el elemento de salida generado. El resultado de ejecutar

la regla es un conjunto de *links*, que luego son agregados a la memoria de la transformación.

En el caso de que la regla tenga guarda (especificación de condiciones que los elementos del tipo de entrada deben cumplir para pasar por la regla), la misma se ejecuta utilizando la función *r'*, muy similar, pero recibiendo la función filtro como parámetro y filtrando la lista de entrada:

```
applyGuard : {A : Set} -> (A -> Bool) -> List A -> List A
applyGuard  guard [] = []
applyGuard  guard (x   xs)  with guard x
... | true = x   ( applyGuard guard xs)
... | false =  ( applyGuard guard xs)
```

```
r' : { A B : Set } -> (m : Memory) ->  List A -> ( A -> String) ->
                   (Memory -> A -> B) -> ( A -> Bool ) ->
                   List (Link B)
r' m list id rule guard = r m (applyGuard guard list) id rule
```

En forma general, el algoritmo para implementar una transformación concreta es el siguiente:

1. Se generan los tipos necesarios para representar los meta-modelos de origen y destino.
2. Se agrega al archivo la implementación de las funciones de ATL y OCL.
3. El compilador analiza sintácticamente el archivo ATL de entrada, identifica las reglas y construye una tabla identificando tipos de origen y destino para cada regla.
4. A partir de la tabla construida se genera un *record* llamado *Memory* con *fields* para almacenar todos los *links* de transformación.
5. En función de las inicializaciones (*bindings*) que se encuentren dentro de las reglas, se establece un orden de precedencia para la ejecución.
6. Se genera el código ATL para representar cada regla, incluida su guarda.
7. Se genera el código para ejecutar las reglas en el órden calculado, utilizando las funciones *r* o *r'* segun corresponda, y guardando el resultado de la ejecución en la memoria.
8. Se toman de la memoria los elementos creados y se popula con ellos el modelo de salida.

Todos estos pasos son realizados en forma automática por el compilador, fijando la semántica pre definida por nuestra interpretación de ATL en la implementación de esta transformación en particular. En el Capítulo 6 se detalla

este proceso para un ejemplo concreto, y se analizan las posibilidades que ofrecen los resultados.

Capítulo 6

Caso de Estudio

Para ilustrar mejor el enfoque propuesto presentamos un ejemplo conocido de transformación de modelos, llamado *Class2Relational*, que transforma modelos de clases (desde una perspectiva orientada a objetos) a modelos de bases de datos relacionales. Este es un ejemplo muy conocido y utilizado en la bibliografía relacionada a ATL, y puede ser encontrado junto a otros en [96].

Siguiendo el proceso definido en el capítulo anterior, el primer paso consiste en generar la representación de los meta-modelos de entrada y salida en Agda.

6.1. Meta-Modelos

La figura 6.1 muestra la representación gráfica del meta-modelo de Clases, en este caso el de entrada para la transformación.

El código en KM3 es:

```
package Class {

abstract class NamedElt {
attribute name : String;
}

abstract class Classifier extends NamedElt {
}

class DataType extends Classifier {

}

class Class extends Classifier {
```

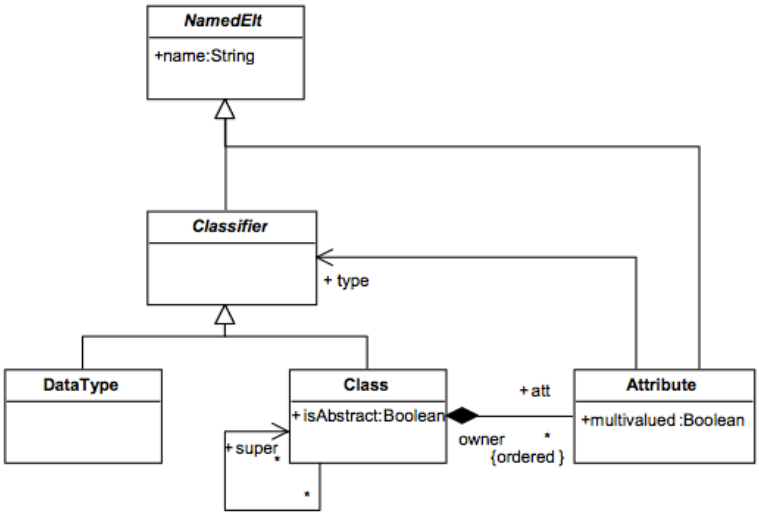

Figura 6.1: Meta-modelo de entrada (Clases)

```
reference super[*] : Class; -- -: no sub reference
reference attr[*] ordered container : Attribute oppositeOf owner;
attribute isAbstract : Boolean;
}

class Attribute extends NamedElt {
attribute multiValued : Boolean;
reference type : Classifier;
reference owner : Class oppositeOf attr; -- +: opposite of attr
}
}
```

Siguiendo los pasos descriptos se genera el siguiente módulo:

```
module ClassKm3 where

module PkgClass where
mutual

  data -NamedElt : Set where
    NamedElt-From-Classifier : -Classifier -> -NamedElt
    NamedElt-From-DataType : DataType -> -NamedElt
    NamedElt-From-Class : Class -> -NamedElt
    NamedElt-From-Attribute : Attribute -> -NamedElt

  data -Classifier : Set where
    Classifier-From-DataType : DataType -> -Classifier
    Classifier-From-Class : Class -> -Classifier
```

```
  record DataType : Set where
    constructor newDataType
    field id : String
          name : String

  record Class : Set where
    constructor newClass
    field id : String
          name : String
          super : List Class
          attr : List Attribute
          isAbstract : Bool

  record Attribute : Set where
    constructor newAttribute
    field id : String
          name : String
          multiValued : Bool
          type : -Classifier
          owner : Class

  record ClassModel : Set where
    constructor newClassModel
    field _DataType : List DataType
          _Class : List Class
          _Attribute : List Attribute

DataType_allInstances : ClassModel -> List DataType
DataType_allInstances m = ClassModel._DataType m


Classifier_id : -Classifier -> String
Classifier_id (Classifier-From-DataType d) = DataType.id d
Classifier_id (Classifier-From-Class c) = Class.id c

_->typeOclIsKindOfDataType : Attribute -> Bool
_->typeOclIsKindOfDataType (
                          newAttribute
                          id
                          n
                          m
                          (Classifier-From-DataType d)
                          o
                            )  = true
_->typeOclIsKindOfDataType _ = false

_->typeOclIsKindOfClass : Attribute -> Bool
_->typeOclIsKindOfClass (
                    newAttribute
```

```
                    id
                    n
                    m
                    (Classifier-From-Class d)
                    o
                    )  = true
_->typeOclIsKindOfClass _ = false
```

Es importante destacar que el modulo generado es mucho más largo. Se eliminan del ejemplo las funciones relacionadas con la herencia e implementaciones de métodos de OCL que no son de utilidad para esta explicación.

La figura 6.2 muestra las representación gráfica del meta-modelo de salida (Relacional). Siguiendo los mismos pasos, se genera el siguiente módulo:

```
module RelationalKm3 where

module PkgRelational where
mutual

  data -Named : Set where
    Named-From-Table : Table -> -Named
    Named-From-Column : Column -> -Named
    Named-From-Type : Type -> -Named

  record Table : Set where
    constructor newTable
    field name : String
          col : List Column
          key : List Column

  record Column : Set where
    constructor newColumn
    field name : String
          owner : Maybe Table
          keyOf : List Table
          type : Maybe Type

  record Type : Set where
    constructor newType
    field name : String

  record RelationalModel : Set where
    constructor newRelationalModel
    field _Table : List Table
          _Column : List Column
          _Type : List Type

Named_name : -Named -> String
```

```
Named_name (Named-From-Table t)  =  Table.name t
Named_name (Named-From-Column c) =  Column.name c
Named_name (Named-From-Type t)   = Type.name t
```

Nuevamente se eliminan los elementos generados que no son de relevancia para el ejemplo. Hay que recordar también que el concepto de "identidad" solo aplica a los elementos del modelo de entrada.

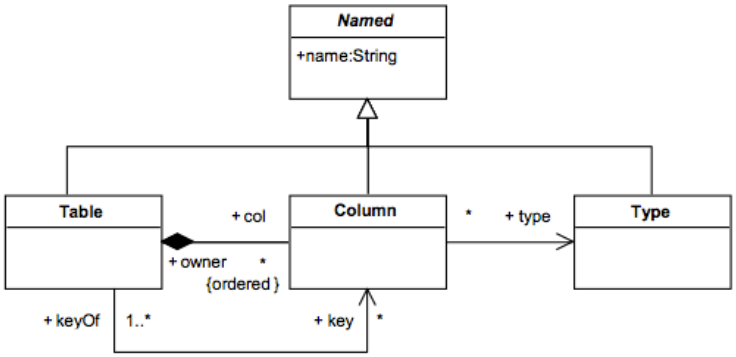

Figura 6.2: Meta-modelo de salida (Relacional)

6.2. Transformación

Luego de generar los módulos correspondientes a los meta-modelos de origen y destino, el compilador identifica las reglas y conforma una tabla que asocia tipos de elementos de entrada con tipos de elementos de salida, a partir de las entradas y salidas de cada regla.

En este caso la tabla queda compuesta de la siguiente manera:

Regla	Tipo de Entrada	Tipo de Salida
DataType2Type	DataType	Type
DataTypeAttribute2Column	Attribute	Column
MultiValuedDataTypeAttribute2Column	Attribute	Table
ClassAttribute2Column	Attribute	Column
MultiValuedClassAttribute2Column	Attribute	Table
Class2Table	Class	Table

Esta tabla determina los tipos de registros que es necesario registrar en la memoria para mantener los *links*. El tipo *Link* es independiente de la transformación, ya que está parametrizado en función del tipo:

```
record Link(A : Set) : Set where
  constructor [_,_]
```

```
  field from : String
        to   : A
```

La memoria deberá guardar colecciones de *Links* para cada combinación de tipos de entrada y salida de la tabla. Queda constituida entonces de la siguiente manera:

```
record Memory : Set where
  constructor newMemory
  field DataType_Type : List (Link Type)
        Attribute_Column :  List (Link Column)
        Attribute_Table : List (Link Table)
        Class_Table : List (Link Table)
```

Cada *field* del tipo *Memory* permite almacenar relaciones entre el identificador del elemento origen (*String*) y el elemento de destino.

Se generan además funciones para trabajar con la memoria, representando operaciones de lectura, búsqueda y escritura:

```
addDataTypeTypeLinks : Memory ->  List (Link Type) -> Memory
addDataTypeTypeLinks m l = record m {
                             DataType_Type =
                                 Data.List._++_ l (Memory.DataType_Type m)}

addAttributeColumnLinks : Memory ->  List (Link Column) -> Memory
addAttributeColumnLinks m l = record m {
                             Attribute_Column =
                           Data.List._++_ l (Memory.Attribute_Column m)}

addAttributeTableLinks : Memory ->  List (Link Table) -> Memory
addAttributeTableLinks m l = record m {
                             Attribute_Table =
                                 Data.List._++_ l (Memory.Attribute_Table m)}

addClassTableLinks : Memory ->  List (Link Table) -> Memory
addClassTableLinks m l = record m {
                             Class_Table =
                                 Data.List._++_ l (Memory.Class_Table m)}


lookUpDataTypeType : Memory -> String -> Maybe Type
lookUpDataTypeType (newMemory [] _ _ _) id = nothing
lookUpDataTypeType (newMemory (x   xs) _ _ _)  id with (Link.from x) == id
... | true = just (Link.to x)
... | false = lookUpDataTypeType (newMemory xs [] [] []) id

lookUpAttributeColumn : Memory -> String -> Maybe Column
```

```
lookUpAttributeColumn (newMemory _ [] _ _) id = nothing
lookUpAttributeColumn (newMemory _ (x   xs)  _ _)  id with (Link.from x) == id
... | true = just (Link.to x)
... | false = lookUpAttributeColumn (newMemory [] xs [] []) id

lookUpAll : {A B : Set} -> (m : Memory) ->
                           List A ->
                           (A -> String) ->
                           (Memory -> String -> Maybe B)
                           -> List B
lookUpAll m [] _ _ = []
lookUpAll m (x   xs) id lookUp with lookUp m (id x)
... | nothing = lookUpAll m xs id lookUp
... | just y =  y   (lookUpAll m xs id lookUp)
```

El siguiente paso consiste en determinar el orden en que deben ejecutarse las reglas. Para encontrarlo se analiza las llamadas implícitas a otras reglas que cada una realiza, estableciendo un orden de precedencia. En este caso la tabla ordenada queda igual a la presentada, dado que aparecen las siguientes dependencias:

- La regla *DataType2Type* no depende de otras.
- Las reglas *DataTypeAttribute2Column*, *MultiValuedDataTypeAttribute2Column*, *ClassAttribute2Column* y *MultiValuedClassAttribute2Column* dependen de la primera (*DataType2Type*), ya que necesitan encontrar el *Type* correspondiente a un *DataType*.
- La regla *Class2Table* depende de todas las anteriores.

Una vez determinado el orden, se genera la implementación de cada regla en Agda, incluída su guarda en los casos que sea necesario.

- Para **DataType2Type**

  ```
  rule DataType2Type {
      from
          dt : Class!DataType
      to
          out : Relational!Type (
              name <- dt.name
          )
  }
  ```

  Se genera:

  ```
  -- Rule DataType2Type --
  DataType2Type : Memory -> DataType -> Type
  DataType2Type m d =  record { name = DataType.name d }
  ```

- Para **DataTypeAttribute2Column**

```
rule DataTypeAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!DataType) and not a.multiValued
        )
    to
        out : Relational!Column (
            name <- a.name,
            type <- a.type
-- explicit use of implicit tracking links
-- (first expected syntax, then present actual syntax)
--owner <- [Class2Type.key]a.owner
--owner <- thisModule.resolveTemp(a.owner, 'key')
}
```

Se genera:

```
-- Rule DataTypeAttribute2Column ----
DataTypeAttribute2Column :  Memory -> Attribute -> Column
DataTypeAttribute2Column m a = record { name = Attribute.name a;
                                        owner = nothing;
                                        keyOf = [];
                                        type = lookUpDataTypeType
                                    m
                                    (Classifier_id (Attribute.type a))
                                     }

-- guard --
gDataTypeAttribute2Column :  Attribute -> Bool
gDataTypeAttribute2Column a = a ->typeOclIsKindOfDataType
not (Attribute.multiValued a)
```

- Para **MultiValuedDataTypeAttribute2Column**

```
rule MultiValuedDataTypeAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!DataType) and a.multiValued
        )
    to
        out : Relational!Table (
            name <- a.owner.name + '_' + a.name,
            col <- Sequence {id, value}
        ),
        id : Relational!Column (
            name <- a.owner.name.firstToLower() + 'Id',
            ype <- thisModule.objectIdType
```

```
        ),
        value : Relational!Column (
            name <- a.name,
            type <- a.type
)
}
```

Se genera:

```
-- Rule MultiValuedDataTypeAttribute2Column --
MultiValuedDataTypeAttribute2Column : Memory -> Attribute -> Table
MultiValuedDataTypeAttribute2Column m a =
record {
                              name = (Class.name (Attribute.owner a)) +'
                               "_" +' (Attribute.name a) ;
                              col = id   value   [] ; key = []}
                    where id = record {
                              name = Class.name (Attribute.owner a) +' ("Id")  ;
                              type = objectIdType;
                              keyOf = [];
                              owner = nothing }
                           value =  record {
                                          name = Attribute.name a;
                                          type =  lookUpDataTypeType m
                                           (Classifier_id (Attribute.type a)) ;
                                          keyOf = [];
                                          owner = nothing
                                    }
-- guard --
gMultiValuedDataTypeAttribute2Column :  Attribute -> Bool
gMultiValuedDataTypeAttribute2Column a = a ->typeOclIsKindOfDataType
(Attribute.multiValued a)
```

- Para **ClassAttribute2Column**

```
rule ClassAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!Class) and not a.multiValued
        )
    to
        foreignKey : Relational!Column (
            name <- a.name + 'Id',
            type <- thisModule.objectIdType
        )
}
```

Se genera:

```
-- Rule ClassAttribute2Column --
ClassAttribute2Column : Memory -> Attribute -> Column
ClassAttribute2Column  m a = record {
                                    name = Attribute.name a +' "Id" ;
                                    type = objectIdType;
                                    keyOf = [];
                                    owner = nothing
                               }
-- guard --
gClassAttribute2Column :  Attribute -> Bool
gClassAttribute2Column a = a ->typeOclIsKindOfClass
not (Attribute.multiValued a)
```

- Para **MultiValuedClassAttribute2Column**

```
rule MultiValuedClassAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!Class) and a.multiValued
        )
    to
        t : Relational!Table (
            name <- a.owner.name + '_' + a.name,
            col <- Sequence {id, foreignKey}
        ),
        id : Relational!Column (
            name <- a.owner.name.firstToLower() + 'Id',
            type <- thisModule.objectIdType
        ),
        foreignKey : Relational!Column (
            name <- a.name + 'Id',
            type <- thisModule.objectIdType
        )
}
```

Se genera:

```
-- Rule MultiValuedClassAttribute2Column --
MultiValuedClassAttribute2Column : Memory -> Attribute -> Table
MultiValuedClassAttribute2Column m  a =
                    record {
                          name =  Class.name (Attribute.owner a) +'
                           "_"  +' Attribute.name a ;
                          col = id   foreignKey   []; key = []}
                where id = record {
                                    name = Class.name (Attribute.owner a) +'
                                      "Id"  ;
                                    type = objectIdType;
                                    keyOf = [];
```

```
                                    owner = nothing }
                        foreignKey =  record {
                                        name = Attribute.name a +' "Id";
                                        type = objectIdType;
                                        keyOf = [];
                                        owner = nothing}
-- guard ---
gMultiValuedClassAttribute2Column :  Attribute -> Bool
gMultiValuedClassAttribute2Column a = a ->typeOclIsKindOfClass
(Attribute.multiValued a)
```

- Para **Class2Table**

```
rule Class2Table {
    from
        c : Class!Class
    to
        out : Relational!Table (
            name <- c.name,
            -- Columns are generated from Attributes in
            -- another rule not explicitly called here !
            col <- Sequence {key}->union(c.attr->select(e | not e.multiValued)),
            key <- Set {key}
        ),key : Relational!Column (
            name <- 'objectId',
            type <- thisModule.objectIdType
        )
}
```

Se genera:

```
-- Rule Class2Table ---
Class2Table : Memory -> Class -> Table
Class2Table m c =
                record {
                   name = Class.name c;
                   col = (Sequence key) ->union(
                    lookUpAll m
                    ((Class.attr c)
                          ->select( \e -> not (Attribute.multiValued e)))
                    Attribute.id lookUpAttributeColumn  )  ;
                   key = Sequence key
                 }
                 where key = record {
                                      name = "objectId";
                                      type = objectIdType;
                                      owner = nothing;
                                      keyOf = []
                                 }
```

Una vez generadas las funciones correspondientes a las reglas, es posible combinarlas para ejecutar la primera fase de la transformación, es decir, crear los elementos del modelo de destino. En este paso nuevamente es importante el orden. La ejecución de las reglas para el ejemplo se codifica de la siguiente manera:

```
Class2Relational_phase1 : ClassModel -> Memory
Class2Relational_phase1 m = addClassTableLinks m6
  (r                                              -- Regla 6, sin guarda (r)
    m6                                            -- Memoria actual
    (ClassModel._Class m)                         -- Elementos de Entrada
    Class.id                                      -- Identificador
    Class2Table                                   -- Funcion de Regla
  )
  where m6 = addAttributeTableLinks m5
    (r'                                           -- Regla 5, con guarda (r')
      m5                                          -- Memoria actual
      (ClassModel._Attribute m)                   -- Elementos de Entrada
      (Attribute.id)                              -- Identificador
      MultiValuedClassAttribute2Column            -- Funcion de Regla
      gMultiValuedClassAttribute2Column           -- Guarda
    )
    where m5 = addAttributeColumnLinks m4
      (r'                                         -- Regla 4, con guarda (r')
        m4                                        -- Memoria actual
        (ClassModel._Attribute m)                 -- Elementos de Entrada
        (Attribute.id)                            -- Identificador
        ClassAttribute2Column                     -- Funcion de Regla
        gClassAttribute2Column                    -- Guarda
      )
      where m4 = addAttributeTableLinks m3
        (r'                                       -- Regla 3, con guarda (r')
          m3                                      -- Memoria actual
          (ClassModel._Attribute m)               -- Elementos de Entrada
          (Attribute.id)                          -- Identificador
          MultiValuedDataTypeAttribute2Column     -- Funcion de Regla
          gMultiValuedDataTypeAttribute2Column    -- Guarda
        )
        where m3 = addAttributeColumnLinks m2
          (r'                                     -- Regla 2, con guarda (r')
            m2                                    -- Memoria actual
            (ClassModel._Attribute m)             -- Elementos de Entrada
            (Attribute.id)                        -- Identificador
            DataTypeAttribute2Column              -- Funcion de Regla
            gDataTypeAttribute2Column             -- Guarda
          )
          where m2 = addDataTypeTypeLinks m1
            (r                                    -- Regla 1, sin guarda (r)
```

```
  m1                                  -- Memoria acual
  (ClassModel._DataType m)            -- Elementos de Entrada
  (DataType.id)                       -- Identificador
  (DataType2Type)                     -- Funcion de Regla
)
where m1 = newMemory [] [] [] []      -- Memoria inicial
```

El paso final para completar la transformación es tomar los elementos generados de la memoria y pasarlos al modelo de destino. Para eso hacemos uso de las funciones generadas automáticamente para leer la memoria:

```
Class2Relational_phase2 : Memory -> RelationalModel
Class2Relational_phase2  m = record {
                                    _Table = Table_allInstances m;
                                    _Column = Column_allInstances m;
                                    _Type = Type_allInstances m
                                    }

Class2Relational : ClassModel -> RelationalModel
Class2Relational m = Class2Relational_phase2 (Class2Relational_phase1 m)
```

La última función generada lleva el nombre de la transformación, y su tipo es $A \rightarrow B$ donde A es el tipo que representa el meta-modelo de entrada y B el tipo que representa el meta-modelo de salida.

6.3. Modelos

Dado que los meta-modelos se encuentran representados por tipos de Agda, es posible definir valores de estos tipos, para así instanciar el modelo. El ejemplo original brinda el siguiente modelo (instancia del meta-modelo de Clases) de guía:

```
package Sample {

  datatype String;
  datatype Integer;

  class Family {
    attribute name : String;
    attribute members[*] : Person;
  }

  class Person {
    attribute firstName : String;
    attribute closestFriend : Person;
```

```
    attribute emailAddresses[*] : String;
  }
}
```

Utilizando los tipos generados, podemos instanciarlo de la siguiente forma:

```
$integer = newDataType "1" "Integer"
$string = newDataType "2" "String"

$person_aux = newClass "1"
                "Person" [] [] false
$firstName =  newAttribute "1"
                "firstName" false
                (Classifier-From-DataType $string)
                $person_aux

$closestFriend =  newAttribute "2"
                    "closestFriend" false
                    (Classifier-From-Class $person_aux)
                    $person_aux

$emailAddresses =  newAttribute "3"
                    "emailAdresses" true
                    (Classifier-From-DataType $string)
                    $person_aux

$person = record $person_aux {
                               attr =  $emailAddresses
                                   $closestFriend
                                   $firstName
                                   Class.attr $person_aux
                             }

$family_aux = newClass "2"
                "Family" [] [] false

$name = newAttribute "4"
            "name" false
            (Classifier-From-DataType $string)
            $family_aux

$members = newAttribute "5"
                "members" true
                (Classifier-From-Class $person)
                $family_aux

$family = record $family_aux {
                               attr =  $name
                                   $members
```

```
                                Class.attr
                                $family_aux
                            }

$model =  newClassModel ( $string  $integer  [])
                        ($family   $person   [] )
                        ($firstName
                            $closestFriend
                            $emailAddresses
                            $name
                            $members  []
                        )
```

6.4. Ejecución

Habiendo instanciado el modelo, y con la implementación de la transformación completa, es posible ejecutar la transformación. Para ello alcanza con normalizar en Agda la siguiente expression:

```
Class2Relational $model
```

Y el resultado obtenido es:

```
newRelationalModel
(newTable "Family"
 (newColumn "objectId" nothing [] (just (newType "Integer"))
  newColumn "name" nothing [] (just (newType "String"))  [])
 (newColumn "objectId" nothing [] (just (newType "Integer"))  [])

 newTable "Person"
 (newColumn "objectId" nothing [] (just (newType "Integer"))
  newColumn "closestFriendId" nothing [] (just (newType "Integer"))
  newColumn "firstName" nothing [] (just (newType "String"))  [])
 (newColumn "objectId" nothing [] (just (newType "Integer"))  [])

 newTable "Family_members"
 (newColumn "FamilyId" nothing [] (just (newType "Integer"))
  newColumn "membersId" nothing [] (just (newType "Integer"))  [])
 []

 newTable "Person_emailAdresses"
 (newColumn "PersonId" nothing [] (just (newType "Integer"))
  newColumn "emailAdresses" nothing [] (just (newType "String"))
  [])
 []
  [])
```

```
(newColumn "closestFriendId" nothing [] (just (newType "Integer"))

 newColumn "firstName" nothing [] (just (newType "String"))
 newColumn "name" nothing [] (just (newType "String"))  [])
(newType "String"  newType "Integer"  [])
```

6.5. Verificación de propiedades

Utilizando la expresividad disponible en Agda gracias a los tipos dependientes, es posible codificar propiedades a verificar como tipos de datos, y sus demostraciones como funciones que reciben representaciones de las hipótesis como argumento y construyen una verificación de la propiedad.

Este es uno de los principales beneficios de contar tanto con representaciones de ATL y KM3 en forma general, como de las transformaciones en particular en Agda. Para ejemplificar esta ventaja, comenzamos por demostrar un par de propiedades en forma general para nuestra interpretación de ATL. Esto es, propiedades que se cumplen para cualquier transformación, independientemente de sus reglas o meta-modelos de origen y destino.

6.5.1. Independientes de la Transformación

La propiedad en cuestión enuncia que la aplicación de cualquier regla de transformación sin guarda, debe generar la misma cantidad de elementos de salida que de entrada. El primer paso será entonces definir el tipo de Agda cuyos valores demuestren que dos listas tienen el mismo tamaño:

```
data _|==|_ {A B : Set} : List A -> List B -> Set where
  empty : [] |==| []
  noempty : forall { x xs y ys} -> xs |==|  ys -> (x  xs) |==| (y  ys)
```

Este tipo especifica que podemos obtener una prueba de que dos listas tienen el mismo tamaño si ambas están vacías (constructor *empty*) o si tenemos una prueba de que ambas listas sin su primer elemento tienen el mismo tamaño.

Las reglas sin guarda se ejecutan en nuestra implementación mediante la aplicación de la funcion r:

```
r : { A B : Set } -> (m : Memory)
                  -> List A ->
                  ( A -> String) ->
                  ( Memory -> A -> B)
                  -> List (Link B)
r  _ [] _ _ = []
r  m (x   xs) id rule = [ id x , rule m x ]  (r m xs id rule)
```

La función toma el estado de la memoria en determinado momento *(M : Memory)*, una lista de elementos de origen *List A*, una función para poder obtener la identidad de los elementos de origen ($A \rightarrow String$), una función que a partir del estado de la memoria y un valor del tipo de origen genera un valor del tipo de destino ($Memory \rightarrow A \rightarrow B$) y genera una lista de *Links*, correspondientes a las representaciones de los objetos transformados.

El objetivo es ahora construir un valor del tipo definido a partir de r, sin importar el resto de los argumentos. Para ello definimos la siguiente función:

```
lem-executeRule : {A B : Set} -> (m : Memory)
                  -> (list : List A) ->
                    (id : A -> String ) ->
                    (rule : Memory -> A -> B)
                    ->   (r m list id rule) |==| list
lem-executeRule  m [] _ _ = empty
lem-executeRule  m (x  xs) id rule = noempty (lem-executeRule m xs id rule)
```

La función *lem-executeRule* toma los mismos argumentos que r y construye una prueba de que el largo del resultado de aplicar esos argumentos a r tiene el mismo tamaño que la lista pasada como argumento.

En el caso de que la lista sea vacía, la aplicación de r devuelve una lista también vacía, de allí el uso del constructor *empty*. Para el caso inductivo, construimos la prueba basandonos en que se cumple la hipótesis inductiva, es decir, que se cumple para la lista con un elemento menos.

En forma análoga podemos demostrar que la aplicación de una regla con guarda a una lista de elementos r' genera menor o igual cantidad de elementos de salida que de entrada. El tipo para la propiedad será:

```
data _|<=|_ {A B : Set} : List A -> List B -> Set where
  leq : forall {xs} -> [] |<=| (xs)
  leq-drp : forall { xs y ys} ->
              xs |<=|  ys ->
              xs |<=| (y  ys)
  leq-kp : forall { x xs y ys} ->
              xs |<=|  ys ->
              (x  xs) |<=| (y  ys)
```

En este caso los constructores permiten demostrar que:

- *leq*: Una lista vacía tiene menor o igual largo que cualquier lista.
- *leq-drp*: si tenemos una demostración de que la lista x tiene menor o igual cantidad de elemenos que la lista y, entonces la propiedad se mantiene si agregamos un elemento más a y.

- *leq-kp*: si tenemos una demostración de que la lista x tiene menor o igual cantidad de elemenos que la lista y, entonces la propiedad se mantiene si agregamos un elemento más a x y otro a y y.

Construimos entonces la función *executeRuleWithGuard*, que toma los mismos argumentos que *r'* y construye una prueba de que el largo del resultado de la aplicación de *r'* es menor o igual al largo de la lista de entrada.

```
lem-executeRuleWithGuard :  {A B : Set} ->
                             (m : Memory) ->
                             (list : List A) ->
                             (id : A -> String) ->
                             (rule : Memory ->  A -> B) ->
                             (guard : A -> Bool) ->
                             (r' m list id rule guard) |<=| list
lem-executeRuleWithGuard m [] _ _ _ = leq
lem-executeRuleWithGuard m (x  xs) id rule guard with guard x
... | false = leq-drp (lem-executeRuleWithGuard m xs id rule guard)
... | true = leq-kp (lem-executeRuleWithGuard m xs id rule guard)
```

Como útlimo ejemplo, incluimos una demostración de que la aplicación de una guarda a una lista de elementos (*applyGuard*) genera un subconjunto (sublista) de los elementos originales. Para ello definimos el tipo que representa la propiedad de inclusión de listas:

```
data __ {A : Set} : List A -> List A -> Set where
  stp : []  []
  drp : forall {xs y ys} -> xs  ys -> xs  (y  ys)
  kp : forall {x xs ys} -> xs  ys ->  (x  xs)  (x  ys)
```

y luego una función que crea un valor de ese tipo a partir de una lista, una guarda y el resultado de su aplicación:

```
lem-guard : forall {A} (p : A -> Bool) xs ->
                        applyGuard  p xs  xs
lem-guard p [] = stp
lem-guard p (x  xs) with p x
... | false = drp (lem-guard p xs)
... | true = kp (lem-guard p xs)
```

6.5.2. Dependientes de la Transformación

Un aspecto interesante del formalismo utilizado y la implementación propuesta es que permiten demostrar propiedades a nivel general (ejemplos anteriores) o para una transformación en particular. Siguiendo una mecánica similar

a la utilizada en los teoremas anteriores, y valiendonos del tipo que representa la propiedad de igualdad de largo para listas, podemos demostrar que el modelo relacional obtenido como resultado de cualquier ejecución de la transformación *Class2Relational* tendrá la misma cantidad de elementos *Type* como elementos *DataType* tenga el modelo de entrada.

La demostración del teorema es:

```
lem-typeCount : (m : ClassModel) ->
          (ClassModel._DataType m) |==|
              (RelationalModel._Type (Class2Relational m))
lem-typeCount (newClassModel [] _ _) =
                           empty
lem-typeCount (newClassModel (x  xs) _ _) =
                           noempty (lem-typeCount (newClassModel xs [] []))
```

Creemos que la aplicación a un caso concreto permite visualizar las ventajas que ofrece la propuesta. Las propiedades demostradas son sencillas, ya que el objetivo de este trabajo es describir el mecanismo, no verificar propiedades concretas. De todos modos, entendemos que aunque sencillos, los ejemplos permiten apreciar las ventajas de este enfoque.

Capítulo 7

Conclusiones

El principal aporte de este trabajo es la construcción de una semántica formal para las transformaciones de modelos especificadas mediante ATL y KM3. Dicha semántica se ofrece por medio de una interpretación de sus conceptos y comportamiento en Teoría de Tipos. Creemos que la semántica propuesta se ajusta razonablemente bien a las descripciones informales encontradas en la literatura y brinda un marco para aplicar métodos formales en el contexto de transformaciones de modelos con ATL y KM3.

La solución propuesta permite, mediante un traductor automático, interpretar especificaciones existentes en ATL y KM3, representándo estas especificaciones mediante progrmas en Agda, un lenguaje funcional con tipos dependienes. Consideramos que esta traducción automática puede ayudar a disminuir el costo de construir transformaciones de modelos con propiedades verificadas.

La interpretación ofrecida en este trabajo no solo permite ejecutar transformaciones de modelos, sino también razonar sobre su especificación, utilizando para ello la potencia que ofrece el formalismo de la Teoría de Tipos.

La implementación de la semántica en un lenguaje con tipos dependientes como Agda permite además asegurar algunas propiedades claves de las transformaciones, como el tipado, la terminación y la confluencia. Estas propiedades quedan garantizadas para cualquier transformación válida contruida sobre la interpretación que brindamos. En forma adicional, el uso de Agda permite razonar sobre modelos y transformaciones particulares, pudiendo así verificar formalmente propiedades de los mismos.

Presentamos una descripción detallada de las reglas utilizadas por el traductor para embeber modelos y transformaciones en Agda, así como la aplicación de dichas reglas a un caso de estudio concreto. Mediante el caso de estudio se representan meta-modelos, una transformación completa, se instancia el modelo de origen y se ejecuta la transformación, cubriendo así el ciclo completo propuesto por MDE. Para demostrar los beneficios del enfoque presentado se demuestrán algunas propiedades en forma general sobre la semánica propuesta

y otras en forma particular sobre la transformación del caso de estudio.

Finalmente, consideramos que el trabajo constituye un ejemplo valioso de los beneficios de la utilización de lenguajes funcionales con tipos dependientes en la industria.

Queda como trabajo pendiente la prueba del enfoque con más casos de estudio, más complejos en los modelos y transformacions y con propiedades más interesantes para verificar. Un caso especialmente interesante de aplicación de nuestro marco de análisis es la generación de código de programas a partir de modelos, ya que se podría certificar que los programas generados mantienen propiedades del modelo.

Cabe aclarar también que el traductor no cubre la especificacón completa de ATL (si de KM3), ya que algunas funciones de OCL no fueron implementadas debido a que no son necesarias para completar el caso de estudio.

Referencias Bibliográficas

[1] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 0025–31, 2006.

[2] B. Selic, "The pragmatics of model-driven development," *Software, IEEE*, vol. 20, no. 5, pp. 19–25, 2003.

[3] "Object Management Group," http://www.omg.org/, 2013, [Online; Accedido Nov-2013].

[4] "MDA Guide Version 1.0. 1," http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf, 2013, [Online; Accedido Nov-2013].

[5] S. Kent, "Model driven engineering," in *Integrated Formal Methods*. Springer, 2002, pp. 286–298.

[6] J.-M. Favre, "Towards a basic theory to model model driven engineering," in *3rd Workshop in Software Model Engineering, WiSME*. Citeseer, 2004.

[7] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[8] A. G. Kleppe, J. Warmer, W. Bast, and M. Explained, "The model driven architecture: practice and promise," 2003.

[9] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3, 2003, pp. 1–17.

[10] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.

[11] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[12] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.

[13] G. Om, "Mof 2.0 query/views/transformation rfp," 2002.

[14] M. Lawley and J. Steel, "Practical declarative model transformation with tefkat," in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006, pp. 139–150.

[15] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "Atl: a qvt-like transformation language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 719–720.

[16] J. Troya and A. Vallecillo, "A rewriting logic semantics for atl." *Journal of Object Technology*, vol. 10, no. 5, pp. 1–29, 2011.

[17] W. A. Howard, "The formulas-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley, Eds. Academic Press, 1980, pp. 479–490, reprint of 1969 article.

[18] U. Norell, "Dependently typed programming in agda," in *Advanced Functional Programming*. Springer, 2009, pp. 230–266.

[19] A. Bove, P. Dybjer, and U. Norell, "A brief overview of agda–a functional language with dependent types," in *Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 73–78.

[20] Y. Sun, Z. Demirezen, T. Lukman, M. Mernik, J. Gray *et al.*, "Model transformations require formal semantics," *Domain-Specific Program Development*, 2008.

[21] "ATL/User Guide - The ATL Language," http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language/, 2013, [Online; Accedido Nov-2013].

[22] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of," *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[23] J. E. Rivera, F. Durán, and A. Vallecillo, "Formal specification and analysis of domain specific models using maude," *Simulation*, vol. 85, no. 11-12, pp. 778–792, 2009.

[24] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, A. Pierantonio *et al.*, "Extending amma for supporting dynamic semantics specifications of dsls," 2006.

[25] B. Courcelle, "The expression of graph properties and graph transformations in monadic second-order logic." in *Handbook of Graph Grammars*, 1997, pp. 313–400.

[26] A. Schürr, A. J. Winter, and A. Zündorf, "The progres approach: Language and environment," 1999.

[27] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of algebraic graph transformation*. Springer Heidelberg, 2006, vol. 373.

[28] R. Heckel, J. M. Küster, and G. Taentzer, "Confluence of typed attributed graph transformation systems," in *Graph Transformation*. Springer, 2002, pp. 161–176.

[29] J. Cabot, R. Clarisó, E. Guerra, and J. De Lara, "Analysing graph transformation rules through ocl," in *Theory and Practice of Model Transformations*. Springer, 2008, pp. 229–244.

[30] ——, "Verification and validation of declarative model-to-model transformations through invariants," *Journal of Systems and Software*, vol. 83, no. 2, pp. 283–302, 2010.

[31] J. De Lara and H. Vangheluwe, "Translating model simulators to analysis models," in *Fundamental Approaches to Software Engineering*. Springer, 2008, pp. 77–92.

[32] L. Baresi and P. Spoletini, "On the use of alloy to analyze graph transformation systems," in *Graph Transformations*. Springer, 2006, pp. 306–320.

[33] K. Anastasakis, B. Bordbar, and J. M. Küster, "Analysis of model transformations via alloy," in *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, 2007, pp. 47–56.

[34] A. Boronat, R. Heckel, and J. Meseguer, "Rewriting logic semantics and verification of model transformations," in *Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 18–33.

[35] A. Boronat and J. Meseguer, "An algebraic semantics for mof," in *Fundamental Approaches to Software Engineering*. Springer, 2008, pp. 377–391.

[36] I. Poernomo, "Proofs-as-model-transformations," in *Theory and Practice of Model Transformations*. Springer, 2008, pp. 214–228.

[37] M. Amrani, L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J. R. Cordy, "A tridimensional approach for studying the formal verification of model transformations," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 921–928.

[38] D. Calegari and N. Szasz, "Verification of model transformations: a survey of the state-of-the-art," *Electronic Notes in Theoretical Computer Science*, vol. 292, pp. 5–25, 2013.

[39] J. M. Küster and M. Abd-El-Razik, "Validation of model transformations–first experiences using a white box approach," in *Models in Software Engineering*. Springer, 2007, pp. 193–204.

[40] D. Varró and A. Pataricza, "Automated formal verification of model transformations," *CSDUML*, pp. 63–78, 2003.

[41] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa, "Dsltrans: A turing incomplete transformation language," in *Software Language Engineering*. Springer, 2011, pp. 296–305.

[42] H. Bruggink, "Towards a systematic method for proving termination of graph transformation systems," *Electronic Notes in Theoretical Computer Science*, vol. 213, no. 1, pp. 23–38, 2008.

[43] H. Ehrig, K. Ehrig, J. De Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay, *Termination criteria for model transformation.* Springer, 2005.

[44] J. M. Küster, "Definition and validation of model transformations," *Software & Systems Modeling*, vol. 5, no. 3, pp. 233–259, 2006.

[45] K. Lano and S. Kolahdouz-Rahimi, "Specification and verification of model transformations using uml-rsds," in *Integrated Formal Methods.* Springer, 2010, pp. 199–214.

[46] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer, "Termination analysis of model transformations by petri nets," in *Graph Transformations.* Springer, 2006, pp. 260–274.

[47] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger, "Right or wrong?-verification of model transformations using colored petri nets," in *Proc. of 9th OOPSLA Workshop on Domain-Specific Modeling, Orlando, USA*. Citeseer, 2009.

[48] F. Hermann, H. Ehrig, F. Orejas, and U. Golas, "Formal analysis of functional behaviour for model transformations based on triple graph grammars," in *Graph Transformations.* Springer, 2010, pp. 155–170.

[49] R. Heckel, J. M. Küster, and G. Taentzer, "Confluence of typed attributed graph transformation systems," in *Graph Transformation.* Springer, 2002, pp. 161–176.

[50] L. Lambers, H. Ehrig, and F. Orejas, "Efficient detection of conflicts in graph-based model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 97–109, 2006.

[51] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Systematic transformation development," *Electronic Communications of the EASST*, vol. 21, 2009.

[52] T. Levendovszky, L. Lengyel, and T. Mészáros, "Supporting domain-specific model patterns with metamodeling," *Software & Systems Modeling*, vol. 8, no. 4, pp. 501–520, 2009.

[53] J. Steel and J.-M. Jézéquel, "On model typing," *Software & Systems Modeling*, vol. 6, no. 4, pp. 401–413, 2007.

[54] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from uml to alloy," *Software & Systems Modeling*, vol. 9, no. 1, pp. 69–86, 2010.

[55] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Automated verification of model transformations based on visual contracts," *Automated Software Engineering*, vol. 20, no. 1, pp. 5–46, 2013.

[56] H. Ledang and H. Dubois, "Proving model transformations," in *Theoretical Aspects of Software Engineering (TASE), 2010 4th IEEE International Symposium on.* IEEE, 2010, pp. 35–44.

[57] E. Planas, J. Cabot, C. Gómez *et al.*, "Two basic correctness properties for atl transformations: Executability and coverage," in *3rd International Workshop on Model Transformation with ATL, Zurich, Switzerland*, 2011.

[58] D. Akehurst, S. Kent, and O. Patrascoiu, "A relational approach to defining and implementing transformations between metamodels," *Software and Systems Modeling*, vol. 2, no. 4, pp. 215–239, 2003.

[59] M. Asztalos, L. Lengyel, and T. Levendovszky, "Towards automated, formal verification of model transformations," in *Software testing, verification and validation (icst), 2010 third international conference on.* IEEE, 2010, pp. 15–24.

[60] E. Cariou, N. Belloir, F. Barbier, and N. Djemam, "Ocl contracts for the verification of model transformations," *Electronic Communications of the EASST*, vol. 24, 2009.

[61] D. Calegari, C. Luna, N. Szasz, and Á. Tasistro, "A type-theoretic framework for certified model transformations," in *Formal Methods: Foundations and Applications.* Springer, 2011, pp. 112–127.

[62] M. Garcia and R. Möller, "Certification of transformation algorithms in model-driven software development." *Software Engineering*, vol. 105, pp. 107–118, 2007.

[63] J. M. Küster, "Systematic validation of model transformations," *WiSME'04 (associated to UML'04)*, 2004.

[64] L. Lengyel, I. Madari, M. Asztalos, and T. Levendovszky, "Validating query/view/transformation relations," in *Model-Driven Engineering, Verification, and Validation (MoDeVVa), 2010 Workshop on.* IEEE, 2010, pp. 7–12.

[65] K. Lano and S. Kolahdouz-Rahimi, "Model-driven development of model transformations," in *Theory and Practice of Model Transformations.* Springer, 2011, pp. 47–61.

[66] B. Schätz, "Verification of model transformations," *Electronic Communications of the EASST*, vol. 29, 2010.

[67] K. Stenzel, N. Moebius, and W. Reif, "Formal verification of qvt transformations for code generation," in *Model Driven Engineering Languages and Systems.* Springer, 2011, pp. 533–547.

[68] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró, "Viatra-visual automated transformations for formal verification and validation of uml models," in *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on.* IEEE, 2002, pp. 267–270.

[69] L. Lúcio, B. Barroca, and V. Amaral, "A technique for automatic validation of model transformations," in *Model Driven Engineering Languages and Systems.* Springer, 2010, pp. 136–150.

[70] A. Narayanan and G. Karsai, "Specifying the correctness properties of model transformations," in *Proceedings of the third international workshop on Graph and model transformations*. ACM, 2008, pp. 45–52.

[71] F. Orejas and M. Wirsing, "On the specification and verification of model transformations," in *Semantics and algebraic specification*. Springer, 2009, pp. 140–161.

[72] A. Schürr and F. Klar, "15 years of triple graph grammars," in *Graph Transformations*. Springer, 2008, pp. 411–425.

[73] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling, "Symbolic invariant verification for systems with dynamic structural adaptation," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 72–81.

[74] L. Baresi, K. Ehrig, and R. Heckel, "Verification of model transformations: A case study with bpel," in *Trustworthy Global Computing*. Springer, 2007, pp. 183–199.

[75] A. Boronat, A. Knapp, J. Meseguer, and M. Wirsing, "What is a multi-modeling language?" in *Recent Trends in Algebraic Development Techniques*. Springer, 2009, pp. 71–87.

[76] B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux, "Essay on semantics definition in mde," *JOURNAL OF SOFTWARE*, vol. 4, no. 9, p. 943, 2009.

[77] G. Engels, J. M. Küster, R. Heckel, and M. Lohmann, "Model-based verification and validation of properties," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 7, pp. 133–150, 2003.

[78] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner, "Towards verified model transformations," in *Proc. of the 3rd International Workshop on Model Development, Validation and Verification (MoDeV 2 a), Genova, Italy, Le Commissariat à lEnergie Atomique-CEA*. Citeseer, 2006, pp. 78–93.

[79] F. Hermann, M. Hülsbusch, and B. König, "Specification and verification of model transformations," *Electronic Communications of the EASST*, vol. 30, 2010.

[80] R. Heckel and S. Thöne, "Behavioral refinement of graph transformation-based models," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 3, pp. 101–111, 2005.

[81] T. Massoni, R. Gheyi, and P. Borba, "Formal refactoring for uml class diagrams," in *19th Brazilian Symposium on Software Engineering (SBES), Uberlândia, Brazil*, 2005, pp. 152–167.

[82] A. Narayanan and G. Karsai, "Towards verifying model transformations," *Electronic Notes in Theoretical Computer Science*, vol. 211, pp. 191–200, 2008.

[83] C. Pons and D. Garcia, "A lightweight approach for the semantic validation of model refinements," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 43–61, 2008.

[84] J. Padberg, M. Gajewsky, and C. Ermel, *Refinement versus verification: Compatibility of net invariants and stepwise development of high-level petri nets*. Citeseer, 1997.

[85] G. Rangel, L. Lambers, B. König, H. Ehrig, and P. Baldan, *Behavior preservation in model refactoring using DPO transformations with borrowed contexts*. Springer, 2008.

[86] J. Wang, S.-K. Kim, and D. Carrington, "Verifying metamodel coverage of model transformations," in *Software Engineering Conference, 2006. Australian*. IEEE, 2006, pp. 10–pp.

[87] L. López, L. Pintos, D. Calegari, and C. Luna, "Estado del arte de testing de transformaciones de modelos."

[88] K. Chan, "Formal proofs for qos-oriented transformations," in *Enterprise Distributed Object Computing Conference Workshops, 2006. EDOCW'06. 10th IEEE International*. IEEE, 2006, pp. 41–41.

[89] M. Chechik, S. Nejati, and M. Sabetzadeh, "A relationship-based approach to model integration," *Innovations in Systems and Software Engineering*, vol. 8, no. 1, pp. 3–18, 2012.

[90] F. Jouault and J. Bézivin, "Km3: a dsl for metamodel specification," in *Formal Methods for Open Object-Based Distributed Systems*. Springer, 2006, pp. 171–185.

[91] "KM3 User Manual," http://www.eclipse.org/gmt/am3/km3/doc/KernelMetaMetaModel%5Bv00.06%5D.pdf, 2013, [Online; Accedido Nov-2013].

[92] J. Warmer and A. Kleppe, *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[93] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006, pp. 128–138.

[94] L. Magnusson and B. Nordström, "The alf proof editor and its proof engine," in *Types for proofs and programs*. Springer, 1994, pp. 213–237.

[95] T. Coquand, "Pattern matching with dependent types," in *Informal proceedings of Logical Frameworks*, vol. 92. Citeseer, 1992, pp. 66–79.

[96] "Eclipse (ATL)," http://www.eclipse.org/atl/atlTransformations/, 2013, [Online; Accedido Nov-2013].

Apéndice A

Definición de KM3

A.1. Definición de KM3 en KM3

```
package KM3 {
    abstract class ModelElement {
    attribute name : String ;
    reference " package " : Package oppositeOf contents ; }

    class Classifier extends ModelElement {}
    class DataType extends Classifier {}
    class Enumeration extends Classifier {
     reference literals [] ordered container : EnumLiteral oppositeOf enum;
    }

    class EnumLiteral extends ModelElement {
     reference enum : Enumeration oppositeOf literals;
    }

    class Class extends Classifier {
     attribute isAbstract : Boolean ;
     reference supertypes [] : Class ;
      reference structuralFeatures [  ] ordered container
                        : StructuralFeature oppositeOf owner;
    }

    class TypedElement extends ModelElement {
     attribute lower : Integer;
     attribute upper : Integer;
     attribute isOrdered : Boolean;
     attribute isUnique: Boolean;
     reference type : Classifier;
    }
```

```
    class StructuralFeature extends TypedElement {
     reference owner : Class oppositeOf structuralFeatures;
     reference subsetOf[*] : StructuralFeature oppositeOf derivedFrom;
     reference derivedFrom[*] : StructuralFeature oppositeOf subsetOf;
    }

    class Attribute extends StructuralFeature {}

    class Reference extends StructuralFeature {
     attribute isContainer : Boolean ;
reference opposite[01] : Reference ;
}

class Package extends ModelElement {
reference contents[] ordered container :
                    ModelElement oppositeOf "package";
reference metamodel : Metamodel oppositeOf contents ;
}

class Metamodel extends LocatedElement {
reference contents [] ordered container : Package oppositeOf metamodel;
}
}

package PrimitiveTypes {
datatype Boolean;
datatype Integer;
datatype String ;
}
```

A.2. Gramática de KM3

package → package *name* { *classifiers* }

classifiers →

classifiers → *classifier classifiers*

classifier → *class*

classifier → *datatype*

classifier → *enumeration*

class → *isabstract* class *name supertypes features*

isabstract →

isabstract → abstract

supertypes →

supertypes → extends *typelist*

typelist → *typeref*

typelist → *typeref*, *typelist*

features →

features → *feature features*

feature → *attribute*

feature → *reference*

attribute → attribute *name multiplicity* : *typeref* ;

reference → reference *name multiplicity* iscontainer :

typeref oppositeOf *name* ;

multiplicity → *bounds*

multiplicity → *bounds* ordered

bounds →

bounds → [*integer* - *integer*]

bounds → [*integer* - *]

bounds → [*]

iscontainer →

iscontainer → container

datatype → datatype *name* ;

enumeration → enumeration *name* *literals*

literals →

literals → *literal literals*

literal → literal *name* ;

typeref → *name*

Apéndice B

Descripción detallada de ATL

B.1. Tipos de Datos

A continuación se describe cada uno de los tipos de datos y se presentan las operaciones disponibles en ATL para todos los tipos de datos en general.

Tipo general

El tipo OclAny ofrece las siguientes operaciones:

- operadores de comparación: =, <>.
- oclIsUndefined() retorna un booleano indicando si el valor de *self* se encuentra definido o no.
- oclIsKindOf(t : oclType) retorna un booleano indicando si *self* es una instancia de *t* o uno de sus sub-tipos.
- oclIsTypeOf(t : oclType) retorna un booleano indicando si *self* es una instancia de *t*.
- toString() retorna un *String* que representa a *self*.
- oclType() retorna el tipo de *self*.
- asSequence(), asSet(), asBag() respectivamente devuelven una instancia de Sequence, Set o Bag conteniendo a *self*.
- refSetValue(name : String, val : oclAny) operación reflexiva que permite establecer el valor de la característica de *self* identificada por *name*, devolviendo un nuevo valor de *self*.

- refGetValue(name : String) operación reflexiva que devuelve el valor de la cartacterística de *self* identificada por *name*
- refImmediateComposite() operación reflexiva que retorna el contenedor inmediato de *self*.
- refInvokeOperation(opName : String, args : Sequence) operacion reflexiva que permite invocar la operación de *self* identificada por *opName*.

El tipo de datos ATLModule es interno a ATL e intenta representar las unidades del lenguaje (modulos, consultas o librerías'). En tiempo de ejecución solo existe una instancia para este tipo, pudiendo los desarrolladores referirse a ella mediante la variable *thisModule*, lo que hace posible acceder a las funciones y atributos definidos en el módulo en ejecución.

Tipos de datos primitivos

OCL define cuatro tipos primitivos de datos:

- Boolean: con *true* y *false* como valores
- Integer: valores numéricos enteros (1, -5, 2, 34, 26524, ...)
- Real: valores numéricos de punto flotante (1.5, 3.14, ...);
- String: textos ('To be or not to be', ...). Los caracteres que componen un String se numeran desde 1 al largo del String.

Las operaciones soportadas por Boolean son:

- operadores lógicos: and, or, xor, not;
- implies(b : Boolean) retorna *false* si *self* es *true* y *b* es *false*, y *true* en otro caso.

Las operaciones soportadas por String son:

- operadores de comparación: <, >, >=, <=;
- el operador de concatenación (+) funciona como atajo de la función concat().
- size() retorna la cantidad de caracteres contenidos en el texto.
- concat(s : String) retorna un nuevo String con *s* concatenado al final de *self*
- substring(lower : Integer, upper : Integer) retorna el sub-texto determinado por el rango especificado.

- toInteger() y toReal(): funciones de conversión.
- toUpper(), toLower() retornan nuevos Strings convirtiendo *self* a mayúsculas y minúsculas respectivamente.
- toSequence() retorna la secuencia de caracteres correspondiente a *s*.
- trim() retorna un nuevo texto sin los espacios en blanco al comienzo o al final.
- startsWith(s : String), endsWith(s : String) retorna un booleano indicando si *self* comienza o termina con *s* respectivamente.
- indexOf(s : String), lastIndexOf(s : String) retorna la posición de la primera/última aparición de *s* en *self*.
- split(regex : String) retorna una secuencia resultado de cortar *self* utilizando *regex* como expresión regular para especificar el separador. La especificación para las expresiones regulares se toma de Java.
- replaceAll(c1 : String, c2 : String) retorna una copia de *self* con cada ocurrencia de *c1* reemplazada por *c2*.
- regexReplaceAll(regex : String, replacement : String) retorna una copia de *self* con cada coincidencia de *regex* reemplazada por *replacement*.

Las operaciones soportadas por los tipos de datos numéricos (tanto Integer como Real) son:

- operadores de comparación: <, >, =>, =<;
- operadores binarios: *, +, -, /, div(), max(), min();
- operadores unarios: abs().
- cos(), sin(), tan(), acos(), asin();
- toDegrees(), toRadians();
- exp(), log(), sqrt().

Además se definen las siguientes operaciones particulares:

- enteros: mod();
- reales: floor(), round().

Colecciones

OCL define un conjunto de tipos de datos para representar colecciones, compuesto por los tipos Set, OrderedSet, Bag y Sequence. Collection es el tipo asbtracto común para todos ellos. Los tipos particulares de colecciones presentan las siguientes características:

- Set es una colección sin duplicados y sin orden.
- OrderedSet es una colección ordenada sin duplicados.
- Bag es una colección que permite duplicados y sin orden.
- Sequence es una colección ordenada que permite duplicados.

Las colecciones pueden verse como tipos de datos parametrizados por otros tipos, por lo que al definir una colección se debe definir el tipo de elementos a contener. Sin importar el tipo particular de colección, su creación siempre debe responder al siguiente esquema:

```
collection_type(element_datatype)
```

Es posible definir una colección por extensión, describiendo su contenido de la siguiente forma:

```
collection_type{elements}
```

ATL provee un listado amplio de operaciones soportadas para las colecciones, y utiliza una sintaxis particular para la ejecución de las mismas:

```
self->operation_name(parameters)
```

Todos los tipos de colecciones comparten un conjunto común de operaciones:

- size() retorna la cantidad de elementos en una colección.
- includes(o : oclAny) retorna un booleano indicando si *o* es parte de la colección *self*.
- excludes(o : oclAny) retorna un booleano indicando si *o* no es parte de la colección *self*.
- count(o : oclAny) retorna el número de apariciones de *o* en *self*.
- includesAll(c : Collection) retorna un booleano indicando si todos los elementos contenidos en *c* están contenidos en *self*.

- excludesAll(c : Collection) retorna un booleano indicando si ninguno los elementos contenidos en *c* están contenidos en *self*.
- isEmpty() retorna un booleano indicando si la colección *self* está vacía. notEmpty() retorna un booleano indicando si la colección *self* no está vacía.
- sum() retorna el valor correspondiente a la adición de todos los elementos comprendidos en la lista.
- asBag(), asSequence(), asSet(): funciones de conversión.

El tipo Sequence responde a las siguientes operaciones particulares:

- union(c : Collection) retorna una secuencia compuesta por todos los elementos de *self* seguidos por los elementos de *c*.
- flatten() retorna una secuencia conteniendo directamente los elementos contenidos por cada uno de los elementos en *self*.
- append(o : oclAny) retorna una copia de *self* con el elemento *o* agregado al final.
- repend(o : oclAny) retorna una copia de *self* con el elemento *o* agregado al principio.
- insertAt(n : Integer, o : oclAny), retorna una copia de *self* con el elemento *o* agregado en la posición *n*.
- subSequence(lower : Integer, upper : Integer) retorna la subsecuencia delimitada por el rango especificado.
- at(n : Integer) retorna el elemento ubicado en la posición *n*.
- indexOf(o : oclAny) retorna la posición de la primera aparición del elemento especificado.
- first() retorna el primer elemento o oclUndefined si la lista está vacía.
- last() retorna el último elemento o oclUndefined si la lista está vacía.
- including(o : oclAny) retorna una copia de *self* con el elemento agregado al final.
- excluding(o : oclAny) retorna una copia de *self* quitando el elemento especificado.

El tipo Set responde a las siguientes operaciones particulares:

- union(c : Collection) retorna un Set compuesta por todos los elementos de *self* seguidos por los elementos de *c* y sin los repetidos.

- intersection(c : Collection) retorna un Set con los elementos en la intersección de *self* y *c*.
- operator - (s : Set) retorna un Set con los elementos *self* que no están en *s*.
- including(o : oclAny) retorna una copia de *self* con el elemento agregado al final (si no existía).
- excluding(o : oclAny), retorna una copia de *self* quitando el elemento especificado.
- symetricDifference(s : Set) retorna un nuevo Set con los elementos que se encuentran en *self* o *s*, pero no en ambos.

El tipo OrderedSet responde a las siguientes operaciones particulares:

- append(o : oclAny) retorna una copia de *self* con el elemento al final si es que no aparecía.
- prepend(o : oclAny) retorna una copia de *self* con el elemento al inicio si es que no aparecía.
- insertAt(n : Integer, o : oclAny), retorna una copia de *self* con el elemento en la posición indicada si es que no aparecía.
- subOrderedSet (lower : Integer, upper : Integer) retorna el subconjunto determinado por el rango especificado.
- at(n : Integer) retorna el elemento ubicado en la posición especificada.
- indexOf(o : oclAny) retorna la posición de la primera aparición del elemento especificado.
- first(), last(): retorna el primer/último elemento contenido.
- union(c : Collection) retorna un OrderedSet compuesto por todos los elementos de *self* seguidos por los elementos de *c* y sin los repetidos.
- flatten() retorna un OrderedSet conteniendo directamente los elementos contenidos por cada uno de los elementos en *self* y elminiando repetidos.
- including(o : oclAny) retorna una copia de *self* con el elemento agregado al final (si no existía).
- excluding(o : oclAny) retorna una copia de *self* quitando el elemento especificado.

El tipo Bag responde a las siguientes operaciones particulares:

- flatten() retorna un Bag conteniendo directamente los elementos contenidos por cada uno de los elementos en *self* y elminiando repetidos.

- including(o : oclAny) retorna una copia de *self* con el elemento agregado al final (si no existía).
- excluding(o : oclAny) retorna una copia de *self* quitando el elemento especificado.

La especificación de OCL incluye un conjunto de operaciones iterativas sobre colecciones, también llamadas expresiones iterativas, aplicables sobre todos los tipos de colecciones. La definición de una expresión iterativa incluye la colección sobre la que se itera, las variables de iteración declaradas y la expresión o cuerpo a utilizar en la iteración. La sintaxis general es:

```
source->operation_name(iterators | body)
```

ATL soporta el siguiente conjunto de operaciones iterativas:

- exists(body) retorna *true* si *body* evalúa como *true* al menos para un elemento en *self*.
- forAll(body) retorna *true* si *body* evalúa como *true* para todos los elementos en *self*.
- isUnique(body) retorna un booleano indicando si *body* evalúa distinto para cada uno de los elementos en *self*.
- any(body) retorna un elemento de la colección para el cual *body* evalúa como *true* y oclUndefined si no hay ninguno.
- one(body) retorna un booleano indicando si *body* evalúa distinto para exactamente uno de los elementos en *self*.
- collect(body) retorna una colección de elementos resultado de aplicar *body* a cada elemento en *self*.
- select(body) retorna el subconjunto de la colección para el cual *body* evalúa como *true*.
- reject(body) retorna el subconjunto de la colección para el cual *body* evalúa como *false*.
- sortedBy(body) retorna una colección ordenada de acuerdo al resultado de *body*, ordenado de mayor a menor.

Enumerados

Los enumerados, a diferencia de los tipos presentados hasta ahora, deben ser definidos en los meta-modelos de entrada y salida de cada transformación. En la especificación de OCL esto se logra especificando el nombre de la enumeración, seguida por doble dos puntos y el valor, por ejemplo Sexo::Femenino. La

implementación de ATL difiere, ya que el nombre del tipo no es necesario y el valor solo debe ser precedido de un numeral, por ejemplo: #Femenino.

Las enumeraciones no tienen operaciones particulares asociadas.

Tuplas

Este tipo permite componer varios valores en uno único. Una tupla está compuesta por un número fijo de partes nombradas, cada una tipada en forma particular. Las tuplas en si misma no son nombradas, por lo que para identificar una tupla es necesario especificar su declaración completa. Cada parte de una tupla está asociada a un OclType e identificada por un nombre único. Su declaración debe corresponde con la siguiente sintaxis, siendo irrelevante el orden de las partes:

```
TupleType(var_name1 : var_type1, ..., var_nameN : var_typeN)
```

La sintaxis para su inicialización es la siguiente:

```
Tuple{var_name1 [: var_type1]? = init_exp1, ..., var_namen [: var_typen]? = init_expn}
```

Las diferentes partes de una tupla son accesibles utilizando la misma notación de punto que para la invocación de operaciones.

Además de las operaciones comunes a todos los tipos, ATL provee la operación asMap() para las tuplas, que retorna un mapa en el que el nombre de cada parte de la tupla se encuentra asociado con su valor.

Mapas

Este tipo de datos permite manejar colecciones en las que cada valor se encuentra asociado a una clave única que permite su acceso. La sintaxis para declarar un map es:

```
Map(key_type, value_type)
```

Y para su inicialización:

```
Map{(key1, value1), ..., (keyn, valuen)}
```

Además del conjunto de operaciones comunes, ATL provee las siguientes operaciones sobre los mapas:

- get(key : oclAny) retorna el valor asociado con la clave en el mapa (OclUndefined si la clave no se encuentra); including(key : oclAny, val : oclAny) retorna una copia de *self* en la cual la pareja (key, val) ha sido insertada si key ya no estaba presente.
- union(m : Map) retorna un mapa todos los elementos de *self* más los de *m* si no estaban.
- getKeys() retorna un *Set* con todas las claves de *self*.
- getValues() retorna un *Bag* con todas las claves de *self*.

ModelElement

En general los meta-modelos definen un conjunto de elementos propios, también llamados clases. En ATL, los elementos variables en función del meta-modelo (clases) son referenciados mediante la notación *metamodelo!clase*, en la cual, "metamodelo" identifica uno de los meta-modelos involucrados en la transformación por su nombre, mientreas que clase referencia una clase particular contenida dentro del meta-modelo.

Las clases pueden contener características, las que pueden ser atributos o referencias, ambos accesibles mediante la notación *self.* característica. En ATL la única forma de crear instanciar elementos es mediante las reglas, especificando la inicialización de sus distintas características, lo que se consigue mediante los enlaces (*bindings*) especificados en la sección de destino (*target*).

El uso de los elementos ModelElement para los usuarios en las versiones actuales de ATL está restringido unicamente a los elementos del tipo Class.

Apéndice C

Transformación Clases a Relacional

C.1. Meta-modelo de Entrada: Classes.km3

```
package Class {

    abstract class NamedElt {
        attribute name : String;
    }

    abstract class Classifier extends NamedElt {
    }

    class DataType extends Classifier {

    }

    class Class extends Classifier {
        -- -: no sub reference
        reference super[*] : Class;
        reference attr[*] ordered container : Attribute oppositeOf owner;
        attribute isAbstract : Boolean;
    }

    class Attribute extends NamedElt {
        attribute multiValued : Boolean;
        reference type : Classifier;
        -- +: opposite of attr
        reference owner : Class oppositeOf attr;
    }
}
```

```
package PrimitiveTypes {
    datatype Boolean;
    datatype Integer;
    datatype String;
}
```

C.2. Meta-modelo de Salida: Relational.km3

```
package Relational {

    abstract class Named {
        attribute name : String;
    }

    class Table extends Named {
        reference col[*] ordered container : Column oppositeOf owner;
        reference key[*] : Column oppositeOf keyOf;
    }

    class Column extends Named {
        -- +: opposite of col
        reference owner : Table oppositeOf col;
        -- +: opposite of key
        reference keyOf[0-1] : Table oppositeOf key;
        reference type : Type;
    }

    class Type extends Named {
        -- +: "extends Named" missing in DSTC paper

    }
}

package PrimitiveTypes {
    datatype Boolean;
    datatype Integer;
    datatype String;
}
```

C.3. Transformación

```
module Class2Relational;
```

```
create OUT : Relational from IN : Class;

uses strings;


-- if there is a configuration problem with the library string,
-- delete the two firstToLower() operations
-- and you no longer need the library "strings"
-- the transfomarion will execute


-- inheritance not supported yet

-- issue: choose an object-id Type (Integer, String?).
-- We choose Integer here, assuming this type is defined in the source model.

-- global variable
-- context
helper def: objectIdType : Relational!Type =
    Class!DataType.allInstances()->select(e | e.name = 'Integer')->first();

rule Class2Table {
    from
        c : Class!Class
    to
        out : Relational!Table (
            name <- c.name,
            -- Columns are generated from Attributes
            -- in another rule not explicitly called here !
            col <- Sequence {key}->union(c.attr->select(e | not e.multiValued)),
            key <- Set {key}
        ),
        key : Relational!Column (
            name <- 'objectId',
            type <- thisModule.objectIdType
        )
}

rule DataType2Type {
    from
        dt : Class!DataType
    to
        out : Relational!Type (
            name <- dt.name
        )
}

rule DataTypeAttribute2Column {
    from
        a : Class!Attribute (
```

```
            a.type.oclIsKindOf(Class!DataType) and not a.multiValued
        )
    to
        out : Relational!Column (
            name <- a.name,
            type <- a.type
-- explicit use of implicit tracking links (first expected syntax,
-- then present actual syntax)
--          owner <- [Class2Type.key]a.owner
--          owner <- thisModule.resolveTemp(a.owner, 'key')
        )
}

rule MultiValuedDataTypeAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!DataType) and a.multiValued
        )
    to
        out : Relational!Table (
            name <- a.owner.name + '_' + a.name,
            col <- Sequence {id, value}
        ),
        id : Relational!Column (
            name <- a.owner.name.firstToLower() + 'Id',
            type <- thisModule.objectIdType
        ),
        value : Relational!Column (
            name <- a.name,
            type <- a.type
        )
}

rule ClassAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!Class) and not a.multiValued
        )
    to
        foreignKey : Relational!Column (
            name <- a.name + 'Id',
            type <- thisModule.objectIdType
        )
}

rule MultiValuedClassAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!Class) and a.multiValued
        )
```

```
    to
        t : Relational!Table (
            name <- a.owner.name + '_' + a.name,
            col <- Sequence {id, foreignKey}
        ),
        id : Relational!Column (
            name <- a.owner.name.firstToLower() + 'Id',
            type <- thisModule.objectIdType
        ),
        foreignKey : Relational!Column (
            name <- a.name + 'Id',
            type <- thisModule.objectIdType
        )
}
```

Apéndice D

Traducción Completa de Class2Model

```
module Class2Relational where

open import Data.Nat
open import Data.String
open import Data.Bool
open import Data.List
open import Data.Maybe
------------------------------
--  ATL & helper functions
------------------------------

_->first : { A : Set } -> List A -> Maybe A
_->first []  = nothing
_->first (x   xs) = just x


_->select_ : {A : Set} -> List A -> ( A -> Bool) -> List A
_->select_ l p = Data.List.filter ( \e -> p e) l

_->union_ : {A : Set} -> List A -> List A -> List A
_->union_ l1 l2 with l1
... | [] = l2
... | (x   xs) = (x   (xs ->union l2))

_+'_ : String -> String -> String
_+'_ a b = Data.String._++_ a b


Sequence : {A : Set} -> A -> List A
Sequence a = (a   [])
```

```
applyGuard : {A : Set} -> (A -> Bool) -> List A -> List A
applyGuard  guard [] = []
applyGuard  guard (x   xs)  with guard x
... | true = x   ( applyGuard guard xs)
... | false =  ( applyGuard guard xs)

record Link(A : Set) : Set where
  constructor [_,_]
  field from : String
        to   : A


module ClassKm3 where

module PkgClass where
mutual

  data -NamedElt : Set where
    NamedElt-From-Classifier : -Classifier -> -NamedElt
    NamedElt-From-DataType : DataType -> -NamedElt
    NamedElt-From-Class : Class -> -NamedElt
    NamedElt-From-Attribute : Attribute -> -NamedElt

  data -Classifier : Set where
    Classifier-From-DataType : DataType -> -Classifier
    Classifier-From-Class : Class -> -Classifier

  record DataType : Set where
    constructor newDataType
    field id : String
          name : String

  record Class : Set where
    constructor newClass
    field id : String
          name : String
          super : List Class
          attr : List Attribute
          isAbstract : Bool

  record Attribute : Set where
    constructor newAttribute
    field id : String
          name : String
          multiValued : Bool
          type : -Classifier
          owner : Class

  record ClassModel : Set where
```

```
    constructor newClassModel
    field _DataType : List DataType
          _Class : List Class
          _Attribute : List Attribute

DataType_allInstances : ClassModel -> List DataType
DataType_allInstances m = ClassModel._DataType m


Classifier_id : -Classifier -> String
Classifier_id (Classifier-From-DataType d) = DataType.id d
Classifier_id (Classifier-From-Class c) = Class.id c

_->typeOclIsKindOfDataType : Attribute -> Bool
_->typeOclIsKindOfDataType (
newAttribute
id
n
m
(Classifier-From-DataType d) o
)  = true
_->typeOclIsKindOfDataType _ = false

_->typeOclIsKindOfClass : Attribute -> Bool
_->typeOclIsKindOfClass (
 newAttribute
 id
 n
 m
 (Classifier-From-Class d)
 o
 )  = true
_->typeOclIsKindOfClass _ = false

module RelationalKm3 where

module PkgRelational where
mutual

  data -Named : Set where
    Named-From-Table : Table -> -Named
    Named-From-Column : Column -> -Named
    Named-From-Type : Type -> -Named

  record Table : Set where
    constructor newTable
    field name : String
          col : List Column
          key : List Column
```

```
  record Column : Set where
    constructor newColumn
    field name : String
          owner : Maybe Table
          keyOf : List Table
          type : Maybe Type

  record Type : Set where
    constructor newType
    field name : String

  record RelationalModel : Set where
    constructor newRelationalModel
    field _Table : List Table
          _Column : List Column
          _Type : List Type

Named_name : -Named -> String
Named_name (Named-From-Table t)  =  Table.name t
Named_name (Named-From-Column c) =  Column.name c
Named_name (Named-From-Type t)   = Type.name t



record Memory : Set where
  constructor newMemory
  field DataType_Type : List (Link Type)
        Attribute_Column :  List (Link Column)
        Attribute_Table : List (Link Table)
        Class_Table : List (Link Table)


for : {A : Set} -> Memory ->
List A  ->
(Memory -> A -> Memory)
-> Memory
for m [] f = m
for m (x  xs) f = for m1 xs f
                  where m1 = f m x

addDataTypeTypeLinks : Memory ->  List (Link Type) -> Memory
addDataTypeTypeLinks m l = record m {
 DataType_Type = Data.List._++_
l
(Memory.DataType_Type m)
}

addAttributeColumnLinks : Memory ->  List (Link Column) -> Memory
addAttributeColumnLinks m l = record m {
Attribute_Column = Data.List._++_ l (Memory.Attribute_Column m)}
```

```
addAttributeTableLinks : Memory ->  List (Link Table) -> Memory
addAttributeTableLinks m l = record m {
Attribute_Table = Data.List._++_
l
(Memory.Attribute_Table m)
   }

addClassTableLinks : Memory ->  List (Link Table) -> Memory
addClassTableLinks m l = record m { Class_Table = Data.List._++_
l
(Memory.Class_Table m)}


lookUpDataTypeType : Memory -> String -> Maybe Type
lookUpDataTypeType (newMemory [] _ _ _) id = nothing
lookUpDataTypeType (newMemory (x   xs) _ _ _)  id with (Link.from x) == id
... | true = just (Link.to x)
... | false = lookUpDataTypeType (newMemory xs [] [] []) id

lookUpAttributeColumn : Memory -> String -> Maybe Column
lookUpAttributeColumn (newMemory _ [] _ _) id = nothing
lookUpAttributeColumn (newMemory _ (x   xs)  _ _)  id with (Link.from x) == id
... | true = just (Link.to x)
... | false = lookUpAttributeColumn (newMemory [] xs [] []) id

lookUpAll : {A B : Set} -> (m : Memory) ->
List A ->
(A -> String) ->
(Memory -> String -> Maybe B) ->
List B
lookUpAll m [] _ _ = []
lookUpAll m (x   xs) id lookUp with lookUp m (id x)
... | nothing = lookUpAll m xs id lookUp
... | just y =  y   (lookUpAll m xs id lookUp)

Type_allInstances : Memory -> List Type
Type_allInstances m = Data.List.map
(\e -> Link.to e)
(Memory.DataType_Type m)

Column_allInstances : Memory -> List Column
Column_allInstances m = Data.List.map
(\e -> Link.to e)
(Memory.Attribute_Column m)

Table_allInstances : Memory -> List Table
Table_allInstances m = Data.List._++_
                (Data.List.map (\e -> Link.to e) (Memory.Class_Table m))
                (Data.List.map (\e -> Link.to e) (Memory.Attribute_Table m))
```

```
r : { A B : Set } -> (m : Memory) ->
List A ->
( A -> String) ->
( Memory -> A -> B) ->
List (Link B)
r  _ [] _ _ = []
r  m (x   xs) id rule = [ id x , rule m x ]  (r m xs id rule)

r' : { A B : Set } -> (m : Memory) ->
List A -> ( A -> String) ->
(Memory -> A -> B) ->
( A -> Bool ) ->
List (Link B)
r' m list id rule guard = r m (applyGuard guard list) id rule

infixl 40 _+'_
infixl 60 _->union_
------------------
-- Transformacion
------------------

objectIdType :  Maybe Type
objectIdType  = just record { name = "Integer"}


-- Rule DataType2Type --
DataType2Type : Memory -> DataType -> Type
DataType2Type m d =  record { name = DataType.name d }

-- Rule DataTypeAttribute2Column ----
DataTypeAttribute2Column :  Memory -> Attribute -> Column
DataTypeAttribute2Column m a = record { name = Attribute.name a;
                                        owner = nothing;
                                        keyOf = [];
                                        type = lookUpDataTypeType
                                         m
                                         (Classifier_id (Attribute.type a))
                                      }

gDataTypeAttribute2Column :  Attribute -> Bool
gDataTypeAttribute2Column a = a ->typeOclIsKindOfDataType
not (Attribute.multiValued a)

-- Rule MultiValuedDataTypeAttribute2Column --
MultiValuedDataTypeAttribute2Column : Memory -> Attribute -> Table
MultiValuedDataTypeAttribute2Column m a = record
{
             name = (Class.name (Attribute.owner a)) +'  "_" +' (Attribute.name a) ;
             col = id   value   [] ; key = []}
```

```
    where id = record {
              name = Class.name (Attribute.owner a) +' ("Id")  ;
              type = objectIdType;
              keyOf = [];
              owner = nothing }
           value =  record {
                           name = Attribute.name a;
                           type =  lookUpDataTypeType
                            m
                            (Classifier_id (Attribute.type a)) ;
                           keyOf = [];
                           owner = nothing
                 }
gMultiValuedDataTypeAttribute2Column :  Attribute -> Bool
gMultiValuedDataTypeAttribute2Column a = a ->typeOclIsKindOfDataType
 (Attribute.multiValued a)




-- Rule ClassAttribute2Column --
ClassAttribute2Column : Memory -> Attribute -> Column
ClassAttribute2Column  m a = record {
                                    name = Attribute.name a +' "Id" ;
                                    type = objectIdType;
                                    keyOf = [];
                                    owner = nothing
                               }

gClassAttribute2Column :  Attribute -> Bool
gClassAttribute2Column a = a ->typeOclIsKindOfClass  not (Attribute.multiValued a)

-- Rule MultiValuedClassAttribute2Column --
MultiValuedClassAttribute2Column : Memory -> Attribute -> Table
MultiValuedClassAttribute2Column m  a = record
{
      name =  Class.name (Attribute.owner a) +' "_"  +' Attribute.name a ;
      col = id   foreignKey   []; key = []}
where id = record {
                name = Class.name (Attribute.owner a) +' "Id"  ;
                type = objectIdType;
                keyOf = [];
                owner = nothing }
   foreignKey =  record {
                   name = Attribute.name a +' "Id";
                   type = objectIdType;
                   keyOf = [];
                   owner = nothing}
```

```
gMultiValuedClassAttribute2Column :  Attribute -> Bool
gMultiValuedClassAttribute2Column a = a ->typeOclIsKindOfClass
 (Attribute.multiValued a)

-- Rule Class2Table ---
Class2Table : Memory -> Class -> Table
Class2Table m c = record
{
       name = Class.name c;
       col = (Sequence key) ->union(  lookUpAll
        m
        ((Class.attr c)
        ->select( \e ->
        not (Attribute.multiValued e)))
        Attribute.id lookUpAttributeColumn
        )  ;
       key = Sequence key
     }
     where key = record {
                          name = "objectId";
                          type = objectIdType;
                          owner = nothing;
                          keyOf = []
                 }

Class2Relational_phase1 : ClassModel -> Memory
Class2Relational_phase1 m = addClassTableLinks m6
  (r                                              -- Regla 6, sin guarda (r)
    m6                                            -- Memoria actual
    (ClassModel._Class m)                         -- Elementos de Entrada
    Class.id                                      -- Identificador
    Class2Table                                   -- Funcion de Regla
  )
  where m6 = addAttributeTableLinks m5
    (r'                                           -- Regla 5, con guarda (r')
      m5                                          -- Memoria actual
      (ClassModel._Attribute m)                   -- Elementos de Entrada
      (Attribute.id)                              -- Identificador
      MultiValuedClassAttribute2Column            -- Funcion de Regla
      gMultiValuedClassAttribute2Column           -- Guarda
    )
    where m5 = addAttributeColumnLinks m4
      (r'                                         -- Regla 4, con guarda (r')
        m4                                        -- Memoria actual
        (ClassModel._Attribute m)                 -- Elementos de Entrada
        (Attribute.id)                            -- Identificador
        ClassAttribute2Column                     -- Funcion de Regla
        gClassAttribute2Column                    -- Guarda
      )
      where m4 = addAttributeTableLinks m3
```

```
        (r'                                          -- Regla 3, con guarda (r')
          m3                                         -- Memoria actual
          (ClassModel._Attribute m)                  -- Elementos de Entrada
          (Attribute.id)                             -- Identificador
          MultiValuedDataTypeAttribute2Column        -- Funcion de Regla
          gMultiValuedDataTypeAttribute2Column       -- Guarda
        )
        where m3 = addAttributeColumnLinks m2
          (r'                                        -- Regla 2, con guarda (r')
            m2                                       -- Memoria actual
            (ClassModel._Attribute m)                -- Elementos de Entrada
            (Attribute.id)                           -- Identificador
            DataTypeAttribute2Column                 -- Funcion de Regla
            gDataTypeAttribute2Column                -- Guarda
          )
          where m2 = addDataTypeTypeLinks m1
            (r                                       -- Regla 1, sin guarda (r)
              m1                                     -- Memoria acual
              (ClassModel._DataType m)               -- Elementos de Entrada
              (DataType.id)                          -- Identificador
              (DataType2Type)                        -- Funcion de Regla
            )
            where m1 = newMemory [] [] [] []         -- Memoria inicial

Class2Relational : ClassModel -> RelationalModel
Class2Relational m = Class2Relational_phase2 (Class2Relational_phase1 m)
```

Apéndice E

Compilador

E.1. Km3 a Agda

Implementación de un parser basada en javacc.

```
options {
  JAVA_UNICODE_ESCAPE = true;
}

PARSER_BEGIN(Km3Parser)

import java.util.*;

public class Km3Parser {

  static ArrayList<Paquete> paquetes = new ArrayList<Paquete>();
  static Paquete paquete = null;

  public static void main(String args[]) {
    Km3Parser parser = null;
    StringBuilder resultado = new StringBuilder();
    resultado.append("module " + args[0] + " where\n");

    resultado.append("open import Data.Nat\n");
    resultado.append("open import Data.String\n");
    resultado.append("open import Data.Bool\n");
    resultado.append("open import Data.List\n");
    resultado.append("open import Data.Maybe\n");

    if (args.length > 1) {
      for(int i=1;i<args.length;i++) {
        System.out.println("Leyendo archivo " + args[i] + ".km3");
```

```
try {
  if(i==1)
    parser = new
     Km3Parser(new java.io.FileInputStream(args[i] + ".km3"));
  else
    parser.ReInit(new java.io.FileInputStream(args[i] + ".km3"));

  paquetes = new ArrayList<Paquete>();

  try {
    parser.Model();
    resultado.append("\n");
    resultado.append("module " + args[i] + "Km3 where\n");
    for(Paquete p : paquetes){
      if(p.clases.size()>0){
        resultado.append("\n");
        resultado.append("module Pkg" + p.nombre + " where\n");
        resultado.append("");
        resultado.append("mutual\n");
        resultado.append("");
        for(Clase c : p.clases){
          if(!c.esAbstracta){
            resultado.append("\n");
            resultado.append("  record " + c.nombre + " : Set where\n");
            resultado.append("    constructor new" + c.nombre + "\n" );
            resultado.append("    field id : String\n" );

            ArrayList<Campo> campos = p.buscarCampos(c);
            boolean primero = true;
            for(Campo m:campos){

              boolean poly =  p.buscarSubClases(m.tipo).size() >0;

              if(m.esMultiple)
                resultado.append("          " +
                m.nombre + " : List " + (poly?"-":"")
                + m.calcularTipo() + "\n");
              else
                resultado.append("          " +
                m.nombre + " : " + (poly?"-":"")
                + m.calcularTipo() + "\n");
            }
          }

          ArrayList<String> subs = p.buscarSubClases(c.nombre);

          if(subs.size()>0){
            resultado.append("\n");
            resultado.append("  data -" + c.nombre + " : Set where\n");
            for(String sub: subs) {
```

```
                    boolean trans =  p.buscarSubClases(sub).size() >0;

                    resultado.append("    " + c.nombre +
                    "-From-" + sub + " : " + (trans?"-":"") + sub +
                     " -> -" + c.nombre + "\n" );
                  }
                }
              }
            }
          }


          for(Paquete p : paquetes){
            if(p.clases.size()>0){
              resultado.append("\n");
              resultado.append("  record " + p.nombre
              + "Model : Set where\n");
              resultado.append("    constructor new"
              + p.nombre + "Model\n");
              boolean primero = true;
              for(Clase c : p.clases){
                if(!c.esAbstracta){
                  resultado.append((primero?"    field ":"          ") +
                                   "_" + c.nombre + " : List "
                                    + c.nombre + "\n");
                  primero = false;
                }
              }
            }
          }

        } catch (ParseException e) {
          System.out.println("Se encontraron errores en el modelo de entrada: " + e.getM
        }
      } catch (java.io.FileNotFoundException e) {
        System.out.println("Archivo no encontrado: " + e.getMessage());
        return;
      }
    }

    System.out.println(resultado.toString());

  } else {
    System.out.println("uso: <nombre_archivo> <modelo_1> ... <modelo_n>");
    return;
  }

}
```

```
}

PARSER_END(Km3Parser)


SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}
SPECIAL_TOKEN : /* COMMENTS */
{
  <SINGLE_LINE_COMMENT: "--" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}

TOKEN : /* RESERVED WORDS AND LITERALS */
{
  < ABSTRACT: "abstract">
| < ATTRIBUTE: "attribute">
| < CLASS: "class">
| < CONTAINER: "container">
| < DATATYPE: "datatype">
| < ENUMERATION: "enumeration">
| < LEFTBRACKET: "[">
| < LITERAL: "literal">
| < MANY: "*" >
| < OPPOSITEOF: "oppositeOf">
| < ORDERED: "ordered">
| < PACKAGE: "package">
| < REFERENCE: "reference">
| < RIGHTBRACKET: "]">
| < SCORE: "-">
| < SEMICOLON: ";" >
| < SUPERTYPE: "extends" >
| < TWOPOINTS: ":">
}

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
| < INT: <DIGIT> (<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
```

```
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}



/*****************************************
 *          GRAMATICA DE KM3             *
 *****************************************/

/*
 * Program structuring syntax follows.
 */

void Model() :
{

}
{
  (Package())*
```

```
  <EOF>
}

void Package() :
{
  String name;
  paquete = new Paquete();
}
{
  "package" name=<IDENTIFIER>.image "{" Classifiers() "}"

  {
    paquete.nombre = name;
    paquetes.add(paquete);
    paquete = new Paquete();
  }
}

void Classifiers() :
{}
{
  (Classifier())*
}

void Classifier() :
{
  Clase clase;
}
{
  clase=Class() { paquete.clases.add(clase); } |
  DataType() |
  clase = Enumeration() { paquete.clases.add(clase); }
}


Clase Class() :
{
  String nombre;
  ArrayList<String> superClases;
  ArrayList<Campo> campos = new ArrayList<Campo>();
  boolean esAbstracta=false;
}
{
  ["abstract" { esAbstracta=true; }]
  "class" nombre=<IDENTIFIER>.image superClases=SuperTypes()
  "{"
      [ campos=Features() ]
  "}" {
        Clase clase = new Clase();
        clase.nombre = nombre;
```

```
        clase.superClases = superClases;
        clase.campos = campos;
        clase.esAbstracta = esAbstracta;
        return clase;
      }
}

ArrayList<String> SuperTypes() :
{
   ArrayList<String> superClases = new ArrayList<String>();

}
{
   ["extends" superClases=TypeList()] {return superClases;}
}

ArrayList<String> TypeList() :
{
  ArrayList<String> lista = new ArrayList<String>();
  ArrayList<String> subLista;
  String tipo;
}
{
  tipo=TypeRef() { lista.add(tipo); }
  [ "," subLista=TypeList()
    {
      lista.addAll(subLista);
    }
  ] { return lista; }
}

ArrayList<Campo> Features() :
{
  ArrayList<Campo> lista = new ArrayList<Campo>();
  ArrayList<Campo> subLista = new ArrayList<Campo>();
  Campo c;
}
{
  c=Feature() [ subLista=Features() ]
  {
    lista.add(c);
    lista.addAll(subLista);
    return lista;
  }
}

Campo Feature() :
{
  ArrayList<Campo> lista = new ArrayList<Campo>();
  Campo c = null;
```

```
}
{
  (
  c=Attribute() |
  c=Reference()
  ) { return c;}
}

Campo Attribute() :
{
  Campo campo = new Campo();
  String nombre, tipo;
  boolean multi=false;
}
{
  "attribute" nombre=Name() multi=Multiplicity() ":" tipo=TypeRef() ";"
  {
    campo.nombre = nombre;
    campo.tipo = tipo;
    campo.esMultiple = multi;
    return campo;
  }
}

Campo Reference() :
{
  Campo campo = new Campo();
  String nombre, tipo;
  boolean multi;
}
{
  "reference" nombre=Name() multi=Multiplicity()
  ["container"] ":" tipo=TypeRef() ["oppositeOf" Name()] ";"
   {
    campo.nombre = nombre;
    campo.tipo = tipo;
    campo.esMultiple = multi;
    return campo;
  }
}

boolean Multiplicity() :
{
  boolean multiValued = false;
}
{
  [ multiValued=Bounds() ["ordered"] ] { return multiValued; }
}

boolean Bounds() :
```

```
{}
{
    LOOKAHEAD(3) "[" <INT> "-" <INT> "]"  {return true;} |
    LOOKAHEAD(3) "["<INT> "-" "*" "]" {return true;} |
    "[" "*" "]" {return true;}
}

void DataType() :
{
  String name;
}
{
  "datatype" name=<IDENTIFIER>.image ";"
}

Clase Enumeration() :
{
  Clase e = new Clase();
  String nombre;
  ArrayList<String> valores = new ArrayList<String>();
}
{
  "enumeration" nombre=<IDENTIFIER>.image "{" valores=Literals() "}"
  {
    e.nombre = nombre;
    e.valores = valores;
    e.esEnumerador = true;
    return e;
  }
}

ArrayList<String> Literals() :
{
  ArrayList<String> valores=new ArrayList<String>();
  ArrayList<String> subList=new ArrayList<String>();
  String nombre;
}
{
    nombre=Literal() [ subList=Literals() ] {
    valores.add(nombre);
    valores.addAll(subList);
    return valores;
  }
}

String Literal() :
{
  String name;
}
{
```

```
  "literal" name=Name() ";" { return name; }
}

String TypeRef() :
{
  String nombre = "";
}
{
  nombre=Name() { return nombre; }
}

String Name() :
{
  Token tk;
}
{
  tk=<IDENTIFIER>
  ( LOOKAHEAD(2) "." <IDENTIFIER>
  )* { return tk.image;}
}
```