Especificación y Verificación de Transformaciones de Modelos

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Estudiantes: Horacio López, Fernando Varesi, Marcelo Viñolo
Tutores: Daniel Calegari, Carlos Luna

Instituto de Computación, Facultad de Ingeniería, UDeLaR

Montevideo, Julio 2010

Resumen

El Desarrollo de Software Guiado por Modelos (Model-Driven Development, MDD) es un enfoque de ingeniería de software basado en el modelado de un sistema como la principal actividad del desarrollo y la construcción del mismo guiada por transformaciones de dichos modelos. Su éxito depende fuertemente de la disponibilidad de lenguajes y herramientas apropiados para realizar las transformaciones y validar su corrección. El proyecto de grado documentado en este informe provee un relevamiento del estado del arte de los lenguajes y las herramientas de transformación de modelos, y un análisis de diferentes técnicas de verificación de transformaciones. Los lenguajes y las herramientas fueron categorizados según sus enfoques y descritos en base a una taxonomía que resalta sus principales características. De igual forma, se agruparon las diferentes técnicas de verificación en categorías y se mencionaron sus fortalezas y debilidades. Por último, se desarrolló un prototipo que constituye un primer paso hacia la semi-automatización de una técnica de verificación de transformación de modelos, basada en la utilización de métodos formales. Entre los resultados de este proyecto se destacan la publicación de dos reportes técnicos y el desarrollo del prototipo mencionado.

Palabras clave: Desarrollo de Software Guiado por Modelos - Transformación de Modelos - Lenguaje de Transformación de Modelos - Herramienta de Transformación de Modelos - Verificación de Transformación de Modelos

Índice general

Capítulo 1

Introducción

Los lenguajes de tercera generación de propósito general (ej: Java, C#, Python) han demostrado ser una herramienta muy potente para el desarrollo de diversos sistemas. Sin embargo, no han logrado resolver problemas de integración y portabilidad del sistema desarrollado con las diferentes plataformas tecnológicas que lo implementan. Este hecho resulta en un considerable esfuerzo por parte de los desarrolladores para adaptar las aplicaciones a las características y complejidades de cada plataforma con sus respectivas versiones. Otro problema importante, es que no proveen por sí mismos una representación conceptual del sistema que sea fácilmente visualizada y comunicada en un equipo de desarrollo [56].

El Desarrollo de Software Guiado por Modelos (Model-Driven Development, MDD) es un enfoque de ingeniería de software que ofrece soluciones para los problemas planteados. Este enfoque se basa en el modelado de un sistema como la principal actividad del desarrollo. La utilización de modelos permite una mejor visualización, ya que la representación del modelo puede ser facilmente adaptada a las necesidades y el conocimiento de los actores involucrados. A modo de ejemplo, un matemático podría manejar modelos en su área (ejemplo: redes de Petri [54]) sin tener conocimientos de programación general. Otro de los objetivos de este enfoque es posibilitar la resolución de los problemas de integración mediante la separación de la especificación de las funcionalidades, de las plataformas tecnológicas que las implementan. La utilización de modelos independientes de plataforma (Platform Independent Models, PIM) es fundamental para llevar a cabo este objetivo [33]. De esta forma, la construcción de los sistemas en este paradigma es guiada por transformaciones de dichos modelos, terminando en la generación automática de modelos y código específicos para una plataforma (Platform Specific Models, PSM). En la actualidad, empresas como Motorola están empleando este paradigma y desarrollando enfoques dentro del mismo para lograr introducir automatismo en sus procesos de desarrollo [8].

Como se puede apreciar, las transformaciones de modelos juegan un papel clave en MDD. Las aplicaciones dentro de MDD comprenden, entre otras, la generación de modelos de bajo nivel (llegando a código fuente) a partir de modelos de alto nivel, la sincronización entre modelos (frente a modificaciones en un modelo, otro debe actualizarse), la evolución de los mismos (ejemplo: realizando un refactoring), y las tareas de ingeniería inversa de modelos de bajo nivel para generar modelos de mayor nivel de abstracción [19]. De forma de asegurar la correctitud del sistema desarrollado, se deben contar con técnicas de verificación que permitan verificar la correctitud de las transformaciones involucradas.

Estas consideraciones motivaron el trabajo del proyecto de grado documentado en este informe, en el contexto de un proyecto de investigación general en esta área del grupo COAL y el grupo de Métodos Formales del Instituto de Computación (InCo). El objetivo principal de este proyecto de grado fue realizar un estudio del estado del arte sobre lenguajes y herramientas existentes para la especificación y verificación de transformaciones de modelos de software. Este relevamiento fue documentado en dos reportes técnicos publicados y anexos a este informe: "Estado del Arte de Lenguajes y Herramientas

de Transformación de Modelos" ([39], de aquí en más *primer reporte técnico*) y "Estado del Arte de Verificación de Transformación de Modelos" ([40], de aquí en más *segundo reporte técnico*). Un resultado adicional es el desarrollo de un prototipo de herramienta que asista al usuario en la aplicación de un enfoque de verificación de interés para el proyecto de investigación general. Este prototipo se enmarca dentro del método deductivo de verificación descrito en [15], el cual se documenta en el segundo reporte técnico y se resume en la sección 4.3 de este informe.

En el capítulo 2 del informe se presentarán los conceptos principales en torno a la transformación de modelos. Se definirán los elementos que participan en la especificación de estas transformaciones y que serán referenciados posteriormente. A continuación, el capítulo 3 presentará un resumen del relevamiento realizado para el primer reporte técnico. En primer lugar se expondrán las características que permiten describir enfoques de transformación de modelos, para luego presentar las categorías principales en los que se agrupan. El capítulo 4 resume el análisis realizado en el segundo reporte técnico sobre las diferentes técnicas de verificación de transformación. El último objetivo de este proyecto es documentado en el capítulo 5, donde se describe el contexto del prototipo realizado, el alcance del mismo y las decisiones de diseño. El apéndice A describe en detalle un caso de estudio que sirve como demostración del prototipo. A continuación, el apéndice B describe los detalles técnicos del prototipo. Por último, el apéndice C muestra las diferentes fases del proyecto y su duración.

Capítulo 2

Transformación de Modelos

2.1. Conceptos básicos de transformación de modelos

Un *modelo* es una abstracción de un sistema que permite a diferentes actores describir un sistema y efectuar cambios sobre el mismo [19]. Esta definición es muy amplia y comprende desde código fuente de programas (al tratarse de una abstracción del lenguaje máquina que se genera) hasta especificaciones de interfaz, esquemas de datos o máquinas de estado. Los modelos pueden expresarse mediante diferentes lenguajes como texto plano (ej: código Java, XML, KM3 [41]) o diagramas (ej: UML). Un *metamodelo* es un caso particular de modelo que describe un conjunto de modelos especificando la sintaxis abstracta de la notación de modelos. Se dice entonces que estos modelos conforman a este metamodelo.

Los modelos descritos sirven como entrada y son la salida de las transformaciones de modelos. Tomaremos la definición de *transformación de modelos* planteada en [34], en donde se define una transformación como la generación automática de un *modelo destino* desde un *modelo origen*, de acuerdo a una definición de transformación. Una *definición de transformación* es un conjunto de reglas de transformación que describe cómo un modelo en el lenguaje origen puede ser transformado en un modelo en el lenguaje destino. Una *regla de transformación* es una descripción de cómo una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.

Los elementos básicos de toda transformación de modelos se pueden apreciar en la Figura 2.1. Aquí se puede ver como una transformación se define en base a metamodelos origen y destino. Esta definición de transformación es tomada por el motor de transformaciones el cual genera un modelo destino a partir del modelo origen. Este esquema se puede extender naturalmente para transformaciones que operan sobre varios metamodelos de entrada y salida. Las transformaciones representadas por esta figura se denominan transformaciones modelo-modelo, pues llevan de un modelo origen a un modelo destino que conforman con sus respectivos metamodelos. Otra categoría general de transformaciones son las modelo-texto, donde el resultado de la transformación es un texto que no conforma con un metamodelo específico (ej: código fuente de un programa). Estas y otras categorizaciones de las transformaciones de modelos serán presentadas en el capítulo siguiente.

El concepto de *lenguaje de transformación* de modelos no cuenta aún con una definición académica consensuada. En nuestro trabajo, aproximamos la definición de lenguaje de transformación como aquel que permite definir reglas de transformación que serán ejecutadas por un motor, las cuales se basan en una especificación formal de metamodelos para las instancias de los modelos de origen y destino de la transformación.

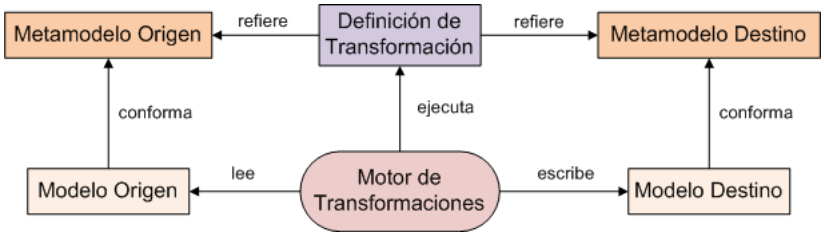

Figura 2.1: Elementos de una transformación de modelos

2.2. Caso de estudio: transformación de modelo de clases UML a modelo relacional

A continuación se describirá en forma general, un caso de estudio de transformación de modelo de clases UML a modelo relacional. Este caso de estudio es el ejemplo más común en el ámbito de MDD y fue seleccionado para la aceptación de los trabajos publicados en el workshop realizado en el 2005 sobre cuestiones prácticas de las transformaciones de modelos [11]. A pesar de su relativa simplicidad, este ejemplo permite mostrar las características más relevantes de las transformaciones de modelos. En particular, se utilizó en el primer reporte técnico para describir los distintos enfoques de transformación de modelos y será referenciado en este informe para mostrar ejemplos de reglas de transformación.

Para fijar ideas, se presentarán ejemplos de metamodelos origen y destino para dicha transformación, y luego se mostrarán ejemplos de instancias de estos metamodelos. En la Figura 2.2 se presenta un metamodelo que representa los diagramas de clases UML a transformar.

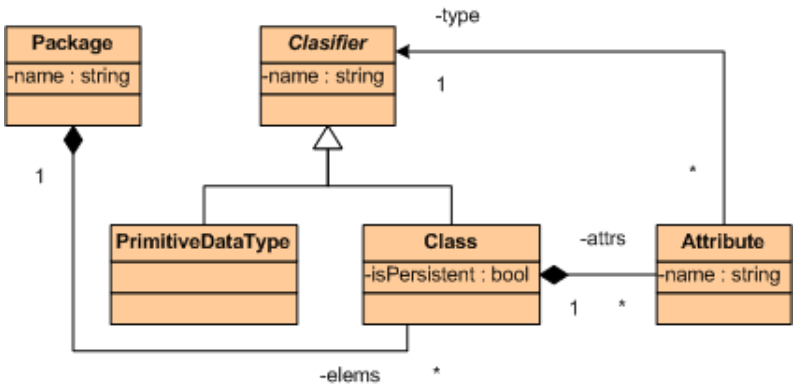

Figura 2.2: Metamodelo de clases UML origen

De acuerdo con este metamodelo las clases (*Class)* poseen un nombre (atributo *name*) y atributos (asociación con la clase *Attribute*). Estas clases pueden ser declaradas como persistentes o no persistentes (atributo *isPersistent*). Los atributos de las mismas poseen un clasificador (*Classifier*) que corresponde al tipo de los mismos, el cual puede ser un tipo primitivo (*PrimitiveDataType*) o una clase. Notar que un atributo cuyo clasificador es una clase es la representación de una asociación en UML. En la Figura 2.3 se puede observar una posible instancia del metamodelo origen. Entre otros elementos, esta instancia modela una clase Customer que tiene asociada la clase Address (pertenecientes al package App) y posee como atributo una cadena de caracteres name.

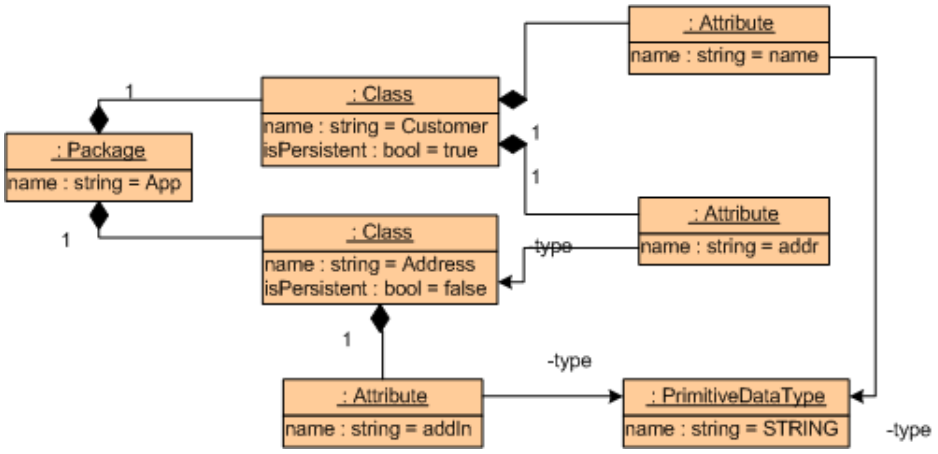

Figura 2.3: Modelo de clases de UML

El modelo destino respetará un metamodelo que describa a los modelos relacionales. Dicho metamodelo puede observarse en la Figura 2.4.

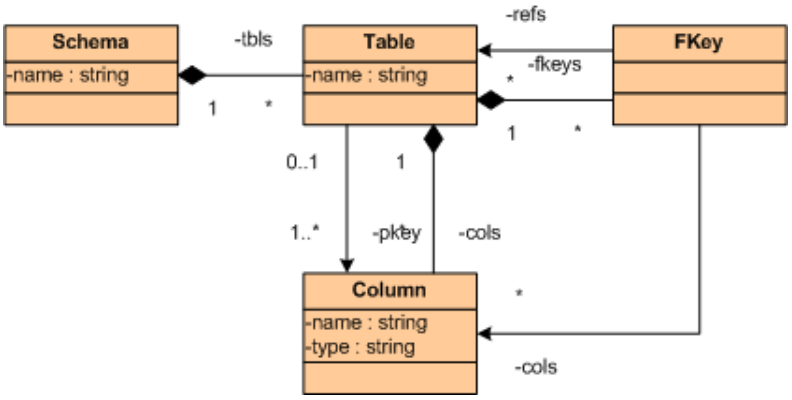

Figura 2.4: Metamodelo relacional destino

Todo esquema destino contará con un número de tablas. Cada tabla está compuesta de columnas (asociación con *Column*), de las cuales alguna será la clave primaria de la misma. Una tabla puede estar asociada a cero o más de una claves foráneas (foreign key). Cada clave foránea puede referirse a un conjunto de columnas de una tabla que constituyen la clave.

Luego de haber presentado los metamodelos que intervendrán en la transformación, veamos algunas de las tareas básicas que deberá realizar dicha transformación:

- Las clases marcadas como persistentes serán transformadas en tablas
- Los atributos y asociaciones de las clases serán transformados en columnas de las respectivas tablas
- Cada atributo de un tipo primitivo es transformado a una columna. Si el atributo es considerado un identificador, la columna resultado será clave primaria de la tabla
- Cada atributo cuyo tipo no sea primitivo, se resolverá y se crearán las estructuras necesarias en el modelo destino que lo representen.

En la Figura 2.5 puede observarse el modelo resultante tras aplicar la transformación al modelo de la Figura 2.3.

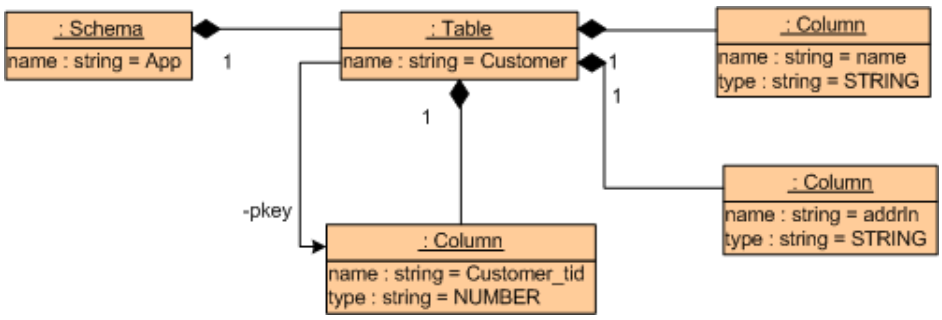

Figura 2.5: Modelo relacional destino

Si el lector desea profundizar en los detalles del ejemplo o en un ejemplo completo de trasformación para el caso de estudio presentado, puede consultar [11] y [31].

Capítulo 3

Lenguajes y Herramientas de Transformación de Modelos

Este capítulo resume el análisis y las conclusiones documentadas en el primer reporte técnico [39] sobre el estado del arte de los lenguajes y herramientas de transformación de modelos. El reporte técnico describe los lenguajes en el contexto del enfoque de transformación que los inspira, y posteriormente menciona las principales herramientas con las cuales se trabaja.

Los trabajos dedicados al estudio de transformaciones de modelos frecuentemente refieren en su análisis a aproximaciones o enfoques de transformación en lugar de considerar lenguajes de transformación. El primer motivo de esta distinción es que el área de investigación es relativamente reciente, por lo que aún se trabaja activamente en la conformación de estándares y los distintos trabajos realizados en el área apuntan en diferentes direcciones. El segundo motivo es que algunos de estos enfoques no implican necesariamente la utilización de un lenguaje de transformación, usando en su lugar herramientas que no manejan un lenguaje de transformación. Estas herramientas no manejan una especificación flexible de la transformación, sino que realizan un conjunto acotado de operaciones sobre los modelos. Los diferentes enfoques de transformación emplean un medio u otro dependiendo del dominio y el propósito de la transformación.

Este capítulo está organizado de la siguiente manera. En la sección 3.1 mencionaremos las características que describen a los enfoques de transformación de modelos. Estas características permiten analizarlos y determinar cual es el enfoque más apropiado para un problema determinado. En las secciones posteriores se describirán los aspectos generales de los distintos enfoques de desarrollo de transformaciones modelo-modelo, brindando ejemplos de cada uno.

3.1. Caracterización de los enfoques de transformación de modelos

Dada la gran diversidad de enfoques existentes para la transformación de modelos, se requiere de criterios que permitan clasificar a estos enfoques y seleccionar el más adecuado para resolver un problema dado. Uno de los trabajos más destacados en este sentido es el realizado por Czarnecki y Helsen [19], el cual brinda una taxonomía que provee una base sólida para trabajos posteriores y para el análisis realizado en este proyecto. Otro trabajo de interés se documenta en [42], aunque este desplaza la atención del enfoque de la transformación hacia las características de las herramientas (ej: usabilidad). Toda la bibliografía relacionada a la clasificación y el análisis de enfoques de transformación se basan fuertemente en estas

publicaciones. Estos trabajos constituyen el framework más genérico desarrollado hasta el momento para analizar las transformaciones sin profundizar en un área de aplicación específica.

En estos trabajos se realiza una primer distinción entre transformaciones modelo-modelo y transformaciones modelo-texto. Como se mencionó en la sección 2.1, un texto puede ser considerado como modelo pero este no conforma con un metamodelo al tratarse de una secuencia de caracteres. Existen dos enfoques básicos para las transformaciones modelo a texto: visitor-based y template-based. En el enfoque *visitor-based* se recorre la representación interna del modelo para generar progresivamente la salida. En el enfoque *template-based* trabaja con un texto objetivo (*template*) que contiene segmentos de metácodigo, el cual es procesado y reemplazado posteriormente. En el primer reporte técnico se puede encontrar una explicación más detallada de estos enfoques y ejemplos de herramientas que lo aplican. Por otro lado, las transformaciones modelo-modelo son más expresivas y son de especial interés para este proyecto de grado, por lo que el reporte técnico mencionado y el contenido restante de esta sección se centrará en las mismas.

En [19] se aplican técnicas de análisis de dominio sobre los diferentes enfoques de transformación de modelos. De esta forma se obtuvo un diagrama de características, que refleja los aspectos comunes y variables que caracterizan al conjunto de instancias de un concepto. En este caso, el concepto es el conjunto de aproximaciones a la transformación de modelos. En particular, se resalta que algunas de las características no necesariamente estarán presentes en los todos los enfoques. Este diagrama se presenta y explica detalladamente en el primer reporte técnico, donde se toma el diagrama y se complementa para describir los enfoques analizados.

Las principales características de los enfoques de transformación extraídas del diagrama son:

- los mecanismos de especificación de la transformación provistos (ej: restricciones expresadas en OCL),
- la posibilidad de recibir varios modelos origen y producir varios modelos destino,
- la posibilidad y/o requerimiento de producir el resultado sobre el propio modelo origen (*in-place*) o generar un nuevo modelo como salida,
- la *incrementalidad* (cómo responde la transformación frente a cambios en el modelo origen o en el modelo destino, considerando esfuerzo de reanálisis y de actualización),
- la posible multidireccionalidad de su ejecución
- y los mecanismos de trazabilidad[1].

En lo que respecta a los metamodelos es importante analizar:

- si la transformación opera con un único metamodelo que describe el modelo origen y el modelo destino (*transformación endógena*) o con diferentes metamodelos (*transformación exógena*).
- si los metamodelos involucrados en la transformación se encuentran en el mismo nivel de abstracción (*transformación horizontal*) como en el caso de un refactoring o una traducción, o en diferentes niveles (*transformación vertical*) como en el caso de un refinamiento.

Como se puede apreciar en el Cuadro 3.1 estas dimensiones de análisis son ortogonales. La tabla presenta ejemplos para las posibles combinaciones de estas características. A modo de ejemplo, un refactoring es una transformación donde se preserva el nivel de abstracción y el metamodelo empleado.

Algunas de las características relacionadas a las reglas de transformación a tener en cuenta son:

[1] Algunos enfoques no proveen soporte explícito para trazabilidad, pero permiten implementarlo

	horizontal	**vertical**
endógena	refactoring	refinamiento
exógena	traducción o migración de lenguaje	generación de código

Cuadro 3.1: Dimensiones ortogonales de las transformaciones

- la posible existencia de una separación sintáctica entre lado izquierdo (patrón sobre el cual se aplica la regla) y lado derecho (patrón que determina el cambio producido),
- la posibilidad de especificar precondiciones (*guardas*) para la aplicación,
- la posibilidad de parametrizar las reglas para permitir su reuso,
- la posible utilización de estructuras intermedias que no pertenecen al modelo origen ni al modelo destino,
- la posibilidad de utilizar mecanismos como reflection y aspects[2],
- cómo se determina donde aplicarlas, ya sea mediante métodos deterministas (ej: recorrer una representación interna), no determinista (ej: aplicación concurrente en todos los elementos que cumplan con cierto criterio) o interactivos con el usuario de la transformación,
- y en qué forma son aplicadas, cómo se determina el orden y cómo se resuelven posibles conflictos.

Estas características fueron complementadas en el primer reporte técnico con ideas tomadas de [42] y criterios propios que listamos a continuación. Estos criterios refieren a las herramientas que implementan los enfoques mencionados.

- *Areas de aplicación*: esta característica pretende describir la clase de problemas que la herramienta puede resolver, los casos de estudio en que fue aplicada y las aplicaciones industriales.
- *Sugerencia de aplicación de transformaciones*: se relaciona con la interactividad con el usuario descrita en la sección previa y describe en que forma la herramienta asiste al usuario para descubrir aquellas reglas de transformación que pueden ser de interés en un determinado modelo a transformar.
- *Tipo de licencia de distribución*: la herramienta puede ser software propietario o distribuírse con algún tipo de licencia de código abierto.
- *Historia de la herramienta*: comprende el origen de la misma (por ejemplo: académico o emprendimiento privado), su antigüedad y la continuidad de su soporte.
- *Soporte brindado al usuario*: refiere al tipo de documentación brindada: manuales, tutoriales, ejemplos, foros y todo tipo de soporte disponible.
- *Comunidad de usuarios y aceptación*: refiere a la cantidad de usuarios activos de la herramienta y la aceptación obtenida por la misma dentro de la comunidad.
- *Mecanismos ofrecidos para garantizar correctitud de las transformaciones*: refiere a las garantías ofrecidas por la herramienta de que el resultado producido es correcto sintáctica y semánticamente.

[2] El mecanismo de reflection permite una mayor navegabilidad de las propiedades de los elementos a transformar y potencialmente también de las reglas que los afectan. La utilización de aspects brinda los beneficios de mayor modularidad y facilidad para separar distintas visiones sobre las transformaciones.

- *Manejo de modelos incompletos o inconsistentes*: refiere a como maneja la herramienta situaciones en las cuales el modelo no está especificado por completo o contiene inconsistencias.
- *Mecanismos ofrecidos para verificación y validación de transformaciones*: a diferencia de la correctitud, esto se refiere a los mecanismos ofrecidos para verificar que una transformación desarrollada presenta el comportamiento deseado por el programador. Un ejemplo de esto podría ser asistir al usuario en la generación de casos de prueba.
- *Escalabilidad de transformaciones y de modelos*: refiere a la capacidad de la herramienta para manejar complejidad y tamaño creciente ya sea en las transformaciones o en los modelos a transformar.

3.2. Enfoques de manipulación directa

En estos enfoques se trabaja con un lenguaje de propósito general y con una representación de los modelos adecuada para el lenguaje. Esta representación debe emplear estructuras de datos (ej: arreglos, árboles, grafos) que pueden ser manipuladas por el programa. La transformación se implementa como un programa que manipula la representación para generar el modelo destino. Al utlizarse un lenguaje de propósito general, no se cuentan con mecanismos específicos para transformación de modelos (ej. trazas, multidireccionalidad de ejecución, planificador de ejecución).

Este enfoque resulta adecuado para problemas simples, donde la especificación de los modelos y la transformación no sean modificados. En caso de requerir agregar nuevos elementos al modelo, modificar la representación del modelo o modificar la lógica de la transformación, se deberá reescribir el programa que implementa dicha transformación. Por este motivo es un enfoque poco flexible en términos de reusabilidad del software.

Dentro de esta categoría, en el primer reporte técnico se describió el framework SiTra [47] que utiliza los lenguajes Java y C# para la manipulación de modelos.

3.3. Enfoques operacionales

Los enfoques operacionales se asemejan al paradigma de programación imperativa. Toman como base los enfoques de manipulación directa e incorporan facilidades para brindar soporte dedicado a la transformación de modelos. Estas facilidades pueden ser compatibilidad con diferentes lenguajes de metamodelado (ej: Ecore), mecanismos de trazabilidad y/o mecanismos de reuso y composición de transformaciones.

El orden de ejecución de la transformación se encuentra explícitamente definido en la especificación de la misma. Al igual que los lenguajes imperativos de propósito general, ofrecen mecanismos para control de flujo como funciones, condiciones y bucles. Estas características producen que este enfoque resulte intuitivo para los desarrolladores no familiarizados con transformaciones de modelos.

Esta categoría comprende enfoques como Kermeta [25] y QVT-Operational [43]. El Código 3.1 muestra un extracto del ejemplo de transformación de modelo de clases UML a modelo relacional, codificado en el lenguaje QVT-Operational.

Código 3.1 Ejemplo de enfoque operacional - QVT-Operational

```
main() { --punto de entrada de la transformación
-- primer paso
srcModel.objects()[Class]->map class2table();
-- segundo paso
srcModel.objects()[Association]->map asso2table();
}
--regla que mapea clases persistentes a tablas,
--con una columna por atributo
mapping Class::class2table () : Table

     when {self.kind='persistent';}

{

     -- sección para realizar inicializaciones
     init {
          --invocación a la regla attr2LeafAttrs
          self.leafAttributes := self.attribute->map attr2LeafAttrs("","");
     }
     -- seccion que expande la tabla
     name := 't_' + self.name;
     column := self.leafAttributes->map leafAttr2OrdinaryColumn("");
     -- sección anidada para expandir las claves primarias
     key_ := object Key {
          name := 'k_'+ self.name;
          column := result.column[kind='primary'];
     };
     ...

}
```

En el ejemplo se pueden observar algunas de las características de los enfoques operacionales como la especificación explícita de el orden de ejecución de las reglas definido en el método main. Además se observa la palabra reservada mapping utilizada para expresar relaciones entre elementos del modelo origen y destino. Este tipo de definición provee de una estructura común para la definición de una regla de transformación: guardas, inicialización de atributos y una sección de código imperativo donde es posible definir la salida e invocar a otras reglas.

3.4. Enfoques relacionales

Esta categoría comprende aquellos enfoques declarativos donde el concepto principal son las relaciones matemáticas. Se deben establecer relaciones entre los tipos de elementos del modelo origen y el modelo destino mediante restricciones. Estos enfoques manejan un alto nivel de abstracción que resulta en un código legible.

La creación de los elementos en el modelo destino es implícita (se infiere de las reglas de transformación pero no es especificada directamente por el desarrollador), por lo que estos enfoques no presentan efectos secundarios en la salida. Además, la mayoría de los enfoques relacionales no permiten actualizaciones in-place, es decir que no es posible ejecutar transformaciones para un mismo modelo origen y destino. A diferencia de los enfoques imperativos, el orden de ejecución no es explícito. El desarrollador no puede asumir el orden en que se ejecutarán las reglas, aunque existen algunos mecanismos para influir en el orden (ej: establecer como precondición de una regla la existencia de un elemento generado por otra regla).

Los desarrolladores no familiarizados con enfoques declarativos deben sortear ciertas dificultades iniciales para especificar la transformación. Esto es debido - entre otras causas - a la imposibilidad de determinar en qué orden se ejecutarán las reglas y a la dificultad de visualizar la interacción de las mismas. Para emplear apropiadamente estos lenguajes, es necesario razonar el problema en forma diferente a la empleada en los paradigmas imperativos.

Dentro de estos enfoques, el reporte técnico citado documenta Tefkat [2] y QVT-Relations [43]. El Código 3.2 ejemplifica la codificación en QVT-Relations del ejemplo de transformación entre diagrama de clases UML y modelo relacional.

Código 3.2 Ejemplo de enfoque relacional - QVT-Relations

```
transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)
{
     key Table (name, schema);
     key Column (name, owner);
     key Key (name, owner);
     --regla que mapea un paquete a un esquema
     top relation PackageToSchema
     {
          pn: String;
          checkonly domain uml p:Package {name=pn};
          enforce domain rdbms s:Schema {name=pn};
     }
     --regla que mapea cada clase persistente a una tabla
     top relation ClassToTable
     {
          cn, prefix: String;
          checkonly domain uml c:Class {namespace=p:Package {},
               kind='Persistent', name=cn};
          --creación de instancia
          enforce domain rdbms t:Table {schema=s:Schema {}, name=cn,
               column=cl:Column {name=cn+'_tid', type='NUMBER'},
               key=k:Key {name=cn+'_pk', column=cl}};
          when {
               PackageToSchema(p, s);
          }
          where {
               prefix = '';
               AttributeToColumn(c, t, prefix);
          }
     }
     ...
}
```

En este ejemplo, a diferencia del Código 3.1, no existe un método main como punto de partida de transformación. Es decir, no se puede definir de forma explícita el orden de ejecución. Además, la creación de los elementos de salida no es explícita, sino que se define de forma implícita al establecer los mapeos entre el modelo origen y el modelo destino.

3.5. Enfoques basados en transformaciones de grafos

Estos enfoques de transformación de modelos se basan en el trabajo teórico sobre transformación de grafos. En [29] se puede encontrar una descripción general de los elementos básicos en una transformación de grafos. Los grafos utilizados en este tipo de transformaciones son grafos tipados, con atributos y etiquetas. Las reglas de transformación se componen de un lado izquierdo (LHS) y un lado derecho (RHS), los cuales se expresan como patrones. El patrón LHS determina los elementos del grafo origen sobre los cuales se aplicará la regla, mientras que el patrón RHS determina los elementos a crear en el grafo destino.

Debido a la teoría de grafos subyascente, las transformaciones de grafo presentan propiedades matemáticas de interés para la verificación (ejemplo: alcanzabilidad de un nodo). Este hecho motiva en la medida en que esto sea posible, la representación de los modelos como grafos aún cuando esta representación no sea la más intuitiva. Esta traducción del modelo original permite aprovechar toda la potencia de las herramientas matemáticas. Estas características favorables para la verificación serán comentadas en la sección 4.2.

Para estos enfoques se emplean con frecuencia herramientas dedicadas a la transformacion de grafos como AGG [58] y AToM3 [21]. Otras aplicaciones emplean lenguajes como Viatra [60] y MOLA [22]. El Código 3.3 muestra un extracto del ejemplo de transformación de modelo de clases UML a modelo relacional, codificado en el lenguaje Viatra.

Código 3.3 Ejemplo de enfoques basados en grafos - Viatra

```
machine uml2reldb {

    //regla que define el punto de entrada de la transformación
    rule main (in UMLStr, in RefStr, in DBStr){
        call initModels(UMLStr, RefStr, DBStr);
        forall C below models("uml")
            with apply class2tableR(C) do skip;
        forall C below models("uml"), A below models("uml")
            with apply attr2columnR(C, A) do skip;
        forall A below models("uml")
            with apply attrOfStringTypeR(A) do skip;
        forall A below models("uml")
            with apply attrOfIntTypeR(A) do skip;
        forall A below models("uml")
            with apply assoc2tableR(A) do skip;
    }
    ...
    //regla que mapea una clase a una tabla
    gtrule class2tableR(in Cls) = {
        //precondiciones de la regla que aplican a LHS
        precondition pattern lhs (Cls, ClsNM) = {
            Class(Cls) below models("uml");
            NamedElement.name(N1, Cls, ClsNM);
            String(ClsNM) below models("uml");
        }
        //acciones de la regla
        action{
            let T = undef
            in let R = undef
            in let RS = undef
            in let RT = undef in seq {
              call createNewTable(value(ClsNM), T);
              call createPrimaryKeyInTable(T);
              new (class2table(R) in models("ref"));
              new (class2table.srcRef(RS, R, Cls));
              new (class2table.trgRef(RT, R, T));
              print("Class " + fqn(Cls) + "-> Table"
                + fqn(T) + "\n");
            }
        }
        ...
    }
```

Como se observa, este ejemplos se asemeja al de los enfoques operacionales donde los elementos en el modelo destino son construídos explícitamente y se define explícitamente el orden de ejecución en un método principal main. La ventaja del uso de este lenguaje en el ejemplo no radica en la forma de desarrollo de la transformación, sino en las posibilidades que brinda la representación de grafos.

3.6. Enfoques híbridos

El enfoque híbrido basicamente combina distintas características que proveen los enfoques anteriores para sumar sus ventajas[61].

Dentro de este enfoque se analizaron en el primer reporte técnico los lenguajes QVT [43], ATL [1] y RubyTL [18]. En este informe se describirán particularmente los lenguajes QVT y ATL, dado que el primero constituye un estándar de referencia para otros lenguajes y el segundo fue empleado en la implementación del caso de estudio descrito en el capítulo 5. Ambos lenguajes se encuentran descritos en detalle en el primer reporte técnico.

QVT

QVT resultó de particular interés en el relevamiento realizado por tratarse de un estándar para la transformación de modelos definido por el OMG. Depende de los estándares OCL y MOF[3] para la especificación de los metamodelos y modelos. En QVT solo son soportadas transformaciones de modelo a modelo, donde un modelo es una entidad que conforma con un metamodelo MOF. Además toda transformación QVT es un modelo, por lo que esta debe ajustarse también a un metamodelo MOF. Por lo tanto, la sintaxis abstracta de QVT en su conjunto conforma con un metamodelo MOF. Finalmente, es importante notar que la especificación QVT cuenta con una naturaleza híbrida (declarativa/imperativa).

En la Figura3.1 se observa la arquitectura de QVT. La parte declarativa de la especificación se encuentra estructurada en dos capas, donde cada capa define un lenguaje declarativo. Una de las capas define al lenguaje QVT Relations (QVT-R), el cual cuenta con una especificación de alto nivel amigable al usuario. La capa que define al lenguaje QVT Core (QVT-C) cuenta con una especificación de nivel más bajo. Se puede realizar una analogía entre estos dos lenguajes y la arquitectura Java, donde el lenguaje QVT-C juega el papel de Java Byte Code, el lenguaje QVT-R simboliza al lenguaje Java, y la transformación entre QVT-R y QVT-C puede representarse como la especificación del compilador Java que produce Byte Code. QVT-R brinda una especificación declarativa de las relaciones entre los modelos MOF, soporta reconocimiento de patrones y puede ser representado mediante una declaración textual o gráfica. El Código 3.2 ejemplifica la definición textual de una transformación expresada en QVT-R. QVT-C es un lenguaje tan potente como QVT-R y su semántica puede ser definida de forma más simple.

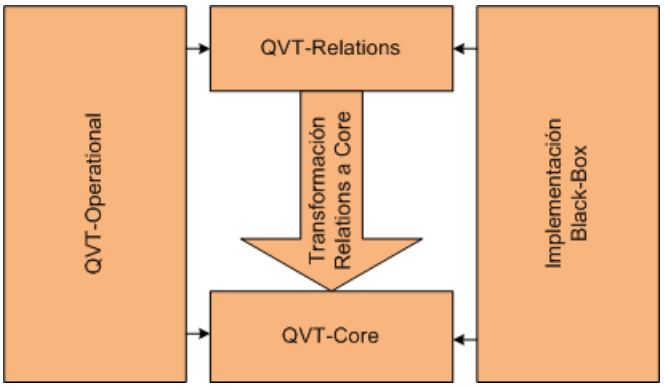

Figura 3.1: Arquitectura QVT

La parte imperativa de la especificación se encuentra definida por el lenguaje QVT Operational

[3] MOF es un estándar desarrollado por OMG para definir el UML y los profiles a emplear en MDE. Se pretende que sea un subconjunto de UML (auténtico metamodelado) y que permita definir transformaciones entre metamodelos [33].

(QVT-O). Provee mecanismos para extender el lenguaje QVT-R, combinando la naturaleza declarativa de QVT-R con su naturaleza imperativa. Las expresiones imperativas en QVT-O utilizan características funcionales de OCL y construcciones utilizadas en los lenguajes de propósito general (if-then-else, loops, variables, etc.). El Código 3.1 muestra un fragmento de código del ejemplo de transformación entre diagramas de clases UML a modelo relacional expresado en QVT-O.

Por último, la implementación BlackBox permite invocar facilidades de transformaciones expresadas en otros lenguajes (XSLT, XQuery). Es útil para integrar librerías y transformaciones no desarrolladas con QVT.

ATL

El lenguaje ATL es de particular interés debido a su utilización en la implementación del caso de estudio. Este lenguaje fue desarrollado como parte de la plataforma AMMA (ATLAS Model Management Architecture) siguiendo los lineamientos del QVT RFP [4]. La sintaxis abstracta de ATL es especificada mediante un metamodelo MOF, y provee de lenguajes en modo textual y gráfico para representar las reglas de transformación. Las transformaciones en ATL son unidireccionales: operan sobre modelos origen de sólo lectura (read only) y producen modelos destino de sólo escritura (write only). Durante la ejecución de una transformación, los modelos origen pueden ser navegados pero no se permiten cambios sobre ellos; en cambio los modelos destino no pueden ser navegados.

Las reglas de transformación en ATL se organizan en módulos. Un módulo contiene una sección de cabezal obligatoria, una sección de imports, y un conjunto de helpers y reglas de transformación. En el Código 3.4 se presenta un ejemplo de módulo para el caso de transformación de modelos UML a modelo relacional presentado en la sección 2.2.

Código 3.4 Módulo ATL

```
-- cabezales
module SimpleClass2SimpleRDBMS;
create OUT : SimpleRDBMS from IN : SimpleClass;
-- helpers
helper context SimpleClass!Class def :

     allAttributes : Sequence(SimpleClass!Attribute) =

          self.attrs->union(

              if not self.parent.oclIsUndefined()
              then  self.parent.allAttributes->select(attr | not

                self.attrs->exists(at | at.name = attr.name)  )

              else Sequence {} endif

          )->flatten();

...
-- reglas de transformacion
rule PersistentClass2Table{

     from

          c : SimpleClass!Class (

              c.is_persistent and c.parent.oclIsUndefined()

          )

     to

          t : SimpleRDBMS!Table (

              name <- c.name,

              ...

          )

}
```

La sección del cabezal da el nombre al módulo de transformación y declara los modelos de origen y los modelos de destino. A continuación se muestra un ejemplo para una sección de cabezal.

Un helper puede ser visto como la equivalencia en ATL a los métodos de Java. Hacen posible la reutilización de código ATL y pueden ser invocados desde distintos puntos de la transformación. Los helpers pueden ser usados para definir operaciones que realicen navegaciones sobre el modelo origen o para definir pseudo-atributos que faciliten la definición de las transformaciones. Pueden ser especificados en base a un tipo de OCL o a un elemento perteneciente al metamodelo origen (dado que el modelo destino no es navegable).

La regla de transformación es el constructor básico en ATL para expresar la lógica de la trasformación. Existen dos tipos de reglas que se corresponden con las dos formas diferentes de programación que provee ATL: las *matched rules* del estilo declarativo y las *called rules* del estilo imperativo.

Una *matched rule* está compuesta por un patrón origen y un patrón destino. El patrón origen especifica un conjunto de elementos del metamodelo origen o del conjunto de tipos disponible en OCL. Además, el patrón puede especificar una guarda definida como una expresión booleana en OCL. De acuerdo a esta especificación, el patrón es evaluado sobre un conjunto de elementos en el modelo origen para determinar si se debe aplicar la regla de transformación. El patrón destino está compuesto por un conjunto de elementos definidos en el metamodelo destino y un conjunto de bindings. Un *binding* especifica una expresión cuyo valor se utiliza para inicializar el valor de la expresión que se encuentra a su izquierda. Un caso particular de matched rule son las lazy rules. Una *lazy rule* es una regla de transformación que no es ejecutada si no es invocada en forma explícita por otra regla.

El Código 3.4 muestra la regla denominada *PersistentClass2Table* , que define el patrón origen *c* y el

patrón destino *t*. El patrón *c* es evaluado para los elementos de tipo *Class* del modelo origen *SimpleClass* que cumplen con la restricción *c.is_persistent and c.parent.oclIsUndefined()*. El patrón *t* genera el elemento *Table* del modelo de destino *SimpleRDBMS*, cuyo elemento *name* es inicializado por su respectivo binding a la derecha del simbolo <-.

Las *called rules* proveen a los desarrolladores facilidades de la programación imperativa. Pueden ser vistas como un tipo particular de helper, ya que deben ser llamadas explícitamente para ser ejecutadas y pueden aceptar parámetros. Una called rule puede ser invocada desde una sección de código imperativo en otra called rule o desde una matched rule.

Durante la ejecución de la transformación se crean en forma automática traceability links internos al motor de transformación. Estos links relacionan tres componentes: la regla, los elementos reconocidos y los nuevos elementos creados. Luego se evalúan y resuelven los bindings, para utilizar sus valores sobre los correspondientes elementos en el modelo de destino. La estrategia de ubicación de reglas dentro del modelo a transformar es determinista para las reglas definidas de forma imperativa, y no determinista para las reglas definidas de forma declarativa. La aplicación de reglas de forma interactiva no es soportada.

3.7. Resumen

El primer reporte técnico publicado sobre las herramientas y lenguajes para transformación de modelos provee un relevamiento del estado del arte en el área, que consideramos útil para conocer las alternativas existentes. Existe una gran diversidad de técnicas y lenguajes de transformación de modelos que abarcan desde técnicas de manipulación directa (ej: transformaciones desarrolladas en Java) hasta lenguajes dedicados como son las implementaciones basadas en el estándar QVT. Esta heterogeneidad también se manifiesta en las herramientas brindadas para especificar y ejecutar las transformaciones. El enfoque empleado es dependiente de la naturaleza del problema a resolver y no existe un único enfoque que contemple todas las posibles necesidades.

El reporte presenta una taxonomía que permite clasificar los diferentes enfoques de forma de conocer sus características y permitir la selección del más adecuado para resolver un problema específico. La tabla 3.2 muestra en forma resumida las principales características descritas en la sección 3.1 para los lenguajes analizados en el primer reporte técnico. Para cada uno de ellos se indican las caracterísiticas descritas en la sección, dejando en blanco aquellos para los cuales no se encontró información concluyente. El símbolo de tick indica que el lenguaje posee el atributo, mientras que la cruz indica que no lo posee.

			Kermeta	Tefkat	Mola	MATRA	Atom3	QVT Relations	QVT Operational	RubyTL	Epsilon	ATL	YATL
Reglas de transformación	Separación sintáctica		✗	✓	✗	✗	✓			✓	✓	✓	✓
	Multidireccionalidad		✗[1]	✗		✗	✗	✓	✗	✗[1]		✗	
	Precondiciones		✓	✓		✓	✓	✓	✓		✓	✓	
	Parámetros			✓				✓		✓		✓	✓
	Estructuras intermedias		✓		✓	✓			✓	✓		✓	
	Reflection		✓	✓						✓		✓	
	Aspects		✓										
Relación orígen/destino	Nuevo destino		✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓
	In-place			✗		✓	✓	✓	✓			✗	✓
Determinación de la ubicación	Determinista		✗		✓			✓	✓			✓	✓
	No determinista		✗			✓	✓			✓		✓	
	Interactiva		✗									✗	
Planificación de reglas	Forma	Implícita	✗	✓				✓				✓	
		Explícita	✓	✗	✓	✓[3]		✓	✓		✓	✓	✓
	Selección	Condición explicita	✗		✓							✓	
		No determinista	✗				✓						
		Resolución de conflictos	✗										
		Interactiva	✗				✓						
	Iteración de reglas	Recursión	✓									✓	
		Looping	✓		✓				✓	✓	✓		✓
		Fixpoint				✓	✓			✓			
	Fases		✗							✓			
Organización de reglas	Mecanismos para modularización		✓		✓	✓		✓		✓	✓	✓	
	Mecanismos para reuso		✓	✓	✓	✓			✓	✓	✓	✓	✓
	Estructura organizacional	Orientada al origen		✓	✓							✓	
		Orientada al destino						✓					
		Independiente	✓										
Incrementalidad			✗[1]	✓[2]				✓	✗				✗
Multidireccionalidad			✗[1]	✗	✗	✗	✗	✓	✗	✗		✗	✗
Trazabilidad	Soporte Explícito		✓			✓							✓
	Soporte Implícito		✗	✓	✓			✓	✓			✓	

Cuadro 3.2: Características presentes en los lenguajes de transformación de modelos

Notas

1. Puede ser simulado
2. Desarrollado por proyecto externo
3. Planificación explícita externa mediante máquina de estados finita

La primer distinción interesante en estos enfoques es si los mismos emplean herramientas o lenguajes. Algunos enfoques emplean herramientas no dedicadas a las transformaciones de modelos (ejemplo: AGG), que cumplen efectivamente con el propósito de la transformación y cuentan con el aval de toda su comunidad de usuarios (incluso de aquellos que las emplean con fines distintos al de transformación de modelos). Estas herramientas fueron probadas exhaustivamente por lo que su uso y posibilidades suele estar bien documentado. Otros enfoques utilizan lenguajes de transformación de modelos, con el fin de aprovechar los mecanismos adicionales que ofrecen para especificar transformaciones (ejemplo:

compatibilidad con diferentes lenguajes de metamodelado, mecanismos de trazabilidad, mecanismos de reuso y composición de transformaciones). Debido a la reciente aparición de estos lenguajes, los entornos y herramientas que lo implementan aún se encuentran inmaduras y en permanente perfeccionamiento [27].

Se presentó QVT como una especificación estándar de transformación de modelos. Esta especificación es el resultado de varios años de trabajo y cooperación del OMG y constituye una referencia para el desarrollo de la mayoría de los lenguajes de transformación de modelos. Resulta claro que los lenguajes híbridos juegan un rol dominante en los trabajos recientes sobre transformación de modelos.

Como se mencionó, ATL se basa en la especificación de QVT y sigue la línea de los enfoques híbridos. Actualmente es uno de los lenguajes de transformación de modelos más utlizados y difundidos, contando con una gran cantidad de publicaciones relacionadas. La herramienta que soporta el lenguaje ATL fue desarrollada como plug-in del IDE Eclipse. Al desarrollarse sobre esta plataforma con varios años de desarrollo y uso, se asegura cierta robustez. En general, las herramientas de transformación disponibles ofrecen características básicas como el reconocimiento de sintaxis y un debugger. Sin embargo, las herramientas cuentan con pocos años de desarrollo y aún existen funcionalidades deseables sin implementar como la detección de errores en la codicación de transformación.

Capítulo 4

Verificación de Transformación de Modelos

Este capítulo resume el relevamiento realizado en el segundo reporte técnico sobre el estado del arte de los diferentes enfoques y técnicas de verificación de transformaciones de modelos empleados para MDD.

Las técnicas relevadas pertenecen a tres diferentes enfoques de verificación: basado en casos de prueba, verificación de modelos (model-checking) y métodos deductivos. El primer enfoque centra la verificación de la transformación a nivel de los modelos particulares que son transformados, trabajando sobre un conjunto de modelos origen y sus correspondientes modelos destino. La correctitud en este caso se prueba para un conjunto representativo del dominio con lo que se generan ciertos niveles de confianza en la correctitud esperada para el resto de los modelos a los cuales se aplicará la transformación. El segundo enfoque permite probar en forma automática propiedades sobre los modelos origen y destino, y sobre la transformación en caso de que estos puedan ser representados como grafos. Las principales ventajas de este enfoque radican en la facilidad de uso de los model-checkers y en el hecho de que no se requiere la asistencia del usuario para realizar las pruebas. La verificación formal de transformaciones bajo este enfoque nos permite asegurar la corrección de las mismas utilizando, en general, herramientas semi-automáticas. El último enfoque centra la verificación sobre los metamodelos en lugar de los modelos. Las técnicas dentro de este enfoque utilizan definiciones formales de los metamodelos y expresan la transformación como un conjunto de assertions (ej: fórmulas lógicas del tipo $\forall\exists$), construyendo de esta forma un framework que permite realizar razonamientos formales. De esta forma, la prueba de propiedades a este nivel garantiza que éstas se cumplan para todos los modelos que conformen con dicho metamodelo. Las técnicas de verificación sobre modelos utilizan un menor nivel de abstracción al de las técnicas de verificación formal sobre metamodelos, por lo que el rango de propiedades que se pueden validar es mayor que en el segundo caso, pero no pueden asegurar corrección en términos absolutos [24].

Este capítulo está organizado de la siguiente manera. En la sección 4.1 de este documento se presentan las técnicas de verificación basadas en casos de prueba. Se exponen las principales dificultades de este enfoque y se presentan las adaptaciones de técnicas conocidas como caja blanca, caja negra y mutation testing. En la sección 4.2 se presenta el enfoque basado en model-checking. Las técnicas aquí referidas tienen puntos de contacto con las de otras categorías pero resulta de interés distinguir el uso de las herramientas de model-checking. Por último, la sección 4.3 agrupa técnicas basadas en la utilización de métodos deductivos. Este enfoque requiere el uso estricto de lenguajes formales de especificación y de asistentes de prueba para probar en forma absoluta propiedades de una transformación.

4.1. Verificación utilizando Casos de Prueba

Para los enfoques tradicionales de desarrollo de software (como el paradigma de orientación a objetos), se han definido y estudiado una amplia variedad de técnicas de testing. Es posible adaptar estas técnicas a MDD con las complejidades propias de este contexto, por ejemplo definiendo los conceptos de testing de caja blanca y negra como la condición de utilizar o no las reglas de transformación para la definición de los casos de prueba. En esencia, las técnicas aplicadas a MDD son similares a las técnicas tradicionales, basándose en la generación de casos de prueba, su validación y su posterior ejecución para verificar transformaciones.

En esta sección se describen en primer lugar las dificultades generales de aplicar el enfoque a la verificación de transformaciones de modelos. Posteriormente se presentan diferentes técnicas de generación de casos de prueba y finalmente técnicas de validación de los mismos.

Dificultades generales

Los elementos que participan de una transformación de modelos presentan una complejidad tal que requieren adaptar las técnicas. A continuación mencionaremos las principales dificultades que se presentan, las cuales se detallan con mayor profundidad en en el segundo reporte técnico.

En primer lugar, los metamodelos son estructuras complejas que determinan un espacio de modelos mediante restricciones estructurales y posiblemente restricciones no estructurales expresadas en otro lenguaje diferente del utilizado para definir el metamodelo (ej: OCL, lenguaje natural). Los modelos origen que participan de una transformación deben pertenecer a este espacio. La complejidad de los datos especificados por estos metamodelos afecta la generación de casos de prueba en términos de memoria y tiempo de ejecución para explorar el espacio de los modelos [9].

Otra dificultad asociada a la complejidad de datos es la de manejar oráculos para verificación. La generación, en forma manual o automática, del resultado esperado así como la comparación con el resultado de la transformación son complejas. En el caso de las transformaciones de grafos, comparar el resultado esperado con la salida de la transformación es un problema NP-completo (problema de isomorfismo de grafos). En el caso de transformaciones que produzcan modelos ejecutables, una técnica de verificación posible es comprobar que estos modelos ejecutan correctamente. Otra alternativa es desarrollar oráculos parciales que verifiquen un conjunto de propiedades mediante la utilización de patrones para comprobar precondiciones y postcondiciones de la transformación.

La gran diversidad de técnicas y lenguajes de transformación de modelos, así como de las herramientas empleadas, repercute también en la selección de las técnicas de verificación. A modo de ejemplo, las técnicas de caja blanca presentan la desventaja de estar fuertemente acopladas al lenguaje de transformación y deben ser adaptadas o completamente redefinidas si el desarrollador opta por utilizar un lenguaje diferente. Por otra parte, las técnicas de caja negra pueden utilizarse con cualquier enfoque de transformación de modelos en forma indistinta, dado que se basan únicamente en el cubrimiento del espacio de modelos definido por el metamodelo origen.

Generación de casos de prueba

La verificación de transformaciones utilizando casos de prueba requiere de modelos origen que respeten el metamodelo y las meta-restricciones definidas. La especificación manual de dichos modelos es una tarea tediosa y compleja, ya que los mismos deben respetar simultáneamente una variedad de restricciones. Las técnicas analizadas a continuación tienen como objetivo verificar tranformaciones de modelos utilizando conjuntos de casos de prueba generados de forma automática.

En [35] se presentan técnicas de testing de *caja blanca*. En primer lugar, se propone realizar el cubrimiento de las reglas mapeando cada una de ellas a un template con el cual se compone un nuevo metamodelo.

A partir de este metamodelo se instancian casos de prueba. El cubrimiento que los casos realizan de los templates garantizan el cubrimiento de las reglas de transformación. Además, es necesario complementar esta técnica con la utilización de las restricciones no estructurales para generar los casos de prueba, de forma de evitar posibles omisiones por elementos no considerados en las reglas. Las restricciones no estructurales deben considerarse también para asegurar que los casos de prueba generados sean válidos. Por último, se propone utilizar pares de reglas para detectar posibles errores de confluencia: el resultado de ejecutar las reglas debe ser el mismo sin importar el órden de ejecución.

Dentro de las técnicas de *caja negra* podemos encontrar diversos enfoques. En [13] se presenta un algoritmo simple, similar a las técnicas de caja negra empleadas en otros paradigmas distintos de MDE. En primer lugar se realiza una partición de los atributos simples y las cardinalidades en los modelos, para luego generar fragmentos de modelo que realicen un cubrimiento apropiado de cada partición. Por último, se combinan estos fragmentos siguiendo una estrategia de verificación de forma de obtener los casos de prueba. Este último paso es el más crítico, por lo que se incorporan varios *puntos de variación* en el algoritmo (ej: tamaño de modelos, creación de asociaciones) de forma de permitir al verificador seleccionar la estrategia más adecuada.

Otro enfoque enmarcado en las técnicas de caja negra es el empleado en [57]. En este trabajo se propone la utilización de una herramienta para generar un sistema de restricciones a partir de la especificación del dominio de entrada de la transformación. Posteriormente se utiliza un solucionador SAT[1] para resolver este sistema y obtener modelos que lo satisfagan. Estos modelos constituyen el conjunto de casos de prueba a utilizar.

Finalmente, las técnicas de *mutation generation* también pueden ser adaptadas al paradigma de MDD. En este enfoque se busca generar casos de prueba mediante la introducción de pequeñas variaciones en modelos existentes. En [20] se propone utilizar una representación de circuitos combinatorios en los cuales se introducen las variaciones.

Validación de casos de prueba

En la verificación mediante casos de prueba no basta con generar un conjunto de casos de prueba que conformen con el metamodelo. Dado que no es posible generar un conjunto exhaustivo que permita verificar todos los casos posibles, es necesario validar de alguna forma este conjunto y brindar cierta confianza en que el mismo será útil para detectar fallas. En el contexto de MDD, dado un conjunto de instancias del metamodelo origen y la transformación a verificar, se busca conocer que tan bueno es dicho conjunto para cubrir el espacio de modelos origen.

Como se mencionó previamente, las técnicas de caja negra permiten al verificador independizarse del lenguaje en el que fue implementada la transformación. El dominio de entrada se define por el metamodelo origen. La idea básica es evaluar que tan adecuados son los casos de prueba con respecto al cubrimiento de los objetos del metamodelo origen. Estos casos de prueba deben instanciar al menos una vez cada clase y propiedad del metamodelo origen. Para clasificar los posibles valores de las propiedades se utiliza una adaptación de la técnica de testing conocida como partición de equivalencias [49]. Esta técnica consiste en descomponer el dominio de los modelos origen en un conjunto finito de subdominios no solapados y escoger datos para la prueba de cada uno de los subdominios.

La contribución de [26] son criterios para cubrir los metamodelos origen. Estos criterios capturan todas las nociones importantes del metamodelo y permiten evaluar los casos de prueba de las transformaciones. Dado el metamodelo origen, se pueden elegir diferentes estrategias para la selección de los casos de prueba y se utiliza un framework para verificar si los datos de prueba son adecuados. El framework automáticamente analiza un conjunto de casos de prueba y provee al usuario de información faltante

[1] Un solucionador SAT determina si las variables de una fórmula booleana pueden ser asignadas de forma tal que la expresión sea evaluada a TRUE.

para lograr el cubrimiento de la estrategia seleccionada. Esta información puede ser utilizada para mejorar iterativamente el conjunto de casos de prueba.

Otra técnica para evaluar un conjunto de casos de prueba es la técnica de mutation testing [44, 45]. La técnica consiste en crear sistemáticamente versiones de un programa con faltas (llamadas mutaciones) y chequear la eficiencia de los conjuntos de casos de pruebas para revelar fallas debido a las faltas introducidas. El interés principal de la técnica es proveer un valor estimado de la calidad del conjunto de casos de prueba en proporción a la fallas detectadas. Para ser efectivo, el proceso de mutación debe crear programas con faltas de forma realista. Es decir, posibles faltas que pueda insertar un programador experimentado en el lenguaje de transformación. El trabajo referenciado realiza un estudio de los problemas comunes que tienen los métodos convencionales de mutación de programas para enfoques distintos a MDD, aplicados en un enfoque MDD. La idea planteada para aplicar mutation testing es definir operadores de mutación para el enfoque MDD. Con este propósito, se identifican operaciones comunes que realizan la mayoría de los lenguajes de transformación y para éstas se definen operadores de mutación. Con estos operadores se generan las mutaciones del programa. En el caso MDD, mutaciones de la transformación.

4.2. Verificación utilizando Model Checking

Muchas de las técnicas planteadas en diferentes publicaciones involucran el uso herramientas de model-checking para verificar transformaciones de modelos. Como sucede con otras técnicas, esta técnica varía según la noción de verificación tomada. Podría tratarse de una verificación sintáctica, una verificación semántica, ambas, o podría definirse una noción de verificación para un caso concreto que sea de interés. A partir de una noción de verificación particular, se define un conjunto de propiedades que puedan ser comprobadas de forma automática por el model-checker. Las propiedades pueden ser restricciones sobre el modelo origen, sobre el modelo destino o sobre la propia transformación siempre .

Las técnicas de model-checking parten del supuesto que el modelo (o transformación) pueda expresarse como un sistema de transiciones (grafo dirigido) que conste de un conjunto de vértices y arcos. En este modelo un conjunto de proposiciones atómicas se asocia opcionalmente a cada nodo. Los nodos representan los estados posibles de un sistema y los arcos evoluciones del mismo mediante ejecuciones permitidas (que alteran el estado). Las proposiciones representan las propiedades básicas que se satisfacen en cada punto de la ejecución.

La principal limitación de esta técnica radica en el tamaño del grafo a verificar, ya que el análisis crece exponencialmente con el número de estados del sistema. Otra limitación es que el lenguaje para expresar propiedades debe ser restringido a fin de asegurar automaticidad.

Redes de Petri

Existe una línea de trabajos ([53], [59]) que tienen como objetivo tomar modelos de comportamiento y transformarlos a redes de Petri, para luego analizar las propiedades de esta última. Las redes de Petri [54] constituyen un lenguaje de modelado formal y gráfico. Estas cuentan con abundantes y sólidas técnicas de análisis que permiten verificar propiedades estructurales y de comportamiento de un modelo. De esta forma, el análisis sobre una red de Petri permite estudiar propiedades sobre el modelo original que fue transformado a dicha red.

OCL

Otros trabajos ([52], [46], [16], [14]) emplean herramientas de model-checking para trabajar con restricciones expresadas en OCL. Si bien UML y OCL carecen de la semántica formal necesaria para realizar la transformación, son lenguajes suficientemente expresivos para visualizar la transformación y conocidos por toda la comunidad de desarrolladores. Esto convierte a OCL en una alternativa interesante frente

a lenguajes formales como Z que cuentan con la formalidad requerida, pero son complejos y requieren herramientas de análisis formal que no se utilizan frecuentemente en la práctica. La desventaja de este enfoque es que el número de herramientas para trabajar con OCL es reducido en comparación con el número de herramientas de lenguajes formales [37]. Los trabajos mencionados proponen extraer en forma estática información de la especificación de la transformación para luego verificar en forma dinámica, verificando propiedades tras aplicar la transformación. La verificación dinámica es imprescindible para asegurar que se obedecen las condiciones de la transformación. Es aconsejable complementar esta verificación con verificación estática (ej: chequeo de tipos cuando este es posible) al momento de escribir la transformación.

Transformaciones de grafos tipados

Un caso de especial interés para el proyecto de grado son las transformaciones de modelos de comportamiento. Los modelos de comportamiento son descritos por metamodelos dinámicos. Estos metamodelos pueden ser formalizados como un sistema de transformaciones de grafos tipados. Debido a la naturaleza de los modelos en la teoría de grafos, los sistemas de transformaciones de grafos y su soporte tecnológico proveen un ambiente adecuado para formalizar y verificar trasformaciones de modelos. Informalmente, un sistema de transformaciones de grafos tipados consiste en:

- un *grafo tipado* que define el vocabulario de los elementos del modelo permitidos y sus relaciones,
- un conjunto de restricciones
- y un conjunto de reglas de transformación de grafos.

Los grafos tipados y las restricciones pueden ser vistos como el análogo a un metamodelo estático. En el caso de sistemas dinámicos, una instancia de un grafo modela el estado del sistema en un punto específico del tiempo. Para modelar las evoluciones de un sistema, el metamodelo dinámico provee reglas de transformación de grafos. Las mismas son especificaciones ejecutables que pueden ser utilizadas para definir reglas de transformación que especifiquen posibles computaciones, comunicaciones u operaciones que puedan ser aplicadas a estados e impliquen transiciones a nuevos estados.

La idea planteada en [30] para refinamientos de modelos de comportamientos, es formular los caminos que definen los sistemas de transiciones del refinamiento, como un problema de "alcanzabilidad" en el sistema de transiciones concreto. Formalmente, se considera un grafo G instancia del sistema abstracto y una instancia G' del sistema concreto, donde G' es un refinamiento de G. La técnica consiste en verificar que para cada paso de la transformación $G \rightarrow H$ en el sistema abstracto, exista una secuencia de $G' \stackrel{*}{\rightarrow} H'$ en el sistema concreto que refine H.

En [12] se propone traducir los conceptos de la representación de sistemas de transformación de grafos a una representación de lógica de reescritura. Esta traducción genera una formalización ejecutable para transformaciones del estilo QVT, que puede ser manipulada por herramientas que realizan model-checking de invariantes y de propiedades lógicas temporales.

Lenguajes formales específicos

En [48] se analiza en primer lugar la especificación de transformaciones declarativas mediante la utilización de triple graph. El concepto de *triple graph* es definido en [55] para describir transformaciones de modelos cuando los modelos y metamodelos son representados como grafos. Esta transformación está caracterizada por la especificación de modelos origen y destino, y la conexión entre las correspondencias de elementos del modelo origen con los elementos del modelo destino. Para ello se utiliza un grafo intermedio (grafo de conexiones) y se introduce un mapeo entre este grafo y los grafos origen y destino.

En [48] se adapta el concepto utilizando álgebras en lugar de grafos. Se presenta el concepto de *álgebra triple*, que consiste en un conjunto de tres álgebras: un álgebra de origen, una de destino y una de conexión, que expresa las relaciones del álgebra origen con el álgebra destino. Siguiendo la idea de los enfoques triple graph, se muestra como pueden ser usados los patrones (patrones triples) para describir las propiedades que se desean verificar en una transformación. Los patrones triples describen propiedades que deben ser satisfechas por un álgebra triple. Utilizando estos patrones se puede definir la transformación, especificando un conjunto de axiomas como fórmulas lógicas que indican lo que debe estar presente (patrones algebraicos positivos) y lo que no (patrones algebraicos negativos) en el modelo. El trabajo introduce la noción de restricción algebraica como la generalización de la noción de restricción sobre grafos.

Por último, en [6] se describe como utilizar el lenguaje Alloy para automatizar el proceso de verificación a partir de una especificación declarativa de una transformación. En [36] se plantea un enfoque similar, transformando la especificación a un formalismo que permite utilizar herramientas de model-checking.

4.3. Verificación utilizando Métodos Deductivos

La verificación utilizando casos de prueba es de gran ayuda para asegurar la calidad de la transformación, explotando la especificación de la transformación y de los modelos para derivar casos de prueba. Sin embargo, solo es posible probar un número finito de casos, lo cual resulta insuficiente en aplicaciones críticas para asegurar la correctitud. En esta sección se describen técnicas de verificación donde se utilizan métodos deductivos que permiten asegurar con certeza absoluta propiedades de una transformación.

La verificación mediante métodos deductivos se basa en la especificación formal de la transformación y de los metamodelos origen y destino. No se emplean instancias de los metamodelos como casos de prueba, por lo que se trata de un análisis estático (no se realiza mediante la ejecución) que produce resultados formales para todas las posibles instancias válidas. Se trabaja sobre las especificaciones mencionadas utilizando lenguajes formales y asistentes de prueba. Por lo general, este tipo de verificación se realiza en forma manual o semi-automática [7].

Si bien la verificación mediante métodos deductivos cuenta con madurez en el campo de investigación de la ingeniería de software, su aplicación práctica se limita a sistemas embebidos y a aplicaciones donde la confiabilidad es crítica. Entre los motivos se encuentran la complejidad de las técnicas de especificación formal y la falta de entrenamiento de los desarrolladores de software para la aplicación de dichas técnicas [24].

En [37] se muestra como puede utilizarse el lenguaje formal B para incorporar semántica objetiva a una especificación realizada en un subconjunto de UML. Como vimos en la sección 4.2, UML no cuenta con la semántica necesaria para realizar verificación. Este trabajo presenta una alternativa al empleo de OCL para incorporar semántica objetiva a UML.

Existe otra línea de trabajos dentro de la utilización de métodos deductivos que se basa en la teoría de tipos. En [50] utiliza esta teoría para implementar una transformación a partir de una prueba formal de su especificación. Para esto emplea el formalismo matemático de alto orden denominado *Teoría de Tipos Constructivos (*Constructive Type Theory, CTT) y el método formal de síntesis *Proof-as-Programs* [17, 51]. CTT es utilizado para expresar relaciones de alto orden entre modelos y metamodelos sobre un subconjunto de MOF. Este subconjunto restringido excluye de la implementación de la transformación a características como reflection. Empleando estos formalismos se demuestra como es posible obtener la implementación de una transformación a partir de la prueba formal de la misma. Sugiere además utilizar probadores de teoremas para asistir al usuario en la construcción de las pruebas formales.

En [15] se parte también de la teoría de tipos con un enfoque ligeramente diferente al trabajo anterior. Mientras [50] se enfoca en la construcción de transformaciones con una correctitud certificada, [15] se concentra en probar semiformalmente la correctitud de transformaciones existentes. Además, se prueba

la factibilidad del uso de la teoría de tipos mediante su aplicación utilizando el lenguaje de metamodelado KM3, el lenguaje de transformación ATL y el asistente de pruebas Coq [3]. El trabajo propone un framework basado en una versión de la teoría de tipos denominada *Cálculo de Construcciones Inductivas* (Calculus of Inductive Constructions, CIC). La técnica de verificación consiste en la traducción de la especificación de los metamodelos a construcciones de CIC. Esta traducción comprende a los elementos del metamodelo, a las restricciones estructurales y a las restricciones no estructurales. Posteriormente, se incorpora la especificación de la transformación como un conjunto de proposiciones lógicas de la forma $\forall\exists$. Utilizando entonces esta nueva especificación formal, se realizan pruebas formales para demostrar propiedades de interés.

4.4. Resumen

En este relevamiento se presentaron una gran variedad de enfoques y técnicas para la verificación de transformaciones de modelos. Algunas técnicas toman la experiencia obtenida en la verificación dentro de otros paradigmas de desarrollo de software y la adaptan a las características de MDD. Esto muestra que los diferentes paradigmas comparten muchos conceptos de verificación como el análisis de caja blanca y caja negra, las mutaciones y las técnicas de cubrimiento. En cambio, otras técnicas toman en consideración que toda transformación tiene como entrada y salida uno o más modelos que conforman un metamodelo. Esta característica es la que explotan los enfoques basados en model-checkers, donde se busca comprobar en forma automática, bajo ciertas restricciones, características de los modelos involucrados.

Una de las principales diferencias entre las distintas técnicas documentadas radica en la certeza que éstas brindan. Las técnicas de verificación basadas en el uso de casos de prueba sólo brindan certeza sobre los casos de prueba verificados. Por este motivo, las técnicas de partición y cubrimiento permiten afirmar, sólo con cierto grado de confiabilidad, que las propiedades se preservarán para modelos no verificados. Por otro lado, las técnicas basadas en model-checking y en métodos deductivos brindan certeza absoluta sobre las propiedades que se analizan para todos los modelos posibles. Bajo las técnicas de métodos deductivos, muchas veces se realizan demostraciones constructivas que permiten generar transformaciones certificadas como correctas (ej: usando teoría de tipos).

Surge entonces la interrogante sobre por qué no se utilizan siempre métodos deductivos. El primer motivo es que los métodos deductivos requieren la utilización de lenguajes formales y herramientas matemáticas con los que los desarrolladores no están habituados a trabajar. Es un requisito indispensable el obtener (en el mejor de los casos de forma semiautomática) una especificación formal de la transformación, de los metamodelos y los modelos involucrados para poder luego utilizar asistentes de pruebas. Para superar esta dificultad existen algunas propuestas donde se intenta reducir la cantidad de formalismos mediante el uso de especificaciones semiformales. Sin embargo, estas técnicas que emplean semiformalismos suelen perder alcance y generalidad en los resultados de la verificación, o asumen la existencia de una transformación intermedia que preserva propiedades semánticas. En nuestro criterio, esa suposición sobre la transformación intermedia correcta nos devuelve al principio sobre el problema de verificación.

El segundo motivo que parece aún más difícil de superar radica en el tipo de propiedades que se puede probar dependiendo del enfoque empleado. Los métodos deductivos utilizan un mayor nivel de abstracción para expresar propiedades, al basarse en la especificación de metamodelos. Los métodos basados en el uso de casos de prueba en cambio permiten probar propiedades más específicas sobre la transformación.

Estas consideraciones generales sobre la verificación de transformaciones son motivo de la existencia de una gran variedad de enfoques. Ninguno de estos enfoques es capaz de contemplar por sí mismo todas las necesidades de verificación, por lo que deben emplearse criterios para determinar cuál es la técnica más apropiada para el problema concreto teniendo en cuenta la criticidad de la aplicación. Debido a esta gran variedad, las herramientas de verificación no se encuentran aún integradas a las herramientas de desarrollo de MDD. La tendencia en herramientas de desarrollo MDD es permitir la especificación de la

mayor diversidad posible de transformaciones. Dada esta diversidad de transformaciones que se pueden implementar, es difícil pensar en herramientas específicas de verificación para integrar a los entornos de desarrollo.

Para concluir este capítulo, se presenta en el Cuadro 4.1 un resumen de los diferentes enfoques de verificación junto con las técnicas que lo implementan y las características más generales.

	Técnicas	**Ventajas**	**Desventajas**
Casos de prueba	Caja blanca [35] Caja negra [26, 52, 57, 13, 38] Mutation testing [20, 32, 44]	Similaridad con técnicas tradicionales en otros paradigmas de desarrollo.	Complejidad de explorar espacio de modelos. Complejidad de comparar resultado de la transformación con resultado esperado Se prueba un número finito de casos.
Model-checking	Redes de Petri [53, 59] Sistema de Transformaciones de Grafos [30, 12] Lenguajes formales [6, 48, 36]	Permite probar propiedades en forma automática. Se apoyan en formalismos conocidos que permiten utilizar herramientas de ánalisis potentes.	Lenguaje restringido para asegurar automaticidad. Solo se puede aplicar sobre grafo. Complejidad crece en forma exponencial con respecto al tamaño.
Métodos deductivos	Triple Graph Grammars [28] Teoría de Tipos [15, 50]	Las propiedades son probadas para todas las transformaciones posibles.	Requiere conocimientos de formalismos utilizados. Las pruebas son manuales o semiautomáticas.

Cuadro 4.1: Resumen de enfoques de verificación

Capítulo 5

Caso de Estudio

A los efectos de concluir el trabajo y de mostrar algunos de los conceptos expuestos en un ejemplo práctico, se implementó un caso de estudio en donde se emplean los lenguajes de transformación de modelos para semi-automatizar una de las técnicas de verificación de transformación. El caso de estudio desarrollado se enmarca en la técnica de verificación descrita en la sección 4.3 y documentada en [15]. En esta técnica se traduce la especificación de los metamodelos que participan en una transformación a una especificación en CIC. Las restricciones no estructurales y las reglas de transformación se traducen a proposiciones lógicas del tipo $\forall\exists$. Utilizando esta nueva especificación, es posible realizar pruebas formales para verificar si se cumplen ciertas propiedades de interés con certeza absoluta. Las herramientas utilizadas para la implementación del caso de estudio fueron el lenguaje de metamodelado KM3, el lenguaje de transformación de modelos ATL y el asistente de pruebas Coq. El prototipo desarrollado se concentra en el primer paso del proceso de verificación: la traducción de los metamodelos expresados en KM3 a una representación en Coq.

El prototipo a desarrollar constituye un primer paso hacia la automatización de esta técnica de verificación. Por este motivo se busca obtener una transformación que cubra los aspectos básicos de la traducción de los metamodelos involucrados en la transformación a verificar. Se plantea como trabajo a futuro el complementar este prototipo con la automatización de la traducción de restricciones estructurales, no estructurales y la traducción de las reglas de transformación a proposiciones lógicas.

El prototipo define una transformación a nivel de metametamodelos como muestra la Figura 5.1. El objetivo del prototipo es transformar metamodelos expresados en KM3 a un metamodelo que emplee los constructores de Coq. A modo de ejemplo, se podrá traducir el metamodelo de UML y el metamodelo relacional de la transformación descrita en la sección 2.2 a sus respectivas especificaciones que empleen los constructores de Coq. Luego, será posible probar propiedades de la transformación utilizando el asistente de pruebas Coq.

Es importante notar que la transformación implementada en el prototipo utiliza el metametamodelo KM3 como metamodelo origen, y a un metamodelo expresado en KM3 como modelo origen. Esto refleja una de las características de las transformaciones descritas en el capítulo 2: los metamodelos son casos particulares de modelos que pueden servir como modelos origen o destino de una transformación.

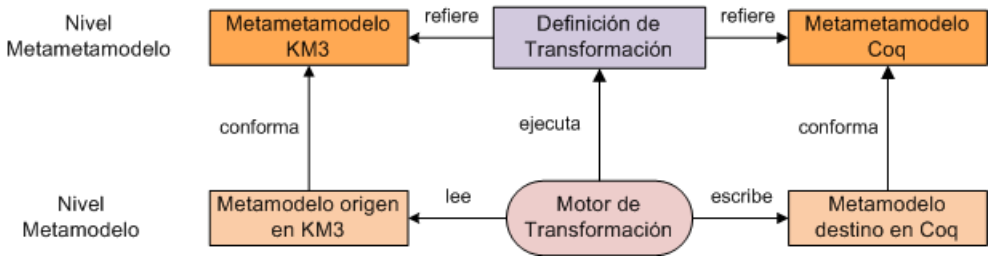

Figura 5.1: Transformación principal del prototipo

Este capítulo está organizado de la siguiente manera. En la sección 5.1 de este capítulo se presentará el lenguaje de metamodelado KM3 con el que se expresan el metamodelo origen de la transformación implementada. A continuación, en la sección 5.2 se describen aquellos elementos en Coq considerados de interés para representar el modelo destino. Finalmente, en la sección 5.3 se describen el diseño y los detalles técnicos de la implementación del prototipo que realiza la traducción. Para esto se explicarán en primer lugar los detalles de la transformación principal modelo-modelo que fue implementada en ATL, para luego detallar la transformación modelo-texto desarrollada en Xtext.

5.1. Kernel MetaMetaModel (KM3)

El lenguaje de metamodelado utlilizado para el prototipo fue KM3. La utilización de KM3 fue determinada por el espacio técnico descrito en [15]. Es un lenguaje popular en la comunidad de usuarios de ATL, por su simplicidad y expresividad en modo textual. En la Figura 5.2 se puede observar el metamodelo de KM3. El recuadro indica los elementos considerados por la transformación implementada en el prototipo. Los elementos del metamodelo que han quedado fuera, no son considerados dado que la traducción a realizar no es afectada por los valores de los mismos.

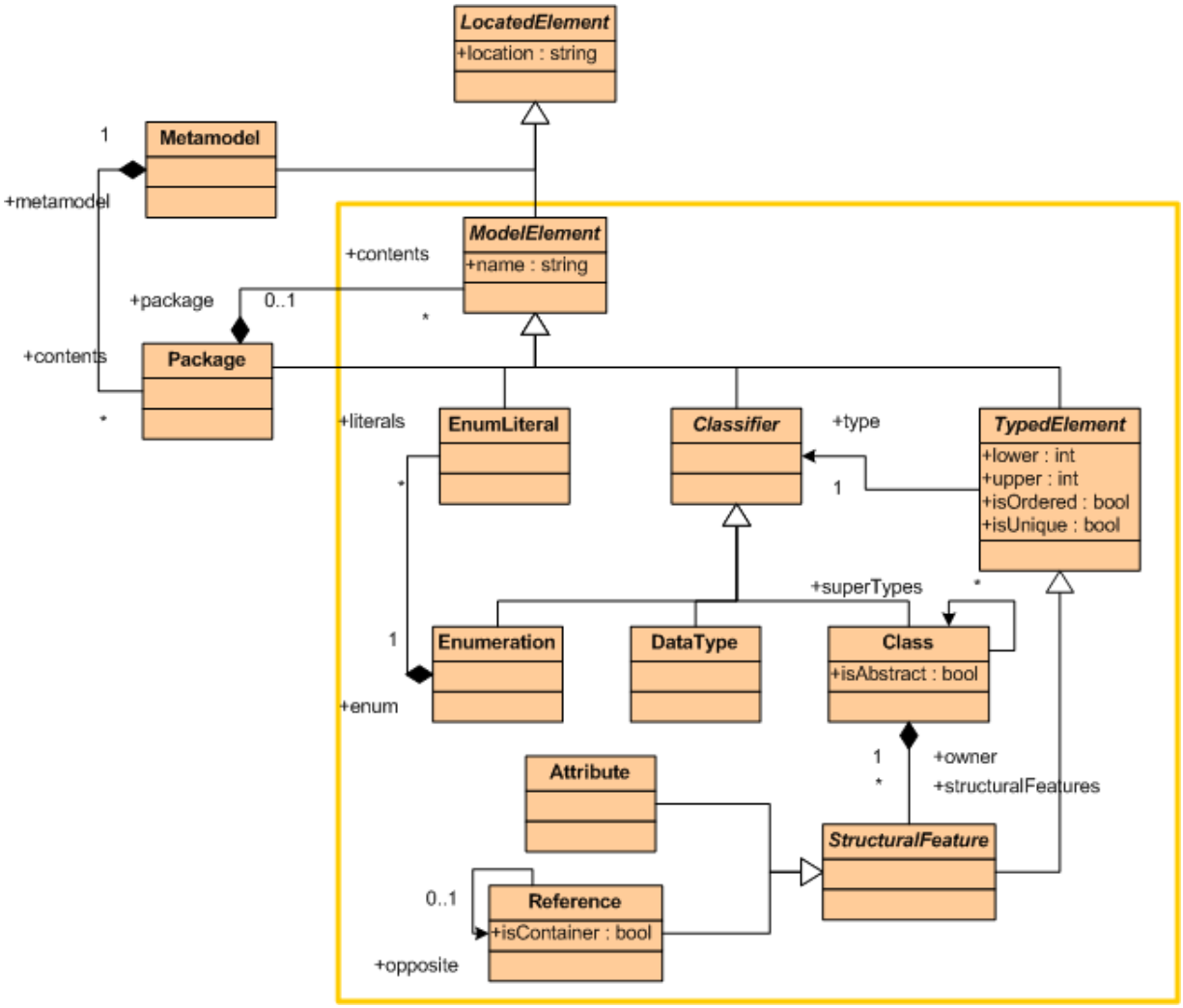

Figura 5.2: Metamodelo KM3

El manual de KM3 [41] describe el metamodelo KM3 de la siguiente forma:

> Un **Metamodelo** KM3 se compone de elementos **Package**. Un **Package** contiene algunas entidades abstractas **ModelElement** (**TypedElement**, **Classifier**, **EnumLiteral** y entidades **Package**, ya que un **Package** es también un **ModelElement**). Cada **ModelElement** se caracteriza por su *name* (nombre). Se debe notar que un **ModelElement** no necesariamente se encuentra contenido dentro de un **Package**. Un **ModelElement**, así como un **Metamodel**, hereda de la entidad abstracta **LocatedElement**. Esta última define un atributo *location* cuyo objetivo es codificar en una cadena de texto la ubicación de la declaración del elemento correspondiente dentro del archivo *.km3*.
>
> Un **Classifier** puede ser un **Enumeration**, un **DataType** o un **Class**. Un **Enumeration** se compone de elementos **EnumLiteral**. Todo **EnumLiteral** se encuentra contenido dentro de un **Enumeration**. El elemento **DataType** permite definir data types primitivos. Finalmente, el elemento **Class** define un atributo booleano *isAbstract* que permite declarar clases

abstractas. Además, un **Class** puede ser extendido mediante *supertypes* (supertipos) directos (elementos **Class**).

Un **Class** se compone de un conjunto de elementos abstractos **StructuralFeature**. Un **StructuralFeature** hereda de la entidad abstracta **TypedElement**. Esta entidad define los atributos *lower*, *upper*, *isOrdered* e *isUnique*. Los dos primeros atributos definen una cardinalidad mínima y máxima de un **TypedElement**. Los atributos booleanos *isOrdered* e *isUnique* representan el hecho de que diferentes instancias de un **TypedElement** son respectivamente ordenadas y únicas. Un **TypedElement** evidentemente tiene un *type*, que corresponde al elemento **Classifier**. Se debe notar que un **StructuralFeature** pertenece a un único propietario de tipo **Class**. En otras palabras, aunque un **StructuralFeature** es también un **ModelElement**, es contenido por un **Class** en lugar de un **Package**.

Un **StructuralFeature** es o bien un **Reference** o un **Attribute**. El elemento **Attribute** permite definir un atributo de una clase. Un **Reference** define el atributo booleano *isContainer* que representa el hecho de que los elementos apuntos por la referencia son contenidos por esta última. Un **Reference** puede tener también un *opposite reference* (referencia opuesta).

A modo de ejemplo, en la Figura 5.3 se puede observar la representación parcial del metamodelo UML descripto en la sección 2.2 como instancias del metametamodelo KM3. Como se puede apreciar, este metamodelo se compone de un conjunto de clases que se relacionan entre sí mediante diferentes tipos de asociación. En la figura se representan parcialmente la clase *Class*, su atributo *isPersistent* y su asociación con la clase *Attribute*.

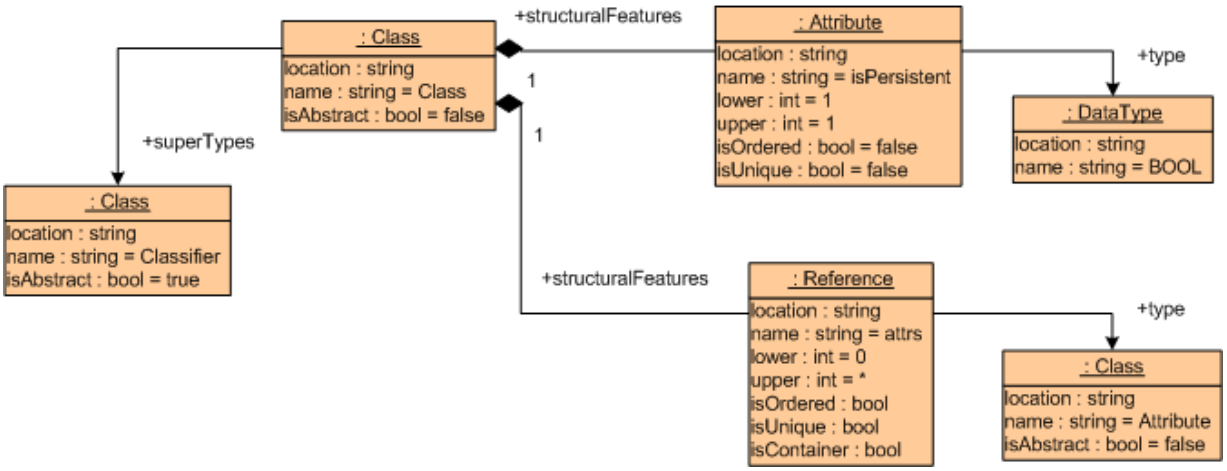

Figura 5.3: Representación parcial de metamodelo UML con un modelo KM3

En el Código 5.1 se puede ver una representación textual en KM3 del metamodelo de UML.

Código 5.1 Representación textual en KM3 del metamodelo UML

```
package UML {

    abstract class Classifier {
        attribute name : String;
    }
    class Class extends Classifier {
        attribute isPersistent : Boolean;
        reference pack : Package oppositeOf elems;
        reference attrs[*] ordered container : Attribute oppositeOf from;
    }
    class PrimitiveDataType extends Classifier {}

    class Package {
        attribute name : String;
        reference elems[*] ordered container : Class oppositeOf pack;
    }

    class Attribute {
        attribute name : String;
        reference from : Class oppositeOf attrs;
        reference type : Classifier;
    }

    datatype Boolean;
    datatype Integer;
    datatype String;

}
```

5.2. COQ

Como se describió brevemente en la técnica de verificación, se sugiere utilizar Coq como asistente de pruebas para demostrar propiedades de la transformación que permitan concluir que la misma es correcta. Coq es un asistente de pruebas que permite expresar proposiciones matemáticas, verificar la correctitud de pruebas para estas proposiciones y asistir a la generación de pruebas formales. Además permite generar programas correctos a partir de una demostración constructiva de la especificación formal del programa.

La sintaxis del lenguaje Coq está basada en el *Cálculo de Construcciones Inductivas* (Calculus of Inductive Constructions, CIC), el cual es un derivado del cálculo de construcciones. CIC es una teoría de tipos con tipos dependientes que permite expresar formulas lógicas de alto orden sobre objetos que se clasifican en una jerarquía de tipos. Los tipos se asemejan a los de los lenguajes funcionales fuertemente tipados en donde existen tipos básicos, tipos recursivos definidos por inducción (*tipo inductivo*) y tipos función [15].

Para ilustrar los conceptos presentados se utiliza como ejemplo la definición del metamodelo de árboles finitarios no vacíos de enteros en KM3. El Código 5.2 muestra la definición del metamodelo expresada en KM3. Como se puede apreciar, esta especificación es muy similar a la utilizada para cualquier lenguaje dentro del paradigma de orientación a objetos. Sin embargo, Coq no es un lenguaje orientado a objetos por lo que no permite representar directamente constructores como las generalizaciones y se debe emplear una aproximación diferente para la representación de los tipos.

Código 5.2 Metamodelo de árboles binarios desbalanceados de enteros en KM3

```
package MMTree {

    abstract class Tree {
    }
    class Node extends Tree {
        reference left : Tree;
        reference right : Tree;
    }
    class Leaf extends Tree {
        attribute value : Integer
    }

}
```

A continuación, el Código 5.3 muestra una definición constructiva de los tipos *tree* y *forest*[1]. Se puede observar en los constructores de cada tipo que *tree* y *forest* dependen entre sí. Dos tipos se consideran dependientes entre sí, si la definición de cada uno de ellos hace referencia a otro. Estos tipos se denominan *mutuamente inductivos*, y en este caso se debe generar una definición común a ambos de forma de resolver la dependencia en Coq. En la definición constructiva se utiliza el tipo básico *nat* que representa a los naturales. Los tipos básicos como *string*, *boolean y nat* son soportados en forma nativa por Coq mediante librerías. Con el fin de definir recursivamente los elementos del metamodelo, se definen los tipos *tree* y *forest* como tipos inductivos.

Código 5.3 Definición constructiva de árboles finitarios no vacíos de enteros en Coq

```
Inductive tree : Set :=

    | node (f : forest)
    with forest : Set :=
        | emptyf (value : nat)
        | consf (t : tree) (f : forest).
```

A partir de esta definición, se definen instancias del metamodelo como la exhibida en el Código 5.4. Esta especificación emplea los constructores de la definición de los tipos *tree* y *forest* de forma de construir la instancia.

Código 5.4 Instancia de árbol finitario no vacío de enteros

```
Definition arbol : tree :=

    node (
        consf (node (emptyf 1))
        (consf(node (emptyf 2)) (emptyf 3))
    ).
```

5.3. Diseño del prototipo

Como se comentó brevemente en la introducción del capítulo, la transformación modelo-modelo se implementó utilizando el lenguaje de transformación ATL. En la Figura 5.4, se muestra como la trans-

[1] Es importante mencionar, que en Coq todo nombre que identifica a un elemento es único.

formación ATL toma como entrada un metamodelo expresado en KM3 y genera un metamodelo que conforma con el metametamodelo Coq. A continuación, se lleva el modelo generado para Coq a su representación textual mediante Xtext [23]. Este último paso es necesario para generar un texto que respete la sintaxis del asistente de pruebas Coq. De esta forma se utilizan las dos categorías de enfoques mencionadas en la sección 3.1: una transformación principal modelo-modelo y una transformación secundaria de modelo-texto.

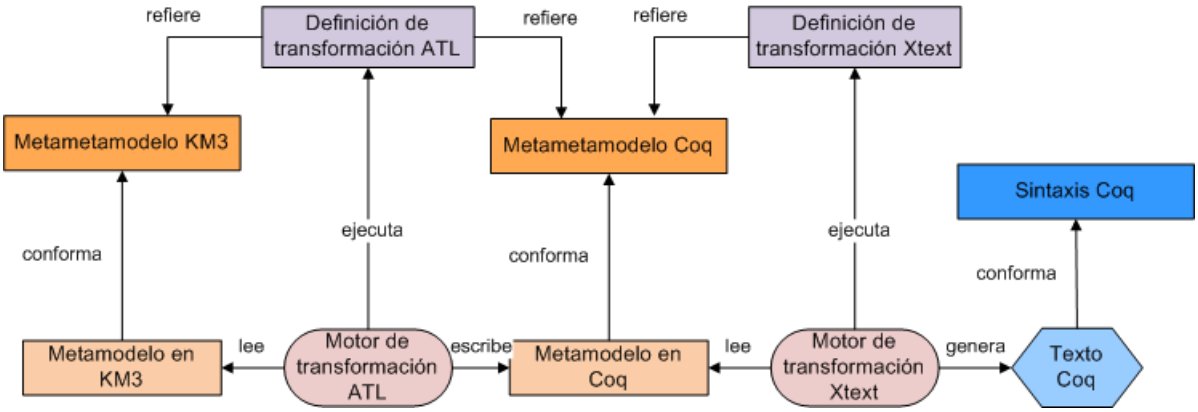

Figura 5.4: Proceso del prototipo

5.3.1. Transformación ATL

Como se mencionó en la introducción de este capítulo, esta es la transformación modelo-modelo central del prototipo. La misma toma como entrada un metamodelo expresado en KM3 y genera un metamodelo expresado con constructores de Coq. En la Figura 5.5 se puede observar el metametamodelo Coq definido para la transformación.

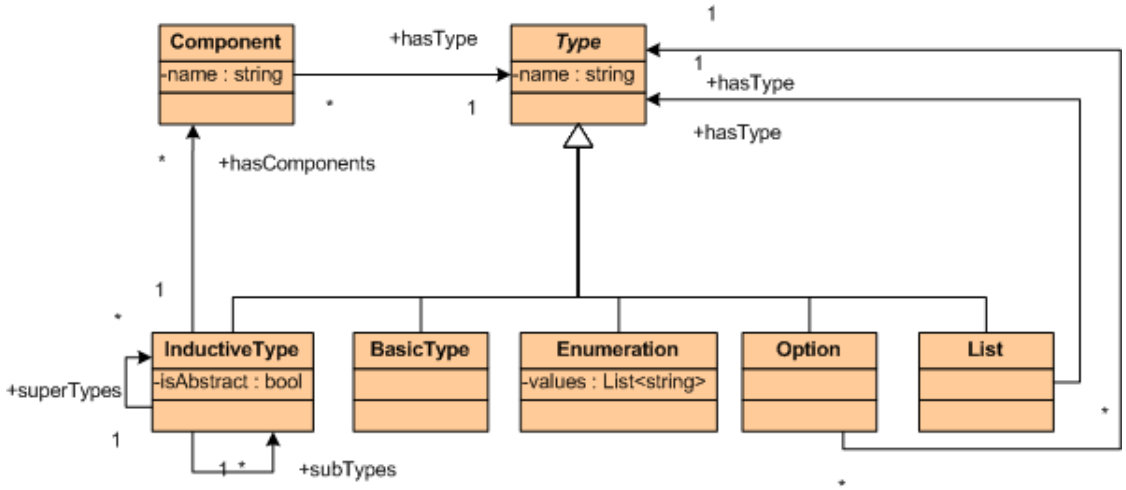

Figura 5.5: Metamodelo Coq

El metamodelo Coq definido consta basicamente de un conjunto clases que heredan de la clase ab-

stracta **Type**. Un **Type** puede ser **InductiveType**, **BasicType**, **Enumeration**, **Option** o **List**. La clase **InductiveType** representa a los tipos inductivos y posee tres asociaciones: una asociación con sus padres directos (denominada *superTypes*), una asociación a sus hijos directos (denominada *subTypes*) y una asociación con la clase **Component** (*hasComponents*). A su vez, cada **Component** posee un nombre (*name*) y tiene asociado un único *Type* por medio de la asociación *hasType*.

Las clases **List** y **Option** representan tipos especiales de Coq, las cuales se encuentran a su vez asociadas con un único **Type** (*hasType*). Notar que esta representación permite tipos más complejos, como por ejemplo una clase **List** cuyo tipo sea **List** y este último tenga como tipo asociado **BasicType**.

La clase **Enumeration** representa los tipos enumerados. Consta del atributo *values* definido como una lista de strings, donde cada elemento de esta lista es un valor posible para el enumerado. Finalmente, el tipo **BasicType** representa los tipos básicos de Coq (*nat*, *bool* y *string*).

Como se mencionó previamente, la utilización de KM3 fue determinada por el espacio técnico descrito en [15]. Esto plantea la necesidad de definir el metametamodelo KM3 como metamodelo a utilizar por la transformación ATL. Se analizaron dos posibilidades para expresar el metamodelo KM3 que participa como modelo origen en la transformación: utilizar XMI o utilizar la propia sintáxis de KM3.

El lenguaje ATL maneja en forma nativa modelos expresados en XMI que conforman con metamodelos expresados en Ecore. Por este motivo, la utilización de XMI resulta una alternativa natural aunque menos legible que KM3 para expresar los modelos origen. Como la transformación pretende partir de una representación en KM3, en este caso sería necesario definir una transformación texto-modelo que tome la representación del modelo origen en KM3 y la transforme a una representación en XMI. Además, se requeriría expresar el metametamodelo KM3 en Ecore para que sea utilizado por la transformación. Finalmente se optó por la segunda alternativa (emplear una representación textual que utiliza la sintaxis de KM3) ya que el entorno de desarrollo Eclipse cuenta con un plugin[2] que permite utilizar metamodelos estándar predefinidos. En particular, provee un metamodelo KM3 predefinido que permite que la transformación implementada tome como entrada directamente modelos origen expresados en KM3. A diferencia del metamodelo origen, el metamodelo destino fue expresado en Ecore mientras que el modelo destino es expresado en XMI.

La transformación se implementó en ATL utilizando una especificación declarativa. Uno de los mecanismos provistos que resultó de utilidad para la implementación fueron las *lazy rules*, permitiendo tener cierto control sobre el orden de ejecución de las reglas. Como se explicó en la sección 3.6, una lazy rule es una regla de transformación que no es ejecutada si no es invocada en forma explícita por otra regla.

A continuación, se describirán los pasos de la transformación ATL detallando las reglas de transformación implementadas. En primer lugar se describirán las reglas involucradas en la transformación de clases de KM3 a tipos inductivos de Coq. Luego, se detallarán aquellas reglas relacionadas a la transformación de atributos y referencias a componentes de los tipos inductivos.

Traducción de clases a tipos inductivos

Cada clase de KM3 (*Class* en el modelo de la Figura 5.2) se representa como un único tipo inductivo (*InductiveType* en Coq, Figura 5.5). En la definición de estos tipos inductivos se declaran componentes (*Component* Figura 5.5) para representar los atributos y las asociaciones de la clase. Es decir, cada tipo inductivo está asociado a un componente que corresponde a un atributo o asociación de la clase. De aquí en más se utilizarán los términos "clase", "componente" y "tipo inductivo" para refererirse a las instancias de los elementos en los metamodelos correspondientes. A su vez, un componente está asociado a un tipo en Coq determinado por el tipo del atributo o asociación de la clase KM3 a transformar. En

[2] En fase de desarrollo solo fue posible utilizar este plugin en un bundle específico que contenía al mismo. Los detalles de este bundle se encuentran descritos en el apéndice B. En algunos foros de usuarios se menciona que esta característica no se encuentra presente en la última versión (3.6) del entorno Eclipse.

el Código 5.5 se presenta la regla de transformación correspondiente a la transformación de clase a tipo inductivo.

Código 5.5 Regla de transformación ATL *Class* a *InductiveType*

```
rule Class2InductiveType {

     from
          i: KM3!Class ( not i.isAbstract )
     to
          o: COQ!InductiveType (
               isAbstract <- i.isAbstract,
               name <- i.name,
               hasComponents <-
                 Sequence{thisModule.BuildOidComponent(i)}
                   ->union(i.structuralFeatures),
               subTypes <- i.subclasses, superTypes <- i.supertypes )
}
rule AbstractClass2InductiveType {

     from
          i: KM3!Class (i.isAbstract)
     to
          o: COQ!InductiveType (
               isAbstract <- i.isAbstract,
               name <- i.name,
               hasComponents <- i.structuralFeatures,
               subTypes <- i.subclasses,
               superTypes <- i.supertypes
          )

}
```

La regla *Class2InductiveType* cubre todas las clases del modelo origen que no sean abstractas, mientras que la clase *AbtractClass2InductiveType* cubre todas las clases que sí lo son. Las dos reglas realizan tareas similares, mapeando atributos de las clases a atributos del tipo inductivo. Más adelante en esta sección se explicará la necesidad de diferenciar estos dos casos junto con la regla BuildOidComponent. Notar en la definición de las reglas, que la asociación *subclasses* de las clases no existe en el metamodelo origen. La misma, es una pseudo-asociación definida utilizando el helper que se muestra en el Código 5.6. Este helper construye la colección de subclases de una clase *self* recorriendo todas las posibles clases en el modelo origen y seleccionando aquellas que tienen como padre directo a *self*.

Código 5.6 helper definición de la pseudo-asociación *subclasses*

```
helper context KM3!Class def :

     subclasses : Collection(KM3!Class) =
          KM3!Class.allInstances()->select(c | c.supertypes->exists(x | x = self));
```

Para resolver las reglas presentadas en el Código 5.5, se deben resolver los bindings[3] para cada atributo de *InductiveType*. Para los atributos *isAbstract* y *name* el binding es directo, asignando el valor de los atributos de la clase directamente a los del tipo inductivo.

[3] el concepto de binding fue definido en la sección 3.6

En el caso de las asociaciones de la clase con sus sub-clases y sus super-clases (denominadas *subclasses* y *supertypes* respectivamente*)*, ATL resuelve los bindings aplicando nuevamente las reglas de transformacion de clases del Código 5.5. Dado que Coq no soporta constructores que permitan expresar herencia, esta relación se representa como una asociación más entre las clases involucradas en la jerarquía.

Traducción de atributos y asociaciones a componentes

El binding que resta por resolver es el que corresponde a los atributos y asociaciones de la clase (asociación *structuralFeatures* entre *Class* y *StructuralFeature,* Figura 5.2). Como se ha mencionado anteriormente, cada atributo o asociación de una clase se mapea como un componente en Coq. Así mismo, un componente está asociado con un tipo (asociación *hasType* entre *Component* y *Type*, Figura 5.5) el cual es determinado por la multiplicidad y tipo del atributo o asociación de la clase. El procedimiento es el siguiente:

- Si la clase a transformar posee atributos o asociaciones con cardinalidad mínima y máxima 1 (multiplicidad 1-1), cada atributo y/o asociación se mapea a un componente donde su tipo se resuelve en base al tipo del atributo o asociación. En el caso que el tipo corresponda a un tipo básico o a un enumerado (*BasicType* y *Enumeration* respectivamente, Figura 5.2), este se transforma a un tipo básico o enumerado en Coq según corrresponda (*BasicType* y *Enumeration*, Figura 5.5). En el Código 5.7 se presentan la reglas de transformación para los tipos básicos y enumerados. En la regla de transformación de tipos básicos a enumerados, se traducen además los nombres de los tipos básicos en KM3 a sus correspondientes en Coq. Por último, en el caso que el tipo de la asociación o atributo de una clase sea también una clase, el tipo asociado al componente es el resultado de la transformación de la dicha clase.

Código 5.7 Regla *Datatype2BasicType* y *Enumeration*

```
rule Enumeration {

     from
          i : KM3!Enumeration
     to
          out : COQ!Enumeration(
              name <- i.name,
              values <- i.literals->collect(literal | literal.name )
          )

}
rule Datatype2BasicType {

     from
          i : KM3!DataType
     using {
          -- Se consideran los tipos básicos de KM3: Integer, String y Boolean
          typeName : String =
              if (i.name = 'Integer') then
                'nat'
              else
                if (i.name = 'String') then
                  'string'
                else
                  'bool'
                endif
              endif; }
     to
          basicType : COQ!BasicType( name <- typeName )

}
```

- Si la clase a transformar posee atributos o asociaciones con cardinalidad mínima 0 y máxima 1 (multiplicidad 0-1), cada uno de ellos es mapeado a un componente de tipo option (*Option*, Figura5.5*)* en Coq. Un tipo option está asociado también a un tipo, el cual es determinado de la misma manera que que en el caso de la multiplicidad 1-1.
- Si la clase a transformar posee atributos o asociaciones con cardinalidad máxima mayor a 1, cada uno de ellos es mapeado a un componente asociado a un tipo *list* (*List*, Figura5.5*)* en Coq. De forma similar que para el caso anterior, el tipo list está también asociado a un tipo que es resuelto de la misma manera que para el caso de multiplicidad 1-1.

En el Código 5.8 se presentan las reglas de transformación ATL para lograr la transformación de atributos/asociación a componentes descripta.

Código 5.8 Reglas de transformación de *StructuralFeature* a *Component*

```
rule Feature2Component_Basic {

     from
          i : KM3!StructuralFeature (
              i.lower = 1 and i.upper = 1
          )
     to
          o : COQ!Component(
              name <- i.name,
              hasType <- i.type
              )

}
rule Feature2Component_Option {

     from
          i : KM3!StructuralFeature (
              i.lower = 0 and i.upper = 1
          )
     to o : COQ!Component(
          name <- i.name,
          hasType <- thisModule.Type2Option(i)
     )

}
rule Feature2Component_List {

     from
          i : KM3!StructuralFeature (
              i.upper < 0 or i.upper > 1
          )
     to o : COQ!Component(
          name <- i.name,
          hasType <- thisModule.Type2List(i)
     )

}
```

Como se mencionó, las multiplicidades en los atributos y asociaciones a tranformar son las que determinan la aplicación o no de una regla, y por consiguiente la salida de la transformación. La regla *Feature2Component_Basic* considera el caso más simple, donde la clase transformada posee atributos y/o asociaciones con multiplicidad 1-1. La regla *Feature2Component_Option* cubre los casos en que la clases a transformar posean atributos y/o asociaciones con multiplicidad 0-1, mientras que la regla *Feature2Component_List* cubre los casos en que las clases posean atributos o asociaciones cuya multiplicidad máxima sea mayor a 1[4]. Estas dos últimas reglas hacen uso de reglas *lazy* para lograr su cometido. La regla *Feature2Component_Option* utiliza la regla *Type2Option* para realizar la asociación del tipo option resultado con el tipo asociado a dicho option. La misma tarea realiza el helper *Type2List* para la regla *Feature2Component_List* pero para el caso del tipo list. Las reglas lazy descritas se presentan en el Código 5.9.

[4] Notar que en la Figura 5.8, la guarda en la regla *Feature2Component_List* considera las multiplicidades máximas a 1 y menores a 0. Eso es debido a que el metamodelo Coq fue definido en Ecore, y la representación de la cardinalidad * en este lenguaje de metamodelado es el valor -1.

Código 5.9 Reglas lazy Type2Option y Type2List

```
lazy rule Type2Option {

    from
        i:KM3!StructuralFeature
    to
        option : COQ!Option (
            name <- i.name,
            hasType <- i.type
        )
    }

lazy rule Type2List {

    from
        i:KM3!StructuralFeature
    to
        o : COQ!List (
            name <- i.name,
            hasType <- i.type
        )

}
```

Un detalle importante en la transformación de clase a tipo inductivo, surge de la necesidad de tener dos reglas diferenciando clases abstractas de las que no lo son. La diferencia entre las reglas es introducida para representar la identidad de objetos en el modelo Coq destino. Como se puede apreciar en el Código 5.5, la diferencia entre las reglas se introduce para el atributo *hasComponents*. La regla *lazy BuildOidComponent* agrega un componente extra llamado *oid y* con tipo asociado *Integer* para cada clase, de forma de establecer la identidad de las instancias de la misma en Coq[5]. Debido a que Coq solo permite comparar elementos por el valor de sus propiedades, este atributo *oid* permite comparar la igualdad entre instancias. Observando nuevamente el Código 5.5, donde solo se utiliza la regla *lazy BuildOidComponent* para las clases que no son abstractas, se ve que es razonable no agregar este componente a aquellas clases declaradas como abstractas ya que por definición no pueden ser instanciadas. La regla *BuildOidComponent* puede observarse en el Código 5.10.

Código 5.10 Regla lazy BuildOidComponent

```
lazy rule BuildOidComponent {

    from i : KM3!Class
    to o : COQ!Component (
        name <- 'oid',
        hasType <- i.getBasicType('Integer')
        )
    }
```

Como muestra el Código anterior, es una regla simple en donde para cada clase pasada como parámetro a la regla, se crea un componente de nombre *oid*, que tiene asociado mediante *hasType* el tipo *Integer*.

[5] Debido a un bug conocido en el prototipo desarrollado, todo modelo origen en KM3 de la transformación ATL debe declarar el datatype Integer. El motivo es que este el componente oid generado emplea este tipo y el mismo no es generado si no es declarado explícitamente.

Para la obtención del tipo del componente, se debe navegar el modelo origen y obtener el tipo deseado utilizando un *helper*. En el Código5.11 se presenta el helper definido. Como se puede obeservar, dicho helper itera en todas las posibles intancias de *DataType* del modelo origen, y retorna la clase identificada por el nombre especificado como parámetro al helper (variable name).

Código 5.11 helper getBasicType

```
helper context KM3!Class def : getBasicType(name: String) : KM3!DataType =

    KM3!DataType.allInstances()->any(c | c.name = name);
```

A continuación se presenta un ejemplo de la transformación descrita, transformando de forma parcial el metamodelo UML presentado en la seccion 2.2, Figura 2.3. La porción del modelo origen transformado es la siguiente:

```
package PartialUML {

    abstract class Classifier {

        attribute name : String;

    }
    class Class extends Classifier {

        attribute isPersistent : Boolean;
        attrs[*] ordered container : Attribute;

    }
    class Attribute {

        attribute name : String;

    }
    datatype Boolean;
    datatype Integer;
    datatype String;

}
```

Utilizando este ejemplo como modelo origen de la transformación ATL, el modelo resultante es el presentado en la Figura 5.3.

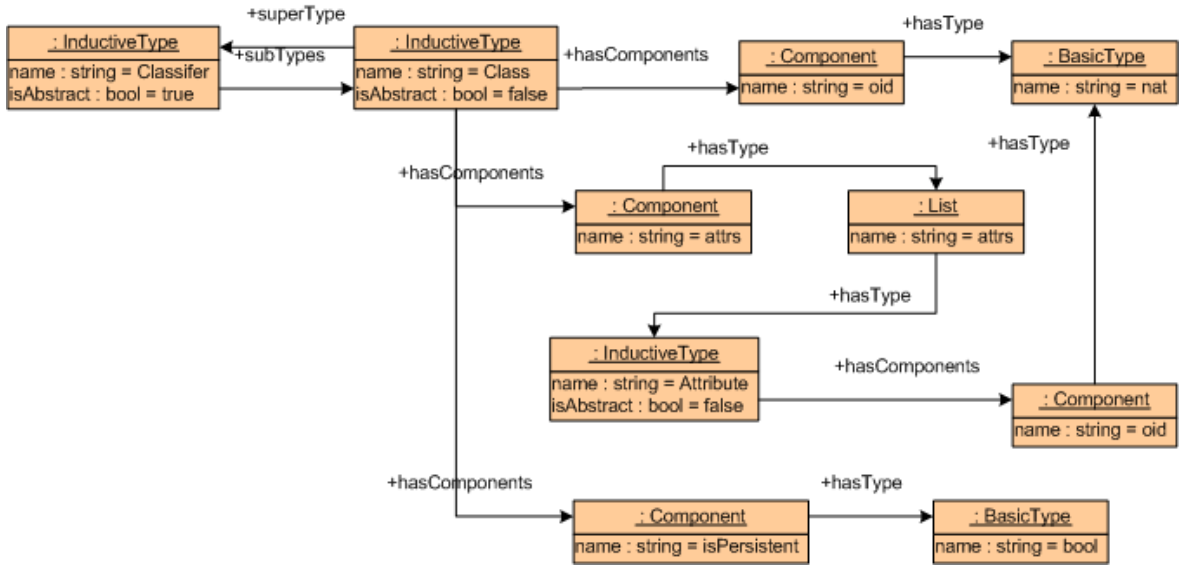

Figura 5.6: Resultado parcial de transformación de metamodelo UML

En la figura se representa la salida de la transformación ATL como un diagrama de clases. Puede observarse como cada clase (*Classifier*, *Class* y *Attribute*) del modelo origen es trasformada a un tipo inductivo de igual nombre. Así mismo, se ve cómo los atributos y/o asociaciones de casa clase son mapeados a componentes de igual nombre, relacionados con el tipo correspondiente. Notar además, la aparición del componente de nombre *oid*, que como se ha mencionado es utilizado en Coq para representar la identidad de los objetos.

Como se ha mencionado, la salida de la transformación ATL es un archivo XMI pero se representó como un diagrama de clases para facilitar la legibilidad. El Código 5.12 ilustra la salida del modelo de la Figura 5.3 en XMI.

Código 5.12 Resultado parcial de transformación de metamodelo UML en XMI

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

    <xmi:XMI ... >
    <coq:InductiveType xmi:id="a1" name="Classifier" isAbstract="true" hasComponents="a6"
    subTypes="a2"/>
    <coq:InductiveType xmi:id="a2" name="Class" hasComponents="a18 a7 a8 a13" superTypes="a1"/>
    <coq:InductiveType xmi:id="a5" name="Attribute" hasComponents="a21 a10"/>
    <coq:Component xmi:id="a6" name="name" hasType="a17"/>
    <coq:Component xmi:id="a7" name="isPersistent" hasType="a15"/>
    <coq:Component xmi:id="a9" name="name" hasType="a17"/>
    <coq:Component xmi:id="a10" name="name" hasType="a17"/>
    <coq:Component xmi:id="a13" name="attrs" hasType="a22"/>
    <coq:BasicType xmi:id="a15" name="bool"/>
    <coq:BasicType xmi:id="a16" name="nat"/>
    <coq:BasicType xmi:id="a17" name="string"/>
    <coq:Component xmi:id="a18" name="oid" hasType="a16"/>
    <coq:Component xmi:id="a21" name="oid" hasType="a16"/>
    <coq:List xmi:id="a22" name="attrs" hasType="a5"/>

</xmi:XMI>
```

Este modelo Coq expresado en XMI es idéntico al presentado en el diagrama de la Figura 5.3. Como ejemplo, puede observarse los tipos inductivos expresados con el tag *<coq:InductiveType />*. El atributo *xmi:id* en los tags del XMI son identificadores únicos de cada elemento. De esta manera, es posible relacionar los elementos definidos en el XMI. En el Código 5.12, se puede observar un ejemplo para el caso del componente de nombre *isPersistent* relacionado con su tipo básico *bool*, mediante el *xmi:id* de este último de valor *a15*.

El ejemplo aquí presentado fue utilizando en la transformación de modelo de clases UML a modelo relacional descrito en la sección 2.2. Sin embargo, el caso de estudio sobre el cual se probó el prototipo fue el de transformación del propio metamodelo KM3 expresado en KM3, a Coq. Este ejemplo completo puede consultarse en el apéndice A.

5.3.2. Transformación Xtext

El objetivo de esta transformación secundaria es obtener una representación textual que respete la sintaxis de Coq. Esta es una transformación modelo-texto para la cual se evaluaron dos alternativas diferentes en lo que respecta a herramientas: TCS y Xtext.

En primer lugar se evaluó realizar la implementación en TCS. Esta herramienta no cuenta con instalador para descargar[6] pero se encuentra pre-instalado en algunos versiones de Eclipse más viejas. Sin embargo, la falta de documentación y ejemplos de uso de la misma fue un factor decisivo para descartar su uso.

Finalmente se optó por Xtext como herramienta para la transformación modelo-texto. La elección se debió a su clara documentación y la versatilidad que provee al permitir utilizar extensiones de tipo OCL y código Java embebido. Xtext puede ser clasificada como una herramienta template-based (sección 3.1), dado que la transformación a texto es llevada a cabo utilizando templates. Un *template* es un archivo que contiene segmentos de metacódigo, el cual es procesado y reemplazado posteriormente por el texto correspondiente. En Xtext el lenguaje utilizado para la definición de templates es denominado Xpand. Además, Xtend permite agregar extensiones al código de los templates mediante expresiones similares a OCL e incluso código Java. Estas extensiones se pueden emplear para navegar y refinar el modelo origen[7].

La transformación modelo-texto desarrollada toma como entrada un modelo expresado en XMI resultante de la transformación principal en ATL y su salida es expresada en texto plano. Como se mencionó en la sección anterior sobre la transformación ATL, este archivo XMI conforma con el metamodelo Coq (Figura 5.5) el cual fue expresado en Ecore. Para que la herramienta pueda interpretar el modelo XMI como modelo origen, se debe definir previamente una gramática que describa la sintaxis de dicho archivo.

Definiendo la gramática antes mencionada se encontraron algunas fallas importantes en la herramienta que derivaron en nuevas decisiones de diseño para intentar evitarlas. Una de las fallas encontradas se relaciona con el procesamiento de los caracteres ' y " (comilla simple y comilla doble) en el modelo origen. Más precisamente, si el modelo origen está expresado en un archivo cuyo contenido presenta este tipo de caracteres, Xtext no es capaz de procesarlo. En el caso del archivo tomado de la salida de la transformación ATL (ejemplo Código 5.12) , estos caracteres se encuentran presentes en su sintaxis.

Otro problema de la herramienta sobre la definición de la gramática, es la imposiblidad de que la representación textual del modelo origen contenga una cadena constante de un largo mayor a 255 caracteres[8]. En caso de que el modelo origen contenga una cadena mayor a esta longitud, la herramienta conduce a un error de ejecución al intentar generar un archivo con el texto de la cadena como nombre.

[6] En la página de descargas de Generative Modelling Tools de Eclipse (http://www.eclipse.org/gmt/download/) se muestra el mensaje de que los binarios se encontrarán disponibles a la brevedad desde fines del año 2009.

[7] Para el desarrollo de la transformación modelo-texto fue de utilidad la documentación sobre Xtext publicada en http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html

[8] Este límite no se encuentra documentado y fue descubierto durante el desarrollo en un caso concreto. El largo de la cadena mas larga corresponde al máximo largo para un nombre de archivo permitido por Windows.

Los dos problemas anteriormente mencionados, fueron solucionados pre-procesando la representación textual del modelo origen. Esto se realizó mediante la implementación de un plugin del entorno Eclipse, que toma el archivo que representa el modelo origen de la transformación y resuelve los problemas mencionados. Los detalles específicos sobre el plugin y el pre-procesamiento del archivo pueden consultarse en el apéndice B.

Unfragmento de la gramática definida en Xtext para el procesamiento del modelo origen en XMI se presenta en el Código 5.13. El código completo del la gramática se encuentra en el apéndice D.

Código 5.13 Gramática de Coq

```
Model :

    '<xmi>'
        (definitions+=Definition)*
    '</xmi>'

;
Definition:

    Enumeration | InductiveType | Component | BasicType | Option | List

;
InductiveType:

    '<coq:InductiveType xmi:id=' name=ID
        'name=' typeName=ID
        ('isAbstract='isAbstract='true')?
        'hasComponents=' (components+=[Component])+
        ('subTypes=' (subtypes+=[InductiveType])+)?
        ('superTypes=' (supertypes+=[InductiveType])+)? '/>'

;
Component:

    '<coq:Component xmi:id=' name=ID
        'name=' compName=ID
        'hasType=' hasType=[Type] '/>'

;
Type:

    InductiveType | BasicType | Enumeration | Option | List

;
BasicType:

    '<coq:BasicType xmi:id=' name=ID 'name=' typeName=ID '/>'

;
...
...
```

Xtext utiliza esta gramática para construir un parser y un nuevo metamodelo origen expresado en Ecore. Los elementos de este nuevo metamodelo se construyen a partir de las etiquetas definidas en la gramática (ejemplo: *Model*, *Definition*, *Enumeration,* etc). De esta forma es posible navegar una instancia del nuevo metamodelo usando elementos definidos a partir de estas etiquetas como se describirá más adelante.

La transformación desarrollada tiene como punto de entrada el template principal *Main*.xpt (Código 5.14). Este template contiene las reglas de transformación principales que invocan a su vez reglas auxiliares. Para facilitar la legibilidad y la organización del código, las diferentes reglas se agruparon en

distintos templates auxiliares según el elemento sobre el cual se aplican y las estructuras a generar:

- Enumeration.xpt
- InductiveType.xpt
- Projections.xpt
- Model.xpt
- TypeOperators.xpt
- Projections.xpt
- Ancestors.xpt

El detalle de cada uno de estos templates será descrito más adelante en esta sección.

Código 5.14 Template principal Main.xpt

```
«IMPORT coq2Text»
«DEFINE main FOR Model »

    «FILE "salida.v"»
    Require Import String.
    Require Import List.
    Require Import Arith.
        «EXPAND Enumeration::build FOREACH
            this.definitions.typeSelect(Enumeration)»
        «EXPAND InductiveType::build FOR this »
        «EXPAND Projections::build FOREACH
            this.definitions.typeSelect(InductiveType)»
        «EXPAND Model::build FOR this »
        «EXPAND TypeOperators::build FOREACH
            this.definitions.typeSelect(InductiveType)»
        «EXPAND Projections::abstractoid FOREACH
            this.definitions.typeSelect(InductiveType)»
        «EXPAND TypeOperators::beq FOREACH
            this.definitions.typeSelect(InductiveType)»
        «EXPAND Ancestors::build FOREACH
            this.definitions.typeSelect(InductiveType)»
    «ENDFILE»

«ENDDEFINE»
```

A continuación, se describe en términos generales el template principal Main.xpt (Código 5.14). Las sentencias de metacódigo en los templates en lenguaje Xpand, son delimitadas por los caracteres « y ». La primera sentencia del template importa el metamodelo que contiene las entidades definidas por la gramática (en el ejemplo: «IMPORT coq2Text»).

Luego, se define la regla principal que será invocada en primera instancia al ejecutar la transformación. Esta regla, es definida como regla main y se aplica sobre el elemento **Model** declarado en la gramática. Este elemento contiene a todos los elementos restantes del metamodelo permitiendo la navegación hacia los mismos. El template principal Main.xpt navega el modelo origen invocando a cada uno de los templates auxiliares (definiciones EXPAND). Dicho template define además el nombre del archivo de salida donde se ubicará el resultado de la transformación a texto (en el ejemplo: "salida.v").

```
«DEFINE main FOR Model »
    «FILE "salida.v"»
```

En primer lugar, el template principal genera en el archivo de salida una serie de importaciones a librerías requeridas para las definiciones de tipos que contendrá el texto Coq generado: *String*, *List* y *Arith*. . Luego, el template traduce los diferentes elementos distinguiéndolos según su tipo y aplicando reglas específicas de traducción. A modo de ejemplo, la primer regla del template recorre todos los elementos de tipo *Enumeration* y aplica sobre los mismos la regla *build* definida en el template auxiliar *Enumeration*. Este template auxiliar es el encargado de generar el texto correspondiente para los enumerados en Coq.

```
«EXPAND Enumeration::build FOREACH
    this.definitions.typeSelect(Enumeration)»
```

En lo que resta de esta sección, se describirá el contenido de cada template auxiliar y sus reglas.

Template Enumeration

En este template se define la regla *build* que es aplicada para cada elemento Enumeration de forma de crear el tipo inductivo que lo representa.

Código 5.15 Código de la regla build del Template Enumeration

```
«DEFINE build FOR Enumeration»
    Inductive «this.typeName» : Set :=
    «FOREACH this.enumValues AS e»
        | «e.value» : «this.typeName»
    «ENDFOREACH ».
«ENDDEFINE»
```

Esta regla crea un tipo inductivo de igual nombre que el enumerado y con la estructura que se muestra en la Figura 5.16. En el ejemplo se puede observar la traducción de un enumerado *Color* con los valores *Red, Green y Yellow* en sintaxis Coq

Código 5.16 Representación enumerado en Coq

```
Inductive Color : Set :=
    | Red : Color
    | Green : Color
    | Yellow : Color.
```

Template InductiveType

En este template se define la regla *build* que es aplicada al elemento *Model*. Su objetivo es generar las definiciones de tipos mutuamente inductivos en Coq a partir de los tipos representados en modelo origen. Para cada InductiveType se iteran sobre sus componentes y subtipos generando los constructores que lo representan.

Código 5.17 Código de la regla build del Template InductiveType

```
«EXTENSION templates::Extensions»
«DEFINE build FOR Model »

    «FOREACH this.definitions.typeSelect(InductiveType) AS
        t ITERATOR iter »
        «IF iter.firstIteration »
            Inductive « t.typeName » : Set :=
        «ELSE»
            with « t.typeName » : Set :=
        «ENDIF »
        | Build_«t.typeName»
        «EXPAND component FOREACH t.components »
        «EXPAND subComponent FOREACH t.subtypes »
    «ENDFOREACH ».

« ENDDEFINE »
«DEFINE component FOR Component»

    («this.compName» : « getTypeName(this.hasType) »)

« ENDDEFINE »
«DEFINE subComponent FOR InductiveType »

    (sub« this.typeName » : option « this.typeName »)

« ENDDEFINE »
```

Para obtener el tipo de datos de cada componente a traducir se utiliza una extensión mediante Xtendrealizada en Java que se denomina *getTypeName*. El código completo de esta extensión se presenta en el apéndice D (ver método en clase *Util.java*). Las extensiones son importadas en los template mediante la directiva EXTENSION y declaradas en el archivo Extensions.ext. Para ver una descripción más detallada de la estructura del proyecto Xtext, referirse al apéndice B.

Un ejemplo del resultado de la aplicación de esta regla se puede ver en el Código 5.18. Aquí se puede apreciar un fragmento de la definición de tipos mutuamente inductivos para el metamodelo UML presentado en la sección 2.2. Se puede observar el empleo de algunos de los tipos básicos de Coq y las referencias a tipos definidos por la transformación. Es importante notar también que debido a la multiplicidad * del atributo *attrs* se emplea el tipo *list* (lo cual fue mencionado en la sub-sección 5.3.1).

Código 5.18 Constructor de tipos mutuamente inductivos

```
Inductive
    Attribute : Set := |

        Build_Attribute (oid : nat) (name : string) (from : Class) (type : Classifier)
    with
        Class : Set := |

            Build_Class (oid : nat) (isPersistent : bool) (pack : Package) (attrs : list Attribute)
    with
...
```

Como se explicó en la sección 5.2, si los tipos son mutuamente inductivos se debe generar una definición común a ambos de forma de resolver la dependencia entre los tipos en Coq. Si en cambio el tipo inductivo

solo depende de tipos que han sido definidos previamente, este puede ser definido en forma aislada. Sin embargo, determinar la interdependencia entre tipos de forma de reconocer aquellos tipos mutuamente inductivos no es una tarea trivial. Por este motivo se decidió en conjunto con los tutores considerar a todos los tipos como mutuamente inductivos, aunque no exista una interdependencia real entre todos ellos. De esta forma, se asegura que la definición de tipos sea procesable por la herramienta Coq.

Template Projections

Para permitir el acceso a los atributos de un tipo inductivo definido en Coq, se deben generar las llamadas proyecciones. Las *proyecciones* son funciones que se invocan sobre un elemento y devuelven el valor de un atributo. En el Código 5.19 se puede ver la proyección del componente *name* para el tipo *Attribute* presentado en el Código 5.18.

Código 5.19 Proyección del componente *name* en el tipo *Attribute*

```
Definition Attribute_name (o : Attribute) : string :=
match o with

     | (Build_Attribute _ a10 _ _) => a10
     end.
```

En el template *Projections* se definen dos reglas para generar estas proyecciones: *build* (Código 5.20) y *abstractoid* (Código 5.21). La regla *build* es aplicada para cada *InductiveType* creando las proyecciones para los componentes correspondientes del tipo. Como se mencionó en la sección 5.3.1, los atributos y asociaciones de las clases se mapean a componentes en Coq cuyo tipo se determina en base a la multiplicidad y los tipos originales de los atributos y/o asociaciones. En el caso de la herencia de clases, se mantienen referencias unidireccionales con sus subtipos. Para generar el texto de las proyecciones se utilizó la extensión *paramPrinter* realizada en Java. Esta función recibe como parámetros el nombre del atributo para la proyección, su posición dentro de la lista de atributos y la cantidad de atributos del tipo inductivo. La función retorna el string que se debe ubicar luego del nombre del constructor del tipo inductivo (en el ejemplo del Código 5.19 debe ubicarse luego de *Build_ Attribute*). El código de la función *paramPrinter* puede consultarse en el apéndice D, clase *Util.java*.

Código 5.20 Código de la regla *build* del Template Projections

```
«EXTENSION templates::Extensions»
«DEFINE build FOR InductiveType »

    «FOREACH this.components AS comp ITERATOR iter »
        Definition « this.typeName »_« comp.compName »
            (o : « this.typeName ») :
            « getTypeName(comp.hasType) » :=
            match o with
              |(Build_« this.typeName » «
              paramPrinter(comp.name, iter.counter1,
              iter.elements + this.subtypes.size) »
              ) => « comp.name »
        end.
    «ENDFOREACH »
    «FOREACH this.subtypes AS t ITERATOR iter »
        Definition « this.typeName »_sub« t.typeName »
            (o : « this.typeName ») : option « t.typeName » :=
            match o with
              | (Build_« this.typeName » « paramPrinter(t.name,
              this.components.size + iter.counter1,
              this.components.size + iter.elements) »)
              => « t.name »
        end.
    «ENDFOREACH »

« ENDDEFINE »
```

La regla *abstractoid* es aplicada para cada *InductiveType* abstracto creando la proyección que obtiene el valor del componente *oid*. Para generar dicha proyección se definió la extensión *abstractOidPrinter* realizada en Java, la cual puede consultarse en el apéndice D, clase *Util.java*.

Código 5.21 Código de la regla *abstractoid* del Template Projections

```
«DEFINE abstractoid FOR InductiveType »

    «IF this.isAbstract == "true" »
        « abstractOidPrinter(this.typeName, this.subtypes) »
    «ENDIF »

« ENDDEFINE »
```

En el Código 5.22 se presenta un ejemplo de proyección del atributo *oid* para la clase Classifier introducida en la sección 2.2.

Código 5.22 Proyección del atributo oid para clase Classifier

```
Definition Classifier_oid (i : Classifier) : nat :=

    match (asTypeClass i) with
        | None => match (asTypePrimitiveDataType i) with
                    | None => 0
                    | Some a3 => PrimitiveDataType_oid a3
                  end
        | Some a2 => Class_oid a2
    end.
```

Template Model

Este template toma la definición del modelo y genera la representación del modelo en Coq. Un modelo es simplemente un registro (record) con una lista para cada uno de los tipos inductivos generados.

Código 5.23 Código de la regla build del Template Model

```
«DEFINE build FOR Model »

    (* ---------- Model Definition ---------- *)
    « LET "COQModelName" AS modelName »
        Record « modelName » : Set :=
        « modelName » {
        «FOREACH this.definitions.typeSelect(InductiveType)
            AS t ITERATOR iter »
            MCoq_« t.typeName.toFirstLower() »
              AllInstances : list « t.typeName »
            «IF !iter.lastIteration »;« ENDIF »
        «ENDFOREACH » }.
    « ENDLET »

«ENDDEFINE»
```

En el Código 5.24 se muestra como ejemplo la especificación en Coq de un modelo, basándose en el metamodelo UML presentado en la sección 2.2.

Código 5.24 Código del modelo UML

```
Record UML : Set :=

    mkUML {
        MUml_classifierAllInstances : list Classifier;
        MUml_classAllInstances : list Class;
        MUml_primitiveDataTypeAllInstances : list PrimitiveDataType;
        MUml_packageAllInstances : list Package;
        MUml_attributeAllInstances : list Attribute
    }.
```

Template TypeOperators

Para todo tipo, generado a partir de una clase abstracta o no, deben generarse funciones que permitan comparar sus instancias. Además, deben generarse funciones que permitan consultar sobre el tipo concreto

de un objeto (isTypeOf) y realizar casting a un tipo concreto (asType) como las exhibidas en el Código 5.25.

Código 5.25 Funciones de comparación, consulta de tipo y casting

```
Definition beq_Attribute (o1 : Attribute) (o2 : Attribute) : bool :=

     beq_nat (Attribute_oid o1) (Attribute_oid o2).

Definition isTypeOfClass (a1 : Classifier) : bool :=

     match (Classifier_subClass a1) with
          | None => false
          | Some a => true
     end.

Definition isTypeOfPrimitiveDataType (a1 : Classifier) : bool :=

     match (Classifier_subPrimitiveDataType a1) with
          | None => false
          | Some a => true
     end.

Definition asTypeClass (a1 : Classifier) : option Class :=

     Classifier_subClass a1.

Definition asTypePrimitiveDataType (a1 : Classifier) : option PrimitiveDataType :=

     Classifier_subPrimitiveDataType a1.
```

En el template TypeOperators se definen dos reglas para generar dichas funciones: : *build* (Código 5.26) y *beq* (Código 5.27).

La regla *build* es aplicada sobre cada *InductiveType* para generar funciones que devuelven el tipo concreto de un objeto (isTypeOf) y permiten castear al tipo concreto (asType). Para generar las funciones isTypeOf y asType se itera sobre los subtipos directos de cada tipo inductivo.

Código 5.26 Código de la regla build del Template TypeOperators

```
«DEFINE build FOR InductiveType »

     «FOREACH this.subtypes AS t ITERATOR iter »
          Definition isTypeOf« t.typeName » (« this.name » :
              « this.typeName ») : bool :=
              match (« this.typeName »_sub« t.typeName »
              « this.name ») with
                 | None => false
                 | Some a => true
          end.
     «ENDFOREACH »
     «FOREACH this.subtypes AS t ITERATOR iter »
          Definition asType« t.typeName » (« this.name » :
          « this.typeName ») : option « t.typeName » :=
          « this.typeName »_sub« t.typeName » « this.name».
     «ENDFOREACH »

« ENDDEFINE »
```

Por otro lado, la regla *beq* es aplicada para cada *InductiveType* y genera las funciones de comparación

sobre la identidad de objetos. Notar que la comparación de los objetos se realiza mediante la comparación de los componentes oid del tipo inductivo. En el ejemplo del Código 5.25 se presenta la función de comparación para el tipo *Attribute* del Código 5.18.

Código 5.27 Código de la regla beq del Template TypeOperators

```
«DEFINE beq FOR InductiveType »

    Definition beq_« this.typeName » (o1 : « this.typeName »)
    (o2 : « this.typeName ») : bool :=

        beq_nat (« this.typeName »_oid o1)
        (« this.typeName »_oid o2).

« ENDDEFINE »
```

Template Ancestors

En este template (por claridad el código del template se presenta en el apéndice D, Ancestors.xpt) se generan definiciones y funciones para obtener el valor de los componentes del tipo padre, para los tipos que tengan al menos un padre. La traducción desarrollada se limita a casos de herencia simple, donde una clase no puede heredar de múltiples clases.

Para determinar los valores del componente del tipo padre, en primer lugar se evalúa si la colección de padres del tipo inductivo en cuestión (*supertypes*) no es vacía. En este caso, se toma el primer elemento como base y a partir de él se generan:

- La función *Fixpoint* que retorna una referencia al padre
- Las definiciones *Definition* que permiten obtener los valores de los componentes del tipo padre

Para obtener los atributos del padre se itera sobre el conjunto de componentes del padre y se genera una proyección para cada uno. Este conjunto es obtenido por medio de la extensión *getSuperComponents*. En el Código 5.25 se presenta un ejemplo para el tipo **Class** del Código 5.18.

Código 5.28 Funciones de valores de los componentes del tipo *Classifier* padre de *Class*

```
(* Obtiene la instancia de Classifier padre de Class *)
Fixpoint Class_super (l : list Classifier) (a2 : Class) : option Classifier :=

    match l with
        | nil => None
        | cons m l2 => match (Classifier_subClass m) with
            | None => Class_super l2 a2
            | Some x => match (beq_Class a2 x) with
              | true => Some m
              | false => Class_super l2 a2
            end
        end
    end.

(* Se accede al atributo name definido en Classifier desde el tipo Class *)
Definition Class_name (l : list Classifier) (a2 : Class) : option string :=

    match (Class_super l a2) with
        | None => None
        | Some x => Some (Classifier_name x)
    end.
```

Capítulo 6

Conclusiones y Trabajo Futuro

Debido a la reciente aparición de MDD en relación a otros enfoques de desarrollo y a su reducida aplicación en la industria del software, restan resolver aún varias limitaciones del mismo. Existe una gran diversidad de herramientas de desarrollo guiado por modelos que difieren en la forma de especificar una transformación de modelos y en el tipo de problemas que son capaces de resolver. La mayoría de estas herramientas aún se encuentran en fase de prueba y no cuentan con un soporte sólido en documentación. En este sentido, existe un importante esfuerzo de estandarización por parte del Object Management Group y es previsible que en un futuro próximo las diferentes aproximaciones de las herramientas converjan a estos estándares. Otro problema que se presenta es el relacionado con la verificación y validación de las transformaciones de modelo. Muchos investigadores han presentado diferentes aproximaciones a este problema que varían en el dominio de problemas que resuelve una transformación y en la factibilidad de la aproximación, ya que muchas son simplemente conceptualizaciones teóricas que aún no han sido puestas en práctica.

Como mencionamos en la introducción, en el transcurso de este proyecto se publicaron dos reportes técnicos. El primer reporte técnico publicado sobre herramientas y lenguajes para transformación de modelos provee un relevamiento del estado del arte en el área, que consideramos útil para conocer las alternativas existentes. Existe una gran diversidad de técnicas y lenguajes de transformación de modelos que abarcan desde enfoques de manipulación directa (ej: transformaciones desarrolladas en Java) hasta lenguajes dedicados como son las implementaciones basadas en el estándar QVT. Esta heterogeneidad se manifiesta también en las herramientas brindadas para especificar y ejecutar las transformaciones. El enfoque empleado es dependiente de la naturaleza del problema a resolver y no existe un único enfoque que contemple todas las posibles necesidades.

El reporte presenta una taxonomía que permite clasificar los diferentes enfoques de forma de conocer sus características y permitir la selección del más adecuado para resolver un problema específico. La primer distinción interesante en estos enfoques es si los mismos emplean herramientas o lenguajes. Algunos enfoques emplean herramientas no dedicadas a transformaciones de modelos (ejemplo: AGG), que cumplen efectivamente con el propósito de ciertas transformaciones y cuentan con el aval de toda su comunidad de usuarios (incluso de aquellos que las emplean con fines distintos al de transformación de modelos). Estas herramientas son, en general, probadas exhaustivamente por lo que su uso y posibilidades suelen estar bien documentados. Otros enfoques utilizan lenguajes de transformación de modelos con el fin de aprovechar los mecanismos adicionales (ejemplo: compatibilidad con diferentes lenguajes de metamodelado, mecanismos de trazabilidad, mecanismos de reuso y composición de transformaciones) que ofrecen para especificar transformaciones. Debido a la reciente aparición de estos lenguajes, los entornos y las herramientas que los implementan aún se encuentran inmaduros y en permanente perfeccionamiento [27].

Se presentó QVT como una especificación estándar de transformación de modelos. Esta especificación es el resultado de varios años de trabajo y cooperación del OMG y constituye una referencia para el desarrollo de la mayoría de los lenguajes de transformación de modelos. Como se mencionó, ATL se basa en la especificación de QVT y sigue la línea de los enfoques híbridos. Actualmente es uno de los más utilizados y difundidos. En particular cuenta con una gran cantidad de publicaciones relacionadas. La principal herramienta de ATL, al igual que un gran número herramientas de transformación de modelos, fue desarrollada como plug-in del IDE Eclipse. Esta plataforma posee varios años de desarrollo y uso, por lo que el desarrollo sobre la misma asegura cierta robustez. En general, las herramientas disponibles en este y otros entornos ofrecen características básicas como el reconocimiento de sintaxis y un debugger. Sin embargo, las herramientas cuentan con pocos años de desarrollo y aún existen funcionalidades deseables sin implementar, como la detección de errores en la codificación de transformaciones.

En el segundo reporte técnico se presentaron múltiples enfoques y técnicas para la verificación de transformaciones de modelos. Algunas técnicas toman la experiencia obtenida en la verificación dentro de otros paradigmas de desarrollo de software y la adaptan a las características de MDD. Esto muestra que los diferentes paradigmas comparten muchos conceptos de verificación como el análisis de caja blanca y caja negra, las mutaciones y las técnicas de cubrimiento. En cambio, otros métodos toman en consideración que toda transformación tiene como entrada y salida uno o más modelos que conforman con sus respectivos metamodelos. Esta característica es la que explotan los enfoques basados en model-checkers, donde se busca comprobar en forma automática, bajo ciertas restricciones, características de los modelos involucrados.

Una de las principales diferencias entre los distintos métodos documentados radica en la certeza que éstos brindan. Los métodos de verificación basados en el uso de casos de prueba sólo brindan certeza sobre los casos de prueba verificados, por lo que mediante técnicas de partición y cubrimiento se puede afirmar, sólo con cierto grado de confiabilidad, que las propiedades se preservarán para modelos no verificados. Por otro lado, las técnicas basadas en model-checking y en métodos deductivos brindan certeza absoluta sobre las propiedades que se analizan. Bajo métodos deductivos, muchas veces se realizan demostraciones constructivas que permiten generar transformaciones certificadas como correctas (ej: usando teoría de tipos).

Surge entonces la interrogante sobre por qué no se utilizan siempre métodos deductivos. El primer motivo es que los métodos deductivos requieren la utilización de lenguajes formales y herramientas matemáticas con los que la mayoría de los desarrolladores no están habituados a trabajar. Es un requisito indispensable el obtener (en el mejor de los casos de forma semiautomática) una especificación formal de la transformación, de los metamodelos y los modelos involucrados para poder luego utilizar asistentes de pruebas. Para superar esta dificultad existen algunas propuestas donde se intenta reducir la cantidad de formalismos mediante el uso de especificaciones semiformales. Sin embargo, estas técnicas que emplean semiformalismos suelen perder alcance y generalidad en los resultados de la verificación, o asumen la existencia de una transformación intermedia que preserva propiedades semánticas. En nuestro criterio, esa suposición sobre la transformación intermedia correcta nos devuelve al principio sobre el problema de verificación. El segundo motivo que parece aún más difícil de superar radica en el tipo de propiedades que se puede probar usando un enfoque u otro. Los métodos deductivos utilizan un mayor nivel de abstracción para expresar propiedades, al basarse en la especificación de metamodelos. Los métodos basados en el uso de casos de prueba en cambio permiten probar propiedades más específicas sobre las transformaciones.

Estas consideraciones generales sobre la verificación de transformaciones son motivo de la existencia de una gran variedad de enfoques. Ninguno de estos enfoques es capaz de contemplar por sí mismo todas las necesidades de verificación, por lo que deben emplearse criterios para determinar cuál es la técnica más apropiada para un problema concreto, teniendo en cuenta la criticidad de la aplicación. Debido a esta gran variedad, las herramientas de verificación no se encuentran aún integradas a las herramientas de desarrollo de MDD. La tendencia en herramientas de desarrollo MDD es permitir la especificación más general posible de transformaciones. Dada esta diversidad de transformaciones que

se pueden implementar, es difícil pensar en herramientas específicas de verificación para integrar a los entornos de desarrollo.

La última fase del proyecto, consistente en la implementación de un caso de estudio para una de las técnicas de verificación presentadas, afianzó la comprensión de los diferentes enfoques de implementación de transformaciones y las características de los mismos. Al utilizar un enfoque declarativo para la implementación de la transformación principal en ATL, se puso de manifiesto la necesidad de razonar el problema de forma diferente a la empleada en los paradigmas de desarrollo más habituales. En la transformación modelo-texto para obtener un texto procesable por Coq, se experimentó con la utilización de un enfoque basado en templates y el uso de extensiones en Java que siguen el enfoque imperativo. De esta forma, a lo largo del desarrollo se emplearon algunos de los enfoques descritos en el capítulo 3.

Como se planteó en el capítulo 5, este prototipo constituye un primer paso hacia la semi-automatización de la técnica descrita en [15], y resumida en el segundo reporte técnico y en el presente informe. El prototipo traduce los principales elementos de todo metamodelo expresado en KM3. Debido al alcance planteado por los tutores para este prototipo, resta aún cubrir algunos puntos considerados como trabajo a futuro. El primer punto refiere a las restricciones estructurales. Estas son aquellas que se encuentran definidas en forma implícita en las construcciones del diagrama del modelo a transformar. Estas restricciones deben especificarse en forma explícita en Coq, generando definiciones que puedan ser usadas posteriormente para probar propiedades. Un ejemplo de este tipo de restricciones es la multiplicidad en los extremos de las asociaciones. Si bien nuestro prototipo considera estas multiplicidades en el momento de determinar si un atributo debe traducirse como una Option o una List, deben incorporarse funciones que permitan verificar que la cantidad de elementos de una List respetan las cotas impuestas por el modelo original. De igual forma, se deben considerar las restricciones no estructurales. En caso que estas sean expresadas en un lenguaje formal como OCL, es posible realizar traducciones automáticas [10].

Apéndice A

Transformación KM3-Coq

La transformación ATL recibe como entrada el siguiente modelo KM3, expresado en KM3.

```
package KM3 {
    abstract class LocatedElement {
        attribute location : String;
    }
    abstract class ModelElement extends LocatedElement {
        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;
    }
    class EnumLiteral extends ModelElement {}
    class Class extends Classifier {
        attribute isAbstract : Boolean;
        reference supertypes[*] : Class;
        reference structuralFeatures[*] ordered container : StructuralFeature
        oppositeOf owner;
    }
    class StructuralFeature extends ModelElement {
        attribute lower : Integer;
        attribute upper : Integer;
        attribute isOrdered : Boolean;
        attribute isUnique : Boolean;
        reference owner : Class oppositeOf structuralFeatures;
        reference type : Classifier;
    }
    class Attribute extends StructuralFeature {}
    class Reference extends StructuralFeature {
        attribute isContainer : Boolean;
        reference opposite[0-1] : 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;
}
```

Luego de la ejecución de la transformación ATL, se obtiene como resultado la siguiente representación del modelo KM3 que conforma con la definición del metamodelo Coq.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <xmi:XMI xmi:version="2.1"
    xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:coq="http://proymde.fing.edu.uy/coq"
    xsi:schemaLocation="http://proymde.fing.edu.uy/coq
    platform:/resource/KM3Coq/metamodels/COQ.ecore">
    <coq:InductiveType xmi:id="a1" name="Classifier" hasComponents="a34"
    subTypes="a2 a3 a5" superTypes="a12"/>
    <coq:InductiveType xmi:id="a2" name="DataType" hasComponents="a35"
    superTypes="a1"/>
    <coq:InductiveType xmi:id="a3" name="Enumeration" hasComponents="a36 a26"
    superTypes="a1"/>
    <coq:InductiveType xmi:id="a4" name="EnumLiteral" hasComponents="a37"
    superTypes="a12"/>
    <coq:InductiveType xmi:id="a5" name="Class" hasComponents="a16 a27 a28 a38"
    superTypes="a1"/>
    <coq:InductiveType xmi:id="a6" name="StructuralFeature"
    hasComponents="a18 a21 a19 a39 a22 a20 a17" subTypes="a7 a8" superTypes="a12"/>
    <coq:InductiveType xmi:id="a7" name="Attribute" hasComponents="a40"
    superTypes="a6"/>
    <coq:InductiveType xmi:id="a8" name="Reference" hasComponents="a23 a41 a25"
    superTypes="a6"/>
    <coq:InductiveType xmi:id="a9" name="Package" hasComponents="a24 a29 a42"
    superTypes="a12"/>
    <coq:InductiveType xmi:id="a10" name="Metamodel" hasComponents="a30 a43"
    superTypes="a11"/>
    <coq:InductiveType xmi:id="a11" name="LocatedElement" isAbstract="true"
    hasComponents="a13" subTypes="a12 a10"/>
    <coq:InductiveType xmi:id="a12" name="ModelElement" isAbstract="true"
    hasComponents="a14 a15" subTypes="a1 a4 a6 a9" superTypes="a11"/>
    <coq:Component xmi:id="a13" name="location" hasType="a33"/>
    <coq:Component xmi:id="a14" name="name" hasType="a33"/>
    <coq:Component xmi:id="a15" name="package" hasType="a9"/>
    <coq:Component xmi:id="a16" name="isAbstract" hasType="a31"/>
    <coq:Component xmi:id="a17" name="lower" hasType="a32"/>
    <coq:Component xmi:id="a18" name="upper" hasType="a32"/>
    <coq:Component xmi:id="a19" name="isOrdered" hasType="a31"/>
    <coq:Component xmi:id="a20" name="isUnique" hasType="a31"/>
    <coq:Component xmi:id="a21" name="owner" hasType="a5"/>
    <coq:Component xmi:id="a22" name="type" hasType="a1"/>
    <coq:Component xmi:id="a23" name="isContainer" hasType="a31"/>
    <coq:Component xmi:id="a24" name="metamodel" hasType="a10"/>
    <coq:Component xmi:id="a25" name="opposite" hasType="a44"/>
    <coq:Component xmi:id="a26" name="literals" hasType="a45"/>
```

```
    <coq:Component xmi:id="a27" name="supertypes" hasType="a46"/>
    <coq:Component xmi:id="a28" name="structuralFeatures" hasType="a47"/>
    <coq:Component xmi:id="a29" name="contents" hasType="a48"/>
    <coq:Component xmi:id="a30" name="contents" hasType="a49"/>
    <coq:BasicType xmi:id="a31" name="bool"/>
    <coq:BasicType xmi:id="a32" name="nat"/>
    <coq:BasicType xmi:id="a33" name="string"/>
    <coq:Component xmi:id="a34" name="oid" hasType="a32"/>
    <coq:Component xmi:id="a35" name="oid" hasType="a32"/>
    <coq:Component xmi:id="a36" name="oid" hasType="a32"/>
    <coq:Component xmi:id="a37" name="oid" hasType="a32"/>
    <coq:Component xmi:id="a38" name="oid" hasType="a32"/>
    <coq:Component xmi:id="a39" name="oid" hasType="a32"/>
    <coq:Component xmi:id="a40" name="oid" hasType="a32"/>
    <coq:Component xmi:id="a41" name="oid" hasType="a32"/>
    <coq:Component xmi:id="a42" name="oid" hasType="a32"/>
    <coq:Component xmi:id="a43" name="oid" hasType="a32"/>
    <coq:Option xmi:id="a44" name="opposite" hasType="a8"/>
    <coq:List xmi:id="a45" name="literals" hasType="a4"/>
    <coq:List xmi:id="a46" name="supertypes" hasType="a5"/>
    <coq:List xmi:id="a47" name="structuralFeatures" hasType="a6"/>
    <coq:List xmi:id="a48" name="contents" hasType="a12"/>
    <coq:List xmi:id="a49" name="contents" hasType="a9"/>

</xmi:XMI>
```

Luego de la ejecución del plugin realizado para Eclipse, el representación anteriormente mostrada es preprocesada obteniendo el siguiente resultado.

```
<xmi>

    <coq:InductiveType xmi:id=a1 name=Classifier hasComponents=a34
    subTypes=a2 a3 a5 superTypes=a12/>
    <coq:InductiveType xmi:id=a2 name=DataType hasComponents=a35
    superTypes=a1/>
    <coq:InductiveType xmi:id=a3 name=Enumeration hasComponents=a36 a26
    superTypes=a1/>
    <coq:InductiveType xmi:id=a4 name=EnumLiteral hasComponents=a37
    superTypes=a12/>
    <coq:InductiveType xmi:id=a5 name=Class hasComponents=a16 a27 a28 a38
    superTypes=a1/>
    <coq:InductiveType xmi:id=a6 name=StructuralFeature
    hasComponents=a18 a21 a19 a39 a22 a20 a17 subTypes=a7 a8 superTypes=a12/>
    <coq:InductiveType xmi:id=a7 name=Attribute hasComponents=a40 superTypes=a6/>
    <coq:InductiveType xmi:id=a8 name=Reference hasComponents=a23 a41 a25
    superTypes=a6/>
    <coq:InductiveType xmi:id=a9 name=Package hasComponents=a24 a29 a42
    superTypes=a12/>
    <coq:InductiveType xmi:id=a10 name=Metamodel hasComponents=a30 a43
    superTypes=a11/>
    <coq:InductiveType xmi:id=a11 name=LocatedElement isAbstract=true hasComponents=a13
    subTypes=a12 a10/>
    <coq:InductiveType xmi:id=a12 name=ModelElement isAbstract=true
    hasComponents=a14 a15 subTypes=a1 a4 a6 a9 superTypes=a11/>
    <coq:Component xmi:id=a13 name=location hasType=a33/>
    <coq:Component xmi:id=a14 name=name hasType=a33/>
    <coq:Component xmi:id=a15 name=package hasType=a9/>
    <coq:Component xmi:id=a16 name=isAbstract hasType=a31/>
    <coq:Component xmi:id=a17 name=lower hasType=a32/>
    <coq:Component xmi:id=a18 name=upper hasType=a32/>
    <coq:Component xmi:id=a19 name=isOrdered hasType=a31/>
    <coq:Component xmi:id=a20 name=isUnique hasType=a31/>
```

```
    <coq:Component xmi:id=a21 name=owner hasType=a5/>
    <coq:Component xmi:id=a22 name=type hasType=a1/>
    <coq:Component xmi:id=a23 name=isContainer hasType=a31/>
    <coq:Component xmi:id=a24 name=metamodel hasType=a10/>
    <coq:Component xmi:id=a25 name=opposite hasType=a44/>
    <coq:Component xmi:id=a26 name=literals hasType=a45/>
    <coq:Component xmi:id=a27 name=supertypes hasType=a46/>
    <coq:Component xmi:id=a28 name=structuralFeatures hasType=a47/>
    <coq:Component xmi:id=a29 name=contents hasType=a48/>
    <coq:Component xmi:id=a30 name=contents hasType=a49/>
    <coq:BasicType xmi:id=a31 name=bool/>
    <coq:BasicType xmi:id=a32 name=nat/>
    <coq:BasicType xmi:id=a33 name=string/>
    <coq:Component xmi:id=a34 name=oid hasType=a32/>
    <coq:Component xmi:id=a35 name=oid hasType=a32/>
    <coq:Component xmi:id=a36 name=oid hasType=a32/>
    <coq:Component xmi:id=a37 name=oid hasType=a32/>
    <coq:Component xmi:id=a38 name=oid hasType=a32/>
    <coq:Component xmi:id=a39 name=oid hasType=a32/>
    <coq:Component xmi:id=a40 name=oid hasType=a32/>
    <coq:Component xmi:id=a41 name=oid hasType=a32/>
    <coq:Component xmi:id=a42 name=oid hasType=a32/>
    <coq:Component xmi:id=a43 name=oid hasType=a32/>
    <coq:Option xmi:id=a44 name=opposite hasType=a8/>
    <coq:List xmi:id=a45 name=literals hasType=a4/>
    <coq:List xmi:id=a46 name=supertypes hasType=a5/>
    <coq:List xmi:id=a47 name=structuralFeatures hasType=a6/>
    <coq:List xmi:id=a48 name=contents hasType=a12/>
    <coq:List xmi:id=a49 name=contents hasType=a9/>
</xmi>
```

Al aplicar la transformacion Xtext al modelo Coq preprocesado, se obtiene el siguiente código Coq.

```
Require Import String.
Require Import List.
Require Import Arith.
Inductive Classifier : Set :=
    | Build_Classifier (oid : nat) (subDataType : option DataType)
    (subEnumeration : option Enumeration) (subClass : option Class)
    with
        DataType : Set :=
            | Build_DataType (oid : nat)
    with
        Enumeration : Set :=
            | Build_Enumeration (oid : nat) (literals : list EnumLiteral)
    with
        EnumLiteral : Set :=
            | Build_EnumLiteral (oid : nat)
    with
        Class : Set :=
            | Build_Class (isAbstract : bool) (supertypes : list Class)
            (structuralFeatures : list StructuralFeature) (oid : nat)
    with
        StructuralFeature : Set :=
            | Build_StructuralFeature (upper : nat) (owner : Class)
            (isOrdered : bool) (oid : nat) (type : Classifier)
            (isUnique : bool) (lower : nat) (subAttribute : option Attribute)
            (subReference : option Reference)
```

```
     with
          Attribute : Set :=
               | Build_Attribute (oid : nat)
     with
          Reference : Set :=
               | Build_Reference (isContainer : bool) (oid : nat)
               (opposite : option Reference)
     with
          Package : Set :=

               | Build_Package (metamodel : Metamodel) (contents : list ModelElement)
               (oid : nat)
     with
          Metamodel : Set :=
               | Build_Metamodel (contents : list Package) (oid : nat)
     with
          LocatedElement : Set :=
               | Build_LocatedElement (location : string)
               (subModelElement : option ModelElement) (subMetamodel : option Metamodel)
     with
          ModelElement : Set :=
               | Build_ModelElement (name : string) (package : Package)
               (subClassifier : option Classifier) (subEnumLiteral : option EnumLiteral)
               (subStructuralFeature : option StructuralFeature)
               (subPackage : option Package).

Definition Classifier_oid (o : Classifier) : nat :=

     match o with
          | (Build_Classifier a34 _ _ _) => a34
     end.

Definition Classifier_subDataType (o : Classifier) : option DataType :=

     match o with
          | (Build_Classifier _ a2 _ _) => a2
     end.

Definition Classifier_subEnumeration (o : Classifier) : option Enumeration :=

     match o with
          | (Build_Classifier _ _ a3 _) => a3
     end.

Definition Classifier_subClass (o : Classifier) : option Class :=

     match o with
          | (Build_Classifier _ _ _ a5) => a5
     end.

Definition DataType_oid (o : DataType) : nat :=

     match o with
          | (Build_DataType a35) => a35
     end.

Definition Enumeration_oid (o : Enumeration) : nat :=

     match o with
```

```
          | (Build_Enumeration a36 _) => a36
     end.

Definition Enumeration_literals (o : Enumeration) : list EnumLiteral :=
     match o with
          | (Build_Enumeration _ a26) => a26
     end.

Definition EnumLiteral_oid (o : EnumLiteral) : nat :=
     match o with
          | (Build_EnumLiteral a37) => a37
     end.

Definition Class_isAbstract (o : Class) : bool :=
     match o with
          | (Build_Class a16 _ _ _) => a16
     end.

Definition Class_supertypes (o : Class) : list Class :=
     match o with
          | (Build_Class _ a27 _ _) => a27
     end.

Definition Class_structuralFeatures (o : Class) : list StructuralFeature :=
     match o with
          | (Build_Class _ _ a28 _) => a28
     end.

Definition Class_oid (o : Class) : nat :=
     match o with
          | (Build_Class _ _ _ a38) => a38
     end.

Definition StructuralFeature_upper (o : StructuralFeature) : nat :=
     match o with
          | (Build_StructuralFeature a18 _ _ _ _ _ _ _ _) => a18
     end.

Definition StructuralFeature_owner (o : StructuralFeature) : Class :=
     match o with
          | (Build_StructuralFeature _ a21 _ _ _ _ _ _ _) => a21
     end.

Definition StructuralFeature_isOrdered (o : StructuralFeature) : bool :=
     match o with
          | (Build_StructuralFeature _ _ a19 _ _ _ _ _ _) => a19
     end.

Definition StructuralFeature_oid (o : StructuralFeature) : nat :=
     match o with
          | (Build_StructuralFeature _ _ _ a39 _ _ _ _ _) => a39
     end.

Definition StructuralFeature_type (o : StructuralFeature) : Classifier :=
```

```
     match o with
          | (Build_StructuralFeature _ _ _ _ a22 _ _ _ _) => a22
     end.

Definition StructuralFeature_isUnique (o : StructuralFeature) : bool :=

     match o with
          | (Build_StructuralFeature _ _ _ _ _ a20 _ _ _) => a20
     end.

Definition StructuralFeature_lower (o : StructuralFeature) : nat :=
match o with

     | (Build_StructuralFeature _ _ _ _ _ _ a17 _ _) => a17

end.
Definition StructuralFeature_subAttribute (o : StructuralFeature) : option Attribute :=

     match o with
          | (Build_StructuralFeature _ _ _ _ _ _ _ a7 _) => a7
     end.

Definition StructuralFeature_subReference (o : StructuralFeature) : option Reference :=

     match o with
          | (Build_StructuralFeature _ _ _ _ _ _ _ _ a8) => a8
     end.

Definition Attribute_oid (o : Attribute) : nat :=

     match o with
          | (Build_Attribute a40) => a40
     end.

Definition Reference_isContainer (o : Reference) : bool :=

     match o with
          | (Build_Reference a23 _ _) => a23
     end.

Definition Reference_oid (o : Reference) : nat :=
match o with

     | (Build_Reference _ a41 _) => a41

end.
Definition Reference_opposite (o : Reference) : option Reference :=
match o with

     | (Build_Reference _ _ a25) => a25

end.
Definition Package_metamodel (o : Package) : Metamodel :=

     match o with
          | (Build_Package a24 _ _) => a24
     end.

Definition Package_contents (o : Package) : list ModelElement :=
match o with

     | (Build_Package _ a29 _) => a29

end.
Definition Package_oid (o : Package) : nat :=

     match o with
```

```
          | (Build_Package _ _ a42) => a42
     end.

Definition Metamodel_contents (o : Metamodel) : list Package :=
match o with

     | (Build_Metamodel a30 _) => a30

end.
Definition Metamodel_oid (o : Metamodel) : nat :=

     match o with
          | (Build_Metamodel _ a43) => a43
     end.

Definition LocatedElement_location (o : LocatedElement) : string :=

     match o with
          | (Build_LocatedElement a13 _ _) => a13
     end.

Definition LocatedElement_subModelElement (o : LocatedElement) : option ModelElement :=

     match o with
          | (Build_LocatedElement _ a12 _) => a12
     end.

Definition LocatedElement_subMetamodel (o : LocatedElement) : option Metamodel :=

     match o with
          | (Build_LocatedElement _ _ a10) => a10
     end.

Definition ModelElement_name (o : ModelElement) : string :=

     match o with
          | (Build_ModelElement a14 _ _ _ _ _) => a14
     end.

Definition ModelElement_package (o : ModelElement) : Package :=

     match o with
          | (Build_ModelElement _ a15 _ _ _ _) => a15
     end.

Definition ModelElement_subClassifier (o : ModelElement) : option Classifier :=

     match o with
          | (Build_ModelElement _ _ a1 _ _ _) => a1
     end.

Definition ModelElement_subEnumLiteral (o : ModelElement) : option EnumLiteral :=

     match o with
          | (Build_ModelElement _ _ _ a4 _ _) => a4
     end.

Definition ModelElement_subStructuralFeature (o : ModelElement) : option StructuralFeature :=

     match o with
          | (Build_ModelElement _ _ _ _ a6 _) => a6
     end.

Definition ModelElement_subPackage (o : ModelElement) : option Package :=
```

```
     match o with
          | (Build_ModelElement _ _ _ _ _ a9) => a9
     end.

(*-----------------------------------------------------------------------*)
(* ---------- Model Definition ---------- *)
Record KM3Model : Set :=

     KM3Model {
          MCoq_classifierAllInstances : list Classifier;
          MCoq_dataTypeAllInstances : list DataType;
          MCoq_enumerationAllInstances : list Enumeration;
          MCoq_enumLiteralAllInstances : list EnumLiteral;
          MCoq_classAllInstances : list Class;
          MCoq_structuralFeatureAllInstances : list StructuralFeature;
          MCoq_attributeAllInstances : list Attribute;
          MCoq_referenceAllInstances : list Reference;
          MCoq_packageAllInstances : list Package;
          MCoq_metamodelAllInstances : list Metamodel;
          MCoq_locatedElementAllInstances : list LocatedElement;
          MCoq_modelElementAllInstances : list ModelElement
     }.

Definition isTypeOfDataType (a1 : Classifier) : bool :=

     match (Classifier_subDataType a1) with
          | None => false
          | Some a => true
     end.

Definition isTypeOfEnumeration (a1 : Classifier) : bool :=

     match (Classifier_subEnumeration a1) with
          | None => false
          | Some a => true
     end.

Definition isTypeOfClass (a1 : Classifier) : bool :=

     match (Classifier_subClass a1) with
          | None => false
          | Some a => true
     end.

Definition asTypeDataType (a1 : Classifier) : option DataType :=

     Classifier_subDataType a1.

Definition asTypeEnumeration (a1 : Classifier) : option Enumeration :=

     Classifier_subEnumeration a1.

Definition asTypeClass (a1 : Classifier) : option Class :=

     Classifier_subClass a1.

Definition isTypeOfAttribute (a6 : StructuralFeature) : bool :=

     match (StructuralFeature_subAttribute a6) with
          | None => false
          | Some a => true
     end.

Definition isTypeOfReference (a6 : StructuralFeature) : bool :=

     match (StructuralFeature_subReference a6) with
```

```
          | None => false
          | Some a => true
     end.

Definition asTypeAttribute (a6 : StructuralFeature) : option Attribute :=

     StructuralFeature_subAttribute a6.

Definition asTypeReference (a6 : StructuralFeature) : option Reference :=

     StructuralFeature_subReference a6.

Definition isTypeOfModelElement (a11 : LocatedElement) : bool :=

     match (LocatedElement_subModelElement a11) with
          | None => false
          | Some a => true
     end.

Definition isTypeOfMetamodel (a11 : LocatedElement) : bool :=

     match (LocatedElement_subMetamodel a11) with
          | None => false
          | Some a => true
     end.

Definition asTypeModelElement (a11 : LocatedElement) : option ModelElement :=

     LocatedElement_subModelElement a11.

Definition asTypeMetamodel (a11 : LocatedElement) : option Metamodel :=

     LocatedElement_subMetamodel a11.

Definition isTypeOfClassifier (a12 : ModelElement) : bool :=

     match (ModelElement_subClassifier a12) with
          | None => false
          | Some a => true
     end.

Definition isTypeOfEnumLiteral (a12 : ModelElement) : bool :=

     match (ModelElement_subEnumLiteral a12) with
          | None => false
          | Some a => true
     end.

Definition isTypeOfStructuralFeature (a12 : ModelElement) : bool :=

     match (ModelElement_subStructuralFeature a12) with
          | None => false
          | Some a => true
     end.

Definition isTypeOfPackage (a12 : ModelElement) : bool :=

     match (ModelElement_subPackage a12) with
          | None => false
          | Some a => true
     end.

Definition asTypeClassifier (a12 : ModelElement) : option Classifier :=

     ModelElement_subClassifier a12.

Definition asTypeEnumLiteral (a12 : ModelElement) : option EnumLiteral :=
```

```
     ModelElement_subEnumLiteral a12.

Definition asTypeStructuralFeature (a12 : ModelElement) : option StructuralFeature :=

     ModelElement_subStructuralFeature a12.

Definition asTypePackage (a12 : ModelElement) : option Package :=

     ModelElement_subPackage a12.

Definition LocatedElement_oid (i : LocatedElement) : nat  :=

     match (asTypeModelElement i) with
          | None => match (asTypeMetamodel i) with
                      | None => 0
                      | Some a10 => Metamodel_oid a10
                    end
          | Some a12 => ModelElement_oid a12
     end.

Definition ModelElement_oid (i : ModelElement) : nat  :=

     match (asTypeClassifier i) with
          | None => match (asTypeEnumLiteral i) with
                      | None => match (asTypeStructuralFeature i) with
                                  | None => match (asTypePackage i) with
                                              | None => 0
                                              | Some a9 => Package_oid a9
                                            end
                                | Some a6 => StructuralFeature_oid a6
                                end
                      | Some a4 => EnumLiteral_oid a4
                    end
          | Some a1 => Classifier_oid a1
     end.

Definition beq_Classifier (o1 : Classifier) (o2 : Classifier) : bool :=

     beq_nat (Classifier_oid o1) (Classifier_oid o2).

Definition beq_DataType (o1 : DataType) (o2 : DataType) : bool :=

     beq_nat (DataType_oid o1) (DataType_oid o2).

Definition beq_Enumeration (o1 : Enumeration) (o2 : Enumeration) : bool :=

     beq_nat (Enumeration_oid o1) (Enumeration_oid o2).

Definition beq_EnumLiteral (o1 : EnumLiteral) (o2 : EnumLiteral) : bool :=

     beq_nat (EnumLiteral_oid o1) (EnumLiteral_oid o2).

Definition beq_Class (o1 : Class) (o2 : Class) : bool :=

     beq_nat (Class_oid o1) (Class_oid o2).

Definition beq_StructuralFeature (o1 : StructuralFeature)
(o2 : StructuralFeature) : bool :=

     beq_nat (StructuralFeature_oid o1) (StructuralFeature_oid o2).

Definition beq_Attribute (o1 : Attribute) (o2 : Attribute) : bool :=

     beq_nat (Attribute_oid o1) (Attribute_oid o2).

Definition beq_Reference (o1 : Reference) (o2 : Reference) : bool :=

     beq_nat (Reference_oid o1) (Reference_oid o2).
```

```
Definition beq_Package (o1 : Package) (o2 : Package) : bool :=

     beq_nat (Package_oid o1) (Package_oid o2).

Definition beq_Metamodel (o1 : Metamodel) (o2 : Metamodel) : bool :=

     beq_nat (Metamodel_oid o1) (Metamodel_oid o2).

Definition beq_LocatedElement (o1 : LocatedElement) (o2 : LocatedElement) : bool :=

     beq_nat (LocatedElement_oid o1) (LocatedElement_oid o2).

Definition beq_ModelElement (o1 : ModelElement) (o2 : ModelElement) : bool :=

     beq_nat (ModelElement_oid o1) (ModelElement_oid o2).

(*Para la siguientes definiciones, se considera herencia simple, solo
al padre directo y los atributos del mismo*)
Fixpoint Classifier_super (l : list ModelElement) (a1 : Classifier) : option ModelElement :=

     match l with
          | nil => None
          | cons m l2 => match (ModelElement_subClassifier m) with
                              | None => Classifier_super l2 a1
                              |Some x => match (beq_Classifier a1 x) with
                                          | true => Some m
                                          | false => Classifier_super l2 a1
                                         end
                             end
     end.

Definition Classifier_name (l : list ModelElement) (a1 : Classifier) : option string :=

     match (Classifier_super l a1) with
          | None => None
          | Some x => Some (ModelElement_name x)
     end.

Definition Classifier_package (l : list ModelElement) (a1 : Classifier) : option Package :=

     match (Classifier_super l a1) with
          | None => None
          | Some x => Some (ModelElement_package x)
     end.

Fixpoint DataType_super (l : list Classifier) (a2 : DataType) : option Classifier :=

     match l with
          | nil => None
          | cons m l2 => match (Classifier_subDataType m) with
                          | None => DataType_super l2 a2
                          | Some x => match (beq_DataType a2 x) with
                                       | true => Some m
                                       | false => DataType_super l2 a2
                                      end
                         end
     end.

Fixpoint Enumeration_super (l : list Classifier) (a3 : Enumeration) : option Classifier :=

     match l with
     | nil => None
```

```
        | cons m l2 => match (Classifier_subEnumeration m) with
                       | None => Enumeration_super l2 a3
                       | Some x => match (beq_Enumeration a3 x) with
                                   | true => Some m
                                   | false => Enumeration_super l2 a3
                                   end
                     end
    end.

Fixpoint EnumLiteral_super (l : list ModelElement) (a4 : EnumLiteral) : option ModelElement :=

    match l with
        | nil => None
        | cons m l2 => match (ModelElement_subEnumLiteral m) with
                       | None => EnumLiteral_super l2 a4
                       | Some x => match (beq_EnumLiteral a4 x) with
                                   | true => Some m
                                   | false => EnumLiteral_super l2 a4
                                   end
                     end
    end.

Definition EnumLiteral_name (l : list ModelElement) (a4 : EnumLiteral) :
option string :=

    match (EnumLiteral_super l a4) with
        | None => None
        | Some x => Some (ModelElement_name x)
    end.

Definition EnumLiteral_package (l : list ModelElement) (a4 : EnumLiteral) :
option Package :=

    match (EnumLiteral_super l a4) with
        | None => None
        | Some x => Some (ModelElement_package x)
    end.

Fixpoint Class_super (l : list Classifier) (a5 : Class) : option Classifier :=

    match l with
        | nil => None
        | cons m l2 => match (Classifier_subClass m) with
                         | None => Class_super l2 a5
                         | Some x => match (beq_Class a5 x) with
                                     | true => Some m
                                     | false => Class_super l2 a5
                                     end
                       end
    end.

Fixpoint StructuralFeature_super (l : list ModelElement) (a6 : StructuralFeature) :
option ModelElement :=

    match l with
        | nil => None
        | cons m l2 => match (ModelElement_subStructuralFeature m) with
                       | None => StructuralFeature_super l2 a6
                       | Some x => match (beq_StructuralFeature a6 x) with
                                   | true => Some m
                                   | false => StructuralFeature_super l2 a6
                                   end
```

```
                        end
     end.

Definition StructuralFeature_name (l : list ModelElement) (a6 : StructuralFeature) :
option string :=

     match (StructuralFeature_super l a6) with
     | None => None
          | Some x => Some (ModelElement_name x)

     end.

Definition StructuralFeature_package (l : list ModelElement) (a6 : StructuralFeature)
: option Package :=

     match (StructuralFeature_super l a6) with
          | None => None
          | Some x => Some (ModelElement_package x)

     end.

Fixpoint Attribute_super (l : list StructuralFeature) (a7 : Attribute) :
option StructuralFeature :=

     match l with
          | nil => None
          | cons m l2 => match (StructuralFeature_subAttribute m) with
                         | None => Attribute_super l2 a7
                         | Some x => match (beq_Attribute a7 x) with
                                     | true => Some m
                                     | false => Attribute_super l2 a7
                                    end
                        end

     end.

Definition Attribute_upper (l : list StructuralFeature) (a7 : Attribute) :
option nat :=

     match (Attribute_super l a7) with
          | None => None
          | Some x => Some (StructuralFeature_upper x)

     end.

Definition Attribute_owner (l : list StructuralFeature) (a7 : Attribute) :
option Class :=

     match (Attribute_super l a7) with
          | None => None
          | Some x => Some (StructuralFeature_owner x)

     end.

Definition Attribute_isOrdered (l : list StructuralFeature) (a7 : Attribute) :
option bool :=

     match (Attribute_super l a7) with
          | None => None
          | Some x => Some (StructuralFeature_isOrdered x)

     end.

Definition Attribute_type (l : list StructuralFeature) (a7 : Attribute) :
option Classifier :=

     match (Attribute_super l a7) with
          | None => None
          | Some x => Some (StructuralFeature_type x)
```

```
     end.

Definition Attribute_isUnique (l : list StructuralFeature) (a7 : Attribute) :
option bool :=

     match (Attribute_super l a7) with
          | None => None
          | Some x => Some (StructuralFeature_isUnique x)
     end.

Definition Attribute_lower (l : list StructuralFeature) (a7 : Attribute) :
option nat :=

     match (Attribute_super l a7) with
          | None => None
          | Some x => Some (StructuralFeature_lower x)
     end.

Fixpoint Reference_super (l : list StructuralFeature) (a8 : Reference) :
option StructuralFeature :=

     match l with
          | nil => None
          | cons m l2 => match (StructuralFeature_subReference m) with
                         | None => Reference_super l2 a8
                         | Some x => match (beq_Reference a8 x) with
                                     | true => Some m
                                     | false => Reference_super l2 a8
                                     end
                         end
     end.

Definition Reference_upper (l : list StructuralFeature) (a8 : Reference) :
option nat :=

     match (Reference_super l a8) with
          | None => None
          | Some x => Some (StructuralFeature_upper x)
     end.

Definition Reference_owner (l : list StructuralFeature) (a8 : Reference) :
option Class :=

     match (Reference_super l a8) with
          | None => None
          | Some x => Some (StructuralFeature_owner x)
     end.

Definition Reference_isOrdered (l : list StructuralFeature) (a8 : Reference) :
option bool :=

     match (Reference_super l a8) with
          | None => None
          | Some x => Some (StructuralFeature_isOrdered x)
     end.

Definition Reference_type (l : list StructuralFeature) (a8 : Reference) :
option Classifier :=

     match (Reference_super l a8) with
          | None => None
          | Some x => Some (StructuralFeature_type x)
     end.
```

```
Definition Reference_isUnique (l : list StructuralFeature) (a8 : Reference) :
option bool :=

     match (Reference_super l a8) with
          | None => None
          | Some x => Some (StructuralFeature_isUnique x)
     end.

Definition Reference_lower (l : list StructuralFeature) (a8 : Reference) :
option nat :=

     match (Reference_super l a8) with
          | None => None
          | Some x => Some (StructuralFeature_lower x)
     end.

Fixpoint Package_super (l : list ModelElement) (a9 : Package) :
option ModelElement :=

     match l with
          | nil => None
          | cons m l2 => match (ModelElement_subPackage m) with
                           | None => Package_super l2 a9
                           | Some x => match (beq_Package a9 x) with
                                         | true => Some m
                                         | false => Package_super l2 a9
                                       end
                         end
     end.

Definition Package_name (l : list ModelElement) (a9 : Package) :
option string :=

     match (Package_super l a9) with
          | None => None
          | Some x => Some (ModelElement_name x)
     end.

Definition Package_package (l : list ModelElement) (a9 : Package) :
option Package :=

     match (Package_super l a9) with
     | None => None
     | Some x => Some (ModelElement_package x)

end.
Fixpoint Metamodel_super (l : list LocatedElement) (a10 : Metamodel) :
option LocatedElement :=

     match l with
          | nil => None
          | cons m l2 => match (LocatedElement_subMetamodel m) with
                          | None => Metamodel_super l2 a10
                          | Some x => match (beq_Metamodel a10 x) with
                                        | true => Some m
                                        | false => Metamodel_super l2 a10
                                      end
                        end
     end.

Definition Metamodel_location (l : list LocatedElement) (a10 : Metamodel) :
option string :=

     match (Metamodel_super l a10) with
```

```
          | None => None
          | Some x => Some (LocatedElement_location x)
     end.


Fixpoint ModelElement_super (l : list LocatedElement) (a12 : ModelElement) :
option LocatedElement :=
     match l with
          | nil => None
          | cons m l2 => match (LocatedElement_subModelElement m) with
                         | None => ModelElement_super l2 a12
                         | Some x => match (beq_ModelElement a12 x) with
                                     | true => Some m
                                     | false => ModelElement_super l2 a12
                                    end
            end
     end.

Definition ModelElement_location (l : list LocatedElement) (a12 : ModelElement) :
option string :=
     match (ModelElement_super l a12) with
          | None => None
          | Some x => Some (LocatedElement_location x)
     end.
```

Apéndice B

Configuración del ambiente y uso del prototipo

En este apéndice se describen los detalles técnicos para la configuración y la ejecución del prototipo. Se detallarán las herramientas requeridas para la ejecución y los pasos a seguir para utilizar el prototipo.

B.1. Configuración del entorno

Para desarrollar la transformación del modelo KM3 al texto con sintáxis Coq, fue necesario utilizar dos versiones diferentes del entorno de desarrollo. Esto se debió a problemas de compatibilidad entre las versiones de la plataforma y los plugins requeridos para integrar ATL y Xtext. En particular, fue necesario utilizar el plugin AM3 [5] para utilizar modelos expresados KM3 como modelos origen de la transformación ATL. La integración del plugin de AM3 con ATL solo fue posible en la version 3.4.2 de la plataforma Eclipse[1]. Para la utilización de Xtext se empleó Eclipse Modelling Tools en su version 3.5.1[2], ya que esta herramienta no se encontraba disponible para el bundle al cual se integró AM3. De esta forma, cada una de las transformaciones (ATL y Xtext) fue desarrollada y ejecutada en versiones diferentes del entorno Eclipse.

B.2. Transformación ATL

A continuación se describirá la estructura del proyecto Km3Coq que implementa la transformación ATL (Figura B.1). Este proyecto se divide en 4 carpetas:

- **Input**: En esta carpeta se encuentran los archivos de entrada para la transformación. Para el caso de estudio propuesto en este proyecto se utilizará la especificación de KM3 escrita en KM3.
- **Metamodels**: Aquí se encuentran los metamodelos involucrados en la transformación. Se puede observar que solo se encuentra definido el metamodelo Coq. El metamodelo KM3 utilizado es el metamodelo predefinido por el plug-in AM3.

[1] Esta versión de la plataforma se obtuvo de un bundle descargado de http://docatlanmod.emn.fr/Eclipse_Bundles/AMMA_Prototype/AMMAPrototypeEclipse.zip. Este bundle solo puede ser utilizado en el sistema operativo Windows.

[2] El bundle de Eclipse Modelling Tools puede ser descargado de http://www.eclipse.org/modeling/downloads. Este bundle se encuentra disponible para diversos sistemas operativos.

- **Output**: En esta carpeta se encuentran los archivos de salida de la transformación.
- **Transformations**: Aquí se encuentra la definición de la transformación ATL.

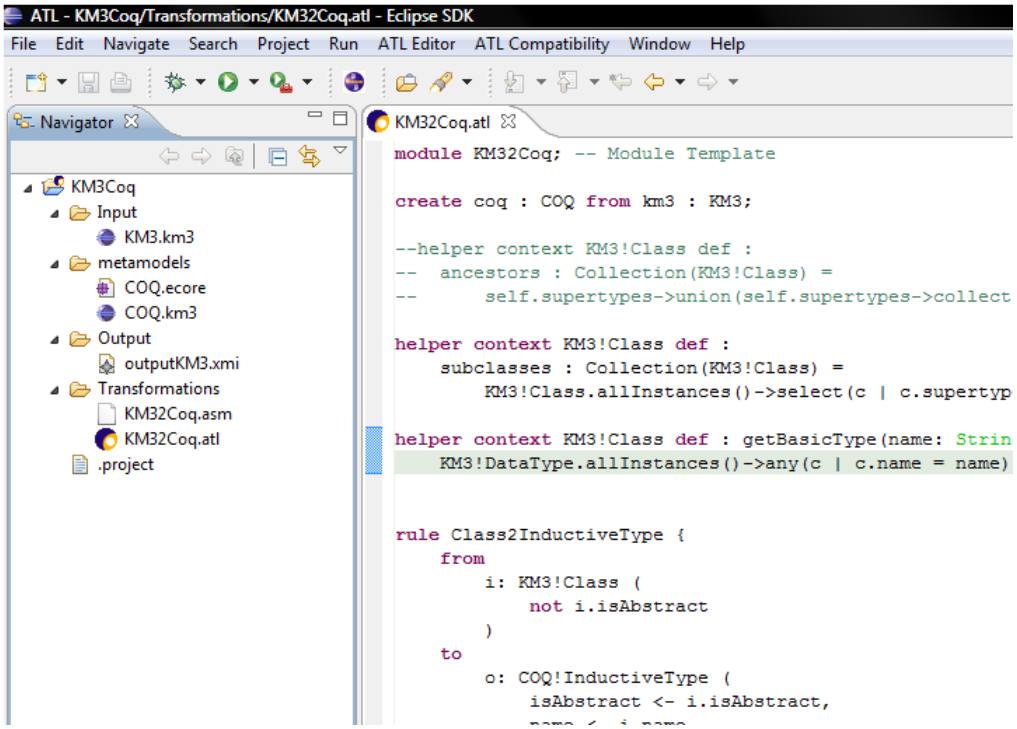

Figura B.1: Transformación ATL - Proyecto Km3Coq

La configuración para la ejecución de la transformación, se realiza de manera similar a la configuración para ejecutar un proyecto Java en Eclipse (dirigiéndose al menú *Run -> Run Configurations*). Luego de crear una nueva configuración de tipo ATL, se observará una ventana similar a la de la Figura B.2. En la misma se muestra la información mínima indispensable que necesita la transformación ATL para ejecutar. En el ejemplo de la figura, se indica el archivo que contiene la definición de la transformacion (*KM33Coq.atl*), el metamodelo origen (pre-definido y ubicado en la uri, *http://www.eclipse.org/gmt/KM3*), el metamodelo destino (*COQ.ecore*), el modelo origen (*KM3.km3*) y el archivo en donde se escribirá el modelo destino (*outputKM3.xmi*).

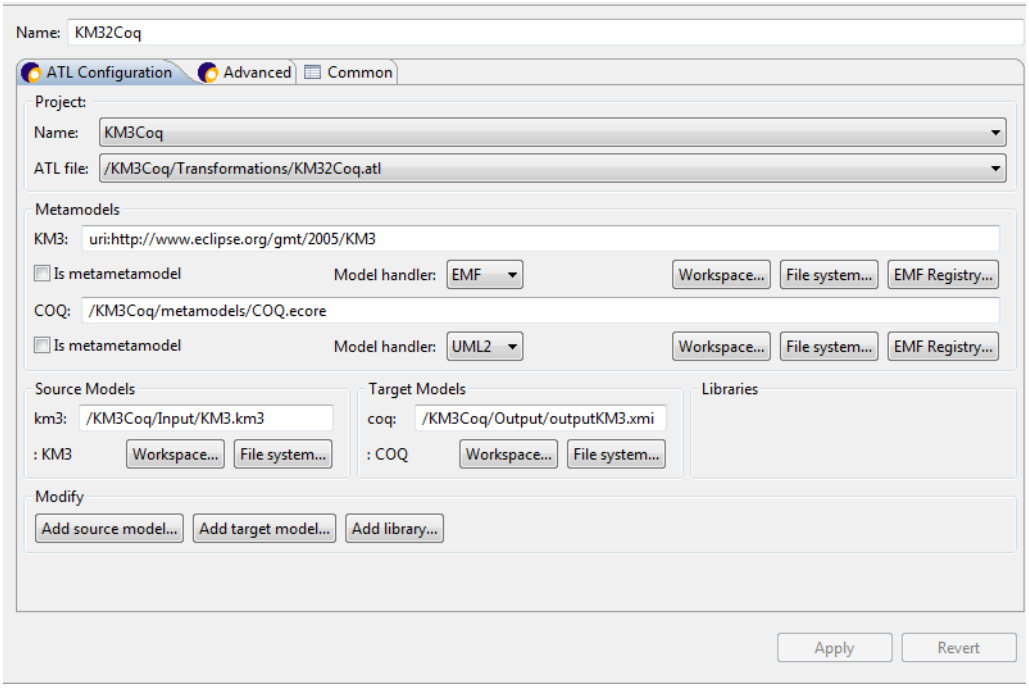

Figura B.2: Configuración de la transformación ATL

B.3. Transformación Xtext

Como se ha mencionado en la sub-sección 5.3.2, la transformación a texto utilizando la herramienta Xtext toma como entrada un modelo expresado en XMI (modelo destino de la transformación ATL) y genera el texto que respeta la sintaxis Coq. La figura B.4muestra los tres proyectos necesarios para la transformación Xtext:

- **org.xtext.example.coq2text**: En este proyecto se encuentra definida la gramática que reconoce la representación intermedia XMI (en la ubicación *src/org.xtext.example/Coq2Text.xtext*). Al compilar la gramática se genera el parser necesario para el reconocimiento de la representacion intermedia e internamente se crea un modelo Ecore que representa el lenguaje definido por la gramática. Para compilar la gramática se deberá ejecutar el workflow *GenerateCoq2Text.mwe* ubicado en el mismo paquete de la gramática. Para ejecutar el workflow se debe hacer click derecho sobre el mismo y seleccionar *Run As/MWE Workflow*.

- **org.xtext.example.coq2text.generator** (de aquí en más *proyecto generador*): En este proyecto se encuentran el modelo origen, los templates Xpand, funciones auxiliares Xtend y el workflow *Coq2TextGenerator.mwe*. Al ejecutar este workflow, los templates recorren la estructura que define la gramática y genera el texto que respeta la sintaxis Coq. La Figura B.3muestra el workflow anteriormente mencionado y el resultado de la transformación, archivo con extensión .v generado bajo el folder */src-gen* del proyecto generador.

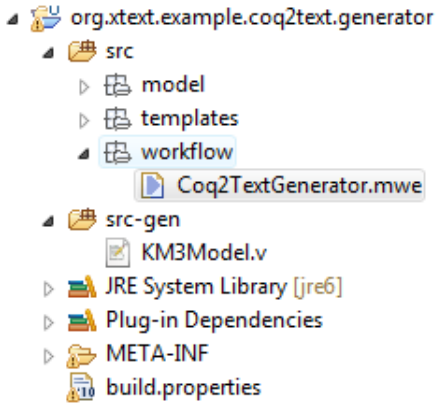

Figura B.3: Workflow generador y resultado de la transformación Xtext

- **org.xtext.example.coq2text.ui**: Este proyecto contiene un editor generado en función de la gramática, que permite escribir textos que respeten la gramática señalando inmediatamente posibles errores de sintaxis. Si bien se utilizó este proyecto para verificar la correctitud de la gramática desarrollada, el mismo no es utilizado por el prototipo.

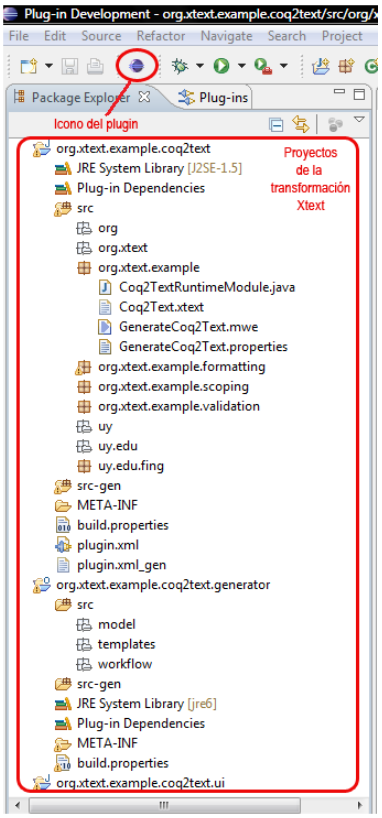

Figura B.4: Proyectos de la transformación Xtext

Para la correcta ejecución de esta transformación modelo-texto, se debe utilizar previamente un plugin de configuración. Este plugin fue desarrollado para evitar errores que produce la herramienta Xtext en ciertos casos particulares. Estos problemas son descritos en la sub-sección 5.3.2dedicada a la transformación Xtext.

En la Figura B.5 se presenta una configuración de ejemplo dentro de la ventana desplegada al presionar sobre el ícono del plugin  (Figura B.4) en la barra de herramientas del entorno. Los datos solicitados por la ventana de configuración son: la ruta absoluta al proyecto generador, un nombre para el modelo destino Coq, y la ruta al modelo destino de la transformación ATL (archivo .XMI). Para más información sobre el uso, instalación y funcionamiento del plugin, el lector podrá referirse a la sección B.3.1.

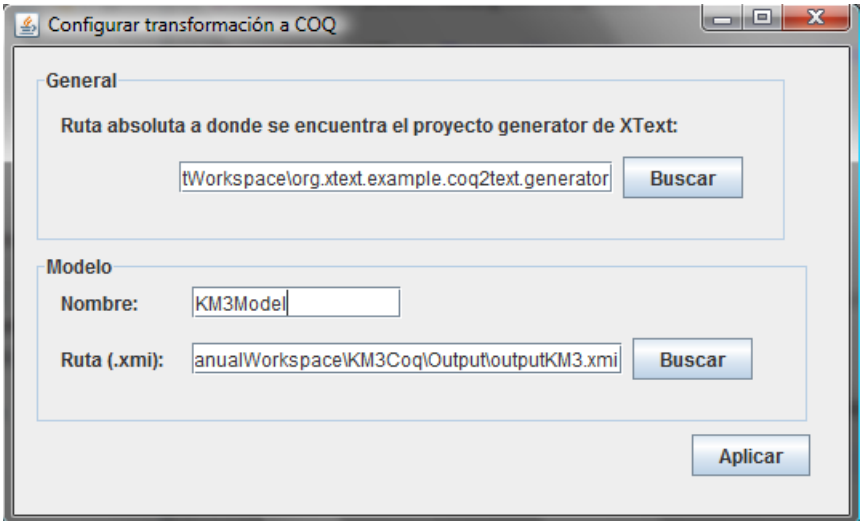

Figura B.5: Ejecución del Plugin de configuración

B.3.1. Plugin de configuración

Desarrollo

La figura B.6 muestra el proyecto de tipo plugin de Eclipse desarrollado bajo la versión 3.5.1 de dicho entorno. Su lógica principal se encuentra en la clase *PluginAction.java*. Para exportar un jar instalable del proyecto se deberá acceder al archivo *MANIFEST.MF* y utilizar el wizard de exportación.

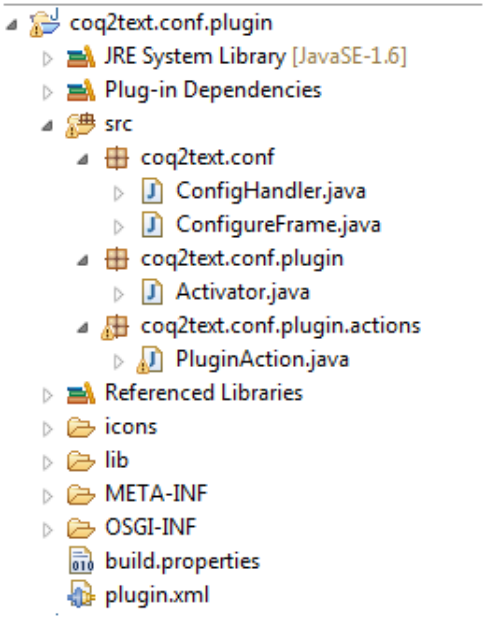

Figura B.6: Estructura de paquetes y clases del plugin

Instalación

Bajo la carpeta *dropins* de la versión 3.5.1 del entorno de desarrollo Eclipse deberá ubicarse el archivo jar anteriormente exportado. Al iniciar nuevamente Eclipse se apreciará un nuevo ícono , desde el cual será posible abrir la ventana de configuración.

Funcionamiento

El plugin permite especificar las propiedades necesarias para la ejecución de la transformación a texto: la ruta absoluta del proyecto generador, el nombre para el modelo Coq a generar y la ruta al modelo destino de la transformación ATL (archivo .XMI).

Al presionar el botón *"Aplicar"*, se realizan las siguientes acciones:

- Se asigna el valor proporcionado en el campo *Nombre* como nombre del archivo de salida de la tranformación Xtext .En el caso que no se ingrese un valor en dicho campo, se tomará el nombre del archivo XMI. Esta tarea es realizada modificando el template Model.xpt, ubicado en la carpeta templates dentro del proyecto generador.
- Se copia el contenido del archivo .xmi hacia la carpeta */src/model* del proyecto generador, y se almacena como *input.c2t*.
- Al contenido del archivo *input.c2t* se le realizan dos modificaciones:
  - Se eliminan las ocurrencias de las comillas dobles. Esto es debido a un bug encontrado en la herramienta, el problema es que la gramática no soporta en el texto que reconoce a estas comillas.
  - Se reemplazan las dos primeras líneas del archivo .xmi por el tag *<xmi>*. Esto se realiza para agregarle portabilidad al proyecto, ya que al momento de la ejecución del workflow que genera

el texto Coq, Xtext crea automáticamente clases con nombre igual a los tags del archivo que procesa. Estos nombres pueden ser demasiado largos y dar problemas en el sistema operativo Windows.

Apéndice C

Progreso del Proyecto

La Figura C.1 muestra el avance del proyecto para sus diferentes tareas y el conjunto de hitos asociados a dichas tareas.

El diagrama de Gantt de la Figura C.2 muestra de forma gráfica este avance.

Id	Nombre de tarea	Comienzo	Fin
1	Contexto Introductorio	mar 17/03/09	jue 04/06/09
2	Primera reunión con tutores y primeros documentos a leer	mar 17/03/09	mar 17/03/09
3	Versión final descripción proyecto	jue 19/03/09	jue 19/03/09
4	Generación de documentación sobre los documentos leidos	sáb 28/03/09	sáb 28/03/09
5	Presentamos resumen de primeros documentos	mié 08/04/09	mié 08/04/09
6	Búsqueda y análisis de docs. sobre verificación de transformaciones	mar 21/04/09	mar 21/04/09
7	Tutores envían documentación sobre verificación	vie 22/05/09	vie 22/05/09
8	Estado del Arte de Lenguajes de Transformación y Herramientas	dom 07/06/09	lun 09/11/09
9	Se envía Borrador Criterios Clasifiación Lenguajes	mié 10/06/09	mié 10/06/09
10	Relevamiento de Lenguajes de Transformación a analizar	mar 14/07/09	mar 14/07/09
11	Borrador documento de Lenguajes de Transformación y Herramientas	mar 15/09/09	mar 15/09/09
12	Devolución tutores - Lenguajes de Transformación y Herramientas	dom 20/09/09	dom 20/09/09
13	2do Borrador Lenguajes de Transformación y Herramientas	dom 01/11/09	dom 01/11/09
14	Versión final doc Lenguajes de Transformación y Herramientas	lun 09/11/09	lun 09/11/09
15	Estado del Arte de Verificación de Transformaciones	mié 16/09/09	mié 07/04/10
16	Borrador sobre Verificación de Transformaciones	mié 16/09/09	mié 16/09/09
17	Devolución tutores - Verificación de Transformaciones	dom 20/09/09	dom 20/09/09
18	2do borrador sobre Verificación de Transformaciones	mar 29/09/09	mar 29/09/09
19	Devolución Tutores Verificación + Introducción al caso de estudio.	mié 30/09/09	mié 30/09/09
20	Borrador doc Verificación de Transformaciones	dom 08/11/09	dom 08/11/09
21	Tutores envían más docs de verificacíon para analizar	vie 20/11/09	vie 20/11/09
22	Borrador doc Verificación de Transformaciones	lun 15/02/10	lun 15/02/10
23	Devolución tutores - Verificación de Transformaciones	mié 24/02/10	mié 24/02/10
24	Versión final doc Verificación de Transformaciones	mié 07/04/10	mié 07/04/10
25	Caso de Estudio	sáb 28/11/09	jue 22/04/10
26	Tutores envían ejemplo de especificación de Coq	lun 30/11/09	lun 30/11/09
27	Problemas técnicos KM3 y Eclipse	sáb 05/12/09	sáb 05/12/09
28	Solución Eclipse con AM3	sáb 26/12/09	sáb 26/12/09
29	Comenzamos implementación del proyecto ATL	jue 28/01/10	jue 28/01/10
30	Se opta por Xtext en lugar de TCS	vie 12/02/10	vie 12/02/10
31	Reunión donde presentamos avances en ATL y Xtext	dom 28/03/10	dom 28/03/10
32	Enviamos versión definitiva del proyecto ATL y Xtext	lun 19/04/10	lun 19/04/10
33	Tutores proponen implementar como caso de estudio KM3	mié 21/04/10	mié 21/04/10
34	Informe final	lun 03/05/10	dom 27/06/10

Figura C.1: Avance del proyecto - Tareas

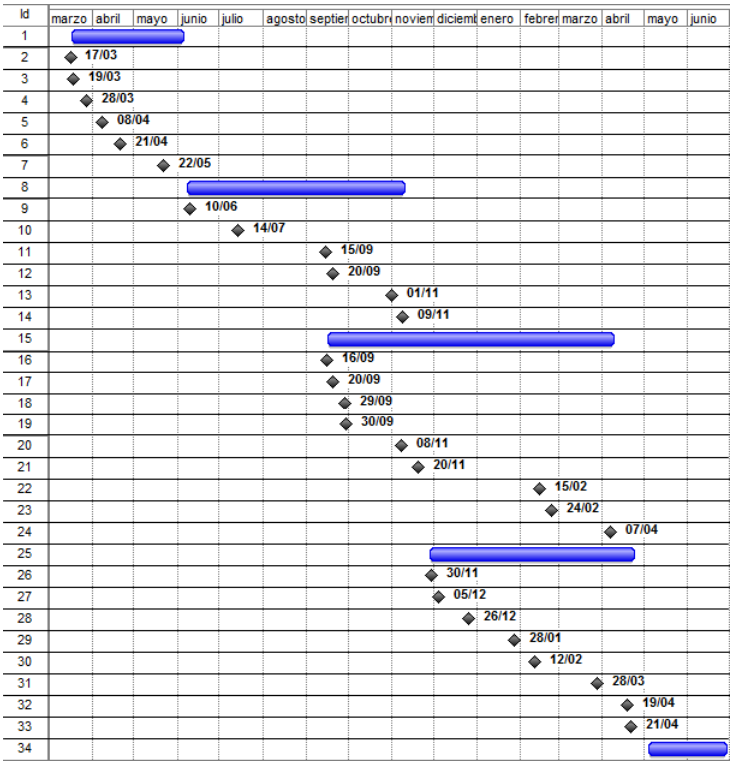

Figura C.2: Avance del Proyecto - Diagrama de Gantt

Apéndice D

Código Fuente

D.1. Transformación ATL

```
module KM32Coq; -- Module Template
create coq : COQ from km3 : KM3;
--helper context KM3!Class def :
-- ancestors : Collection(KM3!Class) =
-- self.supertypes->union(self.supertypes->collect(c | c.ancestors))->flatten();
helper context KM3!Class def :

     subclasses : Collection(KM3!Class) =

         KM3!Class.allInstances()->select(c | c.supertypes->exists(x | x = self));

helper context KM3!Class def : getBasicType(name: String) : KM3!DataType =

     KM3!DataType.allInstances()->any(c | c.name = name);

rule Class2InductiveType {

     from
         i: KM3!Class (
             not i.isAbstract
         )
     to
         o: COQ!InductiveType (
             isAbstract <- i.isAbstract,
             name <- i.name,
             hasComponents <- Set{thisModule.BuildOidComponent(i)}->union(i.structuralFeatures),
             subTypes <- i.subclasses,
             superTypes <- i.supertypes
         )

}
rule AbstractClass2InductiveType {

     from
         i: KM3!Class (
             i.isAbstract
         )
     to
         o: COQ!InductiveType (
```

```
            isAbstract <- i.isAbstract,
            name <- i.name,
            hasComponents <- i.structuralFeatures,
            subTypes <- i.subclasses,
            superTypes <- i.supertypes
        )
}
lazy rule BuildOidComponent {
    from
        i : KM3!Class
    to
        o : COQ!Component (
            name <- 'oid',
            hasType <- i.getBasicType('Integer')
        )
}
rule Feature2Component_Basic {
    from
        i : KM3!StructuralFeature (
            i.lower = 1 and i.upper = 1
        )
    to
        o : COQ!Component(
            name <- i.name,
            hasType <- i.type
        )
}
lazy rule Type2Option {
    from
        i:KM3!StructuralFeature
    to
        option : COQ!Option (
            name <- i.name,
            hasType <- i.type
        )
}
lazy rule Type2List {
    from
        i:KM3!StructuralFeature
    to
        o : COQ!List (
            name <- i.name,
            hasType <- i.type
        )
    }
rule Feature2Component_Option {
    from
        i : KM3!StructuralFeature (
```

```
            i.lower = 0 and i.upper = 1
        )
    to
        o : COQ!Component(
            name <- i.name,
            hasType <- thisModule.Type2Option(i)
        )
}
rule Feature2Component_List {
    from
        i : KM3!StructuralFeature (
            i.upper < 0 or i.upper > 1
        )
    to
        o : COQ!Component(
            name <- i.name,
            hasType <- thisModule.Type2List(i)
        )
}
rule Datatype2BasicType {
    from
        i : KM3!DataType
    using {
        typeName : String =
            if (i.name = 'Integer') then
              'nat'
            else if (i.name = 'String') then
              'string'
              else
                'bool'
              endif
            endif;
        }
    to
        basicType : COQ!BasicType(
            name <- typeName
        )
}
rule Enumeration {
    from
        i : KM3!Enumeration
    to
        out : COQ!Enumeration(
            name <- i.name,
            values <- i.literals->collect(literal | literal.name )
        )
}
```

D.2. Transformación Xtext

Coq2Text.xtext

```
grammar org.xtext.example.Coq2Text with org.eclipse.xtext.common.Terminals
generate coq2Text "http://www.xtext.org/example/Coq2Text"
Model :

     '<xmi>'
         (definitions+=Definition)*
     '</xmi>'

;
Definition:

     Enumeration | InductiveType | Component | BasicType | Option | List

;
Enumeration:

     '<coq:Enumeration xmi:id=' name=ID 'name=' typeName=ID '>'
         (enumValues+=EnumValue)*
     '</coq:Enumeration>'

;
EnumValue:

     '<values>' value=ID '</values>'

;
InductiveType:


     '<coq:InductiveType xmi:id=' name=ID 'name=' typeName=ID ('isAbstract='isAbstract='true')?
     'hasComponents=' (components+=[Component])+ ('subTypes=' (subtypes+=[InductiveType])+)?
     ('superTypes=' (supertypes+=[InductiveType])+)? '/>'

;
Component:

     '<coq:Component xmi:id=' name=ID 'name=' compName=ID 'hasType=' hasType=[Type] '/>'

;
Type:

     InductiveType | BasicType | Enumeration | Option | List

;
BasicType:

     '<coq:BasicType xmi:id=' name=ID 'name=' typeName=ID '/>'

;
Option:

     '<coq:Option xmi:id=' name=ID 'name=' typeName=ID 'hasType=' hasType=[Type] '/>'

;
List:

     '<coq:List xmi:id=' name=ID 'name=' typeName=ID 'hasType=' hasType=[Type] '/>'

;
```

Main.xpt

```
«IMPORT coq2Text»
«DEFINE main FOR Model »

    «FILE "salida.v"»

Require Import String.
Require Import List.
Require Import Arith.

    «EXPAND Enumeration::build FOREACH this.definitions.typeSelect(Enumeration)»

        «EXPAND InductiveType::build FOR this »
        «EXPAND Projections::build FOREACH this.definitions.typeSelect(InductiveType)»
        «EXPAND Model::build FOR this »
        «EXPAND TypeOperators::build FOREACH this.definitions.typeSelect(InductiveType)»
        «EXPAND Projections::abstractoid FOREACH this.definitions.typeSelect(InductiveType)»
        «EXPAND TypeOperators::beq FOREACH this.definitions.typeSelect(InductiveType)»
        (*Para la siguientes definiciones, se considera herencia simple,
        solo al padre directo y los atributos del mismo*)
        «EXPAND Ancestors::build FOREACH this.definitions.typeSelect(InductiveType)»

    «ENDFILE»

«ENDDEFINE»
```

Enumeration.xpt

```
«IMPORT coq2Text»
«DEFINE build FOR Enumeration»
Inductive «this.typeName» : Set := «FOREACH this.enumValues AS e»

    | «e.value» : «this.typeName»«ENDFOREACH ».«ENDDEFINE»
```

InductiveType.xpt

```
«IMPORT coq2Text»
«EXTENSION templates::Extensions»
«DEFINE build FOR Model »
«FOREACH this.definitions.typeSelect(InductiveType) AS t ITERATOR iter »
«IF iter.firstIteration »
Inductive « t.typeName » : Set := «ELSE » with

    « t.typeName » : Set := «ENDIF »

        | Build_«t.typeName»
        «EXPAND component FOREACH t.components »
        «EXPAND subComponent FOREACH t.subtypes »«ENDFOREACH ».« ENDDEFINE »

«DEFINE component FOR Component» («this.compName» : « getTypeName(this.hasType) »)« ENDDEFINE »
«DEFINE subComponent FOR InductiveType » (sub« this.typeName » : option « this.typeName »)« ENDDEFINE »
```

Projections.xpt

```
«IMPORT coq2Text»
«EXTENSION templates::Extensions»
«DEFINE build FOR InductiveType »«FOREACH this.components AS comp ITERATOR iter »
Definition « this.typeName »_« comp.compName » (o : « this.typeName ») :
« getTypeName(comp.hasType) » :=

    match o with

        | (Build_« this.typeName » « paramPrinter(comp.name, iter.counter1,
        iter.elements + this.subtypes.size) ») => « comp.name »

    end.
```

```
«ENDFOREACH »
«FOREACH this.subtypes AS t ITERATOR iter »
Definition « this.typeName »_sub« t.typeName » (o : « this.typeName ») : option « t.typeName » :=

     match o with

          | (Build_« this.typeName » « paramPrinter(t.name, this.components.size + iter.counter1,
          this.components.size + iter.elements) ») => « t.name »

     end.

«ENDFOREACH »
« ENDDEFINE »
«DEFINE abstractoid FOR InductiveType »«IF this.isAbstract == "true" »
« abstractOidPrinter(this.typeName, this.subtypes) » «ENDIF »« ENDDEFINE »
```

Model.xpt

```
«IMPORT coq2Text»
«DEFINE build FOR Model »
(*----------------------------------------------------------------*)
(* ---------- Model Definition ---------- *)
« LET "COQModelName" AS modelName »
Record

     « modelName » : Set :=  « modelName » {
     «FOREACH this.definitions.typeSelect(InductiveType) AS t ITERATOR iter »

          MCoq_« t.typeName.toFirstLower() »AllInstances : list « t.typeName »
          «IF !iter.lastIteration »;« ENDIF »«ENDFOREACH »
          }.« ENDLET »

«ENDDEFINE»
```

TypeOperators.xpt

```
«IMPORT coq2Text»
«DEFINE build FOR InductiveType »

     «FOREACH this.subtypes AS t ITERATOR iter »

Definition isTypeOf« t.typeName » (« this.name » : « this.typeName ») : bool :=

     match (« this.typeName »_sub« t.typeName » « this.name ») with
          | None => false
          | Some a => true

     end.

«ENDFOREACH »«FOREACH this.subtypes AS t ITERATOR iter »
Definition asType« t.typeName » (« this.name » : « this.typeName ») : option « t.typeName » :=
« this.typeName »_sub« t.typeName » « this.name ».
«ENDFOREACH »« ENDDEFINE »
«DEFINE beq FOR InductiveType »


     Definition beq_« this.typeName » (o1 : « this.typeName ») (o2 : « this.typeName ») : bool :=
          beq_nat (« this.typeName »_oid o1) (« this.typeName »_oid o2).

« ENDDEFINE »
```

Ancestors.xpt

```
«IMPORT coq2Text»
«EXTENSION templates::Extensions»
«DEFINE build FOR InductiveType »
« IF !this.supertypes.isEmpty »« LET this.supertypes.get(0) AS super »
Fixpoint « this.typeName »_super (l : list « super.typeName ») (« this.name » : « this.typeName »)
: option « super.typeName » :=

     match l with
          | nil => None
          | cons m l2 => match (« super.typeName »_sub« this.typeName » m) with
              | None => « this.typeName »_super l2 « this.name »
              | Some x => match (beq_« this.typeName » « this.name » x) with
                | true => Some m
                | false => « this.typeName »_super l2 « this.name »
                end
              end
          end.
     «FOREACH getSuperComponents(super) AS t ITERATOR iter »

Definition « this.typeName »_« t.compName » (l : list « super.typeName »)
(« this.name » : « this.typeName ») : option « getSuperTypeName(t.hasType) » :=

     match (« this.typeName »_super l « this.name ») with
          | None => None
          | Some x => « IF !isInstanceOfOption(t.hasType) »Some « ENDIF »
          (« super.typeName »_« t.compName » x)
     end.
     «ENDFOREACH » « ENDLET » « ENDIF »

« ENDDEFINE »
```

Extensions.ext

```
import coq2Text;
String paramPrinter(String componentName, Integer index, Integer size) : JAVA


     uy.edu.fing.Util.paramPrinter(java.lang.String, java.lang.Integer, java.lang.Integer);

String abstractOidPrinter(String typeName, List[InductiveType] subtypes) : JAVA

     uy.edu.fing.Util.abstractOidPrinter(java.lang.String, java.util.List);

Boolean isInstanceOfOption(Type t) : JAVA

     uy.edu.fing.Util.isInstanceOfOption(org.xtext.example.coq2Text.Type);

String getTypeName(Type t) : JAVA

     uy.edu.fing.Util.getTypeName(org.xtext.example.coq2Text.Type);

String getSuperTypeName(Type t) : JAVA

     uy.edu.fing.Util.getSuperTypeName(org.xtext.example.coq2Text.Type);

Collection[Component] getSuperComponents(InductiveType super) :

     super.components.select(e| e.compName != 'oid');
```

Util.java

```
package uy.edu.fing;
import java.util.List;
import org.xtext.example.coq2Text.InductiveType;
import org.xtext.example.coq2Text.Type;
public class Util {

    public static boolean isInstanceOfList(Type t) {
        return t instanceof org.xtext.example.coq2Text.List;
    }
    public static boolean isInstanceOfOption(Type t) {
        return t instanceof org.xtext.example.coq2Text.Option;
    }
    public static boolean isInstanceOfBasicType(Type t) {
        return t instanceof org.xtext.example.coq2Text.BasicType;
    }
    public static boolean isInstanceOfEnumeration(Type t) {
        return t instanceof org.xtext.example.coq2Text.Enumeration;
    }
    public static boolean isInstanceOfInductiveType(Type t) {
        return t instanceof org.xtext.example.coq2Text.InductiveType;
    }
    public static String getTypeName(Type t) {
        return getTypeName(t, false);
    }
    public static String getSuperTypeName(Type t) {
        return getTypeName(t, true);
    }
    public static String paramPrinter(String componentName, Integer index,
        Integer size) {
        String str = "";
        index -= 1;
        for (int i = 0; i < index; i++)
            str += "_ ";
        str += componentName;
        for (int i = index + 1; i < size.intValue(); i++)
            str += " _";
        index++;
        return str;
    }
    public static String abstractOidPrinter(String typeName,
        List<InductiveType> subtypes) {
            int size = subtypes.size();
            if (size > 0) {

              String def = "Definition " + typeName + "_oid (i : " + typeName
                + ") : nat  := \n";
              return def + abstractOidPrinter(subtypes, size) + ".";
            }
            return "";
    }
    private static String getTypeName(Type t, boolean issuper) {
        if (isInstanceOfList(t)) {
```

```
        String pre = "", suf = "";
        if (issuper) {
          pre = "(list ";
          suf = ")";
        } else {
          pre = "list ";
        }
        return pre
          + getTypeName(((org.xtext.example.coq2Text.List) t)
            .getHasType()) + suf;
    } else if (isInstanceOfOption(t)) {
        String pre = "";
        if (!issuper) {
          pre = "option ";
        }
        return pre
          + getTypeName(((org.xtext.example.coq2Text.Option) t)
            .getHasType());
    } else {
        if (isInstanceOfBasicType(t)) {

          return ((org.xtext.example.coq2Text.BasicType) t).getTypeName();
        } else if (isInstanceOfEnumeration(t)) {
          return ((org.xtext.example.coq2Text.Enumeration) t)
            .getTypeName();
        } else {
          return ((org.xtext.example.coq2Text.InductiveType) t)
            .getTypeName();
        }
    }
}
private static String abstractOidPrinter(List<InductiveType> subtypes, int subtypesSize) {
    int size = subtypes.size();
    InductiveType subtype = subtypes.get(0);
    subtypes.remove(0);
    String retorno = "";
    if (subtypesSize-size==0){
        retorno += "\t";
    }
    retorno += "match (asType" + subtype.getTypeName() + " i) with\n";
    if (subtypesSize-size==0){
        retorno += "\t\t";
    } else {
        for(int i=0; i<(subtypesSize-size+1)*2+1; i++)
          retorno += "\t";
    }
    retorno += "| None => ";
    retorno += ((size == 1) ? "0" : abstractOidPrinter(subtypes,subtypesSize)) + "\n";
    if (subtypesSize-size==0){
        retorno += "\t\t";
    } else {
        for(int i=0; i<(subtypesSize-size+1)*2+1; i++)
```

```
                retorno += "\t";
            }
            retorno += "| Some " + subtype.getName() + " => ";
            retorno +=  subtype.getTypeName() + "_oid " + subtype.getName() + "\n";
            if (subtypesSize-size==0){
                retorno += "\t";
            } else {
                for(int i=0; i<(subtypesSize-size+1)*2+1; i++)
                  retorno += "\t";
            }
            retorno += "end";
            return retorno;
        }

}
```

Bibliografía

[1] ATL Project. http://www.eclipse.org/m2m/atl/. Último acceso, Abril 2010.

[2] Tefkat: The EMF Transformation Engine. http://tefkat.sourceforge.net/. Último acceso, Abril 2010.

[3] The Coq Proof Assistant. http://coq.inria.fr/. Último acceso, Abril 2010.

[4] Object Management Group: OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP. http://www.omg.org/spec/QVT/1.0/PDF/, Octubre 2002. Último acceso, Abril 2010.

[5] F. Allilaire, J. Bézivin, H. Brunelière, and F. Jouault. Global model management in Eclipse GMT/AM3. In *Eclipse Technology eXchange workshop in conjunction with ECOOP*, volume 6, 2006.

[6] K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In Benoit Baudry, Alain Faivre, Sudipto Ghosh, and Alexander Pretschner, editors, *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[7] M. Asztalos, L. Lengyel, and T. Levendovszky. A formalism for describing modeling transformations for verification. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10. ACM, 2009.

[8] P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context - Motorola case study. *Model Driven Engineering Languages and Systems*, pages 476–491, 2005.

[9] B. Baudry, S. Ghosh, F. Fleurey, R. France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 2009.

[10] B. Beckert, U. Keller, and P.H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings of the Second Verification Workshop: VERIFY*, volume 2, pages 02–07. Citeseer, 2002.

[11] J. Bezivin, B. Rumpe, A. Schurr, and L. Tratt. Model transformations in practice workshop. *Lecture Notes in Computer Science*, 3844:120, 2006.

[12] A. Boronat, R. Heckel, and J. Meseguer. Rewriting logic semantics and verification of model transformations. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 18–33. Springer-Verlag, 2009.

[13] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. pages 85–94, 2006.

[14] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, August 2009.

[15] D. Calegari, C. Luna, N. Szasz, and A. Tasistro. Experiment with a Type-Theoretic Approach to the Verification of Model Transformations.

[16] E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL contracts for the verification of model transformations.

[17] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, et al. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1986. Último acceso, Abril 2010.

[18] J.S. Cuadrado, J.J.G. Molina, and M.M. Tortosa. RubyTL: un Lenguaje de Transformación de Modelos Extensible.

[19] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[20] A. Darabos, A. Pataricza, and D. Varró. Towards testing the implementation of graph transformations. *Electronic Notes in Theoretical Computer Science*, 211:75–85, 2008.

[21] J. De Lara and H. Vangheluwe. Using AToM as a Meta-CASE Tool. pages 642–649, 2002.

[22] Universidad de Latvia. The MOLA Language Reference Manual. http://mola.mii.lu.lv/mola2fin_refmanual.pdf. Último acceso, Abril 2010.

[23] Eclipse. Xtext User Guide. http://www.eclipse.org/Xtext/documentation/1_0_0/xtext.html. Último acceso, Julio 2010.

[24] G. Engels, J. Küster, R. Heckel, and M. Lohmann. Model-based verification and validation of properties. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[25] F. Fleurey, Z. Drey, D. Vojtisek, C. Faucher, and V. Mahé. Kermeta Language, Reference Manual. *Internet: http://www. kermeta. org/docs/KerMeta-Manual. pdf. IRISA*, 2006.

[26] F. Fleuri, B. Baudry, P. Muller, and Y. Le Traon. Towards dependable model transformations: Qualifying input test data. *Journal of Software and Systems Modeling (SoSyM)*, 2007.

[27] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. 2007.

[28] H. Giese, S. Glesner, J. Leitner, W. Schafer, and R. Wagner. Towards verified model transformations. *MoDeV²a: Model Development, Validation and Verification.*, page 78.

[29] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, 2006.

[30] R. Heckel and S. Thöne. Behavioral refinement of graph transformation-based models. *Electr. Notes Theor. Comput. Sci.*, 127(3):101–111, 2005.

[31] F. Jouault and I. Kurtev. Transforming models with ATL. *Lecture Notes in Computer Science*, 3844:128, 2006.

[32] S. Kandl, R. Kirner, and G. Fraser. Verification of platform-independent and platform-specific semantics of dependable embedded systems. Oct. 2006.

[33] S. Kent. Model driven engineering. *Lecture notes in computer science*, pages 286–298, 2002.

[34] A.G. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[35] J.M. Kuster and M. Abd-El-Razik. Validation of model transformations-first experiences using a white box approach. *Lecture Notes in Computer Science*, 4364:193, 2007.

[36] T. Lacey and S. DeLoach. Verification of agent behavioral models, 2000.

[37] K. Lano. Using b to verify uml transformations. *MODEVA workshop at MoDELS 2006*, 2006.

[38] Yuehua Lin, Jing Zhang, and Jeff Gray. A testing framework for model transformations. pages 219–236, 2005.

[39] H. López, F. Varesi, M. Viñolo, D. Calegari, and C. Luna. Estado del Arte de Lenguajes y Herramientas de Transformación de Modelos. Technical Report RT09-19, InCo-PEDECIBA, Uruguay, 2009. Último acceso, Abril 2010.

[40] H. López, F. Varesi, M. Viñolo, D. Calegari, and C. Luna. Estado del Arte de Verificación de Transformación de Modelos. Technical Report RT10-07, InCo-PEDECIBA, Uruguay, 2010. Último acceso, Abril 2010.

[41] ATLAS Group. *KM3: Kernel MetaMetaModel*. LINA & INRIA, manual v0.3 edition, 2005.

[42] T. Mens, K. Czarnecki, and P. Van Gorp. 04101 discussion – a taxonomy of model transformations. (04101), 2005.

[43] OMG MOF. 2.0 Query/View/Transformation Specification (2007).

[44] J. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. *Lecture Notes in Computer Science*, 4066:376, 2006.

[45] J.M. Mottu, B. Baudry, and Y. Le Traon. Reusable MDA Components: A Testing-for-Trust Approach. *Lecture Notes in Computer Science*, 4199:589, 2006.

[46] A. Narayanan and G. Karsai. Specifying the correctness properties of model transformations. In *GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations*, pages 45–52, New York, NY, USA, 2008. ACM.

[47] The University of Birmingham. Simple Transformer (SiTra). http://www.cs.bham.ac.uk/~bxb/SiTra.html. Último acceso, Abril 2010.

[48] F. Orejas and M. Wirsing. On the specification and verification of model transformations. pages 140–161. 2009.

[49] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[50] I. Poernomo. Proofs-as-Model Transformations. In *ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, pages 214–228, Berlin, Heidelberg, 2008. Springer-Verlag.

[51] I.H. Poernomo, J.N. Crossley, and M. Wirsing. *Adapting proofs-as-programs: the Curry-Howard protocol*. Springer-Verlag New York Inc, 2005.

[52] C. Pons and D. Garcia. Proving the Correctness of Refinement in Model Driven Software Engineerings. *Electronic Notes in Theoretical Computer Science*, 2008.

[53] I. Raedts, M. Petković, A. Serebrenik, J.M van der Werf, L. Somers, and M. Boote. A software framework for automated verification. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1031–1032, New York, NY, USA, 2007. ACM.

[54] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[55] A. Schiirri. Specification of graph translators with triple graph grammars. In *Graph-theoretic concepts in computer science: 20th international workshop, WG'94, Herrsching, Germany, June 16-18, 1994: proceedings*, page 151. Springer Verlag, 1995.

[56] Douglas Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[57] S. Sen, B. Baudry, and J.M. Mottu. Automatic model generation strategies for model transformation testing. *International Conference on Model Transformation, ICMT'09*, 2009.

[58] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. *Applications of Graph Transformations with Industrial Relevance*, pages 446–453, 2004.

[59] D. Varró and A. Pataricza. Automated formal verification of model transformations. (TUM-I0323):63–78, September 2003.

[60] VIATRA. VIATRA2 Home Page. http://www.eclipse.org/gmt/VIATRA2/. Último acceso, Abril 2010.

[61] T.U. Wien. Language Jungle-An Evaluation and Extension of Existing Approaches. 2008.

Índice alfabético