Universidad de Murcia.
Facultad de Informática.

Proyecto Fin de Carrera.

Proceso de Modernización de los Datos de un Sistema de Información aplicando técnicas DSDM.

Francisco Moya Arnao.

Proyecto dirigido por:
Francisco Javier Bermúdez Ruiz.

Departamento de Informática y Sistemas.
Universidad de Murcia.

Febrero 2010.

Agradecimientos.

A Don Francisco Javier Bermúdez Ruiz profesor del Departamento de Informática y Sistemas de la Universidad de Murcia y director de este proyecto por ayudarme a sacarlo adelante en tan poco tiempo. Agradecer a Javier Luis Cánovas Izquierdo y a Jesús Sánchez Cuadrado investigadores del Grupo Modelum del Departamento de Informática y Sistemas de la Universidad de Murcia por su apoyo, compresión y paciencia ante todas las dudas que les he planteado.

A la empresa SAES de Cartagena en la que me encuentro realizando prácticas por darme el tiempo necesario para poder finalizar este proyecto.

A toda mi familia por su apoyo, a María por aguantarme y a todos aquellos que están a mi lado y no se dan por aludidos sino queda reflejado.

El guardián me clavó una mirada desprovista de emoción.
Porque así está establecido –dijo-. Porque es así. De la misma manera
que el sol sale por el este y se pone por el oeste.

Haruki Murakami
"El fin del Mundo y un despiadado País de las Maravillas"

Este trabajo ha sido parcialmente financiado por la Ayuda 08797/PI/08 de la Agencia Regional de Ciencia y Tecnología (Fundación Seneca) de la Región de Murcia.

Índice.

Índice de Figuras.

1 Introducción

La evolución o modernización de software es el área de la ingeniería del software destinada a proporcionar los principios, métodos, técnicas y herramientas necesarias para abordar los posibles escenarios en los que un sistema software existente debe ser transformado en un nuevo sistema para satisfacer nuevas demandas. No cabe duda de que se trata de un área estratégica para las organizaciones y empresas en cuanto les permite adaptar sus sistemas a las nuevas necesidades. En los últimos años, ha surgido la "Modernización basada en modelos" como resultado de aplicar en la evolución del software las técnicas propias del "Desarrollo de Software Dirigido por Modelos", aplicadas con éxito en la creación de nuevos sistemas.

El trabajo de este proyecto se enmarca en el ámbito de la modernización basada en modelos en el escenario del análisis y mejora de la organización de las bases de datos. Se han publicado muy pocos trabajos de investigación sobre modernización basada en modelos y este trabajo es pionero en la aplicación de las técnicas de metamodelado y transformaciones de modelos en un escenario relacionado con las bases de datos relacionales de una aplicación. Se ha ideado un enfoque basado en cuatro etapas: extracción de modelos a partir de la información de una base de datos relacional, generación de modelos de errores de la base de datos actual, corrección de la base de datos y regeneración en tecnologías de persistencia.

1.1 Contexto

En la actualidad las soluciones existentes para ese tipo de mejoras y correcciones son escasas y consiguen una automatización limitada de la transformación de la base de datos. Este trabajo muestra cómo es posible aplicar las técnicas del Desarrollo Dirigido por Modelos en la modernización de bases de datos para conseguir elevar el nivel de automatización del proceso. Para ello, es necesario extraer modelos conformes a un metamodelo a partir de los esquemas textuales expresados en SQL y aplicar transformaciones de modelos. Se ha usado el framework EMF de Eclipse para aplicar desarrollo basado en modelos, de modo que los metamodelos necesarios se han definido en el lenguaje de metamodelado Ecore. Para la extracción de modelos se ha usado el lenguaje Gra2MoL, un DSL (Domain Specific Language) para la extracción de modelos que permite expresar la correspondencia entre los elementos de una gramática y el metamodelo al que debe conformar el modelo obtenido. Para la generación de modelos de errores, corrección de la base de datos y regeneración de la información para tecnologías de persistencia se ha usado RubyTL, un DSL embebido dentro del lenguaje Ruby para transformaciones modelo a modelo y modelo a código.

El Desarrollo de Software Dirigido por Modelos (DSDM) emergió a principios de esta década con el propósito de suponer el paso definitivo hacia la industrialización del software, o al menos proporcionar mejoras significativas en la productividad y calidad. En realidad, el término DSDM se refiere a un conjunto de diferentes paradigmas de construcción de software

que comparten algunos principios básicos, y que surgen, a partir de la iniciativa MDA [MDA] del OMG presentada en noviembre de 2000. Entre ellas podemos destacar: la propia MDA, las factorías de software y el desarrollo específico del dominio.

La principal idea compartida por todos los paradigmas englobados dentro del DSDM es la utilización de lenguajes de modelado o lenguajes específicos del dominio (DSL) para crear modelos que representen diferentes aspectos del sistema a construir, y a partir de los cuales se puede generar código de la aplicación final. Un DSL proporciona conceptos más cercanos al dominio de la aplicación, de modo que tiene un nivel de abstracción más alto que un lenguaje de programación convencional. Mientras que un lenguaje de programación nos permite crear programas, con un DSL creamos modelos.

La teoría subyacente al DSDM se denomina metamodelado y en [Clark, 2004] puede encontrarse una excelente descripción de los diferentes aspectos que abarca: lenguajes de metamodelado, sintaxis abstracta, sintaxis concreta, semántica y transformaciones. Un lenguaje de metamodelado es un lenguaje para describir DSL. Los lenguajes de metamodelo más extendidos son MOF de OMG y Ecore para Eclipse. Una definición de un lenguaje a través de un lenguaje de metamodelado se denomina metamodelo. Cuando un DSL ***L*** es descrito por un metamodelo ***M,*** se dice que un modelo ***m*** expresado en ***L*** conforma con ***M***.

La utilización de un DSL requiere, lógicamente, la existencia de una herramienta de transformación que transforme el modelo expresado con un DSL en código fuente de un lenguaje de programación o en otro modelo expresado en un DSL intermedio como paso previo a la generación de código. Realmente, el código destino puede ser texto que corresponda a cualquier artefacto del sistema ejecutable final (código fuente, archivos de configuración, metadatos, SQL, etc.). En los últimos años ha surgido un área muy activa relacionada con las transformaciones modelo a modelo y se han propuesto diferentes lenguajes entre los que destaca ATL [Jouault, 2005], RubyTL [Sánchez, 2006] y QVT [QVT] el lenguaje propuesto por OMG. También han surgido varios lenguajes de plantillas para las transformaciones modelo a texto como [MOF2Text, MOFScript] o el propio RubyTL

En la actualidad los sistemas software son esenciales para el funcionamiento de la mayoría de organizaciones y empresas de todo el mundo, y es crucial la evolución o modernización de estos sistemas de acuerdo a las necesidades cambiantes de las organizaciones y de su entorno, de modo que no pierdan su valor estratégico. Situaciones comunes que provocan la evolución de un sistema existente o *legacy.* Son cambios en los requisitos funcionales, una mejora o reestructuración del sistema, un cambio de alguna tecnología de implementación, la migración a otra plataforma o la integración con otro sistema.

Como se señala en [Comella-Dorda, 2000] conviene distinguir entre mantenimiento, modernización y reemplazo del sistema. Mientras el mantenimiento es un proceso iterativo e incremental en el que se realizan pequeños cambios para quitar errores o pequeñas mejoras en la funcionalidad que no afectan a la estructura, una modernización supone cambios de más envergadura pero se conserva parte del sistema existente. La modernización se clasifica en dos categorías según sea necesario o no tener un conocimiento de aspectos internos del sistema:

modernización caja-negra vs. modernización caja-blanca. Cuando una modernización es muy costosa, las empresas deben optar por el reemplazo del sistema software existente.

Las técnicas del DSDM, el metamodelado y las transformaciones de modelos, no sólo son aplicables para la construcción de nuevas aplicaciones (ingeniería directa) sino que pueden ser aplicadas también para la evolución o modernización de aplicaciones existentes (ingeniería inversa). Mientras que en el caso de ingeniería directa, los desarrolladores crean modelos del sistema que son conformes a ciertos metamodelos y a partir de los cuales se crea total o parcialmente la aplicación final, en la ingeniería inversa se extraen modelos del código fuente (u otros artefactos del sistema software existente) y entonces a partir de ellos, o bien se modifica el sistema o se crea un nuevo sistema. Así, conforme el DSDM va gozando de mayor aceptación, ha comenzado a emerger una nueva disciplina conocida como "*Modernización de Software Dirigida por Modelos*" (MDM) cuyo objetivo es la automatización de las tareas de evolución del software mediante la aplicación de técnicas DSDM. En la actualidad hay en marcha proyectos en todo el mundo relacionados con esta nueva área de la ingeniería del software que persiguen definir los principios, procesos, técnicas, estándares y herramientas necesarias.

Tras la propuesta de MDA, en 2003 OMG lanzó la iniciativa ADM (Architecture Driven Modernization) de OMG [ADM] destinada a establecer un conjunto de especificaciones estándar (metamodelos) para favorecer la interoperabilidad entre herramientas de modernización. En estos momentos están disponibles los dos metamodelos sobre los que se construirán el resto: ASTM destinado a representar la sintaxis abstracta de cualquier lenguaje de programación y KDM destinado a representar conocimiento sobre ciertos aspectos del software como las interfaces de usuario, los datos y las acciones. Los restantes metamodelos están relacionados con aspectos como métricas, testing y análisis de código fuente.

Según [ADM06], podemos encontrar diversas situaciones a la hora de realizar una modernización de un sistema de software. Estas situaciones son denominadas escenarios, y definen las tareas requeridas en cada uno de ellos y las herramientas necesarias para realizar dichas tareas. Uno de estos escenarios es el de mejora de una aplicación cambiando la estructura de sus datos. Son diversas las razones que pueden llevar a esta decisión como la obsolescencia de las de bases de datos, su calidad de no relacionales o por necesidad de migración a otras bases de datos o motores de persistencia. También podemos encontrarnos en el caso de bases de datos no relacionales que presentan problemas de consistencia de datos, accesibilidad, redundancia y problemas de Integridad. La existencia de ficheros planos, jerárquicos o estructuras de red no están accesibles fácilmente a tecnologías nuevas o distribuidas.

Un problema real al que se enfrentan las empresas es el de trabajar con bases de datos creadas hace muchos años y modificadas varias veces, las cuales no presentan un nivel de normalización y de consistencia adecuado. La forma clásica de afrontar dicho problema consiste en el diseño de un nuevo esquema de base de datos partiendo de cero y su posterior migración de datos. Sin embargo en este proyecto se presenta una nueva forma de abordar el problema haciendo uso de MDM para automatizar el proceso de transformación de la base de

datos, abandonando las prácticas artesanales actuales para reducir costos y tiempo. La modernización de bases de datos relacionales basada en modelos puede aportar otros beneficios como la independencia y consolidación de los datos existentes, o la abstracción del sistema de base de datos utilizado.

Por tanto, este proyecto se enmarca dentro de un **escenario de mejora de una base de datos relacional.** El enfoque MDM propuesto consta de las siguientes fases: ***(i)*** la extracción de modelos a partir de script de bases de datos expresados en DDL y DML, ***(ii)*** la obtención de modelos de errores que pueda contener una base de datos, utilizando para ello un algoritmo de detección de errores, ***(iii)*** la modificación de la base de datos original para eliminar las deficiencias mediante la transformación modelo a modelo (M2M) para la obtención de un nuevo modelo de base de datos corregido y la transformación modelo a código (M2C) para la obtención de un código fuente que represente la nueva base de datos *(es importante remarcar que la herramienta implementada como resultado de la aplicación del proceso de modernización de este proyecto, también incorpora una fase para la asistencia en la elección de los errores a corregir en una base de datos)*

En cuanto a la tecnología empleada en este proyecto, los metamodelos necesarios se han definido en el lenguaje de metamodelado Ecore de EMF/Eclipse[Eclipse] y para la extracción de modelos se ha usado el lenguaje Gra2MoL [Gra2Mol], un DSL específico para esta tarea que permite expresar la correspondencia entre los elementos de una gramática, en este caso la gramática del DDL y DML de SQL, y el metamodelo al que debe conformar el modelo obtenido, en este caso los modelos que representan los scripts. Además se ha empleado el lenguaje de transformación RubyTL [Sánchez, 2006] para las transformaciones M2M, para la generación de código SQL mediante transformaciones modelo a código (M2T) e incluso para la implementación del algoritmo para la inferencia/detección de errores. Por ultimo comentar el uso de la infraestructura que ofrece Eclipse para el desarrollo de plugins (PDE).

1.2 Antecedentes

Como se puede ver en [ADM06], se definen diversos escenarios de modernización de software: Gestión de un conjunto de aplicaciones, Mejora de aplicaciones, Conversión de lenguaje a lenguaje, Migración de plataforma, Integración de aplicaciones no invasiva, Transformación a SOA, Migración de arquitectura de datos, Consolidación de arquitectura de sistema y datos, Desarrollo de Data Warehouse, Selección y desarrollo de paquetes de aplicaciones de terceros, Reuso de componentes, Transformación a MDA y Seguridad de software.

De todos ellos, tres se encuentran dentro del ámbito de la modernización datos. La Migración de arquitectura de datos migra una o más estructuras de datos de un archivo o base de datos no relacional a otra arquitectura relacional. Los requisitos de este escenario son que el usuario está experimentado problemas de consistencia de datos, redundancia, accesibilidad e integridad; el negocio no consigue el mismo tipo de datos a través de sistemas distintos o se

calculan de manera distinta; otros problemas son los conocidos sistemas de datos basados en ficheros, estructuras de red o jerárquicas, no accesibles por nuevas tecnologías o distribuidas. También es frecuente la necesidad de los usuarios de nuevas perspectivas o vistas de los datos existentes. Por último, las estructuras de bases de datos o archivos son obsoletas y son eliminadas.

El segundo escenario relacionado con la modernización de datos es la Consolidación de arquitecturas de sistemas y datos. Muchas organizaciones tienen múltiples sistemas que realizan las mismas funciones básicas, por ejemplo tres sistemas que facturan. La clonación de sistemas también contribuye a este tipo de redundancias. Los sistemas y estructuras de datos redundantes contribuyen a crear inconsistencias, procesos de negocio repetidos, frustración del cliente, problemas de integración y un volumen de trabajo de mantenimiento excesivo. También construir una aplicación simple para múltiples sistemas únicos conlleva este escenario.

Por último tenemos un escenario relacionado con el Desarrollo de bases de datos multidimensionales o Data Warehouse. Este escenario construye un escenario de datos de negocio y crea caminos para acceder a estos datos. La base de datos multidimensional contiene datos que han sido extraídos, analizados y transformados en un repositorio común al que los usuarios pueden acceder, pero no actualizar, como requerimiento. Como requisitos nos encontramos que las funciones del negocio requieren consolidar el acceso a ciertos datos para aerodinamizar las tareas del usuario; otro aspecto típico es que el usuario necesite acceder a datos que traspasan los límites organizacionales y de aplicación. Otro síntoma de cambio son los datos relacionados que están definidos a través de múltiples sistemas, haciendo complicado al usuario acceder a un sumario de la información. Por último ocurre que no hay demasiado tiempo o presupuesto para integrar o modificar el núcleo de las aplicaciones con el fin de conseguir los requisitos deseados.

La MDM es un enfoque que ha emergido recientemente y hasta la fecha se han publicado pocos trabajos que describan principios, enfoques, herramientas y experiencias de procesos de modernización basada en modelos. A continuación comentamos algunos trabajos relacionados con la modernización de los datos de una aplicación.

KDM [KDM] es una especificación diseñada para ser la base de la iniciativa Architecture Driven Modernization (ADM) del OMG. Provee una representación común intermedia para los sistemas de software existentes y sus entornos operativos. KDM es un meta-modelo MOF, y define una API sobre la cual podemos construir herramientas de modernización y aseguramiento del software. KDM está compuesto de paquetes organizados a su vez en las cuatro capas que son Infraestructura, Elementos de Programa, Elementos de recursos y Abstracciones. Dentro de la Capa Elementos de recursos está incluido El paquete Data representa los artefactos relacionados con los datos persistentes, tales como ficheros indexados, bases de datos y otros tipos de almacenes de datos. Estos activos son clave para el software empresarial, ya que representan los metadatos empresariales. El paquete KDM está alineado con otra especificación del OMG, llamada Common Warehouse metamodel (CWM).

En [MAZON+07] se hace uso del MDM para modernización en el mundo de las bases de datos multidimensionales o Data Warehouse. Las bases de datos multidimensionales son sistemas de análisis de datos muy completos y minuciosos, que nada tienen que ver con los lenguajes tradicionales de consulta como SQL. Los autores plantean el paso de una base de datos convencional a un sistema multidimensional enfocado como una modernización de datos en tres fases, obteniendo automáticamente una representación lógica de la base de datos, haciendo anotaciones sobre ella relacionadas con los conceptos de bases de datos multidimensionales, y posteriormente construyendo un modelo conceptual multidimensional a partir del modelo con las anotaciones.

También en [POLO+07] se aplica la modernización de software. En este caso a partir de un sistema de software completo se obtiene, por ingeniería inversa, una especificación de alto nivel, a partir de la cual construir un nuevo sistema. El proceso se desarrolla creando un esquema físico de la base de datos, y que se representará como instancia de un metamodelo independiente de la plataforma para ser transformado en un diagrama de clases que representa un posible esquema conceptual de la base de datos. Este diagrama es utilizado como punto de partida del proceso de generación de código de la nueva aplicación.

Existen publicaciones que abordan la mejora de bases de datos con herramientas no basadas en modelos. En [CES+09] podemos encontrar un análisis de herramientas de mejora de bases de datos relacionales, tratando de mejorar la no existencia de normalización en sus estructuras de datos, y en ningún caso haciendo uso de técnicas de DSDM. Según [ELI+08], desde el punto de vista del diseñador una base de datos normalizada tiene como beneficios la existencia de una estrategia de construcción de las relaciones, mejora de las interfaces con los usuarios finales de actividades no planeadas, eliminación de problemas relacionados con la inserción y eliminación de datos, así como identificación de problemas potenciales que probablemente requieran un análisis adicional y documentación. Por otro lado el usuario obtendrá con una base de datos normalizada una mejora del tiempo de respuesta. [ELI+08] propone algoritmos de obtención automatizada de bases de datos normalizadas, pero también fuera del ámbito del DSDM.

Finalmente comentamos tres proyectos de investigación recientes sobre modernización de aplicaciones empresariales que ilustran bien las fases del proceso.

En [OSC+08], se plantea un caso de estudio de MDM para un escenario de migración de plataformas: la migración de sistemas web desarrollados con Struts a sistemas Java Server Faces (JSF), donde sus fases principales consisten en la extracción de modelos a partir de los artefactos fuente, el rediseño del sistema existente mediante transformaciones modelo a modelo que establecen la correspondencia entre las plataformas origen y destino, y la generación del nuevo sistema a partir de transformaciones modelo a código que permiten obtener código a partir de los modelos obtenidos en la fase anterior. Se ha utilizado Gra2MoL para la extracción de modelos y RubyTL para las transformaciones M2M. También se ha definido un lenguaje específico para extraer código JSP de modelos, puesto que el código HTML no conforma con una gramática.

El [HCMMER08] encontramos otro proyecto de MDM para un caso de estudio industrial de migración de sistemas heredados (legacy systems) a servicios web (arquitectura SOA). En este caso se crea un parser ad-hoc para extraer el código en vez de DSL como Gr2aMoL.

En [AGA+06] se plantea un caso de modernización desde Oracle Forms a la plataforma .NET. Se incide en la dificultad de salvar la diferencia de semántica entre ambas plataformas y se plantea un proceso de desarrollo muy general. El proceso de desarrollo que se propone es completamente artesanal, en vez de usar técnicas DSDM como un medio eficaz para automatizar cada etapa del desarrollo.

Este proyecto se basa en [Bernabe09], tesis de Máster presentada en Septiembre del 2009 por Bernabé Nicolás García. En dicha tesis se realiza la obtención del modelo de la gramática DDL de SQL mediante Gra2Mol y la detección de los errores de dicho modelo mediante un algoritmo de detección de errores implementado en PL/SQL.

Este proyecto amplía y mejora esta tesis. Obtiene, además del modelo de la gramática DDL de SQL, el modelo de la gramática DML de SQL. Además de realizar la detección de los errores (reescribiendo el algoritmo de detección de errores de la tesis al lenguaje RubyTL), realiza la corrección del modelo de errores y la generación de código SQL mediante RubyTL.

1.3 Motivación

Durante las dos últimas décadas el uso de los sistemas gestores de bases de datos relacionales se ha generalizado sustituyendo a los archivos y sistemas jerárquicos y en red. Por ello, en la actualidad existe por todo el mundo un gran número de aplicaciones que usan esquemas relacionales para la definición del modelo que describe la estructura de la información almacenada en la base de datos. Estos modelos o esquemas relacionales se han creado con el lenguaje DDL (Data Definition Languag), un lenguaje de SQL específico del dominio de bases de datos ampliamente conocido, y para sistemas gestores de bases de datos más o menos sofisticadas y potentes.

Estos esquemas tienen que ser conocidos y comprendidos completamente por el analista y normalmente también por el desarrollador de la aplicación. Además el desarrollador debe conocer de qué manera recuperar esos datos a través de las pertinentes órdenes de manejo de datos (DML – Data Manage Lenguage) que es el otro lenguaje importante en SQL.

En un proyecto de modernización o evolución de una aplicación existente que usa una base de datos relacional, es frecuente que el modelo relacional tenga algunas deficiencias en aspectos como la normalización o la definición de reglas de integridad. Por lo tanto, como parte de la modernización del sistema legado, se requiere una transformación del modelo relacional original a otro equivalente pero corregido para eliminar los fallos encontrados. Las mejoras y correcciones a considerar durante el proceso de modernización pueden ser tales como la creación de claves foráneas no existentes, incluso existiendo un alto porcentaje de

coincidencias sobre los datos actuales, o la creación de chequeos sobre los determinados valores de algunas columnas no definidos.

Debido a la existencia de numerosos esquemas de datos en el mercado actual y al pensamiento de la sociedad actual que prefiere crear a reutilizar. Plantearse crear una herramienta para la modernización de los datos de un sistema de información dependiente del esquema es impensable hoy en día. Por ello se debía dirigir el proceso mediante un Desarrollo Software Dirigido por Modelos. Gracias a un proceso DSDM se consigue obtener la independencia entre el origen y el esquema de datos que se desea, pudiendo emplearse la herramienta sobre cualquier esquema. Además el proceso DSDM permite el empleo de tecnologías nuevas que facilitan el proceso de modernización, como la extracción de modelos a partir de gramáticas, transformaciones model-to-model y model-to-code. Y la posibilidad de una vez se ha terminado de modernizar los datos del sistema generar código para cualquier plataforma sobe la que se desee obtener el esquema de datos de nuevo, no teniendo que ser la misma plataforma de la que se partió en la modernización.

1.4 Objetivos

Este proyecto presenta diferentes objetivos. El primero y más importante es el de definir un proceso de modernización de software, centrado en la modernización de los datos de un Sistema de Información, aplicando técnicas y tecnologías propias del Desarrollo de Software Dirigido por Modelos.

Gracias al uso del DSDM en la modernización del sistema de información se puede conseguir definir un proceso de modernización independiente del sistema gestor de bases de datos relacionales. La realización de este proyecto se ha centrado en aislar totalmente la modernización de la base de datos del sistema de información.

Otro objetivo del proyecto es el de analizar la conveniencia y la facilidad de usar un enfoque DSDM en un proceso de modernización de datos (MDM). Se quiere comprobar cómo de útil, fácil y eficiente es el uso de un enfoque DSDM en el proceso de modernización de datos o MDM.

El último de los objetivos de este proyecto es el de construir una herramienta operativa y completa que de soporte al proceso de modernización definido en el proyecto. Se quiere obtener un asistente a partir del cual pasándole los modelos de datos de las gramáticas DDL y DML de SQL se consiga realizar todo la modernización del sistema y la obtención de código. De este modo se consigue crear una de las primeras herramientas operativas y completas que realicen una modernización de los datos aplicando un proceso DSDM del mercado.

Además de todos los objetivos que se han relatado, este proyecto presenta una serie de objetivos secundarios:

- Aprender a aplicar tecnologías de transformación Model-to-Model (M2M) y Model-to-Code (M2C).
- Aprender a aplicar tecnologías de extracción de modelos a partir de gramáticas.
- Aprender a aplicar tecnologías para la navegación y consulta de modelos.

Loa tres objetivos secundarios tienen la misma finalidad: aprender a emplear las diversas tecnologías existentes en el mercado para lograr obtener un proceso DSDM en la modernización de los datos. Para ello se aplicarán herramientas como: Gra2Mol, tecnología de extracción de modelos a partir de gramáticas. RubyTL, tecnología capaz de realizar transformaciones M2M y M2C. EMF, framework de Eclipse que facilita la navegación a través de un modelo Ecore de datos.

No será objetivo de este proyecto aplicar de manera exhaustiva el proceso implementado sobre un caso de estudio profundo (con grandes volúmenes de datos y extensas estructuras de información). Así como tampoco será objetivo de este proyecto el aprender a usar tecnologías de creación de plugins para Eclipse (PDE) ni explicarlas.

1.5 Metodología

El trabajo de este proyecto se ha desarrollado en varias fases. El proyecto se prolongó a lo largo de un periodo de 4 meses y medio con una dedicación semanal media de unas 15 horas, lo que hace un total de 270 horas, más aproximadamente unas 30 horas extras que se invirtieron en las ultimas semanas para ultimar algunos detalles de la documentación y de la presentación. Lo que hace un total de 300 horas.

La primera fase consistió en estudiar, comprender e instalar Gra2Mol [Gra2Mol]. Toda la información de Gra2Mol fue obtenida de la página web de la herramienta. Se instaló Gra2Mol en Eclipse para su prueba. Se realizaron una serie de proyectos de ejemplo para poder comprender mejor Gra2Mol. En realizar todo esto se emplearon unas 2 semanas. Después se invirtieron otras 2 semanas para crear los proyectos DML y DDL. La duración de esta fase fue de 4 semanas.

La siguiente fase fue la relacionada con RubyTL y Ruby [RubyTL]. En ella se estuvo aproximadamente unas 8 semanas. Las 2 primeras semanas se estuvo leyendo la documentación de la página web de RubyTL y la aportada por el director de este proyecto, realizando una serie de proyectos de ejemplo. Las dos semanas siguientes se dedicaron a realizar el proyecto de la integración de los modelos DML y DDL en el modelo DML_DDL. Las siguientes 2 semanas en trasladar el algoritmo de detección de errores de la tesis a Ruby y probar su funcionamiento. Las dos semanas siguientes se realizó el proyecto de obtención de los modelos corregido y residual, además de la generación de código a partir del modelo corregido. En el tiempo de creación de cada uno de los proyectos de esta fase también se incluye el tiempo dedicado a la prueba de lo desarrollado.

A continuación vino la fase de creación del plugin. En ella se invirtió un tiempo de 2 semanas. La mitad de la primera en obtener información acerca del plugin PDE de Eclipse [plugin]. Y la otra mitad en el desarrollo de unos plugin de ejemplo y del plugin final. La segunda semana de ésta fase se dedicó a modificar el plugin para añadirle la funcionalidad deseada. En esta semana también se realizaban búsquedas de información por foros y blogs de internet para la realización de cierta funcionalidad del plugin.

La última fase de implementación fue la del uso de EMF. En esta fase se invirtió un tiempo de 3 semanas. La primera semana en documentarse sobre EMF gracias al libro proporcionado por el director de del proyecto [EMF]. La siguiente semana se realizaron una serie de pruebas para comprobar el funcionamiento de EMF y se incluyo dicha funcionalidad en el plugin. La última semana se empleó en corregir un error que se obtenía en la ejecución del plugin con EMF.

La última fase del proyecto se dedicó a completar la documentación final. Para esta fase se invirtieron unas 3 semanas. La primera de ellas se intercaló con la última semana de la fase de EMF.

1.6 Organización del Documento

En este **capítulo 1** presentamos una introducción al proyecto, el contexto, los antecedentes, la motivación, los objetivos del proyecto y la metodología utilizada.

El **capítulo 2** se centra en el la solución al problema que se ha alcanzado. Se realiza una breve introducción y posteriormente se describe la solución global obtenida en el proyecto.

El **capítulo 3** explica co se realiza la extracción de la meta-información e información de una BB.DD a dos modelos. Se introduce Gra2Mol, que será la herramienta que se use para dicha extracción. Y se estudiarán la extracción de los modelos DDL y DML por separado en sendos apartados.

En el **capítulo 4** se comenta como se realiza la integración de los modelos DDL y DML obtenidos en el capítulo 2. Antes de entrar en detalle en dicha integración, se realiza una introducción a RubyTL, que será la herramienta que se empleará para realizar la integración. Posteriormente se detalla la realización de la integración de los modelos DDL y DML.

El **capítulo 5** explica como se realiza la detección de los errores en el modelo DML_DDL obtenido en el capítulo anterior mediante RubyTL.

El **capítulo 6** explica como se realiza la corrección de los errores almacenados en el modelo Errores, que se han detectado en el capítulo anterior, mediante RubyTL.

El **capítulo 7** trata sobre la generación de código a partir del modelo de datos corregido mediante RubyTL.

El **capítulo 8** comenta la herramienta que se ha desarrollado para integrar las fases del proyecto. Dicha herramienta es un wizard o asistente. Antes de comentar los aspectos más relevantes de este asistente se dará una pequeña introducción a las herramientas usadas en el asistente: PDE y EMF.

El **capítulo 9** contiene las conclusiones y vías futuras. El **capítulo 10** es la bibliografía. Y en el **capítulo 11** tenemos los anexos, con el código mas relevante, un pseudo-manual del usuario y un bug de RubyTL que se ha tenido que solventar.

2 Solución al Problema

2.1 Introducción

Muchas empresas y organismos se encuentran ante la situación de tener que modernizar una determinada aplicación debido a situaciones como a la migración a una nueva plataforma, la mejora de la aplicación, o que ya no se le ofrezca soporte a la tecnología sobre la que se encuentra implementada. La modernización de un sistema de información se puede desglosar en la modernización del aplicativo y en la modernización del sistema de infraestructura de datos.

En nuestro caso nos centramos en el aspecto de los datos con el objetivo de modernizar o evolucionar el sistema para abordar problemas tan importantes como la existencia de inconsistencias generadas por no existir las restricciones en los datos adecuadas, o la redundancia de datos debido a su desnormalización con respecto a las formas normales de Codd o la forma normal de Boyce-Codd [Connolly+04].

Para modernizar o hacer evolucionar el sistema solucionando los problemas antes mencionados, un diseñador se encuentra con que sólo dispone de un script DDL que define el esquema de la base de datos, el cual muchas veces no incluye comentarios sobre los campos y tablas. Por tanto, es necesario enfrentarse a tediosas tareas como las tres siguientes: 1) encontrar posibles dependencias entre tablas no declaradas explícitamente, 2) dependencias entre tablas que han sido desactivadas, o 3) bien chequeos deshabilitados sobre columnas.

2.2 Solución

Mediante este proyecto se pretende modernizar o evolucionar los datos de un sistema de infraestructuras de datos mediante un desarrollo dirigido por modelos.

Primeramente hay que obtener dos modelos de datos, que conformarán a sus respectivos metamodelos. Dichos modelos contendrán toda la información que se almacena en la base de datos del sistema de información. Esta información será relativa a la meta-información del sistema, guardándose en un modelo DDL, y relativa a los datos que contenga el sistema de información, guardándose en un modelo DML.

Una vez que se han creado los dos modelos con toda la información del sistema se tendrían que unir en un único modelo. Se unirán en un modelo DML_DDL necesario para disponer de toda la información relacionada y así realizar la posterior corrección de los datos del sistema de información.

Cuando se haya realizado la integración de ambos modelos en uno sólo, al modelo resultante se le aplica un algoritmo mediante el cuál se obtienen una serie de errores y posibles mejoras

en el sistema de información. Una vez que se tiene el modelo con los errores se procede a su corrección.

En la corrección del modelo con los errores se obtiene un modelo DML_DDL corregido y un modelo DML_DDL residual. En el modelo residual estarán todos los datos que daban lugar a algún error, mientras que en el modelo corregido se habrán eliminado aquellos datos que presentasen algún error y se habrán corregido aquellas estructuras incorrectas.

Al final de todo el proyecto se procede a la generación del código SQL a partir del modelo DML_DDL corregido que se obtuvo anteriormente. Se obtendrán dos scripts capaces de recrear la base de datos de la que se partió pero corregida.

En la siguiente figura se pueden observar en un esquema las etapas del proyecto:

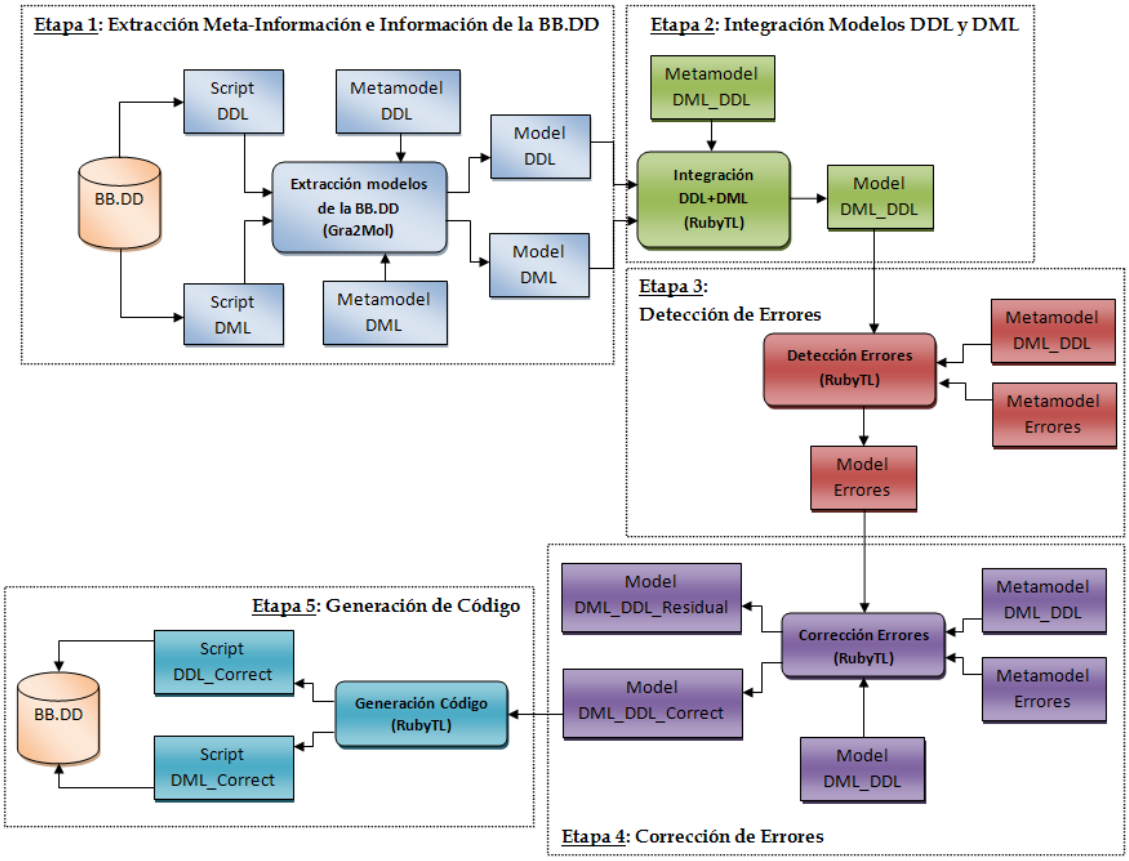

Figura 1: Esquema Global.

En la primera etapa del proyecto, ***'Extracción Meta-Información e Información de la BB.DD'***, se usa Gra2Mol para realizar dicha extracción.

Para modernizar o evolucionar un sistema de información es necesario el conocimiento de su meta-información. Siendo la estructura de un esquema de datos lo principal que necesitamos para poder alcanzar esa modernización.
La meta-información se extrae mediante Gra2Mol a un modelo de datos a partir de un script que contenga sentencias SQL DDL (Data Definition Lenguage). De esta forma se tendrá en el modelo de datos obtenido una descripción del sistema que nos aportará el conocimiento necesario sobre la infraestructura de los datos del sistema, consiguiendo independizar el proceso de modernización del sistema que se quiere evolucionar y de cómo se almacenaba su meta-información.

Una modernización o evolución de un sistema de infraestructuras de datos, para su corrección o migración, no tiene cabida si no se incluyen sus activos más importantes, los datos o información. Debido a esto y a que uno de los motivos principales de éste proyecto es el uso de un Desarrollo del Software Dirigido por Modelos, se debería poder trabajar con dicha información independientemente de cómo el sistema la almacene y la gestione. Es por ello que los datos deben también ser extraidos del origen de datos en cuestión.
La obtención de los datos almacenados en un sistema de información se lleva a cabo mediante Gra2Mol. Dichos datos son exportados a un modelo de datos a partir de un script de sentencias SQL DML (Data Manage Lenguage).

En la segunda etapa, ***'Integración Modelos DDL y DML'***, se procede a realizar la unión de los modelos obtenidos en la primera etapa. Dichos modelos son el modelo DDL, y el modelo DML. En una primera aproximación se pensó extraer los datos y la meta-información directamente a un único modelo (llamado DML_DDL) ya que son necesarios para poder realizar la modernización con los criterios de inferencia de errores que se aplican en este proyecto. Al intentar seguir por ésta vía nos surgió un problema: se comprobó que Gra2Mol, cuando ejecuta una regla, crea el elemento del metamodelo o metaclase sin excepción alguna. Entonces cuando en una misma tabla se quería insertar dos registros, al tratarse de dos sentencias independientes en el scrit DML, Gra2Mol trataba la inserción creando la misma tabla dos veces, cada una con su registro. Se comentó el problema con Javier Luis Cánovas Izquierdo (creador de Gra2Mol) y nos explicó que en Gra2Mol no existen las reglas de copia, como en RubyTL. Cuando en un modelo existe un elemento y se procede a ejecutar una regla que obtiene el mismo elemento pero con distinto contenido, Gra2Mol no añade el contenido al elemento existente, sino que crea uno nuevo.

Debido a esto, se optó por independizar la meta-información del sistema de los datos del mismo, postergando la unificación de ambos modelos a esta fase que se tiene que ejecutar antes de detectar los posibles errores que posee una sistema de información. Mediante RubyTL se realizará una transformación M2M de los modelos DDL y DML a un modelo DML_DDL que integra a los dos modelos anteriores.

En la tercera etapa del proyecto, ***'Detección de Errores'***, es donde se realiza el mayor trabajo. En ella se ejecuta una reimplementación del algoritmo definido en la tesis de master anteriormente comentada.

Mediante dicho algoritmo se consiguen localizar los errores y mejoras identificados en este proyecto y que puedan existir en el modelo DML_DDL obtenido en la etapa anterior.

Una vez que se han localizado los errores del modelo DML_DDL se guardan en un modelo, llamado Errores, que conforma al metamodelo Errores. Por lo que el modelo Errores contendrá los errores identificados del sistema de información.

Al finalizar las tres primeras etapas hemos conseguido obtener en un modelo todo la información (meta-información e información) de un sistema y en otro modelo todos los errores que existen en dicho sistema.

En la cuarta etapa, ***'Corrección de Errores'***, gracias al modelo DML_DDL y al modelo Errores se puede obtener un modelo, llamado DML_DDL_Correct, que conformará al metamodelo DML_DDL con toda la información corregida. Además, por seguridad, se obtendrá otro modelo llamado DML_DDL_Residual, que conformará también al metamodelo DML_DDL y que contendrá toda aquella información errónea que se haya eliminado al realizar la corrección del modelo DML_DDL según el modelo Errores.

En la última etapa del proyecto, ***'Generación de Código'***, se realiza la generación de dos scripts con sentencias SQL. En uno de dichos scripts se almacenarán las sentencias DDL del modelo DML_DDL_Correct, que serán sentencias corregidas con respecto al script que se usó para la obtención del modelo DDL en la primera etapa. Y en el otro script se almacenarán las sentencias DML del modelo DML_DDL_Correct, que serán sentencias con las inserciones de datos que cumplen el nuevo esquema corregido.

Este proyecto se ha centrado en tres tipos de errores/mejoras que se pueden encontrar en una base de datos de aplicaciones existentes que cuentan con deficiencias, básicamente por problemas de integridad de la información. Las tres deficiencias consideradas han sido:

1. Ausencia de claves foráneas de una tabla hacia otra. Se considerará que la tabla padre va a poseer una clave primaria, de la que dependerá la clave foránea de la tabla dependiente.

2. Existencia de una clave foránea de una tabla principal respecto a otra dependiente, pero desactivada por necesidades de los datos, migración de los datos desde una aplicación anterior o por cualquier otra causa.

3. Restricciones en los posibles valores de una columna. Existencia de una clave de chequeo (check constraint) que comprueba que valores son aceptados en una columna, pero que se encuentra desactivada debido a que no todos sus valores la cumplen por necesidades propias de los datos, migración de los datos desde una aplicación anterior o por cualquier otra causa.

Estos serán los tres tipos de errores que se han detectado y corregido, siempre y cuando el usuario quiera corregirlos. De ahora en adelante se les conocerá como *"possible_new_Fk"*, *"Fk_disabled"* y *"Ck_disabled"* respectivamente.

3 Extracción Meta-Información e Información de la BB.DD

En está etapa se pretende obtener toda la información del sistema en dos modelos. En un modelo llamado DDL se obtendrá todo la meta-información de la BB.DD. Mientras que en un modelo llamado DML se obtendrán todos los datos de la BB.DD. Esta extracción se realizará mediante la herramienta Gra2Mol.

3.1 Introducción a Gra2Mol

Gra2Mol (Grammar-to-Model Lenguage) [Gra2Mol] es un lenguaje creado por Javier Luis Cánovas Izquierdo investigador del Grupo Modelum del Departamento de Informática y Sistemas de la Universidad de Murcia. Gra2Mol es un lenguaje ideado para la obtención de modelos que conforman un metamodelo fuente partiendo de un código que conforma una gramática del lenguaje. Fue especialmente diseñado para hacer frente al problema de la extracción de modelos desde un código fuente basado en una gramática. Por lo que se podría pensar que Gra2Mol es un extractor de modelos a partir de gramáticas.

El lenguaje que define Gra2Mol se enmarca dentro de los lenguajes de transformación basados en reglas, como ATL o RubyTL (lenguajes de transformación model-to-model), pero con una gran diferencia y es que el elemento origen de una regla en Gra2Mol es un elemento de la gramática en vez de ser un elemento del metamodelo como ocurre en ATL o RubyTL. Otra diferencia importante es que el lenguaje proporciona características específicas del dominio para abordar temas específicos de las transformaciones grammar-to-model.

3.1.1 Queries

Gra2Mol incorpora un pequeño lenguaje estructural, inspirado en XPath, para permitir la navegación a través del árbol de la sintaxis sin necesidad de especificar de manera declarativa cada uno de los pasos de la navegación. El lenguaje de consulta definido en Gra2Mol permite recorrer en orden y de forma eficiente el árbol de sintaxis concreto (CST – Concrete Syntax Tree), para extraer los elementos de información deseados.

El CST es una estructura de datos que representa el resultado almacenado en memoria que se obtiene del análisis de la sintaxis. Por lo que se podría decir que el CST es lo mismo que el Parsing Tree. El CST presenta una diferencia importante con el AST (Abstract Syntax Tree). Un AST es un árbol que representa una estructura sintáctica de un código fuente que sigue una gramática formal. En el CST los nodos representan los elementos no terminales de la gramática, mientras que las hojas representan a los símbolos terminales de la gramática. Sin embargo un AST es una representación abstracta (simplificada) de la estructura sintáctica de

un código fuente que sigue una gramática. Los nodos internos de un AST representan a los operadores de la gramática, mientras que las hojas son los valores a los que se le aplican dichos operadores. La sintaxis es abstracta en el sentido de que cada nodo representa cada detalle que aparece en la sintaxis real de la gramática. Se puede observar la diferencia entre ambos en la siguiente figura.

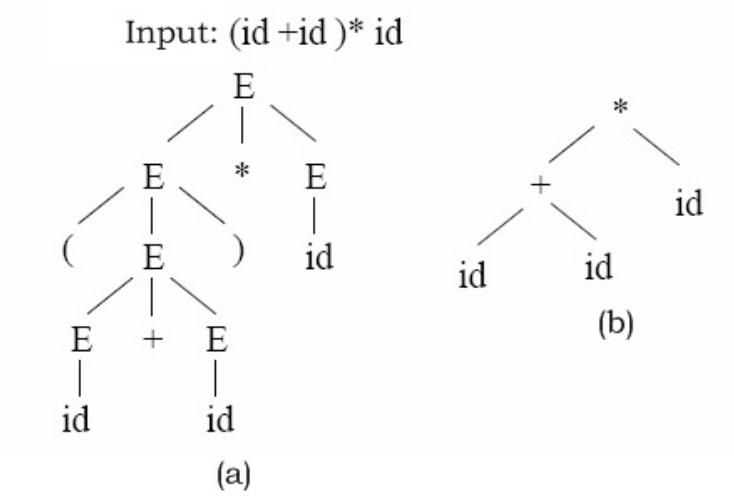

Figura 2: Concrete Syntax Tree vs Abstract Syntax Tree

Las consultas en Gra2Mol usan el CST, facilitando así la definición de reglas a partir de la gramática. El desarrollador solamente necesita conocer la definición de la gramática para especificar reglas en Gra2Mol. Por lo que los operadores y los filtros de las reglas son definidos en relación a los elementos terminales y no terminales del CST.

Las consultas navegan por el árbol de la sintaxis recolectando el conjunto de nodos que satisfagan las condiciones impuestas por dicha consulta. A cada nodo del CST se le asigna un tipo que se corresponde con un símbolo no terminal. La expresión EBNF (Extended BNF - Backus Normal Form) que se usa para definir una consulta es la siguiente (corchetes indican opcionalidad):

```
query : [castingExpression] { ( / | // | /// ) node [filterExpression] [accessExpression] };
```

Las consultas están formadas por secciones. Cada sección tiene el uno de los operadores (/, // o ///) y especifica un *node* correspondiente del CST. La *filterExpression* permite filtrar el resultado de una consulta, y *accesExpression* permite acceder al contenido del CST como si fuese un array. Además, el carácter *'#'* se usa para especificar el tipo de nodo raíz o nodos raíces que se obtendrán como resultado de la consulta. La opción *castingExpression* permite modificar el tipo de los nodos devueltos.

Cada uno de los operadores de una consulta tiene una funcionalidad. A continuación se describe cada uno de ellos por separado.

- *Operador '/'*: este operador devuelve el hijo inmediato de un nodo. Por ejemplo, si necesitamos obtener todos los *nodoB* que son hijos inmediatos de cualquier *nodoA*, la consulta sería:

```
query1 : /nodoA/#nodoB;
```

  Cada instancia del operador permite bajar un nivel en el CST.

- *Operador '//'*: este operador navega a través de todos los nodos hijos (directos o indirectos) devolviendo todos los nodos con el tipo correcto. Permite ignorar nodos intermedios que no nos interesan, haciendo la consulta más sencilla de definir al indicar el tipo de los nodos en los que estamos interesados, e ignorando los demás. Por ejemplo, si queremos obtener todos los hijos de un *nodoA* del tipo *nodoB*, la consulta quedaría como sigue:

```
query2 : /nodoA//#nodoB;
```

- *Operador '///'*: es similar al operador anterior, pero con la diferencia de que este operador permite navegar a través del CST de manera recursiva. El operador '//' navega a través de todos los nodos hijos (directos o indirectos) devolviendo los nodos de un tipo en concreto. Sin embargo, una vez que encuentra un nodo no profundiza en él para encontrar nodos del mismo tipo. El operador '///' permite extraer información de gramáticas con estructuras recursivas. La diferencia entre ambos operadores se puede apreciar mejor en la siguiente figura.

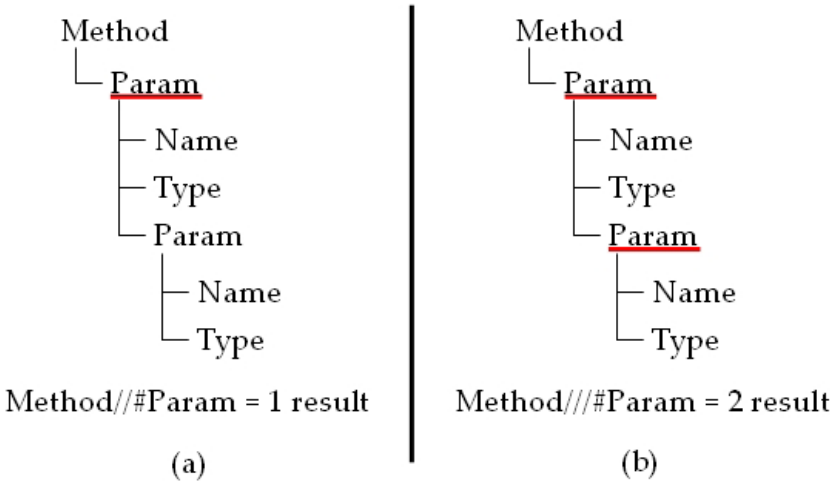

Figura 3: Operador // vs Operador ///

Cabe destacar que los operadores no son excluyentes entre sí, y que se pueden usar de múltiples formas para especificar un path del CST. Por ejemplo, la siguiente consulta obtiene todos los *nodoC* que tiene como hijo un *nodoD*. Además, dichos *nodoC* son hijos directos o indirectos de un *nodoB* que a su vez es hijo directo de un *nodoA*:

```
query3 : /nodoA/nodoB//#nodoC/nodoD;
```

Los filtros que se pueden aplicar en una consulta son expresiones lógicas aplicables en las hojas de los nodos del CST. Cada filtro es una función booleana que comprueba las propiedades de la hoja, como puede ser su existencia o su valor. Solamente los nodos que satisfagan el filtro serán seleccionados en la consulta.

Las *Filter Expression* que existen actualmente son las siguientes:

- *'eq'*: comprueba el valor de una hoja. Por ejemplo, este filtro comprueba si el valor de la hoja id del *nodoA* es igual a *A.12*:

  ```
  filter1 : /#nodoA{id.eq("A.12")};
  ```

- *'exists'*: comprueba la existencia de una hoja. Por ejemplo, el siguiente filtro comprueba si la hoja *id* del *nodoB* existe:

  ```
  filter2 : /#nodoB{id.exists};
  ```

3.1.2 Rule

Como ya hemos comentado, Gra2Mol es un lenguaje basado en reglas como RubyTL o ATL. Una definición de una transformación en Gra2Mol consiste en un conjunto de reglas. Cada una de las reglas especifica una relación entre un elemento de la gramática (un símbolo no terminal) y un elemento del metamodelo. Por lo tanto una regla necesita la especificación de los elementos origen y destino. Mediante la ejecución de la regla, además de crear el elemento del metamodelo destino, se inicializan sus atributos y/o referencias.

A continuación se muestra la estructura de una regla (los corchetes indican opcionalidad):

```
rule 'foo'
   from grammarElement variable
   [context rule_1, rule_2, ..., rule_n]
   to metamodelElement
   queries
      [some-queries]
   mappings
      [some-mappings]
end_rule
```

Cada una de las secciones de la regla se describen a continuación:

- *'from'*: especifica el símbolo no terminal de la gramática, a partir del cual se partirá, y declara una variable que será ligada al nodo del CST una vez que se la regla sea aplicada. Dicha variable puede ser usada por cualquier expresión dentro de la regla.

  En realidad esta sección es una consulta que filtra los nodos de entrada para la regla. Por lo que la regla recibirá todos los nodos del tipo indicado en el *from*. Además se pueden aplicar filtros como los comentados anteriormente, ya que se trata de una consulta. Por lo que al conjunto de nodos de un tipo en concreto, se le puede aplicar además un filtro. Por ejemplo, en el siguiente *from* los nodos que entraran en la regla serán aquellos nodos *blockStatement* que tengan un nodo *statement* que presenten una hoja *TOKEN* con valor *return*:

  ```
  from blockStatement/statement{TOKEN.eq("return")} bst
  ```

- *'context'*: se utiliza para especificar las reglas que pueden llamar a la regla que contiene el *context*. Sirve para restringir el alcance de una regla o conjunto de reglas. Por ejemplo, si la regla *rule1* solamente puede ser llamada por regla *rule2*, en la regla *rule1* debe aparecer el siguiente *context*:

  ```
  context rule2
  ```

- *'to'*: especifica la metaclase destino del metamodelo. Si la metaclase esta incluida en un paquete, se deben usar los *'::'* como separador entre el nombre de la metaclase y el del paquete. Si el metamodelo solamente presenta un paquete, el prefijo no es necesario.

- *'queries'*: contiene un conjunto de expresiones de consulta, con el formato definido anteriormente.

- *'mappings'*: contiene un conjunto de mapeos que asignan valores a las propiedades del elemento destino del modelo. Dentro de la sección de 'mappings' es donde se realizan los *bindings* para asignar un valor a cada una de los atributos del elemento del metamodelo que se quiere crear. El *binding* se usa para especificar la relación entre el elemento fuente de la gramática y el elemento destino el metamodelo. Existen varios tipos de asignaciones, como pueden ser la asignación de un nodo hijo a un atributo del elemento del metamodelo, la creación explícita de un nuevo elemento del metamodelo, la ejecución imperativa (mediante el operador *'execute'*) de una regla, etc. Estos bindings se usan también en los lenguajes de transformación de modelos como ATL o RubyTL.

3.1.3 Ejecución

La ejecución de una transformación en Gra2Mol consta de dos fases.
La primera fase es inherente a Gra2Mol y siempre es llevada acabo para cualquier transformación de manera automática. Se encarga de obtener una gramática enriquecida a partir de la gramática ANTLR que se ha definido para la transformación. Y además crea el CST que usarán las reglas de la trasformación.
La segunda fase que presenta como entradas la gramática enriquecida *G* obtenida por la primera fase, el metamodelo destino MM_T, el programa P_G conforme a *G* y la especificación de Gra2Mol con las reglas que definen el mapeo entre *G* y MM_T. Devolverá como salida el modelo M_T, obtenido de aplicar las reglas de Gra2Mol ($Mapping_{Gra2Mol}$), que conforma el metamodelo MM_T.

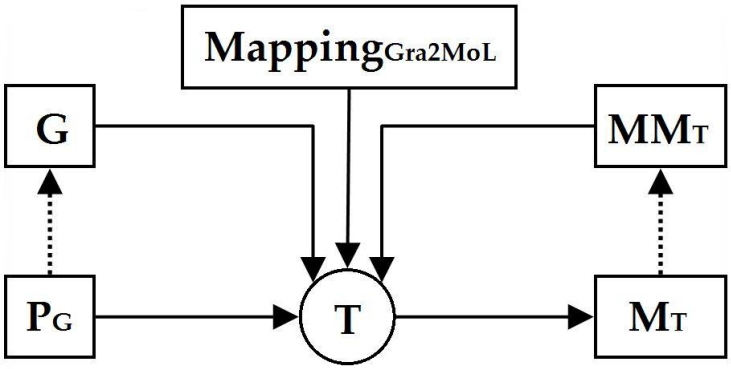

Figura 4: Aproximación de la segunda fase de ejecución de Gra2Mol.

3.2 Extracción Modelo DDL

Gra2Mol necesita como entrada un metamodelo para almacenar la información, el cual conformará el modelo obtenido como salida, una gramática ANTLR, un programa o fuente que conforme dicha gramática, y una serie de reglas de transformación que transformen el programa en el modelo de salida.

Este paso previo a la modernización, o evolución del sistema, se inicia a partir de un script DDL que contiene toda la información almacenada en la base de datos y pretende crear un modelo DDL que conforme el metamodelo DDL de entrada a Gra2Mol.

En la siguiente figura se puede apreciar el esquema que sigue Gra2Mol para obtener el modelo DDL que conforme el metamodelo DDL que se ha definido.

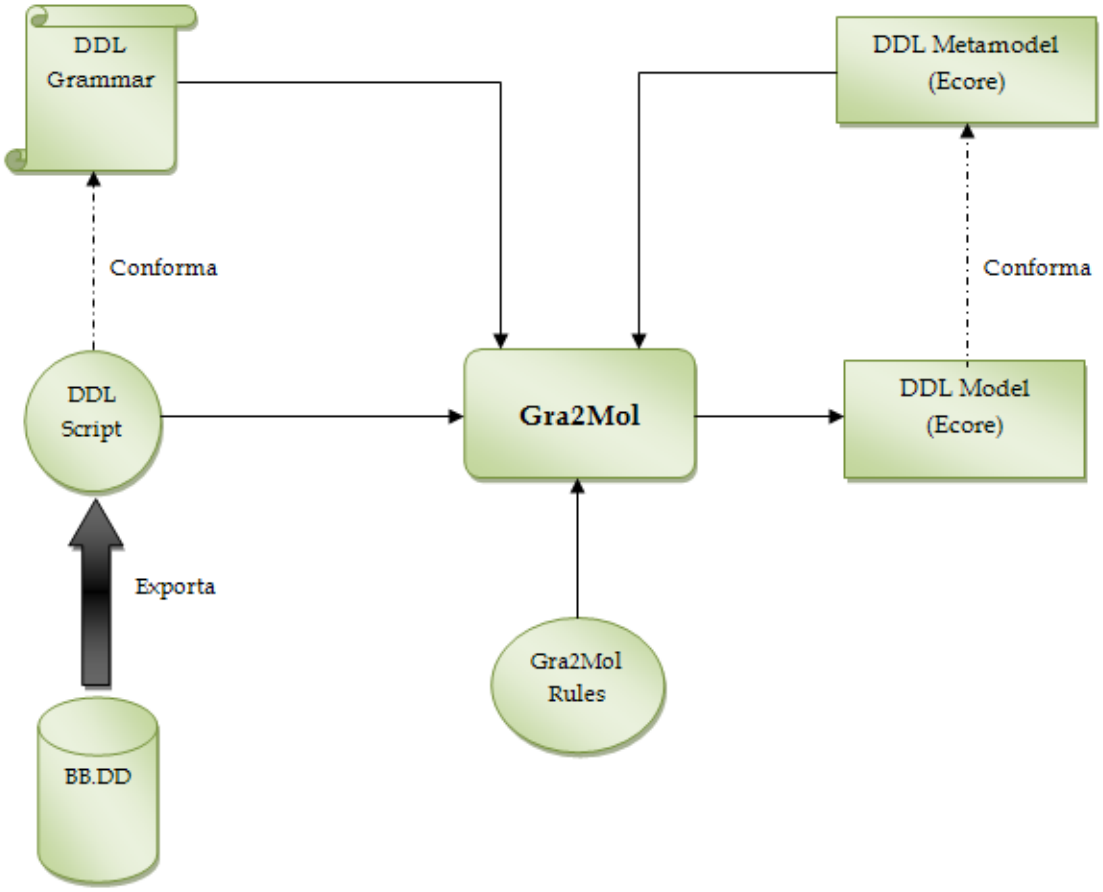

Figura 5: Obtención Modelo DDL con Gra2Mol.

Se va a comentar a continuación la figura anterior.

Para llevar a cabo la obtención del modelo de meta-datos sobre la infraestructura del sistema, se partió de un script en el que se encontraban sentencias SQL para la creación y desactivación de tablas en una BB.DD. Sentencias del tipo:

```
CREATE TABLE <nombre_tabla> (
   <nombre_campo> <tipo_datos(tamaño)> [NULL|NOT NULL] [DEFAULT <valor_por_defecto>],
   …
   CONSTRAINT <nombre> PRIMARY KEY (<nombre_campo>[ ,...n ]),
   CONSTRAINT <nombre> FOREIGN KEY (<nombre_campo>[ ,...n ]) REFERENCES
                                        <tabla_referenciada> (<nombre_campo> [ ,...n ] ) ,
   CONSTRAINT <nombre> CHECK (<nombre_columna> [<,>,=,<=,>=,<>] <valor> [AND|OR]
                                        <nombre_columna> IN (<valores_correctos))
) ;

ALTER TABLE <nombre_tabla> DROP CONSTRAINT <nombre_constraint>;
```

A la hora de obtener el script DDL desde la BB.DD que se esté manejando en el sistema, debemos incluir la opción para que al obtener las sentencias "CREATE TABLE" nos incluya también las sentencias del tipo "ALTER TABLE DROP CONSTRAINT", y así poder identificar que Foreign Keys o Check Constraints están desactivadas. Ya que, de no hacerlo, no habría forma de saber posteriormente que constraints están o no activadas. Al tratarse de un desarrollo dirigido por modelos no se podrá, ni se querrá, acceder mediante sentencias SQL al gestor de la base de datos para obtener la información relativa sobre que constraints están habilitadas o deshabilitadas. Por lo que de ahora en adelante se supondrá lo siguiente:

"Si en el script DDL obtenido de una BB.DD no existe una sentencia que deshabilite una Fk o Ck, se dará por supuesto que dicha Fk o Ck está habilitada."

La gramática usada para la obtención del modelo DDL, es la misma gramática que se uso en la tesis antes comentada [Bernabe09], pero extendida con los tipos del *ANSI SQL-92*. En la gramática de la tesis, se puede observar que los tipos se tratan como simple cadenas de caracteres, no siendo correcto. Por lo que se crearon sentencias en la gramática para definir cada uno de los tipos que soporta el *ANSI SQL-92*. Mencionar que esta definición no es para nada definitiva y se podrá ampliar en un futuro o reducir según las necesidades del desarrollador, pero lo que se intentó fue dar la mayor cobertura posible, intentando definir los tipos más usados y algunos más. A continuación se pueden ver los tipos definidos:

	Tipos del ANSI SQL-92
Exacto	NUMBER, NUMERIC, INTEGER, SMALL INTEGER, DECIMAL, INT, SMALL INT
Aproximado	DOUBLE PRECISION, LONG, LONG RAW, FLOAT, REAL
Characters	CHAR, VARCHAR, VARCHAR2, NVARCHAR2, NCHAR, CHAR VARYING, CHARACTER, CHARACTER VARYING, NATIONAL CHAR, NATIONAL CHAR VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, NCHAR VARYING, CLOB, NCLOB
Bits	BIT, BIT VARYING
Times	DATE, TIME, TIMESTAMP

Intervals	YEAR-MONTH, DAY-TIME
Binaries	BFILE, BLOB, BINARY_DOUBLE, BINARY_FLOAT

Además de los tipos se añadió a la gramática la sintaxis de la sentencia SQL *"ALTER TABLE DROP CONSTRAINT"* para poder conocer que constraints de la tabla están deshabilitadas. Además se han incluido los operados lógicos *"AND"* y *"OR"*, así como los comparadores de *"igualdad", "desigualdad", "mayor que", "menor que", "mayor o igual que"* y *"menor o igual que"*, ya que las check constraint solamente soportaban la sintaxis *"IN (values)"*, pero la sintaxis de la sentencia *"CONSTRAINT CHECK"* también incluye sentencias con comparaciones y operados lógicos.

A continuación se muestran únicamente los cambios introducidos en la gramática de la tesis creada por Bernabé Nicolás García. La gramática completa se puede ver en el apartado "Anexos":

```
disabled_constraint
        : 'ALTER TABLE' table_references 'DROP CONSTRAINT' name_constraint ';'
        ;

log_expresion
        : column_ck (COMPARATOR (NUMBER | CVALUE) LOG_CONJ? | 'IN' '('
(value_list)* ')')
        ;

type
        :       (EXACTO | APROXIMADO | CHARACTERS | BITS | TIMES | INTERVALS
| BINARIES)
        ;

EXACTO
        :       ('NUMBER' | 'NUMERIC' | 'INTEGER' | 'SMALL INTEGER' | 'DECIMAL' |
'INT' | 'SMALL INT')
        ;

APROXIMADO
        : ('DOUBLE PRECISION' | 'LONG' | 'LONG RAW' | 'FLOAT' | 'REAL')
        ;

CHARACTERS
        : ('CHAR' | 'VARCHAR' | 'VARCHAR2' | 'NVARCHAR2' | 'NCHAR' | 'CHAR
VARYING' | 'CHARACTER' | 'CHARACTER VARYING' | 'NATIONAL CHAR' |
'NATIONAL CHAR VARYING' | 'NATIONAL CHARACTER' | 'NATIONAL CHARACTER
VARYING' | 'NCHAR VARYING' | 'CLOB' | 'NCLOB')
        ;
```

```
BITS
        : ('BIT' | 'BIT VARYING')
        ;

TIMES
        : ('DATE' | 'TIME' | 'TIMESTAMP')
        ;

INTERVALS
        : ('YEAR-MONTH' | 'DAY-TIME')
        ;

BINARIES
        : ('BFILE' | 'BLOB' | 'BINARY_DOUBLE' | 'BINARY_FLOAT')
        ;

COMPARATOR                              : '<>' | '>=' | '<=' | '=' | '>' | '<';
LOG_CONJ                        : 'OR' | 'AND';
```

El metamodelo DDL es el mismo que se realizó en la tesis *"Modernización de Base de Datos Relaciones Basada en Modelos"*, pero al cuál se le han añadido las metaclases necesarias para soportar los tipos antes mencionados.

Se tendrán tantas nuevas metaclases como tipos se quieran definir, por lo que se tendrán tantas como se han indicado anteriormente. Habrá una metaclase llamara *"Type"*, que será la clase padre de todos los tipos que se definan. De ella heredarán cada uno de los conjuntos de tipos indicados anteriormente, es decir, *"Exacto", "Aproximado", "Characters", "Bits", "Times", "Intervals"* y *"Binaries"*. Para cada una de estas metaclases, existirá una serie de metaclases que heredarán de ellas, que serán los diferentes tipos que soporte nuestra sentencia SQL.
Además de todo esto, se ha creado una metaclase llamada *"DataTypes"* que engloba todos los tipos que se han usado en la definición de las columnas de las tablas de la base de datos. Lo que queremos conseguir con esto es obtener, al finalizar la transformación de Gra2Mol, una especie de librearía con todos los tipos que se han usado en todas las columnas de la base de datos.

Otra modificación del metamodelo DDL de la tesis es la que tiene que ver con la ampliación de la sintaxis de las check constraint que se ha incluido en la gramática. Para ello se ha creado una nueva metaclase llamada *"ValuesCk"*. En la sentencia "CONSTRAINT CHECK" se especifica el nombre de la constraint a crear y una serie de comparaciones que se repiten una o más veces. Se debe tener en cuenta que en la última repetición, el operador lógico no aparecerá. *"ValuesCk"* almacenará una comparación, es decir, que la metaclase *"Ck"* contiene un conjunto de *"ValuesCk"*, una por cada repetición de la siguiente sintaxis:

```
<columna> <comparador> <valor> <operadorLógico>
```

Mencionar que ha sido incluido un atributo tanto en la metaclase *"Ck"* como en la *"Fk"*, metaclases que representan a la check constraint y a la foreign key respectivamente, para indicar si dicha constraint ésta o no habilitada. Se trata de un string que puede tener únicamente dos valores, *"DISABLED"* o *"ENABLED"*, para indicar su estado.

El metamodelo quedaría como se muestra en la siguiente figura:

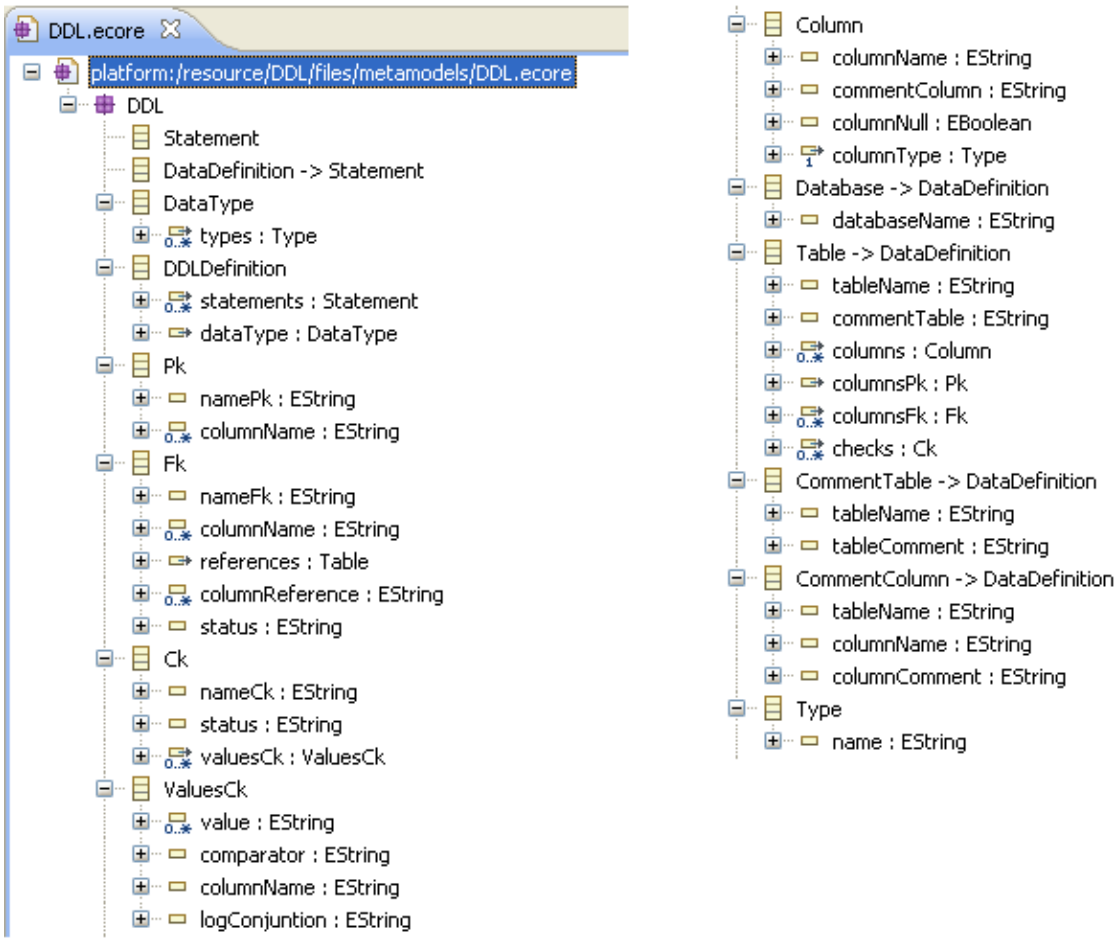

Figura 6: Metamodelo DDL. Las metaclases de todos los tipos han sido omitidas.

A continuación se explicará con más detalle cada una de las nuevas metaclases que se han definido:

- *"DataTypes"*: Se trata de la metaclase que contendrá todas los tipos que se han usado en la definición de las columnas de cada una de las tablas de la base de datos. Contiene un atributo multivaluado de tipo *"Type"* para almacenar los tipos. Como se quería obtener una librería con todos los tipos usados, la metaclase padre del metamodelo *"DDLDefinition"* contiene una referencia multivaluada para almacenar todos los *"Statements"* de creación del script (incluido en la versión de la tesis) y otra

univaluada para incluir la librería de tipos *"DataTypes"*, para con ello aislar los tipos de las sentencias de creación.

- *"Type"*: Es la clase padre de todos los tipos que se han definido en la gramática. Contiene un atributo de tipo string para almacenar el nombre del tipo. Éste atributo se usa en la última fase del proyecto, cuando a partir del modelo corregido de la base de datos queremos obtener un script con las sentencias SQL. Se comentará en dicha fase.

- *"ValuesCk"*: Como se ha comentado anteriormente, ésta metaclase servirá para almacenar una repetición de la check constraint. Contiene los siguientes atributos: el comparador empleado ('=', '<>', '<', '>', '<=', '>='), el nombre de la columna que se ve afectada por la constraint, el operador lógico ('AND', 'OR'), y una lista de valores que debe cumplir la columna. Como una check constraint puede tener más de una comparación y se puede aplicar a más de una columna, en la metaclase *"Ck"* habrá una referencia multivaluada, que indicará el número de restricciones que aplica la check constraint.

El resto de metaclases introducidas, son metaclases que representan los tipos de las columnas en SQL. Por lo que no requieren más explicación.

Una vez que tenemos el metamodelo definido, la gramática y el script con las sentencias SQL de la BB.DD, podemos proceder a la obtención del modelo mediante Gra2Mol. Para ello se han definido una serie de reglas de transformación, las cuales se pueden ver íntegramente en el apartado "Anexos". A continuación se explicaran las reglas más extensas y complejas. Como sucedía anteriormente en la gramática y en el metamodelo, éstas reglas son las mismas que se definieron en la tesis, pero ampliadas para soportar las modificaciones realizadas. Aún así comentaremos las reglas más interesantes, sean nuevas o no.

Se ha querido realizar una librería que englobe a todos los tipos usados en el script en una única metaclase aislada de las metaclases de creación de tablas. Como la librería cuelga de la metaclase *"DDLDefinition"* ésta se debería inicializar cuando se reciba la primera sentencia *"DDLDefinition"*. Además no queremos que la librería tenga tipos repetidos, por lo que se debería comprobar antes si dicho tipo existe, o que se encargase de comprobarlo Gra2Mol de forma transparente. Las reglas relacionas con los tipos son las siguientes:

```
rule 'mapStatements'
        from data_definition df
        to DDLDefinition
        queries
                stats : //#data_definition_statement;
                types : ///#type[0];
        mappings
                dataType = types;
                statements = stats;
end_rule
```

En ésta regla se realiza la inicialización de la librería de tipos. Para ello, y como se sabe que cuelga de la metaclase *"DDLDefinition",* una vez que se recibe el elemento de la gramática que se mapea a la metaclase *"DDLDefinition"* lo que se hace es buscar por todo el scripts los tipos existentes de manera recursiva, usando el operador *'///'*, y de todos los tipos encontrados, únicamente se coge el primero mediante la indexación de arrays (*[]*). Éste único tipo se le asigna a la referencia *"dataTypes"* del metamodelo, lo cual hará que la transformación continúe por la regla *'mapDataType'* inicializando una única librería.
Con esto se consigue que para todos los tipos existentes en el script se cree una única metaclase *"DataType"* para almacenarlos a todos. La siguiente regla es por la que continuaría la transformación.

```
Rule 'mapDataTypes'
        from type t
        to DataType
        queries
                tps : //#type;
        mappings
                types = tps;
end_rule
```

En la regla *'mapDataType'* lo único que se hace es obtener todos los tipos, y asignarlos a la referencia multivaluada que contiene la metaclase *"DataType"*, que serán los tipos de los que se hace uso en el script DDL.

La siguiente regla mapea uno de los muchos tipos de la gramática a su clase en el metamodelo DDL. Como se ha comentado anteriormente, solamente se desea que exista un tipo de la misma clase en la librería, por ello y debido a que Gra2Mol por cada regla que ejecuta crea l elemento del metamodelo asociado con dicha regla, se tiene que realizar un filtro antes de ejecutar la regla. Dicho filtro comprueba antes de la transformación si existe una metaclase de dicho tipo en la librearía, en caso de existir no ejecuta la transformación, en caso contrario creará el nuevo tipo.

```
rule 'mapInteger'
        from type[unique]{EXACTO.eq("INTEGER")} t
        to Integer
        queries
        mappings
                name = t.EXACTO;
 end_rule
```

Esto se consigue indicando en la cláusula *'from'* entre corchetes la opción *'unique'*. Cuando se va a ejecutar la regla, la opción *'unique'* comprueba si el tipo que se va a transformar en la regla existe. En dicho caso la regla no se ejecuta. En caso contrario se realiza la transformación.

La siguiente regla muestra el mapeo de una foreign key a su metaclase. Lo único que habría que realizar sería rellenar los atributos de la metaclase con sus valores concretos, y decidir si

dicha foreign key está o no habilitada. Estará habilitada si no existe en todo el script ninguna sentencia del tipo *"ALTER TABLE DROP CONSTRAINT"* sobre dicha foreign key.

```
Rule 'mapFk'
        from fk fcfk
        to Fk
        queries
                cfk  : /fcfk/#column_list;
                crfk : /fcfk/#references_column_list;
                ctr  : /fcfk/#table_references;
                table : //#data_definition_statement/table{ID.eq(ctr.ID)};
                tab : //data_definition_statement/#table/fk{ID.eq(fcfk.ID)};
                aux : ///#disabled_constraint/table_references{ID.eq(tab.ID)};
                disabled : /#aux/name_constraint{ID.eq(fcfk.ID)};
        mappings
                nameFk = fcfk.ID;
                columnName = cfk.ID;
                columnReference = crfk.ID;
                references = table;
                if (disabled.hasResults) then
                        status = "DISABLED";
                else
                        status = "ENABLED";
                end_if
end_rule
```

Mediante las cuatro primeras sentencias incluidas en la cláusula *'queries'* obtenemos el nombre de la tabla a la que hace referencia la foreign key para buscar la tabla que tiene por nombre o identificador el incluido en la sentencia, ya que en vez de guardar el nombre de la tabla almacenamos la referencia a dicha metaclase. Las columnas de la tabla que formarán la foreign key, y la lista con los nombres de las columnas de la tabla referenciada a las que se referencia.

Se necesita saber si la foreign key ésta desactivada o no, para ello, se realiza lo siguiente:

1. Se busca la tabla que contiene la foreign key que se está tratando mediante la sentencia "tab : //data_definition_statement/#table/fk{ID.eq(fcfk.ID)};". Se busca aquel nodo *"table"*, hijo de sentencias *"data_definition_statement"*, que presente un nodo hijo del tipo *"fk"* cuyo identificador o nombre sea igual al identificador de la foreign key que se está tratando.

2. Se busca por todo el script, de manera recursiva, todos las sentencias *"ALTER TABLE DROP CONSTRAINT"* buscándose aquella sentencia que tenga como identificador de tabla el identificador de la tabla obtenida en el paso 1. Esto se realiza mediante la sentencia: "aux : ///#disabled_constraint/table_references{ID.eq(tab.ID)};". Se buscan aquellos nodos *"disabled_constraint"*, de manera recursiva, que tengan como hijo un

nodo *"table_reference"* cuyo identificador sea igual al identificador de la tabla encontrada en la paso primero.

3. Para todas las sentencias encontradas en el punto 2, se tratan solamente aquellas que hacen referencia a la foreign key de entrada de la regla. Se realiza mediante la siguiente sentencia: "disabled : /#aux/name_constraint{ID.eq(fcfk.ID)};". Se busca en los nodos *"disabled_constraint"* obtenidos en el paso segundo, aquellos que tengan como hijo un nodo *"name_constraint"* cuyo identificador sea igual al identificador de la foreign key que se está tratando.

Una vez realizados éstos tres pasos de la cláusula *'queries'* se comprueba dentro de los *'mappings'* si la variable que almacena todas las sentencias devueltas por el punto 3 tiene algún resultado. En caso de contener algún resultado se supone que dicha foreign key está desactivada y su estado se pone a *"DISABLED"*, en caso contrario se supone activada actualizando su estado a *"ENABLED"*.

Para determinar el estado (activado o desactivado) de las check constraints se realizan los mismos pasos que para las foreign key. La única diferencia entre ambas reglas es la información adicional que se incluye en cada metaclase, siendo diferente para la *Fk* que para la *Ck*. La siguiente regla muestra la transformación de la check constraint.

```
rule 'mapCk'
        from ck fcck
        to Ck
        queries
                cck : /#fcck;
                leck : /fcck/#log_expresion;
                table : //data_definition_statement/#table/ck{ID.eq(cck.ID)};
                aux : ///#disabled_constraint/table_references{ID.eq(table.ID)};
                disabled : /#aux/name_constraint{ID.eq(cck.ID)};
        mappings
                nameCk = cck.ID;
                valuesCk = leck;

                if (disabled.hasResults) then
                        status = "DISABLED";
                else
                        status = "ENABLED";
                end_if
end_rule
```

Como se puede observar en esta regla, se realizan los 3 pasos comentados anteriormente para el caso de las foreigns key. Se busca la tabla que contiene a la propia check constraint, una vez hallada se obtienen todas las sentencias *"ALTER TABLE DROP CONSTRAIN"* que afecten a la tabla encontrada, y de todas ellas obtenemos las que se refieran a la check constraint que se está tratando. Además se le asigna el nombre a la *"Ck"* y se le asigna el identificador incluido

en la sentencia *"check constraint"*. Se asignan también todas las comparaciones de la sentencia a la lista de referencias de la metaclase *"Ck"*.

La sentencia check constraint SQL puede contener dos tipos de restricciones. Las restricciones que usan los operadores de comparación, o las restricciones que usan la sintaxis *"column IN (value1, value2,…, valuen)"*. La metaclase *"ValuesCk"* no hace distinción alguna entre ambos tipos de restricciones. Únicamente presenta una lista de valores que debe cumplir la restricción, un operador de comparación, un operador lógico, y el nombre de la columna a la que aplicar la restricción. La idea es que si la restricción es del tipo *"column IN (value1, value2,…, valuen)"*, asignarle al operador de comparación el valor *"="*, ya que la columna debe ser igual a alguno de los valores que se incluyen entre paréntesis. Y al operador lógico se le asigna el valor nulo.
En caso de no ser de éste tipo, lo que se hace es un mapeo uno o uno.

```
rule 'mapValuesCk'
        from log_expresion le
        to ValuesCk
        queries
                column : /le/#column_ck;
                value_list : /le/#value_list;
                number : /#le{NUMBER.exists};
        mappings
                if (value_list.hasResults) then
                        comparator = "=";
                        value = removeQuotes value_list.CVALUE;
                else
                        comparator = le.COMPARATOR;
                        if (number.hasResults) then
                                value = le.NUMBER;
                        else
                                value = removeQuotes le.CVALUE;
                        end_if
                end_if

                logConjuntion = le.LOG_CONJ;
                columnName = column.ID;
end_rule
```

Como se puede apreciar en la siguiente sentencia de la gramática, una restricción de la check constraint está formada siempre por el nombre de la columna y luego tiene dos opciones. El comparador, el valor a comparar y puede o no estar la conjunción lógica. O el string *'IN'* Con la lista de valores que solamente puede tener la columna a la que se le aplica la restricción.

```
column_ck (COMPARATOR (NUMBER | CVALUE) LOG_CONJ? | 'IN' '(' (value_list)* ')')
```

En el apartado de *'queries'* de la regla *'mapValuesCk'* obtenemos el nombre de la columna, la lista de valores de la segunda opción y el valor de la primera sí existe. Posteriormente en el apartado de *'mappings'* lo primero a realizar es comprobar si la lista de valores contiene algún valor. En caso de contener algún valor, se trata de la segunda opción, por lo que se asigna al comparador el valor "=", se quitan las comillas simples (en caso de ser una cadena de caracteres) y la conjunción lógica tendrá valor nulo, ya que no existe en la sentencia.

En caso de que la lista de valores no tenga ningún valor, se tratará entonces de la primera opción. A continuación si el valor numérico obtenido en *'queries'* presenta algún valor es que el valor a comparar es un número y se asigna al valor de la restricción. En caso de no contener ningún valor será una cadena de caracteres, por lo que se le eliminan las comillas simples y se le asigna al valor de la restricción. Además se le asignan los valores correspondientes al comparador y a la conjunción lógica.

3.3 Extracción Modelo DML

Se ha usado Gra2Mol para obtener el modelo DML con los datos de la base de datos. Como se ha comentado anteriormente, sabemos que como entrada Gra2Mol necesita un metamodelo para almacenar la información, el cual conformará el modelo obtenido como salida, de una gramática ANTLR, un programa o fuente que conforme dicha gramática, y una serie de reglas de transformación que transformen el programa en el modelo de salida. Éste paso previo a la modernización o evolución del sistema se inicia a partir de un script DML (Data Manage Lenguage) que contiene toda la información almacenada en la base de datos y pretende crear un modelo para que, de forma independiente al gestor de base de datos que use el sistema, poder trabajar con dicha información.

Se pretende usar Gra2Mol para a partir de un script con sentencias SQL (obtenido de una base de datos mediante su exportación) que siga la sintaxis definida en una gramática, obtener un modelo (que represente el script) que conforme un metamodelo. Ésta transformación de gramática a modelo se consigue mediante la definición de un conjunto de reglas de Gra2Mol.

En la siguiente figura se puede apreciar la obtención a partir del script DML del modelo DML que conforma al metamodelo DML que se ha definido.

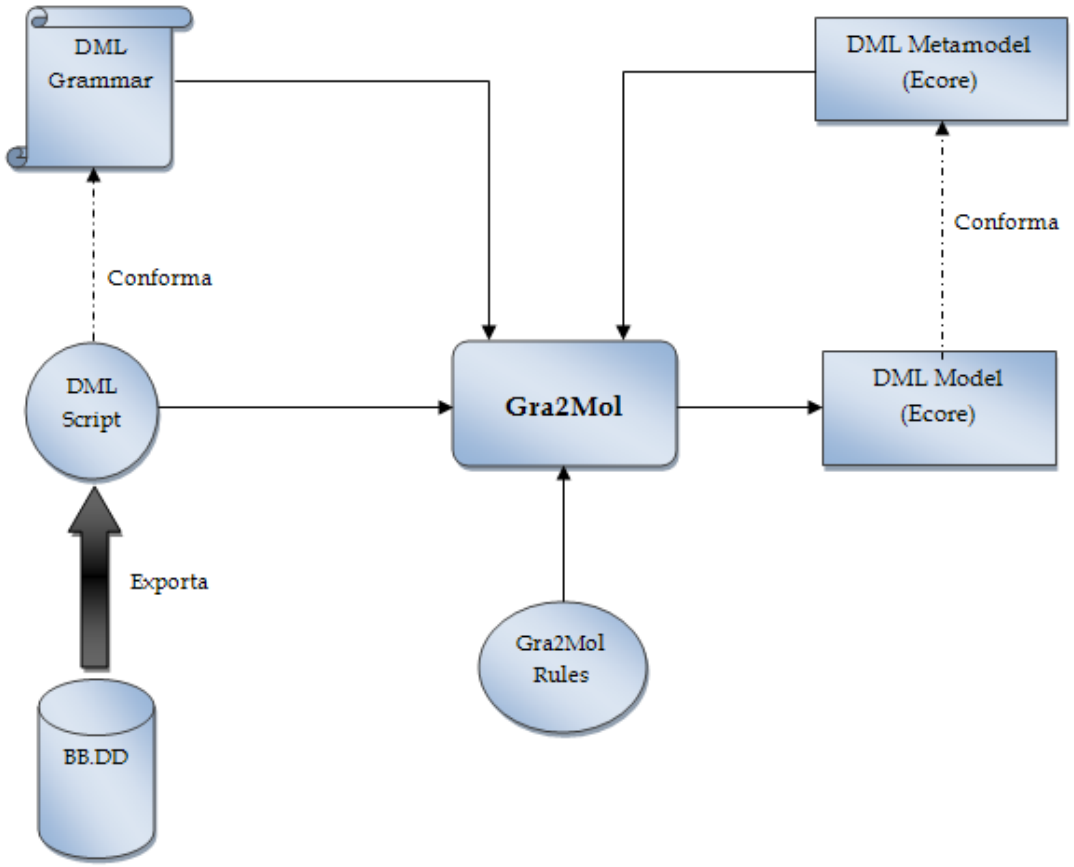

Figura 7: Obtención Modelo DML con Gra2Mol.

Se va ha comentar a continuación cada una de las partes que se pueden apreciar en la figura anterior.

Para llevar a cabo la obtención del modelo de datos sobre los activos del sistema, se partió de un script en el que se encontraban sentencias SQL para la inserción de datos en una BB.DD, sentencias del tipo:

INSERT INTO <nombre_tabla> (<nombre_columnas>[, ...n]) **VALUES** (<valores>[, …n])

A la hora de obtener el script DML desde la BB.DD que se esté manejando en el sistema, debemos incluir la opción para que al obtener las sentencias "INSERT INTO" nos incluya siempre las columnas de la tabla. Ya que según la sintaxis SQL del *ANSI SQL-92*, (que es la que se ha seguido en la realización de éste proyecto), la lista de columnas en la sentencia *"INSERT INTO"* es opcional pero, ya que son necesarias para saber si un valor específico se corresponde con una columna en concreto, para nosotros siempre serán obligatorias.

Para la creación de la gramática necesaria para la obtención del modelo de datos DML, se podrían haber usado herramientas como ANTLR. ANTLR es una herramienta que proporciona un framework para la construcción, interpretación, reconocimiento y transformación de una definición gramatical en un lenguaje destino [ANTLR]. Pero debido a la simplicidad de la sintaxis que la gramática debía soportar (la sentencia SQL de inserción del *ANSI SQL-92*) se optó por la realización manual de la gramática. Dicha gramática íntegra se puede ver en el apartado "Anexos".

Se trata de un simple mapeo de la sentencia de inserción en SQL. Tenemos un conjunto de sentencias *insertInto*. Cada sentencia *insertInto* contiene una lista con los identificadores o nombres de las columnas, y una lista con los valores de cada una de las columnas de la sentencia. Más abajo se definen los tipos básicos de la gramática como valores numéricos, alfa-numéricos y cadenas de caracteres.

Para la obtención del metamodelo DML final, se siguió la misma filosofía que en la obtención de la gramática. Mapear directamente la sentencia SQL a un metamodelo.

Como se puede apreciar en la siguiente figura, tendremos una metaclase para almacenar todas las sentencias de inserción en la base de datos. Para cada una de las sentencias que se indican en el script de inserción, tendremos una metaclase "insertInto".

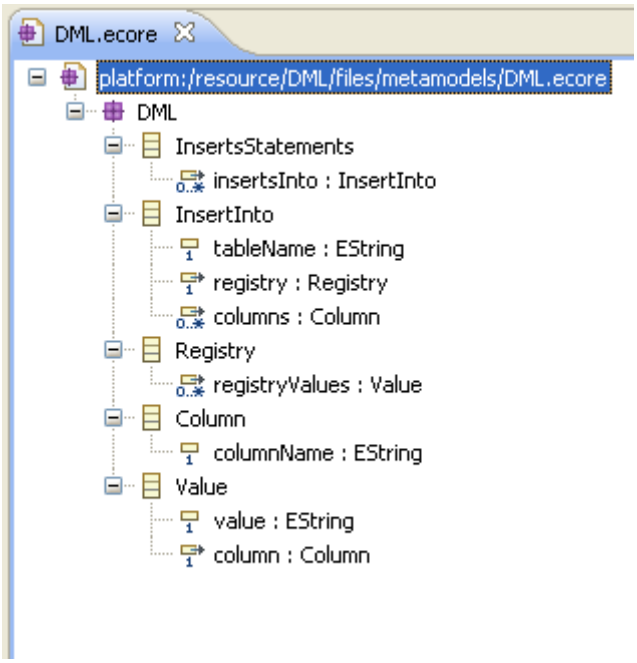

Figura 8: Metamodelo DML.

Se puede ver que dicha metaclase se mapea directamente con una tabla de la base de datos, por lo que se podría haber llamado "table", pero para que no diese lugar a confusiones se le asigno éste nombre. Tendremos una metaclase que refleje las columnas de la tabla incluidas en la sentencia "inserInto", y una metaclase para almacenar los registros que queremos insertar en la tabla. Para cada uno de los valores del registro a insertar tendremos una metaclase que almacena dicho valor.

En el esquema se pueden observar las siguientes metaclases:

- *"InsertsStatements"*: Se trata de la metaclase padre de todas las demás. Referencia a la sentencia en sentido general. Contendrá un conjunto de sentencias *InsertInto*, tantas como haya en el script DML que se obtiene de la base de datos.

- *"InsertInto"*: Ésta metaclase almacenará toda la información de la sentencia, en concreto tendrá un atributo para almacenar el nombre de la tabla y dos referencias una univaluada de registro (los valores que se incluirán en la respectiva tabla) y otra multivaluada de columnas (las columnas que contiene la tabla y que se van a rellenar con los valores de los registros).

- *"Registry"*: Esta metaclase representa a una fila de la tabla. Por lo que en realidad será una serie de valores, uno para cada una de las columnas que contiene la tabla. Esto se refleja en una referencia multivaluada de *Values*, una para cada columna.

- *"Column"*: Representa a una columna de la tabla. Contiene únicamente un identificador o nombre de la columna.

- *"Value"*: Esta metaclase es la que contendrá el valor de un atributo de una tabla. Presenta un atributo, el valor concreto para una columna, y una referencia a la columna que contiene dicho valor.

Una vez que tenemos el metamodelo definido, la gramática y el script con las sentencias SQL de la BB.DD, podemos proceder a la obtención del modelo mediante Gra2Mol. Para ello se han definido una serie de reglas de transformación. Dichas reglas se pueden ver íntegramente en el apartado "Anexos". A continuación se explicaran las dos reglas más extensas y complejas.

La regla "*mapInsertInto"* mapea la sentencia SQL al modelo DML. Para ello lo que se hace es obtener el identificador de la tabla en la que se tiene que realizar la inserción, obteniendo así el conjunto de nombres de las columnas de la tabla y la secuencia de valores que queremos insertar. La regla quedaría como sigue:

```
rule 'mapInsertInto'
        from insertInto insertInto
        to InsertInto
        queries
                col : /insertInto//#column;
                re : /insertInto/#registry;
        mappings
                tableName = insertInto.ID;
                columns = col;
                registry = re;
end_rule
```

Esta regla, llamada *"mapInsertInto"*, crea una metaclase *"InsertInto"* del metamodelo para cualquier sentencia de la gramática del tipo *"insertInto"*. Asignando al atributo *"tableName"* de la metaclase el valor del nodo hoja *"ID"* de la sentencia *"insertInto"*, mostrada a continuación. Además en el apartado *"queries"* de la regla Gra2Mol se obtiene la colección de columnas que se mapean a las referencias de las columnas de la metaclase del metamodelo y el registro que contendrá todos los valores que se incluyen en la tabla.

```
insertInto
  : 'INSERT INTO' ID ('(')? (column)* (')')? 'VALUES' registry ';' ;
```

Otra regla más compleja que la anterior es la regla *"mapValue"*. En ésta regla se extraen los valores del registro y se asocian a una columna de la tabla, incluida en la sentencia. Pero al hacer esto surgió un problema, ya que necesitábamos almacenar para un valor en concreto la columna específica de la tabla a la que hacía referencia.
Como ya se ha dicho anteriormente, cada una de las metaclases *"value"* tiene una referencia a la columna a la que le da valor, por lo que la inserción de los valores de un registro debe ir en concordancia con la asignación, a la referencia *"column",* del valor introducido.

Realizar ésta tarea mediante reglas de Gra2Mol era imposible. Pero Gra2MoL incorpora un mecanismo de extensibilidad que permite, desde la sección de los *"mappings"*, llamar a funciones externas definidas en clases auxiliares. Es la *"vía de escape"* para cuando las consultas no ofrecen la potencia necesaria.

La secuencia de columnas que se incluye en la sentencia SQL de inserción es del mismo tamaño que la secuencia de valores que se quieren insertar en la tabla.Ademas, la correspondencia es inmediata, es decir, a la columna de la posición i-ésima en la secuencia de columnas le corresponderá el valor de la posición i-ésima en la secuencia de valores a insertar. La idea a usar con la *"vía de escape"* es la de obtener la posición de un valor en concreto dentro de su secuencia de valores, para así obtener la columna a la que hace referencia dicho valor, únicamente devolviendo la columna que ocupa la misma posición dentro de su secuencia de columnas.

La *"vía de escape"* básicamente consiste en definir un recorrido en el CST mediante una clase Java. Para ello hay que tener en cuenta una serie de cuestiones técnicas.

Las clases auxiliares deben extender a la clase *"MappingExtensionPoint"*, la cual ofrece un conjunto de métodos que actúan de fachada para acceder a las funciones de Gra2MoL. Básicamente se debe implementar el método *"execute"* y luego están disponibles los métodos *"getParam"* y *"returnNode"* para obtener los parámetros y devolver un Nodo del CST, respectivamente. Estas clases auxiliares deben estar localizadas en la carpeta *"extensions"*, que por defecto debe ser hermana de *"metamodels"* y *"src"* (es decir colgar de la carpeta *"files"*), aunque puede configurarse en el *"build.xml"*. Cuando la carpeta *"extensions"* contiene alguna clase, ésta es compilada y se ejecuta junto con la transformación de Gra2Mol.

Puede producirse un error en la compilación debido a que la carpeta *"extensions"* sólo se compila si antes existe la carpeta *"bin"* (creada en una compilación anterior), dando un error si se ésta generando todo desde cero. Para solventar esto solamente hay que cambiar el target *"build"* del fichero *"build.xml"*. Quedando como sigue:

```
<target name="build" depends="checkEGrammarExists, phaseOne,
checkExtensions, yesExtension, noExtension, phaseTwo">
   <tstamp/>
</target>
```

La forma de llamar a estas clases auxiliares desde los *"mappings"* de una regla es utilizando la palabra clave *"ext"* seguida del nombre de la clase auxiliar y entre paréntesis los parámetros.

La regla *"mapValue"* se muestra a continuación.

```
rule 'mapValue'
        from value v
        to Value
        queries
                id : /#v{ID.exists};
```

```
                cvalue : /#v{CVALUE.exists};
                number : /#v{NUMBER.exists};

                insert : //#insertInto//value{this.check(v)};
        mappings
                if (id.hasResults) then
                        value = v.ID;
                end_if
                if (cvalue.hasResults) then
                        value = removeQuotes v.CVALUE;
                end_if
                if (number.hasResults) then
                        value = v.NUMBER;
                end_if

                column = ext InferColumnFromValue(v,insert);
end_rule
```

Según se ha definido en la gramática, un valor puede ser un identificador (una letra, mayúscula o minúscula, seguida de un número o de una letra), un número, o una cadena de caracteres. Como solamente puede ser de un tipo al mismo tiempo, lo que realizamos mediante la porción de código incluida en *"queries"* es comprobar si esa hoja de la gramática existe o no en la sentencia, es decir, si en la sentencia lo que hay es un identificador, un número o una cadena de caracteres. El método *"exists"* devolverá el número de de veces que esa hoja existe en la sentencia, por lo que en nuestro caso solamente una de las tres variables (id, cvalue, number) podrá ser 1, el resto serán 0.

Posteriormente en la porción de *"mappings"* se comprueba mediante sentencias *"if"* cuál de las tres variables ha tenido algún resultado, siendo ésta asignada al atributo *"value"* del metamodelo.

La clase *"InferColumnFromValue.java"* creada para la asignación de la columna al valor correspondiente puede verse íntegramente en el apartado "Anexos". Mediante el mecanismo de extensibilidad de Gra2Mol, realizamos lo siguiente: la clase recibe como parámetros el nodo *"insertInto"* que contiene al nodo *"value"* así como dicho elemento *"value"* con el que estamos trabajando. Entonces obtenemos el nodo *"registry"* incluido en el nodo *"insertInto"* pasado como parámetro. Una vez obtenido, recorremos sus elementos de tipo *"value"* para descubrir la posición que ocupa nuestro elemento *"value"*, pasado como parámetro, en la secuencia de valores a insertar por parte del nodo *"registry"*. Finalmente, conociendo dicha posición, recuperamos el elemento *"column"* del elemento *"insertInto".* Por lo que como resultado nuestra función auxiliar devolverá la columna que ocupa la misma posición que el valor dentro de su lista de valores, dentro de la lista de columnas.

El resto de reglas no se explican ya que solamente se trata de un simple mapeo de la gramática al metamodelo, por lo que carecen de complejidad y no necesitan que se profundice en ellas.

4 Integración Modelos DDL y DML

En está etapa se pretende integrar los modelos obtenidos en la etapa anterior, para tener en un mismo modelo toda la información del sistema, puesto que resulta conveniente tenerlos juntos para su modernización. Esta integración se realizará mediante RubyTL.

4.1 Introducción a RubyTL

RubyTL [RubyTL] es un lenguaje de transformación basado en reglas que ha sido implementado como un DSL embebido dentro del lenguaje Ruby. Es un lenguaje híbrido, cuya parte declarativa está basada en reglas y bindings, mientras que la parte imperativa viene dada por los constructores proporcionados por el propio lenguaje Ruby.

Se puede apreciar en la siguiente figura la sintaxis abstracta de RubyTL expresada como un metamodelo.

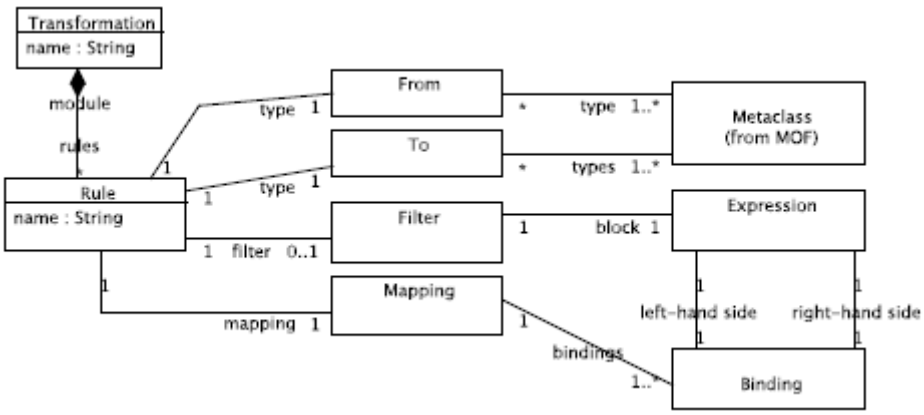

Figura 9: Sintaxis Abstracta de RubyTL.

Como se puede observar, una definición de una transformación esta formada por un conjunto de reglas de transformación empaquetadas en un módulo de transformación. Cada regla presenta un nombre y cuatro partes:

- *'from'*: Especifica la metaclase origen, que pertenece a un metamodelo origen.
- *'to'*: Especifica una o más metaclases destino, correspondientes a un metamodelo destino.
- *'filter'*: Especifica una condición sobre el elemento origen, así que la regla que presente un filtro solamente se lanzará si se satisface la condición; éste filtro es opcional y la regla siempre se lanzará si no presenta ningún filtro.
- *'mapping'*: Especifica las relaciones entre los elementos del modelo origen y destino. Estas relaciones se pueden expresar de forma declarativa, como un conjunto de *bindings*, o de manera imperativa usando Ruby. Se recomienda usar *mappings*

declarativos y usar código imperativo mediante Ruby sólo cuando sea difícil expresar la transformación declarativamente.

El bloque de código de la parte *mapping* de una regla tiene un elemento origen y un elemento destino como parámetros. El primer parámetro es un elemento origen que conforma a la metaclase especificada en la parte *from*. El resto de parámetros corresponden a elementos destino que conforman con las metaclases destino especificadas en la parte *to* (en el mismo orden).

En la siguiente figura se puede observar la sintaxis concreta de RubyTL.

```
module <module-name> do

  rule <rule-name> do
    from <source-metaclass>
    to   {target-metaclass}

    filter do |source_element|
      <expression>
    end

    mapping do |<source_element>, {target_element}|
      {bindings}
      # bindings has the form:
      #     target_element.property = source_element.property
    end
  end

  # one or more rules
end
```

Figura 10: Sintaxis Concreta de RubyTL. Mencionar que '<>' significa una ocurrencia y '{}' una o más ocurrencias.

Está determinada por el hecho de que el lenguaje es implementado como un DSL embebido dentro del lenguaje Ruby (por ejemplo, véase el uso de *'do – end'* para escribir un bloque de código). Se ha usado una técnica '*well-known'* para implementar DSLs embebidos en Ruby, esto quiere decir que cada palabra clave del lenguaje se mapea a una llamada de un método y las estructuras anidadas se mapean a bloques de código parametrizados.

Una regla es definida por el método *rule* que espera dos parámetros: el nombre de la regla en forma de string y un bloque de código cuya estructura debe conformar la sintaxis concreta del elemente *rule*.

Las partes *'from'* y *'to'* de una regla se definen por dos métodos *from* y *to*. Ambos métodos esperan como parámetro una clase, en concreto se tratará de la clase del elemento del metamodelo origen y la clase del elemento del metamodelo destino, respectivamente.

La parte *'filter'* de una regla se define mediante el método *filter* que presenta como parámetro de entrada un bloque con un elemento de una metaclase origen. El filtro se evaluará a

verdadero si el bloque pasado como parámetro es verdadero, en otro caso se devolverá falso. Comentar que en Ruby el resultado de la última expresión evaluada en un bloque es tomado como el valor de retorno de dicho bloque.

La parte *'mapping'* de una regla se define por el método *mapping* que espera como parámetro de entrada un bloque que contendrá un elemento origen y uno o más elementos destino. Dicho bloque consta de un conjunto de *bindings* para implementar la regla si se sigue un estilo declarativo, o de cualquier otro código Ruby si se sigue un estilo imperativo. Un *binding*, que establece una relación entre los elementos origen y destino, se ha implementado sobre-escribiendo el operador de asignación de Ruby.

Puede observarse que la sintaxis de RubyTL tiene algunas peculiaridades. Esto es debido a que RubyTL ha sido creado como un lenguaje embebido en Ruby, por lo que su sintaxis está, hasta cierto punto, determinada por la sintaxis del propio Ruby. Así, las estructuras del lenguaje que tienen elementos internos, como por ejemplo *rule*, *filter* o *mapping*, utilizan bloques de código como mecanismo de anidamiento. Un bloque de código Ruby es un trozo de código encerrado entre *'do – end'* o *'{ }'*. Los parámetros del bloque se especifican con *'| |'*.

Es conveniente tener en cuenta la peculiaridad de que RubyTL es embebido, ya que supone tanto una ventaja como una desventaja.

- Existen restricciones relativas impuestas por la naturaleza del lenguaje. La restricción relativa a la sintaxis más destacable es que los nombres de las reglas deben escribirse entre comillas (simples o dobles), y la forma de especificar los bloques de código.

- Esta característica puede ser aprovechada para hacer uso de todas las construcciones imperativas de Ruby y de sus librerías, esto es, cualquier facilidad o librería de Ruby puede ser utilizada en RubyTL. Por ejemplo, es posible utilizar expresiones regulares dentro de una transformación.

4.1.1 Decoradores

RubyTL permite extender las metaclases de un metamodelo con métodos de utilidad para cierta transformación. Esto es muy útil para hacer las transformaciones más legibles, normalmente factorizando el código de navegación en el decorador.

Un decorador está compuesto de uno o más métodos Ruby (especificados con *'def methodname – end'*). Los métodos podrán ser invocados por cualquier instancia de la metaclase o de cualquiera de sus subclases. Se pueden utilizar variables de instancia dentro de los decoradores pero no es una práctica recomendada.

Es una práctica recomendada escribir los decoradores antes de las reglas de transformación. También, es posible factorizar los decoradores en librerías. Las librerías se suelen guardar en el directorio *helpers*. Una librería es un fichero que contiene decoradores, y cuya extensión es *'.rb'*. Se utiliza la sentencia *'use library'* para cargar una librería desde una transformación. La

sentencia *'use library'* toma una cadena que especifica la ruta de la librería como una URI. En particular, la URI *'helper://'* busca directamente en la carpeta *helpers* del proyecto.

4.1.2 Bindings

El propósito de un *binding* es el de especificar una relación entre un elemento del modelo origen y un elemento del modelo destino. Se escribe como una asignación con el formato '*target.property = source-expresion'*, donde:

- *'source-expresion'*: es una expresión Ruby cuyo resultado es un elemento o una colección de elementos pertenecientes al modelo origen. Por tanto, el tipo del lado derecho de la asignación es dado por el tipo (metaclase) que presenta *source-expresion*. Si el resultado de la expresión es una colección, RubyTL iterará automáticamente sobre la misma al asignarla a la parte izquierda.
- *'target'*: es un parámetro del bloque de código del *'mapping'* que representa un elemento destino a ser creado. Su tipo será el indicado en la parte *'to'* de la regla.
- *'property'*: deber ser una propiedad del elemento destino previamente creado. El tipo del lado izquierdo de la asignación es dado por el tipo de tal *property*.

Es importante remarcar que el operador de *binding* (esto es, =) no tiene a la misma semántica que la asignación normal. En particular, si la propiedad destino es multivaluada (es una colección), entonces la semántica es *"transformar y añadir"*.

4.1.3 Expresions

En RubyTL se utilizan expresiones propias del lenguaje Ruby para escribir filtros, *bindings* y los métodos de los decoradores.

Es muy común en los lenguajes de transformación implementar alguna variante del lenguaje de consulta OCL como lenguaje de navegación y para expresar condiciones. RubyTL en cambio no implementa OCL, sino que simplemente utiliza las facilidades de Ruby y de su librería de colecciones para manipular las colecciones.

La librería de colecciones de Ruby tiene un gran poder expresivo gracias al uso de bloques de código y a la existencia de iteradores internos.

4.1.4 Reglas

En RubyTL hay varios tipos de reglas, y cada una proporciona una funcionalidad distinta para tratar ciertos problemas de transformación. La siguiente figura muestra sus relaciones mediante una jerarquía de herencia. También se muestra la palabra clave que se usa para declarar una regla de ese tipo.

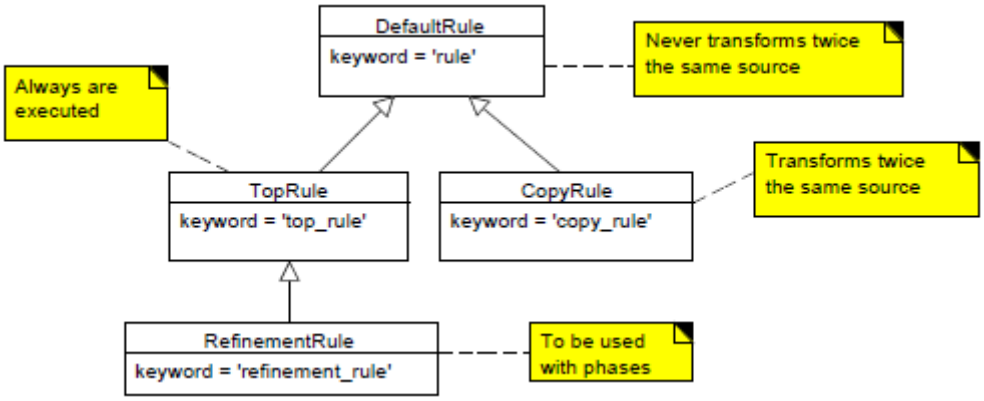

Figura 11: Jerarquía de herencia de los distintos tipos de reglas de RubyTL.

4.1.4.1 Default rule

La *regla por defecto* se nombra simplemente con rule. Tienen *from*, *to*, *filter* y *mapping* como se describe arriba. Una regla por defecto se ejecuta solo en las dos siguientes situaciones:

- Cuando la regla conforma a un *binding* y su filtro es satisfecho por la instancia origen. Se dice que la regla se ejecuta para resolver el *binding*.
- Cuando es la primera regla de una transformación y la transformación no tiene ninguna otra regla top.

La principal característica diferenciadora de este tipo de regla es que la misma regla nunca transforma dos veces el mismo elemento origen. Esto significa que si la regla es llamada por segunda vez (para resolver otro *binding*), se devuelve el resultado de la aplicación anterior.

Éste tipo de regla resuelve problemas de recursividad no transformando dos veces el mismo elemento fuente.

4.1.4.2 Top rule

Una *regla top* es una especialización del tipo de regla por defecto. Al igual que ésta, nunca transforma dos veces el mismo elemento origen. Sin embargo, siempre se ejecuta para cada instancia de la metaclase origen.

Normalmente existe una regla top en cada transformación, a partir de la cual se inicia la misma, mientras que el resto de las reglas son *"no top"*. A partir de la ejecución de una regla top el resto de reglas se van ejecutando, como resultado de la evaluación de *bindings*. Si hay varias reglas top, se ejecutan en el mismo orden que aparecen en la definición de transformación.

4.1.4.3 Copy rule

Una *regla de copia* se comporta como una regla por defecto, excepto que sí que puede transformar más de una vez el mismo elemento fuente. Esto significa que si se aplica una misma regla de copia varias veces sobre el mismo elemento origen, se crearán varios elementos destino, uno por cada aplicación.

Se debe usar esta regla solo cuando no sea posible utilizar reglas por defecto, ya que puede causar recursión infinita si hay ciclos en el grafo de llamadas.

4.1.5 Proceso de transformación

Como se ha comentado anteriormente, RubyTL ha sido implementado como un DSL interno de Ruby. Esta opción de diseño significa que estamos delegando en el intérprete de Ruby para parsear y evaluar una trasformación. El motor de transformación y el parser XMI son implementados también en Ruby.

En la siguiente figura se puede apreciar el proceso de transformación mediante un diagrama, y muestra los componentes y los datos que son involucrados en todo el proceso de transformación.

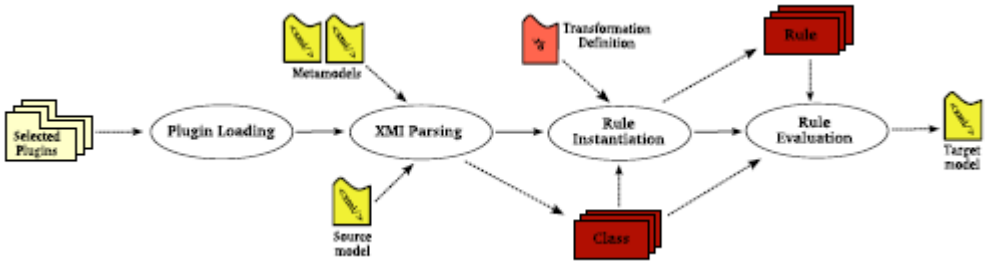

Figura 12: Ejecución de un proceso de ejecución en RubyTL.

Los pasos son los siguientes:

1. Desde que RubyTL presenta un diseño de plugin, el primer paso es el de cargar los plugins necesarios para configurar el lenguaje con ciertas características.

2. El metamodelo fuente, el metamodelo destino y el modelo fuente son ficheros XMI. El parser de Ruby leerá estas entradas y un conjunto de clases de Ruby serán generadas y cargadas en el intérprete de Ruby. Estas clases corresponden a las clases definidas en los metamodelos fuente y destino.

3. Una vez que los metamodelos han sido cargados, la transformación es leída por el intérprete de Ruby, el cual crea un conjunto de reglas. Esas reglas se usarán en el proceso de transformación para llevarla a cabo.

4. La salida del proceso de transformación es un fichero XMI que contiene el modelo destino conforme a un metamodelo destino.

4.1.6 Transformación Model-to-Code

Las transformaciones modelo a código se basan en plantillas o *templates*. Además, un fichero de configuración especial, llamado *'2code'*, permite hacer el *mapping* entre las *templates* y los nombres de los ficheros a ser creados. Éste tipo de ficheros nos permiten iterar sobre un modelo y seleccionar los elementos del modelo que serán transformados a código.

El fichero *'2code'* presenta la siguiente estructura:

```
main do
  compose_file 'file.sql' do |file|
    apply_template 'templates/template.rtemplate', :var => var
  end
end
```

Cada una de las partes se explicará a continuación:

- La palabra clave *'main'* especifica el punto de entrada de la transformación, y siempre debe existir.

- El método *'compose_file'* permite componer un fichero con el contenido que se obtiene como resultado de aplicar varias plantillas de transformación.

- La palabra clave *'apply_template'* se usa para especificar *mappings*. Presenta dos parámetros:

    - El nombre de la plantilla a ser aplicada. El resultado de aplicar dicha *template* será almacenado en el fichero que hay definido en el apartado *'compose_file'*.

- *Mapping* de variables de entrada de la *template*. Son especificadas usando símbolos y variables. Este *mapping* le proporciona a la *template* el contexto necesario para realizar la generación de código.

Las *templates* son ficheros con la extensión *'rtemplate'*. Siguen la misma estructura que los ficheros JSP. Es decir, cualquier contenido de la *template* que no se enmarque dentro de los delimitadores '<% %>' será traducido como texto plano en el fichero de salida obtenido al aplicar dicha *template*, y cualquier contenido dentro de los delimitadores será traducido como código en Ruby y será ejecutado.

Todo el código de Ruby que se usa en la plantilla debe estar enmarcado dentro de los delimitadores indicados anteriormente. Dichos delimitadores presenta distintas opciones:

- *'<%= %>'*: es usado para ejecutar código de Ruby y mostrar en la salida (la salida será el fichero que queremos generar) el resultado obtenido de la ejecución del código como un string.
- *'<% %>'*: se usa para escribir partes de código de Ruby que no deben ser parte de la salida, es decir, que su resultado no aparecerá en el fichero generado.
- *'<% -%>'*: con la opción '-' se omiten los saltos de línea.

4.2 Integración DML+DDL

Una vez que se han obtenido los modelos de datos DDL y DML se procede a unir ambos modelos.

Para llevar a cabo la fase de corrección o modernización es conveniente disponer de la información suministrada por ambos modelos de manera relacionada (tener dicha información separada es ineficiente ya que tendríamos que estar navegando de un modelo a otro, por lo que tener la información en un único modelo es más eficiente). Éste es el motivo por el que se unen ambos modelos en uno sólo.

Se ha hecho uso de RubyTL para transformar los modelos DML y DDL en un tercer modelo llamado DML_DDL, que conformará el metamodelo DML_DDL. El metamodelo DML_DDL no es más que el metamodelo DDL, comentado en el apartado DDL, pero ampliado para que pueda soportar la información almacenada en el metamodelo DML.

Antes de continuar, se dará una visión global de lo que se espera realizar en ésta fase.

Gracias al uso de RubyTL, que permite realizar transformaciones de un modelo de entrada a otro de salida definiendo una serie de reglas de transformación, se han transformado los modelos obtenidos mediante Gra2Mol en un único modelo. Más que una trasformación se podría decir que es una unión. Para ello RubyTL necesita de los modelos obtenidos en Gra2Mol, de los metamodelos que conforman estos modelos y del metamodelo que deberá conformar el modelo obtenido al terminar la transformación. Además se ha tenido que definir una serie de reglas de transformación. Todo esto se puede ver en el siguiente esquema.

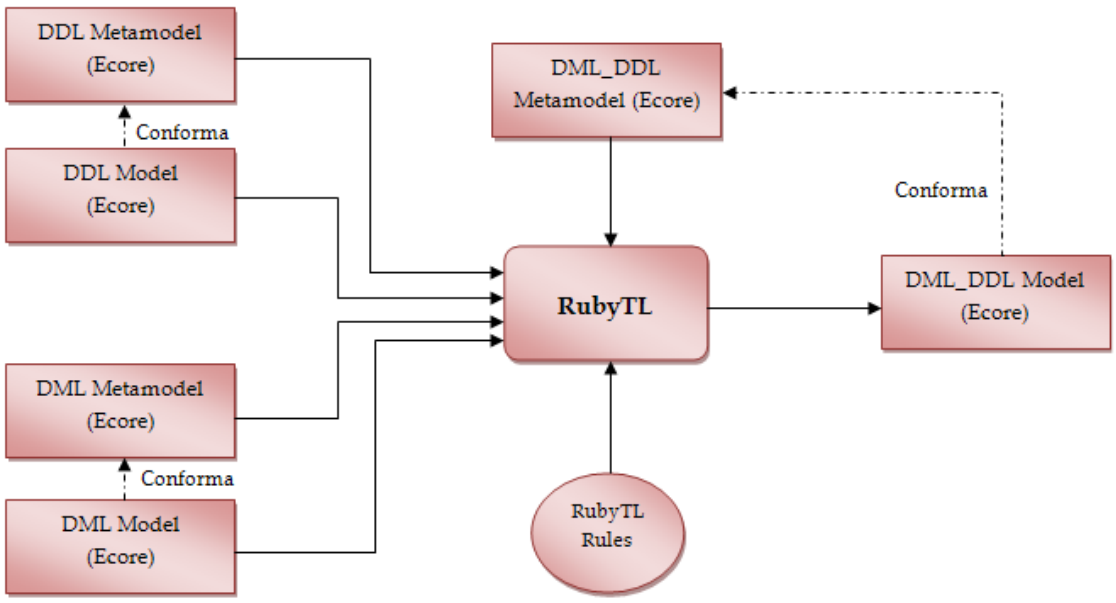

Figura 13: Obtención modelo DML_DDL con RubyTL.

Como se ha comentado, el metamodelo DML_DDL, no es más que una extensión del metamodelo DDL con las estructuras necesarias para soportar la inclusión de los datos almacenados en el modelo DML. Como vimos en el apartado DML, este modelo contenía una metaclase llamada "InsertInto" que contenía una referencia a la metaclase "Registry" para contener el registro o fila que se quería insertar. Dicha metaclase "Registry" contenía un conjunto de referencias a la metaclase "Value" para almacenar cada uno de los valores del registro. "Value" almacenaba el valor propiamente dicho y contenía una referencia a la columna a la que le daba valor. Todo esto se mapea de la siguiente manera en el nuevo metamodelo:

- La metaclase "InsertInto" en realidad ofrece la misma funcionalidad que la metaclase "Table" del metamodelo DDL. Por lo que dicha metaclase no es exportada al nuevo metamodelo, pero sí sus referencias y atributos. Ya que una tabla deberá tener un conjunto de registros, a la metaclase "Table" hay que añadirle un conjunto de referencias a los registros que almacena, es decir, un conjunto de referencias a la metaclase "Registry". El atributo "tableName" ya existe en la metaclase "Table" del metamodelo DDL, al igual que las referencias a la metaclase "Column" por lo que no es necesario exportarlas.

- La metaclase "Registry" hay que exportarla tal cual al nuevo metamodelo. Por lo que habrá que exportar también la metaclase para almacenar los valores del registro, es decir, la metaclase "Value".

- La metaclase "Value" se ha exportado tal cual al nuevo metamodelo. Pero con una diferencia y es que como en el metamodelo DDL existe una metaclase llamada "Column" que contiene información más relevante que la metaclase del mismo nombre del metamodelo DML, se hará uso de la primera metaclase. Por lo que la metaclase "Column" del metamodelo es desechada y la referencia de la metaclase "Value" apuntará a la metaclase "Column" heredada del metamodelo DDL.

El metamodelo DML_DDL quedará de la siguiente manera. En la siguiente figura solamente se muestran expandidas las nuevas metaclases y la metaclase ("Table") que se ha modificado con respecto al metamodelo DDL.

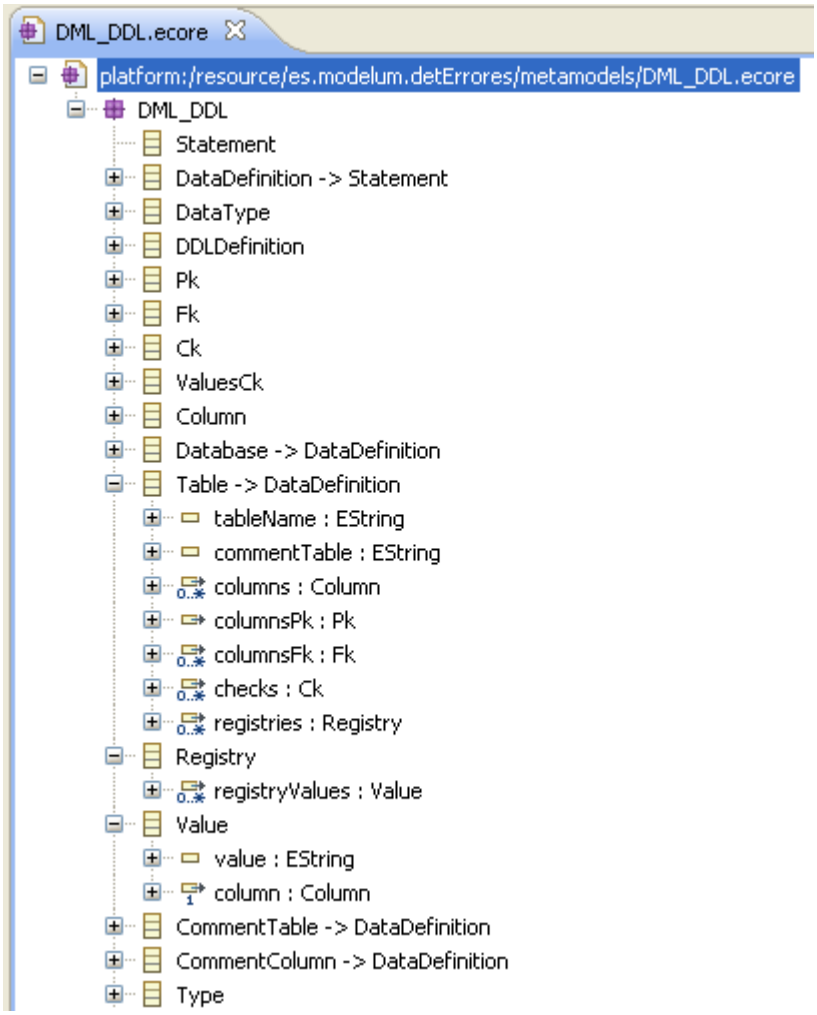

Figura 14: Metamodelo DML_DDL.

Además de los metamodelos DML y DDL, de los modelos DML y DDL obtenidos mediante Gra2Mol y del metamodelo DML_DDL comentado en el párrafo anterior, RubyTL necesita un conjunto de reglas de transformación. Dichas reglas pueden verse íntegramente en el apartado "Anexos". A continuación se comentarán las más relevantes.

La mayoría de las reglas son simples mapeos de una metaclase del metamodelo DDL a una metaclase del metamodelo DML_DDL, por lo que carecen de importancia explicativa al ser sencillas en su compresión. Sin embargo, se va a proceder a comentar las reglas que crearán los nuevos elementos de las nuevas metaclases introducidas en el metamodelo DML_DDL.

En el modelo DML obtenido al ejecutar Gra2Mol obtenemos una serie de elementos de la metaclase "InsertInto". Tantos como sentencias SQL "INSERTO INTO" existan en el script obtenido de la base de datos. Por lo que la estructura de dicho modelo no será la correcta, ya que si para la tabla "A" existen dos sentencias de inserción, en el modelo DML obtenido existirán dos metaclases para la tabla "A" cada una con registro. Pero realmente debería ser que la tabla "A" apareciese una única vez en el modelo de datos y tuviese dos registros.

Además, hay que tener en cuenta que los registros llevan asociados un conjunto de valores que referencian a una columna en concreto. La asignación de los valores al nuevo metamodelo es inmediato, un simple mapeo. El problema reside en la asignación de la referencia de las columnas.

La referencia que contenían los elementos de la metaclase "Value" era hacía las columnas del modelo DML, por lo que dichas referencias no servirán. Habrá que establecer dicha relación de nuevo. Esto implica la búsqueda por todo el modelo DML_DDL de las columnas a las que referenciaban antes los elementos de la metaclase "Value" para establecer de nuevo la referencia de los elementos de la metaclase "Value" del metamodelo DML_DDL que se crearán. Esto se debe a que como se ha comentado anteriormente las metaclases "Column" del metamodelo DML son despreciadas, por lo que no existirán en el nuevo metamodelo DML_DDL. Solamente existirán las metaclases de las columnas del metamodelo DDL que se han exportado al nuevo modelo DML_DDL. Esto es lo que se intenta obtener el unir el modelo DML con el DDL en un único modelo.

Las reglas que realizan todo lo comentado anteriormente son las siguientes:

```
rule 'Table2Table' do
  from DDL::Table
  to DML_DDL::Table
  mapping do |ddl, dml_ddl|
    dml_ddl.tableName = ddl.tableName
    dml_ddl.columns = ddl.columns
    dml_ddl.columnsPk = ddl.columnsPk
    dml_ddl.columnsFk = ddl.columnsFk
    dml_ddl.checks = ddl.checks
    dml_ddl.commentTable = ddl.commentTable

    dml_ddl.registries = DML::InsertInto.all_objects.select{
           |dml_ddl| dml_ddl.tableName == ddl.tableName}.map{
           |dml_ddl| dml_ddl.registry}.flatten
  end
end

rule 'Registry2Registry' do
  from DML::Registry
  to DML_DDL::Registry
  mapping do |dml, dml_ddl|
    dml_ddl.registryValues = dml.registryValues
  end
end

rule 'Value2Value' do
  from DML::Value
  to DML_DDL::Value
  mapping do |dml, dml_ddl|
    dml_ddl.value = dml.value
    dml_ddl.column =
     DML_DDL::Table.all_objects.select{|dml_ddl| dml_ddl.tableName ==
                dml.column.getInsertInto.tableName}.map{|dml_ddl|
                dml_ddl.columns}.flatten.find{|dml_ddl|
                dml_ddl.columnName == dml.column.columnName}
  end
end
```

En la regla *'Table2Table'* para toda tabla del modelo DDL '*from DDL::Table'* se obtiene la misma tabla en el modelo DML_DDL *'to DML_DDL::Table'*. Como se puede observar las seis primeras líneas del apartado *'mapping'* son asignaciones directas ya que en las metaclases de ambos metamodelos existirán éstos mismos seis atributos. Pero sin embargo el atributo "registries" es nuevo en la metaclase "Table" del metamodelo DML_DDL que se pretende crear. Para asignarle a una tabla todos los registros que posee en el modelo DML, lo que se hace es lo siguiente:

1. Se seleccionan todos los elementos "InsertInto" del modelo DML mediante el método *'all_objects'*. Para todos ellos, mediante el método *'select'*, seleccionamos aquellos que tienen el atributo "tableName" con el mismo valor que la tabla del modelo DDL que se quiere transformar.

2. Una vez que tenemos todos los "InsertInto" que corresponden a la tabla que queremos transformar se cogen todos los registros, atributo "registry" de la tabla, y se forma con ellos un conjunto mediante el método *'map'* de Ruby. Dicho conjunto se aplana en una sola dimensión mediante el método *'flatten'*.

3. Dicho conjunto será el que se asigne al atributo "registries" de la nueva tabla creada en el modelo DML_DDL.

Todo lo comentado en los tres puntos anteriores se realiza en la siguiente línea:

```
dml_ddl.registries = DML::InsertInto.all_objects.select{|dml_ddl|
   dml_ddl.tableName == ddl.tableName}.map{|dml_ddl| dml_ddl.registry}.flatten
```

La asignación de todos los registros encontrados a la referencia "registries" de la nueva tabla hace que, para cada uno de los registros encontrados, se ejecute la regla *'Registry2Registry'*. En ella se realiza la asignación de todos los valores que el registro del modelo DML posee al registro del modelo DML_DDL que se quiere crear. Como se puede ver en dicha regla, la parte *'from'* es el registro del modelo DML, *'DML::Registry'* y la parte 'to' es el registro del modelo DML_DDL que se quiere crear *'DML_DDL::Registry'*.

Como pasaba antes, para todo valor de cada uno de los registros que se asigne en la regla *'Registry2Registry'* se ejecutará la regla *'Value2Value'*.

En la regla *'Value2Value'* se realiza la asignación directa del valor contenido en el elemento "Value" del modelo DML al nuevo elemento "Value" del modelo DML_DDL. Además se realiza la búsqueda de la columna perteneciente al modelo DML_DDL que se corresponde a la columna perteneciente al modelo DML que referenciaba "Value" en su referencia "column".

Se creo un decorador para hacer más legible la sentencia de búsqueda. Dicho decorador se encuentra al principio del fichero y realiza lo siguiente:

```
decorator DML::Column do
  def getInsertInto
    DML::InsertInto.all_objects.find{|dml_ddl| dml_ddl.columns.include?(self)}
  end
end
```

Sobre la metaclase "Column" del metamodelo DML, como se indica en la primera línea, se define un método llamado *'getInsertInto'* al que cualquier elemento del modelo DML que tenga como metaclase a "Column" podrá llamar mediante la notación *"."*. Dicho método realiza lo siguiente: selecciona todos los elementos pertenecientes a la metaclase "InsertInto" del modelo DML y para todos ellos busca aquel que cumple la condición encerrada entre corchetes. Es decir, mediante el método de Ruby *'find'* (que devuelve la primera ocurrencia) nos quedamos con aquel "InsertInto" que tiene en su lista de columnas a la columna que ejecuta el decorador. La palabra clave *'self'* se refiere al elemento que ejecuta el decorador, en éste caso a la columna que queremos encontrar que será la que ejecute el decorador.

La sentencia de búsqueda en la regla *'Value2Value'* es la siguiente:

```
dml_ddl.column = DML_DDL::Table.all_objects.select{|dml_ddl|
  dml_ddl.tableName == dml.column.getInsertInto.tableName}.map{|dml_ddl|
   dml_ddl.columns}.flatten.find{|dml_ddl| dml_ddl.columnName ==
    dml.column.columnName}
```

Seleccionamos todas las tablas del modelo DML_DDL que existen mediante la sentencia *'DML_DDL::Table.all_objects'*. De todas ellas se seleccionan aquellas que tengan el mismo nombre que la tabla del modelo DML que contiene la columna que referencia el elemento "Value" que se está transformando. Aquí es donde se usa el decorador comentado anteriormente para obtener la tabla que contiene la columna referenciada por el valor que se pretende transformar. Todo esto se realiza mediante *'select{|dml_ddl|dml_ddl.tablName == dml.column.getInsertInto.tableName}'*. Con todas las tablas seleccionadas (que deberá ser una única tabla) se crea un conjunto con todas sus columnas mediante el operador *'map'*, se aplana dicha colección mediante *'flatten'* y se busca mediante *'find'* aquella columna que tenga el mismo nombre que la columna que referencia el valor que se pasa como parámetro de la regla *'Value2Value'*. Aquella columna que coincida será la que se asigne a la referencia "column" del nuevo "Value" del modelo DML_DDL.

5 Detección de Errores

En está etapa se pretende detectar los errores que puedan existir en el modelo DML_DDL obtenido en la etapa anterior. Para ello se hará uso de RubyTL para, aplicando el algoritmo de detección, obtener todos los errores.

Una vez que se tiene toda al información unificada en un único modelo de datos. El siguiente paso sería la detección de los posibles errores que pueda contener el sistema de información. En esta etapa se ha vuelto a hacer uso de RubyTL.

Para poder ejecutar RubyTL en la detección de errores es necesario el modelo DML_DDL obtenido en el apartado anterior y su metamodelo DML_DDL. Además se necesita definir el metamodelo que deberá conformar el modelo obtenido después de la transformación, dicho metamodelo se llama Errores. Es necesario también el conjunto de reglas que transformarán el modelo DML_DDL de entrada en el modelo Errores de salida. Y en este caso en particular se han tenido que definir una serie de decoradores (almacenados en el directorio *'helpers'*) que son los que realmente aplicarán el algoritmo de detección de errores creado en el proyecto.

En la siguiente figura se puede observar un diagrama general de la fase de detección de errores:

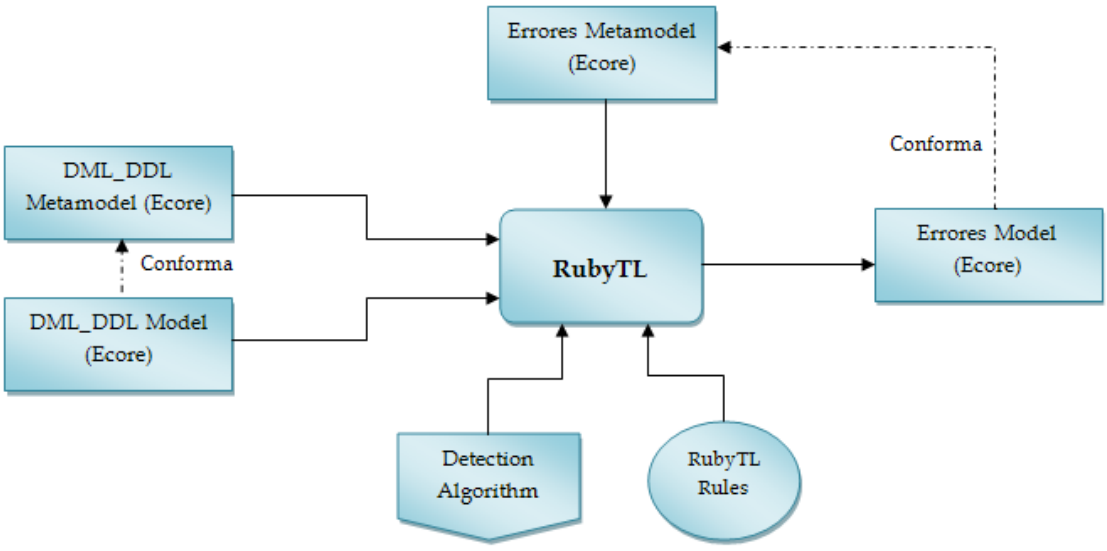

Figura 15: Obtención del modelo Errores mediante RubyTL.

El metamodelo Errores es el mismo que se realizó en la tesis *"Modernización de Base de Datos Relaciones Basada en Modelos"* pero al cuál se le han realizado una serie de modificaciones. Se ha mantenido la estructura del metamodelo pero casi todos sus atributos y referencias han sido modificados. El metamodelo quedaría como sigue:

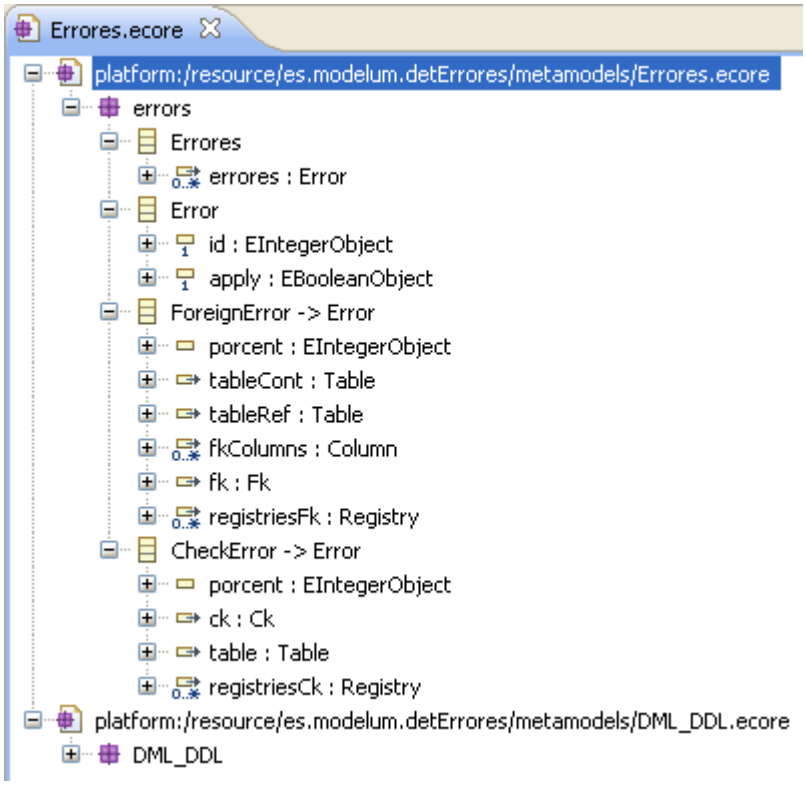

Figura 16: Metamodelo Errores.

La primera diferencia que se puede observar es que el metamodelo de Errores extiende al metamodelo DML_DDL ya que el metamodelo Errores tiene que referenciar a la información contenida en el metamodelo DML_DDL. Referenciará aquellas tablas, check constraints, foreign key, registros y columnas que presenten algún error.
En el metamodelo de la tesis existían metaclases para representar las tablas, las columnas, las check constraint y las foreign key. Tener información duplicada no es eficiente y requiere un mantenimiento extra por parte del desarrollador del proyecto, por lo que se optó por eliminar dichas metaclases y sustituirlas por referencias a las metaclases del metamodelo que las contiene, es decir, referencias al metamodelo DML_DDL.

El resto de modificaciones se explicarán mejor metaclase a metaclase:

- *'Error'*: en esta metaclase se han añadido dos nuevos atributos. *'id'* es un entero que contendrá un identificador para poder referirse de manera inmediata a un error concreto y *'apply'* un booleano que indicará en caso de ser "true" que ese error se quiere corregir e indicará si vale "false" que no se quiere corregir.

- *'ForeignError'*: los atributos *'tableRef'* y *'tableCont'* apuntarán a las tablas referenciada (a la que apunta la foreign key) y contenedora (la que contiene dicha foreign key) almacenadas en el modelo DML_DDL, respectivamente. Un valor de porcentaje que se explicará más adelante. Tendrá una referencia, *'fk'*, a la foreign key del modelo DML_DDL en cuestión, por lo que el atributo *'nameFk'* que existía en la tesis ha sido

eliminado, ya que el propio elemento foreign key del modelo DML_DDL contiene un atributo con su nombre. Una referencia multivaluada *'fkColumns'*, que contiene un conjunto de columnas que forman la foreign key que se quiere corregir. Y un conjunto de referencias a registros *'registriesFk'* que contendrá el conjunto de registros de la tabla contenedora que no cumple la foreign key.

- *'CheckError'*: el atributo 'table' apuntará a la tabla que contiene la check constraint, en el metamodelo de la tesis esta relación se establecía mediante un atributo con el nombre de la tabla a la que pertenecía la check constraint. Un valor de porcentaje que se explicará más adelante. Tendrá una referencia, *'ck'*, a la check constraint del modelo DML_DDL en cuestión, por lo que el atributo *'nameCk'* que existía en la tesis ha sido eliminado ya que el propio elemento check constraint del modelo DML_DDL contiene un atributo con su nombre. Y un conjunto de referencias a registros *'registriesCk'* que contendrá el conjunto de registros de la tabla contenedora que no cumplen la check constraint.

Como se puede observar existen únicamente dos metaclases pero se pretenden corregir tres tipos de errores. Esto se debe a que mediante la metaclase "ForeignError" se pretende reflejar dos errores, en concreto mediante dicha metaclase se representan los dos primeros errores; *"possible_new_Fk"* y *"Fk_disabled"*. Es por esto que la metaclase contiene tantos atributos y referencias, ya que no todos son para los dos tipos de errores.

El atributo *'porcent'* y las referencias *'tableRef'*, *'tableCont'* y *'registriesFk'* son comunes para ambos errores. La diferencia ésta en las referencias *'fkColumns'* y *'fk'*.

- La referencia *'fkColumns'* contendrá las columnas de la posible nueva foreign key que el algoritmo ha detectado, por lo que en el caso de que el error sea de una *"fk_disabled"* ésta referencia valdrá nulo.

- La referencia *'fk'* apunta a la foreing key que se encuentra desactivada y se quiere corregir, por lo que si el error es debido a una *"possible_new_Fk"* ésta referencia será nula ya que no existe dicha foreign key.

El error *"Ck_disabled"* se representa mediante la metaclase *'CheckError'*.

Mediante el conjunto de reglas que se usan como entrada para RubyTL se obtendrán todas las check constraint y foreign key desactivadas o nuevas que se encuentran dentro del elemento raíz *'DDLDefinition'* del modelo DML_DDL. Con todos los posibles elementos del modelo DML_DDL que contienen algún error se obtiene un modelo que conformará el metamodelo Errores. El conjunto de reglas de RubyTL se pueden ver íntegramente en el apartado de "Anexos".

La obtención de todos los errores que se encuentran en el modelo de datos DML_DDL se realiza mediante la siguiente regla.

```
top_rule 'error' do
  from DML_DDL::DDLDefinition
  to Errores::Errores
  mapping do |dml_ddl, error|
    dml_ddl.getErrors

    error.errores = dml_ddl.getCks
    error.errores = dml_ddl.getFks
    error.errores = dml_ddl.getPks
  end
end
```

Para el elemento *'DDLDefinition'* del modelo DML_DDL se obtiene un único elemento *'Errores'* (será el elemento raíz) del modelo Errores que se pretende crear. Como se puede observar en la sección de los *mappings*, lo primero que hace la regla es llamar a un método del decorador asociado a la metaclase *'DML_DDL::DDLDefinition'* que ésta definido en el fichero *'modernizar_BD.rb'* que se encuentra en el directorio *'helpers'*. Para poder hacer uso de este decorador hay que incluirlo usando la directiva *'use_library'* de RubyTL que se comentó en el apartado de RubyTL, a continuación se puede ver como quedaría dicha inclusión:

```
use_library 'helper://modernizar_BD.rb'
```

El decorador definido para la metaclase *'DDLDefinition'* del modelo DML_DDL contiene cuatro métodos. Mediante el método *'getErrores'* obtenemos los diferentes errores que existen en el modelo DML_DDL. Una vez que se tienen todos los errores lo que la regla hace es asignar a la referencia multivaluada *'errores'*, que contiene todos los errores encontrados en el modelo DML_DDL, cada uno de los arrays con los distintos errores de cada uno de los tres tipos. Esta asignación se realiza llamando a los otros tres métodos del decorador. Para cada una de las tres asignaciones de los elementos del modelo DML_DDL que presentan algún error a la referencia *'errores'* hace que se lancen el resto de las reglas que se han definido.

Existen tres reglas más. Cada una para cada tipo de error. Cada regla creará el elemento correspondiente a su tipo de error e inicializará sus atributos y referencias.

En el caso de que exista algún error del tipo *'Ck_disabled'* se ejecutará la regla *'ck_Desactivada'*. En ella se obtiene el porcentaje de error, se asignan las distintas referencias, se le asigna un identificador único y el atributo *'apply'* por defecto se pone a "true". La regla se puede a ver a continuación:

```
rule 'ck_Desactivada' do
  from DML_DDL::Ck
  to Errores::CheckError
  mapping do |dml_ddl, error|
    error.porcent = dml_ddl.getPorcent
    error.set('table', dml_ddl.getTable)
    error.set('ck', dml_ddl)
    dml_ddl.getRegistries.each do |reg|
      error.set('registriesCk', reg)
    end
    error.id = counter.next()
```

```
    error.apply = true
  end
end
```

La regla *'ck_Desactivada'* toma como parámetro el elemento *'Ck'* del modelo DML_DDL que presenta algún error y obtiene un elemento *'CheckError'* del modelo Errores. El nuevo elemento creado tendrá por porcentaje el obtenido al aplicar el algoritmo de modernización que se encuentra en el decorador. Las distintas referencias al modelo DML_DDL se llevan a cabo mediante el método *'set'* de RubyTL. Dicho método hace que a la referencia pasada como primer parámetro, que posee del elemento que ejecuta *'set',* se le asigne el elemento pasado como segundo parámetro. Por lo que se asigna a la referencia *'table'* la tabla obtenida al ejecutar el método *'getTable'* del decorador asociado al elemento *'Ck'* del modelo DML_DDL. A la referencia *'ck'* se le asigna el propio elemento *'Ck'* del modelo DML_DDL que ejecuta la regla y a la referencia multivaluada de los registros erróneos se le asigna cada uno de los registros encontrados al ejecutar el método *'getRegistries'* del decorador asociado al elemento *'Ck'* del modelo DML_DDL. Para ello recorremos el array obtenido del método *'getRegistries'* mediante la sentencia Ruby *'each do |reg|'* la cuál asigna el elemento correspondiente a la iteración concreta a la variable *'reg'* para que pueda ser usada dentro del bucle.

Como se ha dicho anteriormente el atributo *'apply'* es puesto a "true" por defecto, ya que por defecto se desea corregir el error encontrado.

El identificador es asignado mediante una clase que se ha creado en el fichero *'modernizar_BD.rb'*. Dicha clase solamente contiene un entero (definido como una variable global) que se mantiene durante toda la ejecución de la transformación. Llamando al método *'next()'* obtenemos el nuevo identificador del error. Se trata de un simple contador que se inicializa al inicio del fichero de transformación mediante la sentencia:

```
counter = Counter.new
```

Consiguiendo así, que la variable *'counter'* de tipo *'Counter'* sea global para todas las reglas y las sucesivas llamadas aumenten el contador de forma correcta.

La clase se presenta a continuación:

```
class Counter
  @@id = -1
  def next()
    aux = @@id.next
    @@id = aux
    return aux
  end
end
```

Como se puede observar se define mediante la palabra reservada *'class'*. Presenta un único atributo *'id'* que es global a toda la ejecución de la transformación por contener el prefijo *'@@'*. A continuación se define un único método llamado *'next()'*, el cuál ejecuta el método *'.next'* que es un método que contienen las variables enteras en Ruby. Mediante esta llamada

se obtiene el siguiente número al que almacena *'@@id'*. Posteriormente se actualiza *'@@id'* con el valor devuelto, par asegurarnos que *'@@id'* tiene el valor actualizado aunque *'.next'* lo actualice, y se devuelve el valor obtenido.

En el caso de que exista algún error del tipo *'Fk_disabled'* se ejecutará la regla *'fk_Desactivada'*. En ella se obtiene el porcentaje de error, se asignan las distintas referencias, se le asigna un identificador único y el atributo *'apply'* por defecto se pone a "true". La regla se puede a ver a continuación:

```
rule 'fk_Desactivada' do
  from DML_DDL::Fk
  to Errores::ForeignError
  mapping do |dml_ddl, error|
    error.porcent = dml_ddl.getPorcent
    error.set('tableCont', dml_ddl.getTable)
    error.set('fk', dml_ddl)
    error.set('tableRef', dml_ddl.references)

    dml_ddl.getRegistries.each do |reg|
      error.set('registriesFk', reg)
    end

    error.apply = true
    error.id = counter.next()
  end
end
```

La regla *'fk_Desactivada'* toma como parámetro el elemento *'Fk'* del modelo DML_DDL que presenta algún error y obtiene un elemento *'ForeignError'* del modelo Errores. El nuevo elemento creado tendrá por porcentaje el obtenido al aplicar el algoritmo de modernización. Se establecen las distintas las referencias. Se obtiene la tabla del modelo DML_DDL que contiene la foreign key mediante el método *'getTable'* del decorador asociado al elemento *'Fk'* del modelo DML_DDL y dicha tabla es asignada a la referencia *'tableCont'* del nuevo elemento *'ForeignError'* del modelo Errores. A la referencia *'fk'* se le asigna el propio elemento *'Fk'* del modelo DML_DDL que ejecuta la regla. La referencia *'tableRef'* se podría dejar nula, ya que en el propio elemento *'Fk'* del modelo DML_DDL existe una referencia llamada *'references'* que apunta a la tabla que referencia la foreign key, pero para no tener que navegar hasta el elemento *'Fk'* y posteriormente hasta el elemento *'Table'* se lo asignamos a la referencia *'tableRef'* para tener un acceso directo. A la referencia multivaluada de los registros erróneos se le asigna cada uno de los registros encontrados al ejecutar el método *'getRegistries'* del decorador asociado al elemento *'Fk'* del modelo DML_DDL.

Como se ha dicho anteriormente, el atributo *'apply'* es puesto a "true" por defecto, ya que por defecto se desea corregir el error encontrado. Y se le asigna un nuevo identificador.

Como se puede ver, la referencia *'fkColumns'* del elemento *'ForeignError'* no existe en la sección de los *mappings*. Esto indica que su valor se ha puesto a nulo ya que no se trata de un error de una posible nueva foreign key, por lo que no existirán posibles nuevas columnas para la nueva foreign key, sino que la actual foreign key está desactivada y presenta algún error.

En el caso de que exista algún error del tipo *'possible_new_Fk'* se ejecutará la regla *'check_Pk'*. En ella se obtiene el porcentaje de error, se asignan las distintas referencias, se le asigna un identificador único y el atributo *'apply'* por defecto se pone a "true". La regla se puede a ver a continuación:

```
rule 'check_Pk' do
  from DML_DDL::Pk
  to Errores::ForeignError
  mapping do |dml_ddl, error|
    error.porcent = dml_ddl.getPorcent
    error.set('tableRef', dml_ddl.getTable)
    error.set('fkColumns', dml_ddl.getColumnsRef)
    error.set('tableCont', dml_ddl.getTableRef)

    dml_ddl.getRegistries.each do |reg|
      error.set('registriesFk', reg)
    end

    error.apply = true
    error.id = counter.next()
  end
end
```

La regla *'check_Pk'* toma como parámetro el elemento *'Pk'* del modelo DML_DDL que presenta algún error y obtiene un elemento *'ForeignError'* del modelo Errores. El nuevo elemento creado tendrá por porcentaje el obtenido al aplicar el algoritmo de modernización. Se establecen las distintas las referencias. Se obtiene la tabla del modelo DML_DDL que contiene la primary key mediante el método *'getTable'* del decorador asociado al elemento *'Pk'* del modelo DML_DDL y dicha tabla es asignada a la referencia *'tableRef'* del nuevo elemento *'ForeignError'* del modelo Errores. Como se puede observar, la tabla que contiene la primary key será en realidad la tabla a la que referenciará la nueva foreign key que se pretende crear. Es por eso que se ha asignado a la referencia *'tableRef'* y no a la referencia *'tableCont'*. A la referencia *'tableCont'* se asigna la mejor tabla que podría tener la nueva foreign key según el algoritmo de detección de errores implementado en el decorador asociado al elemento *'Pk'*. Mediante el método *'getColumnsRef'* del decorador obtenemos la lista de columnas que formarán la nueva foreign key. Serán tantas como columnas tenga la primary key. A la referencia multivaluada de los registros erróneos se le asigna cada uno de los registros encontrados al ejecutar el método *'getRegistries'* del decorador asociado al elemento *'Fk'* del modelo DML_DDL.

Como se ha dicho anteriormente, el atributo *'apply'* es puesto a "true" por defecto, ya que por defecto se desea corregir el error encontrado. Y se le asigna un nuevo identificador.

Como se puede ver, la referencia *'fk'* del elemento *'ForeignError'* no existe en la sección de los *mappings*. Esto indica que su valor se ha puesto a nulo ya que no se trata de un error de una foreign key desactivada sino de una supuesta nueva foreign key, por lo que la foreign key no existirá en el modelo DML_DDL,. Es por eso que no se puede asignar ningún valor a la referencia *'fk'* del elemento *'ForeignError'* del modelo Errores.

Una vez que se conocen las reglas y su ejecución, hay que saber como funciona el algoritmo de detección de errores para cada uno de los tres tipos de error. Así como comprender los diferentes métodos implementados para cada uno de los decoradores que se han definido.

En el decorador asociado al elemento *'DDLDefintion'* del modelo DML_DDL se definen cuatro métodos:

- *'getErrors'*: en este método se obtienen todos los posibles errores que existen en el modelo DML_DDL. Para ello, hay que tener en cuenta un aspecto relacionado con cada uno de los tres tipos de error:
    - *'Ck_disabled'*: se considerará que una check constraint presenta un error cuando su estado es *"DISABLED"* y su porcentaje de error obtenido al ejecutar el algoritmo de detección de errores es inferior al 70%.
    - *'Fk_disabled'*: se considerará que una foreign key presenta un error cuando su estado es *"DISABLED"* y su porcentaje de error obtenido al ejecutar el algoritmo de detección de errores es inferior al 70%.
    - *'possible_new_Fk'*: se considerará que existe una supuesta nueva foregin key asociada a una primary key cuando el porcentaje de error obtenido al ejecutar el algoritmo de detección de errores es superior al 70%.

  Hay que tener en cuenta que el porcentaje de error devuelto en cada uno de los tres tipos de error tiene un significado diferente, el cuál se comentará más adelante.
- *'getCks'*: se devuelven todas las check constraint que se han encontrado en el método *'getErrors'*.
- *'getFks'*: se devuelven todas las foreing key que se han encontrado en el método *'getErrors'*.
- *'getPks'*: se devuelven todas las primary key que se asocien al error de una posible nueva foreign key encontradas en el método *'getErrors'*.

El decorador presenta el siguiente aspecto:

```
decorator DML_DDL::DDLDefinition do
  def getErrors
    @cks = DML_DDL::Ck.all_objects.select{|dml_ddl| dml_ddl.status ==
"DISABLED" && dml_ddl.getPorcent < 70}
    @fks = DML_DDL::Fk.all_objects.select{|dml_ddl| dml_ddl.status ==
"DISABLED" && dml_ddl.getPorcent < 70}
    @pks = DML_DDL::Pk.all_objects.select{|dml_ddl| dml_ddl.getPorcent > 70}
  end

  def getCks
    return @cks
  end

  def getFks
```

```
    return @fks
  end

  def getPks
    return @pks
  end
end
```

Como se puede ver, en el método *'getErrors'* se definen e inicializan las tres variables que contendrán el conjunto de cada uno de los tres tipos de error que se encuentren en el modelo DML_DDL.

Para obtener todas las check constraint que presenta algún error en la variable *'cks'*, que se define global al decorador al tener el prefijo *'@'*, se almacenan todas las check constraint devueltas mediante la sentencia:

```
DML_DDL::Ck.all_objects.select{|dml_ddl|
                        dml_ddl.status == "DISABLED" && dml_ddl.getPorcent < 70}
```

En ella se obtienen todos elementos *'Ck'* del modelo DML_DDL mediante *'DML_DDL::Ck.all_objects'* y para todos ellos se seleccionan aquellos que presentan algún error, es decir, los que su estado es *"DISABLED"* y su porcentaje de error obtenido es menor del 70%.

Todos estos *'Ck'* se guardarán en la variable *'@cks'* para, posteriormente, en el método *'getCks'* devolver su contenido mediante un simple *'return @cks'*.

Para obtener todas las foreign key que presenta algún error en la variable *'fks'*, que se define global al decorador al tener el prefijo *'@'*, se almacenan todas las foreign key devueltas mediante la sentencia:

```
DML_DDL::Fk.all_objects.select{|dml_ddl|
                        dml_ddl.status == "DISABLED" && dml_ddl.getPorcent < 70}
```

En ella se obtienen todos elementos *'Fk'* del modelo DML_DDL mediante *'DML_DDL::Fk.all_objects'*, y para todos ellos se seleccionan aquellos que presentan algún error, es decir, los que su estado es *"DISABLED"* y su porcentaje de error es menor del 70%.

Todos estos *'Fk'* se guardarán en la variable *'@fks'* para, posteriormente, en el método *'getFks'* devolver su contenido mediante *'return @fks'*.

Para obtener todas las primary key que presentan un error de una posible nueva foreign key en la variable *'pks'*, que se define global al decorador al tener el prefijo *'@'*, se almacenan todas las primary key devueltas mediante la sentencia:

```
DML_DDL::Pk.all_objects.select{|dml_ddl| dml_ddl.getPorcent > 70}
```

En ella se obtienen todos elementos *'Pk'* del modelo DML_DDL mediante *'DML_DDL::Pk.all_objects'*, y para todos ellos se seleccionan aquellos que presentan algún error, es decir, los que su porcentaje de error, obtenido al aplicar el algoritmo de detección de errores es mayor del 70%.

Todos estos *'Pk'* se guardarán en la variable *'@pks'* para, posteriormente, en el método *'getPks'* devolver su contenido mediante *'return @pks'*.

A continuación se detallaran cada uno de los decoradores definidos para detectar cada uno de los tres tipos de error.

La idea del algoritmo para la detección de los errores *'Ck_disabled'* es la siguiente:

Cuando se da la situación de que una check constraint se encuentra deshabilitada, para comprobar si debería habilitarse o continuar deshabilitada se comprueba si todos los registros de la tabla, que contiene a la check constraint, cumplen dicha check constraint. Para ello se comprueban si los valores de las columnas que forman la check constraint en cada uno de los registros cumplen la propia check constraint.

En caso de que un registro cumpla la check constraint se considera un acierto. El porcentaje devuelto al terminar de detectar los errores 'Ck_disabled' es el porcentaje de aciertos que se han obtenido en cada una de las check constraint desactivadas, que se obtiene mediante la siguiente fórmula:

```
porcent = (hits/num_registries) * 100
```

Esta idea se plasma en el decorador asociado a la metaclase '*Ck*' del modelo DML_DDL. En dicho decorador hay definidos una serie de métodos. El más importante es el que aplica el algoritmo de detección comentado anteriormente para obtener el porcentaje: el método *'getPorcent'*. El decorador presenta también dos métodos más. Uno llamado *'getTable'* para obtener la tabla que contiene a la check constraint. Y otro llamado *'getRegistries'* para obtener el conjunto de registros que no cumplen la check constraint, es decir, aquellos registros que han dado un fallo como resultado al aplicar el algoritmo de detección.

El método *'getTable'* presenta el siguiente contenido:

```
def getTable
  # Devolvemos la tabla asociada a una Ck
  table = DML_DDL::Table.all_objects.find{|dml_ddl|
                                          dml_ddl.checks.include?(self)}
  return table
end
```

De todas las tablas que contiene el modelo DML_DDL nos quedamos con aquella que tenga en su lista de checks constraints, atributo *'checks'* de *'Table'*, la check constraint que ejecuta el decorador. Dicha comprobación se realiza mediante el método *'include?'* de Ruby para los

arrays. Dicho método comprueba si el elemento pasado como parámetro pertenece al conjunto que ejecuta el *'include?'*.

En el método que ejecuta el algoritmo de detección de errores se realiza lo siguiente:

Primeramente hay que obtener los valores de las columnas que forman la check constraint, de la tabla que contiene la check constraint deshabilitada. Para ello se crea un array con todos los nombres de las columnas que forman la check constraint. Recorremos todos los *'ValuesCk'* que conforman la check constraint y se guarda el atributo *'columnName'* en un array. Esta inicialización del array con los nombres de las columnas que forman la check constraint se realiza mediante la siguiente porción de código:

```
self.valuesCk.each do |value|
      columns << value.columnName if (!columns.include?(value.columnName))
    end
```

Aparte hay que obtener los valores de todas las columnas que forman la check constraint a partir de los registros de la tabla que contiene la check constraint. Para ello se recorren todos los *'Registry'* de la tabla para obtener su referencia *'registryValues'* con todos los valores que contiene el registro. Para cada uno de los *'registryValues'* de un registro, accedemos cada uno de sus *'Value'* para comprobar que la columna a la que le da valor pertenece a las columnas que forman la check constraint. En caso afirmativo la introducimos en un Hash con clave el nombre de la columna y contenido todos los valores de la columna para todos los registros de la tabla. Además se consigue el número total de registros de la tabla, que será igual al número final de valores que existirá en cualquier entrada del Hash. Todo esto se consigue mediante la siguiente porción de código RubyTL:

```
for i in 0..columns.length-1
  values = Array.new # Array para almacenar los Values.
  registries.each do |r|
    registryValues = r.registryValues
    registryValues.each do |rv|
      if (rv.column.columnName == columns[i])
        values << rv.value
      end
    end
  end

  total = values.length
  valuesColumn[columns[i]] = values
end
```

Una vez que se tienen los valores de cada una de las columnas que forman la check constraint de toda la tabla se procede a comprobar si verifican la check constraint deshabilitada. Para ello para cada contenido del Hash, es decir, para cada conjunto de valores de cada una de las columnas que forman la check, se comprueba si cumple la restricción que la check constraint establece en su definición para esa columna. Se recorre el hash obtenido en el paso anterior y para columna se recorre el conjunto de restricciones, *'ValuesCk'*, de la check constraint. Se comprueba que la columna del conjunto de valores se corresponde con la columna de la *'ValuesCk'*. En caso de ser la misma columna se guarda la conjunción lógica para ser usada en

el siguiente paso del algoritmo y se procede a comprobar que los valores, para dicha columna, de toda tabla cumplen las restricciones impuestas por la check constraint. Esto se puede ver en la siguiente porción de código del decorador:

```
valuesColumn.each_key do |key|
  values = valuesColumn[key]
  r = Array.new
  ckValues = Array.new
  for i in 0..values.length-1
    check = 0
    self.valuesCk.each do |val|
      if (val.columnName.eql?(key))
        @logConj << val.logConjuntion if (val.logConjuntion != nil)
        aux = false
        val.value.each do |v|
……
……
……
```

Si la comparación es correcta, en un array se almacena un *'true'*. En caso de ser incorrecta, es decir, de que el valor no cumpla la restricción en el array se almacena un *'false'*. Se realiza esto por que una vez que tenemos todos los resultados de la comprobación de las diferentes restricciones que contiene la check constraint hay que aplicarle las conjunciones lógicas que unen cada restricción.

```
if ((val.comparator.eql?("=")) && (values[i] == v.gsub(/'/, '')))
  aux = true
  break
end

if ((val.comparator.eql?("<>")) && (!(values[i] == v.gsub(/'/, ''))))
  aux = true
  break
end

if ((val.comparator.eql?("<")) && (values[i] < v.gsub(/'/, '')))
  aux = true
  break
end

if ((val.comparator.eql?(">")) && (values[i] > v.gsub(/'/, '')))
  aux = true
  break
end

if ((val.comparator.eql?("<=")) && (values[i] <= v.gsub(/'/, '')))
  aux = true
  break
end

if ((val.comparator.eql?(">=")) && (values[i] >= v.gsub(/'/, '')))
  aux = true
  break
end

r << aux
```

En caso de que la comprobación de la restricción sea falsa se realiza una operación adicional y es que se guarda en un hash global al decorador, llamado *'@valuesCk',* los valores que no cumplen la restricción para cada columna. Dicho hash tendrá por clave el nombre de la columna y por conjunto de valores, aquellos valores de la columna que no cumplen la check constraint. Esto se hace para, en el método *'getRegistries'*, obtener todos los registros que no cumplen la check constraint.

```
@valuesCK = Hash.new

if (!aux)
  ckValues << values[i]
end

@valuesCK[key] = ckValues
```

Para poder realizar la comprobación final con los resultados obtenidos de las comprobaciones de las restricciones y con el conjunto de operadores lógicos hay que obtener la transpuesta de la matriz con los resultados de las comprobaciones. Dicha matriz contenía tantas columnas como columnas forman la check constraint y tantas filas como registros hay en la tabla que contiene la check constraint. Por lo que para poder aplicar las conjunciones lógicas hay que tener tantas columnas como registros hay en la tabla que contiene la check constraint y tantas filas como columnas forman la check constraint.

Se aplica el conjunto de conjunciones lógicas al conjunto de resultados (booleanos) obtenidos en la comprobación de las columnas de la check constraint y cada valor devuelto a *'true'* se considera un acierto.

```
for i in 0..result.length-1
  log = @logConj[i]
  boolean = true
  for j in 0..result[i].length-1
    if (log == "AND")
      boolean = boolean && result[i][j]
    else
      if (log == "OR")
        if (j == 0)
          boolean = false
        end
        boolean = boolean || result[i][j]
      else
        if (log == nil)
          boolean = result[i][j]
        end
      end
    end
  end

  if (boolean)
    aciertos += 1
  end
end
```

Al final del método se aplica la fórmula para obtener el tanto por ciento definitivo. Pero antes se realiza una comprobación y es que si la tabla que contiene la check constraint deshabilitada

no presenta ningún registro se devuelve un valor de porcentaje igual al 100%. Con esto aseguramos que dicha check constraint aún estando deshabilitada, no se habilite, ya que no tiene ningún valor al que aplicarse. Al devolver un 100% aseguramos que el porcentaje devuelto es mayor o igual que el porcentaje de corte de la restricción del decorador asociado al elemento *'DDLDefinition'* del modelo DML_DDL. Así no se añade dicha check constraint al conjunto de check constraint con un error.

```
if (total == 0)
  # Este caso se daría cuando la tabla que contiene la CK no presenta ningún
  # registro, pero aún así la Ck esta desactivada, por lo que para que no
  # crease un error de Ck, debemos devolver un porcentaje mayor o igual al
  # porcentaje de corte, es decir, mayor o igual a 70. Devolveremos 100
  # para así no tener ninguna duda.

  porcent = 100
else
  porcent = (100*aciertos).div(total)
end

return porcent
```

El método que queda del decorador de la *'Ck'* del modelo DML_DDL, es el método *'getRegistries'*. Mediante dicho método obtenemos los registros que no cumplen la check constraint.

En dicho método se realiza lo siguiente: en dos variables globales al decorador se obtuvieron en el método *'getPorcent'* el conjunto de valores que no cumplían la check constraint todas de las columnas de la tabla que forman la check constraint y en otra el conjunto de conjunciones lógicas.

Lo que se hace es recorrer todos los registros de la tabla que contiene la check constraint e ir comparando dichos valores con los valores almacenados en el hash, obtenidos en el método *'getPorcent'*. Se coge un *'Value'* del conjunto de *'registryValues'* de un registro y se comprueba si ésta columna esta en el conjunto de columnas que forman la check constraint. Es decir, se comprueba que dicha columna es alguna de las claves del hash. En caso de ser así, se procede a comprobar si el valor que contiene el elemento *'Value'* ésta en el conjunto de valores que no cumplen la check constraint asociados a esa columna. En caso de que sea igual a uno de ellos, se sale del bucle y se guarda en un array un *'true'*, en caso de que no sea igual a ninguno de los valores que no cumplen la check constraint se guarda en el array un *'false'*.

```
registries.each do |r|
  results = Array.new
  registryValues = r.registryValues
  registryValues.each do |rv|
    @valuesCK.each_key do |key|
      if (rv.column.columnName == key)
        aux = false
        values = @valuesCK[key].uniq
        for i in 0..values.length-1 do
          if (values[i] != rv.value)
            aux = true
            break
          end
        end
```

```
      results << aux
    end
  end
end
```

Una vez que se tiene el conjunto de resultados *('true', 'false'*) para un registro de la tabla, se procede a aplicar las conjunciones lógicas obtenidas en el método *'getPorcent'* y guardadas en una variable global al decorador. En caso de que el resultado obtenido al aplicar el conjunto de conjunciones lógicas sea *'false'* el registro es añadido al conjunto de registros a devolver por el método.

```
if (!boolean)
  registriesCk << r
end

return registriesCk.uniq
```

La idea del algoritmo para la detección de los errores *'Fk_disabled'* es la siguiente:

Cuando se da la situación de que una foreign key se encuentra deshabilitada, para comprobar si debería habilitarse o continuar deshabilitada se comprueba si todos los registros de la tabla que contiene la foreign key, presentan una correspondencia en la tabla que contiene la primary key. Para ello se comprueba para cada registro si las columnas que forman la foreign key tiene una correspondencia en las columnas de los registros que forman la primary key.

En caso de que un registro cumpla la correspondencia foreign key -> primary key se considera un acierto. El porcentaje devuelto al terminar de detectar los errores 'Fk_disabled' es el porcentaje de aciertos total de todos los registros de la tabla con la foreign key a comprobar. Dicho valor se obtiene mediante la siguiente fórmula:

```
porcent = (hits/num_registries) * 100
```

Esta idea se plasma en el decorador asociado a la metaclase '*Fk*' del modelo DML_DDL. En dicho decorador hay definidos una serie de métodos. El más importante es el que aplica el algoritmo de detección comentado anteriormente para obtener el porcentaje, el método *'getPorcent*'. El decorador presenta también dos métodos más. Uno llamado *'getTable'* para obtener la tabla que contiene la foreign key y otro llamado *'getRegistries'* para obtener el conjunto de registros que no cumplen la foreign key, es decir, aquellos registros cuyos valores de las columnas que forman la foreign key no se encuentran en las columnas de los registros de la tabla a la que la foreign key hace referencia, es decir, la tabla que contiene la primary key.

El método *'getTable'* presenta el siguiente contenido:

```
def getTable
  # Devolvemos la tabla asociada a una Fk
  DML_DDL::Table.all_objects.find{|dml_ddl| dml_ddl.columnsFk.include?(self)}
end
```

De todas las tablas que contiene el modelo DML_DDL nos quedamos con aquella que tenga en su lista de foreign keys, atributo *'columnsFk'* de *'Table'*, la foreign key que ejecuta el decorador.

En el método que ejecuta el algoritmo de detección de errores se realiza lo siguiente:

Primero se obtienen todos los valores de las columnas que forman la foreign key. Para ello se obtienen los *'Registry'* de la tabla con la foregin key que está deshabilitada. Una vez que se tienen todos los registros como en el elemento *'Fk'* tiene un conjunto con todos los nombres de las columnas que forman la foreign key, se recorren todos los registros de la tabla que contiene a la foreign key obteniendo todos sus *'registryValues'*. Para cada *'registryValue'* se accede a su conjunto de *'Values'*. En caso de que el *'Value'* dé valor a la misma columna que ésta en la lista de columnas que forman la foreign key guardamos su valor en un array.

```
self.columnName.each do |col|
  values = Array.new # Array para almacenar los Values.
  registries.each do |r|
    registryValues = r.registryValues # Obtenemos los registryValues de un
                                      # registry en concreto.
    registryValues.each do |rv|
      if (rv.column.columnName == col) # Comprobamos que es el valor de la
                                       # columna de la Fk.
        values << rv.value
      end
    end
  end
  values_fk << values # Almacenamos los valores de una columna.
End
```

Una vez que se tienen todos los valores de cada una de las columnas que forman la foreign key se tiene que realizar lo mismo para obtener los valores de las columnas que forman la primary key a la que referencia la foreign key.

Para ello realizamos una operación análoga a la anterior pero en la tabla de la primary key, con los registros de la tabla de la primary key y con las columnas que forman la primary key. Quedaría como sigue:

```
self.columnReference.each do |col|
  values = Array.new # Array para almacenar los Values.
  registries.each do |r|
    registryValues = r.registryValues # Obtenemos los registryValues de un
                                      # registry en concreto.
    registryValues.each do |rv|
      if (rv.column.columnName == col) # Comprobamos que es el valor de la
                                       # columna de la Pk.
        values << rv.value
      end
    end
  end
  values_pk << values # Almacenamos los valores de una columna.
End
```

Cuando se tienen todos los valores de las columnas que forman la foreign key y la primary key, se tienen que comparar para ver si los valores de las columnas de la foreign key están presentes en el conjunto de valores de las columnas de la primary key.

Para cada una de las columnas que forman la foreign key se obtiene el conjunto de valores que presenta en la tabla en la que está definida, obtenidos anteriormente. Para cada uno de dichos valores, se comprueba si esta en el conjunto de valores de la columna de la primary key a la que la columna de la foreign key referencia. Si cada uno de los valores está presente en las columnas de la primary key a la que hace referencia se considerará un acierto, en otro caso un fallo.

Hay que tener en cuenta que en cuanto se comprueba que un valor del conjunto de valores de la columna que forma la foreign key esta en el conjunto de valores de la columna de la primary key a la que referencia no hace falta seguir comprobando, se sale del bucle dando como resultado un acierto.

```
for i in 0..values_fk.length-1
  v_fk = values_fk[i]
  v_pk = values_pk[i]

  for j in 0..v_fk.length-1
    existe = 0
    for k in 0..v_pk.length-1
      if (v_fk[j].eql?(v_pk[k]))
        existe = 1
        break
      end
    end
    if (existe == 1)
      aciertos += 1
    end
  end
end
```

Al final del método se aplica la fórmula para obtener el tanto por ciento definitivo. Antes se realiza una comprobación y es que si la tabla que contiene la foreign key deshabilitada no presenta ningún registro, se devuelve un valor de porcentaje igual al 100%. Con esto aseguramos que dicha foreign key, aunque deshabilitada, no se habilite, ya que no tiene ningún valor al que aplicarse. Al devolver un 100% aseguramos que el porcentaje devuelto es mayor o igual que el porcentaje de corte de la restricción del decorador asociado al elemento *'DDLDefinition'* del modelo DML_DDL así no se añade dicha foreign key al conjunto de foreign key con un error.

```
if (values_fk[0].length == 0)
  # Este caso se daría cuando la tabla que contiene la FK no presenta nigún
  # registro, pero aún así la Fk esta desactivada, por lo que para que no
  # crease un error de Fk, debemos devolver un porcentaje mayor o igual al
  # porcentaje de corte, es decir, mayor o igual a 70. Devolveremos 100
  # para así no tener ninguna duda.

  porcent = 100
else
  porcent = (aciertos*100).div(values_fk[0].length)
end
```

```
return porcent
```

El método que queda del decorador de la *'Fk'* del modelo DML_DDL, es el método *'getRegistries'*. Mediante dicho método se obtienen los registros que no cumplen la foreign key.

En dicho método se comprueba qué registros de la tabla que tiene la foreign key no la cumplen. Para ello, para cada una de las columnas que forman la foreign key se recorre el conjunto de registros que tiene la tabla. Se obtiene para cada registro el valor correspondiente a la columna de la foreign key. Para dicho valor se recorren todos los registros de la tabla a la que referencia la foreign key, es decir, la tabla que tiene la primary key a la que la foreign key referencia y se busca la columna concreta a la cual referencia la columna de la foreign key. Si sus valores coinciden, el registro al que corresponde el valor no se añade al conjunto de valores a devolver por el método. En otro caso se añade el registro al array que contendrá los registros que no cumplen la foreign key.

```
for i in 0..columnsFk.length-1
  registriesTableCont.each do |registryCont|
    registryValuesCont = registryCont.registryValues
    registryValuesCont.each do |valueCont|
      if (valueCont.column.columnName == columnsFk[i])
        aux = false
        registriesTableRef.each do |registryRef|
          registryValuesRef = registryRef.registryValues
          registryValuesRef.each do |valueRef|
            if ((columnsPk[i] == valueRef.column.columnName) &&
                                          (valueCont.value == valueRef.value))
              aux = true
              break
            end
          end
          if (aux)
            break
          end
        end

        if (!aux)
          registries << registryCont
          break
        end
      end
    end
  end
end

return registries
```

La idea del algoritmo para la detección de los errores *'possible_new_Fk'* es la siguiente:

Se puede dar la situación que en una base de datos una tabla debiera tener una foreign key hacia una primary key de otra tabla. Para que sucediese esto, en la tabla que debería tener la foreign key tienen que existir tantas columnas como presenta la primary key. Que dichas columnas que tuviesen un nombre similar y fuesen del mismo tipo. Por ello lo que se hace es

para toda primary key buscar por todas las demás tablas alguna que reúna las condiciones anteriores, es decir, que tenga tantas columnas como tiene su primary key similares en nombre y del mismo tipo.

En caso de que una tabla sea candidata a tener una posible nueva foreign key se obtiene el tanto por ciento de coincidencia de las columnas de los registros de la tabla de la nueva posible foreign key, con las columnas de los registros de la tabla de la primary key. Al final se obtendrá la mejor tabla. La cuál será aquella que presenta un mayor valor de porcentaje. El porcentaje se obtiene mediante la siguiente fórmula:

```
porcent = (hits/num_registries) * 100
```

Esta idea se plasma en el decorador asociado a la metaclase *'Pk'* del modelo DML_DDL. En dicho decorador hay definidos una serie de métodos. El más importante es el que aplica el algoritmo de detección comentado anteriormente para obtener el porcentaje, el método *'getPorcent'*. El decorador presenta también cuatro métodos más. Uno llamado *'getTable'* para obtener la tabla que contiene la primary key. Otro llamado *'getRegistries'* para obtener el conjunto de registros que no cumplen la supuesta nueva foreign key, es decir, aquellos registros que los valores de las columnas que formarán la nueva foreign key no se encuentran en las columnas de los registros de la tabla a la que la foreign key hace referencia, la tabla que contiene la primary key. Además como se trata de una posible nueva foreign key se tienen que obtener las columnas que formarán la nueva foreign key y la tabla que la contendrá. Para ello en el decorador se definen dos variables globales llamadas *'@tableRef'* y *'@columnsRef'*. Para obtener dichos valores se definen dos métodos más para obtenerlos. Los métodos *'getTableRef'* y *'getColumnsRef'*.

El método *'getTable'* presenta el siguiente contenido:

```
def getTable
  DML_DDL::Table.all_objects.find{|dml_ddl| dml_ddl.columnsPk == self}
End
```

De todas las tablas que contiene el modelo DML_DDL nos quedamos con aquella que tenga una primary key, atributo *'columnsPk'* de *'Table'*, igual a la primary key que ejecuta el decorador.

Los métodos *'getTableRef'* y *'getColumnsRef'* simplemente devuelven el contenido de sus respectivas variables globales mediante la palabra reserva *'return'*. Tendrían el siguiente aspecto:

```
# Definición para devolver la tabla de la Fk
def getTableRef
  return @tableRef
end

# Definición para devolver las columnas de la tabla de la Fk
def getColumnsRef
  return @columnsRef
end
```

En el método que ejecuta el algoritmo de detección de errores se realiza lo siguiente:

Primero se realiza un filtro de las posibles tablas que pueden tener una posible foreign key hacia la primary key que ejecuta el decorador. Dicho filtro elimina aquellas tablas que ya tienen una foreign key relacionada con la primary key en cuestión. A las tablas que pasan éste primer filtro se le aplica un segundo filtro. Este segundo filtro elimina de las tablas que pasan aquellas columnas que ya forman una foreign key y su estado es *"ENABLED"*. Esto se realiza en la siguiente porción de código:

```
# Nos quedamos con aquellas tablas que no tengan ninguna Fk hacia la tabla de
# la Pk.
toDelete = Array.new
tables.each do |table|
  table.columnsFk.each do |fk|
    if (fk.references == table_Pk)
      toDelete << table
    end
  end
end
toDelete.each do |table|
  tables.delete(table)
end

# De todas las columnas de una tabla nos quedamos con aquellas que no son
# FK ENABLED.
tablesColumns = Array.new
tables.each do |table|
  columns = table.columns
  table.columns.each do |column|
    table.columnsFk.each do |fk|
      if((fk.status=="ENABLED")&&(fk.columnName.include?(column.columnName)))
        columns.delete(column)
      end
    end
  end
tablesColumns << columns
end
```

Para eliminar todas las tablas que presenta una foreign key que referencia a la primary key lo que se hace es guardar en el array *'toDelete'* todas aquellas tablas que tiene alguna foreign key que apunta a la primary key. Una vez se obtienen todas estas tablas se eliminan de una lista que contiene todas las tablas de la base de datos, menos la tabla de la primary key mediante el método *'delete'*.

A todas las tablas que restan se le aplica el segundo filtro. Dicho filtro comprueba si cada columna da cada tabla pertenece a alguna de las foreing keys que pueda tener dicha tabla relacionadas con la primary key en cuestión y si su estado es "*ENABLED"*, en dicho caso se elimina de las columnas que se comprobarán para obtener la nueva posible foreign key.

Una vez que se tienen las posibles tablas con las posibles columnas hay que comprobar tabla a tabla y columna a columna si cumplen las restricciones comentadas anteriormente. Para ello

se guardan en dos arrays aparte los diferentes tipos y los valores que tienen las columnas que forman la primary key.

```
# Obtenemos los tipos de las columnas que forman la Pk.
table_Pk.columns.each do |tableColumn|
  columns_Pk.each do |col|
    if (col == tableColumn.columnName)
      type_Pk << tableColumn.columnType
    end
  end
end

# Obtenemos los valores de las columnas que forman la Pk.
columns_Pk.each do |col|
  values = Array.new # Array para almacenar los Values.
  table_Pk.registries.each do |r|
    registryValues = r.registryValues # Obtenemos los registryValues de un
                                      # registry en concreto.
    registryValues.each do |rv|
      if (rv.column.columnName == col) # Comprobamos que es el valor
                                       # de la columna.
        values << rv.value
      end
    end
  end
  values_Pk << values
end
```

Una vez se tienen todos los valores y los tipos de las columnas que forman la primary key se procede a obtener la tabla que albergará la posible nueva foreign key. Esto se realiza mediante la siguiente porción de código del método *'getPorcent'* del decorador:

```
for i in 0..tables.length-1
  table = tables[i]
  columns = tablesColumns[i]

  # Array para almacenar la columna con mejor acierto
  best_column_Pk = Array.new
  # Entero para almacenar el % de aciertos para esa columna
  best_hits_Pk = Array.new

  for j in 0..columns_Pk.length-1
    column_Pk = columns_Pk[j]

    best_hits = -1
    best_column_id = -1
    total = 0
    for k in 0..columns.length-1
      aciertos = 0
      if (((columns[k].columnName.include?(column_Pk)) ||
(column_Pk.include?(columns[k].columnName))) && (columns[k].columnType ==
type_Pk[j]))

        values = Array.new
        # Obtenemos el número total de valores que hay en tablaha comprobar
        total = table.registries.length

        table.registries.each do |r|
          registryValues = r.registryValues # Obtenemos los registryValues
                                            #de un registry en concreto.
          registryValues.each do |rv|
            if (rv.column.columnName == columns[k].columnName)
```

```
              values << rv.value
            end
          end
        end
        values.each do |value|
          if (values_Pk[j].include?(value))
            aciertos += 1
          end
        end

        if (best_hits <= aciertos)
          best_hits = aciertos
          best_column_id = k
        end
      end
    end

    # En este punto tendré la columna con más número de aciertos para una
    # columna de la Pk
    # En best_column_id tendre el index para obtener la columna de columns
    # y en best_hits el numero de aciertos para esa columna

    # Comprobamos que hay algun acierto
    if (best_hits >= 0)
      if (total == 0)
        porcent_column = 0
      else
        porcent_column = (best_hits*100).div(total)
      end
      best_column_Pk << columns[best_column_id]
      best_hits_Pk << porcent_column
    end
  end

  # Calculamos el % total de esta tabla
  porcent = 0
  if (best_column_Pk.length == columns_Pk.length)
    best_hits_Pk.each do |hits|
      porcent += hits
    end
    porcent = porcent.div(columns_Pk.length)
  end

  # Nos quedamos con la tabla de mayor %
  if (best_porcent < porcent)
    best_porcent = porcent
    best_table_id = i
    best_table_columns.clear
    best_column_Pk.each do |col|
      best_table_columns << col
    end
  end
end
```

Para cada una de las tablas obtenidas del filtro se recorren todas las columnas que forman la primary key. Para cada una de las columnas de la primary key se busca aquella columna de la tabla (de las columnas que quedaron cuando se aplico el segundo filtro) cuyo nombre sea similar, es decir, el nombre la columna de la posible foreign key este contenido en el nombre de la columna de la primary key o viceversa y que sean del mismo tipo. En caso de que se encuentre una columna de la tabla que cumpla las restricciones se calcula el porcentaje de aciertos entre ambas columnas. Este porcentaje es usado para decidir, en el caso de que en

una tabla existan más de una posible columna que cumpla las restricciones, cual es la mejor de todas las posibles columnas.

Una vez que se han obtenido las mejores columnas que cumplen las restricciones para una tabla, se comprueba si esa tabla es la mejor para contener la posible foreign key asociada a la primary key. Para realizar dicha comprobación lo que se hace es como se han guardado los valores de tanto por ciento de las mejores columnas de la tabla, se realiza una media, es decir, se suman los valores del tanto por ciento de cada columna y se divide por el número de columnas que formarán la foreign key (que será el mismo que columnas posee la primary key).

Si al finalizar la búsqueda en todas las tablas no se ha encontrado ninguna que cumpla con las restricciones, la variable *'best_table_id'* tendrá el valor *'-1'*, por lo que el porcentaje que se devolverá será 0. De esta manera se asegura que el porcentaje devuelto es menor o igual que el porcentaje de corte de la restricción existente en el decorador asociado al elemento *'DDLDefinition'* del modelo DML_DDL. Así no se añade dicha primary key al conjunto de primary key con una posible nueva foreing key.

```
if (best_table_id == -1)
  return 0
else
  @tableRef = tables[best_table_id]
  @columnsRef = best_table_columns
  return best_porcent
end
```

En el método del decorador de la *'Pk'* del modelo DML_DDL *'getRegistries'* se obtienen los registros que no cumplen la posible nueva foreign key.

En dicho método se comprueba qué registros de la tabla que tiene la supuesta nueva foreign key no la cumplen. Para ello, para cada una de las columnas que formarán la foreign key se recorre el conjunto de registros que tiene la tabla. Se obtiene para cada registro el valor correspondiente a la columna de la foreign key. Para dicho valor se recorren todos los registros de la tabla a la que referencia la foreign key, es decir, se recorren todos los registros de la tabla que contiene la primary key que ejecuta el decorador y se busca la columna a la cual referencia la columna de la foreign key. Si sus valores coinciden, el registro al que corresponde el valor no se añade al conjunto de valores a devolver por el método. En otro caso se añade el registro al array que contendrá los registros que no cumplen la nueva foreign key.

```
for i in 0..columnsFk.length-1
  registriesTableCont.each do |registryCont|
    registryValuesCont = registryCont.registryValues
    registryValuesCont.each do |valueCont|
      if (valueCont.column.columnName == columnsFk[i])
        aux = false
        registriesTableRef.each do |registryRef|
          registryValuesRef = registryRef.registryValues
          registryValuesRef.each do |valueRef|
            if ((columnsPk[i] == valueRef.column.columnName) &&
                                        (valueCont.value == valueRef.value))
              aux = true
              break
            end
```

```
            end
            if (aux)
              break
            end
          end

          if (!aux)
            registries << registryCont
            break
          end
        end
      end
    end
end

return registries
```

Como se puede observar es totalmente análogo al método para obtener los registros erróneos del decorador para la *'Fk'* ya que se hace exactamente lo mismo. Buscar los registros de una tabla que no cumplen la foreign key asociada a una primary key. Es indiferente si esa foreign key existe o se propone para ser creada.

6 Corrección de Errores

En está etapa se pretende corregir todos los posibles errores que se hayan detectado en la etapa anterior, creando un modelo de datos corregido y otro residual. Para llevar a cabo la corrección se ha usado RubyTL.

Una vez que se han conseguido obtener todos los errores que presenta el sistema de información, el siguiente paso sería la corrección de los errores que se han obtenido, así como la obtención de un modelo de datos residual con toda la información que se ha eliminado o modificado del modelo de datos con los errores.

Para la corrección de los errores que pueda contener el sistema de información que se está tratando y obtención del modelo residual se ha vuelto a hacer uso de RubyTL.

Para poder ejecutar RubyTL en la corrección de los errores es necesario el modelo DML_DDL obtenido en el apartado de unificación de los modelos DML y DDL, su metamodelo DML_DDL. Además es necesario el modelo con los errores que se han encontrado en el modelo DML_DDL, el modelo Errores. Se necesita también de su metamodelo Errores. El metamodelo que deberá conformar el modelo corregido será el metamodelo DML_DDL ya que es el que da el soporte para poder almacenar toda la información de un sistema de información. Es necesario también el conjunto de reglas que corregirán el modelo Errores de entrada en el modelo DML_DDL corregido de salida. En éste caso se han tenido que definir una serie de decoradores (almacenados en el directorio *'helpers'*) para ayudar a las reglas a corregir el modelo DML_DDL.

En la siguiente figura se puede observar un diagrama general de la fase de corrección de errores:

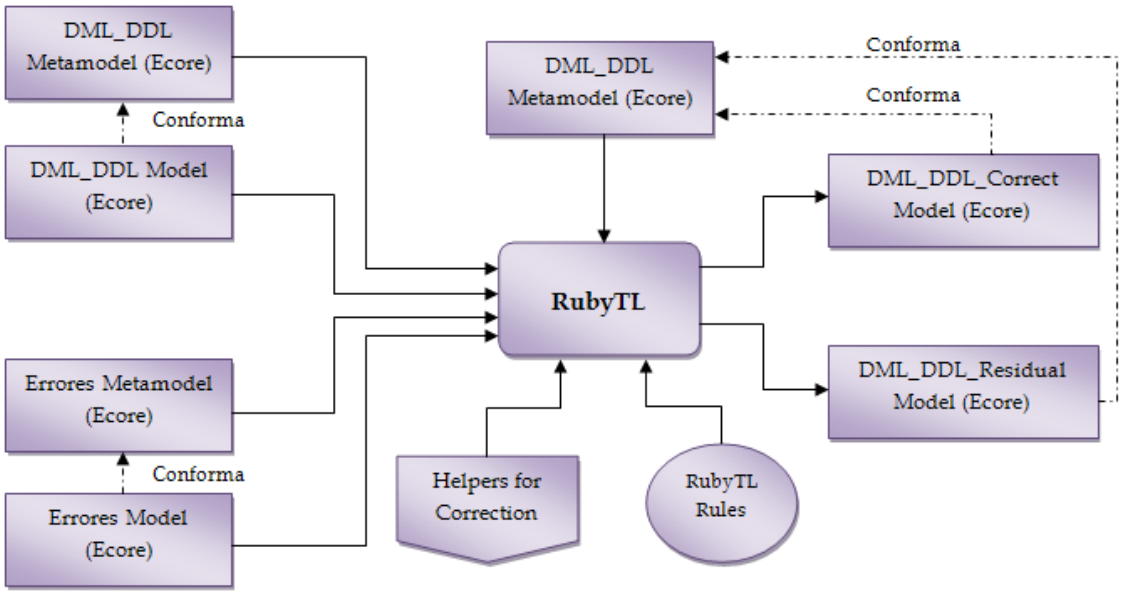

Figura 17: Obtención de los modelos DML_DDL_Correct y DML_DDL_Residual mediante RubyTL.

Como se pude observar en el diagrama anterior se hace uso de los metamodelos definidos anteriormente. En concreto del metamodelo DML_DDL y del metamodelo Errores. Por lo que es ésta fase no es necesario crear ni usar un nuevo metamodelo.
La corrección del modelo DML_DDL mediante el modelo Errores para obtener del modelo DML_DDL_Correct se realiza mediante un conjunto de reglas definidas en un fichero de RubyTL. Mientras que el conjunto de reglas que se usarán para obtener el modelo DML_DDL_Residual a partir del modelo Errores y DML_DDL se definen en otro fichero RubyTL aparte. Se comentarán cada una de las dos fases en las que se compone ésta etapa por separado.

6.1 Corrección

Mediante el conjunto de reglas que se han definido lo que se pretende es obtener un nuevo modelo que conforme el metamodelo DML_DDL en el que los posibles errores que se hayan obtenido en la fase anterior se hayan resuelto.

En realidad lo que las reglas realizan no es más que una mera copia del modelo DML_DDL que se pasa cómo parámetro. Solo en caso de que una check constraint o una foreign key presenten un error se harán las operaciones necesarios para corregirlos y si aparece una posible nueva foreign key se hará lo necesario para crearla.

El conjunto de reglas se encuentran íntegramente en el apartado de “Anexos”. A continuación se comentaran únicamente las reglas relevantes, es decir, aquellas que realizan algo más que pasar del modelo DML_DDL al modelo DML_DDL_Correct.

La corrección del modelo DML_DDL a partir de la información que existe en el modelo Errores consiste en corregir la *‘Ck’* o la *‘Fk’* en caso de que presenten algún error, modificar su estado a *“ENABLED”* y eliminar aquellos registros de la tabla que no cumplen la *‘Ck’* o la *‘Fk’* que se acaba de activar. En el caso de que el error sea del tipo *‘possible_new_Fk’* lo que se tiene que hacer es crear un nuevo elemento *‘Fk’*, con estado *“ENABLED”* y eliminar de la tabla que contendrá la nueva *‘Fk’* aquellos registros que no la cumplen.

Por lo tanto las únicas reglas que se explicarán son las que están relacionadas con las metaclases *‘Table’*, *‘Ck’* y *‘Fk’* ya que el resto serán, como se ha dicho antes, un simple mapeo del modelo DML_DDL al modelo DML_DDL_Correct.

Mediante la regla para corregir las tablas contenidas en el modelo DML_DDL se obtienen los nuevos registros que debería tener la tabla si presenta algún error y las nuevas *‘Fk’* en el caso de que se haya encontrando algún error del tipo *‘possible_new_Fk’*. La regla es la siguiente:

```
rule 'Table' do
  from DML_DDL::Table
  to Correct::Table
  mapping do |dml_ddl, correct|
```

```
    correct.tableName = dml_ddl.tableName
    correct.columns = dml_ddl.columns
    correct.columnsPk = dml_ddl.columnsPk

    correct.columnsFk = dml_ddl.columnsFk

    correct.checks = dml_ddl.checks
    correct.commentTable = dml_ddl.commentTable

    if (dml_ddl.hasError)
      correct.registries = dml_ddl.getRegistries
      newsFk = dml_ddl.getNewsFk
      newsFk.each do |fk|
        correct.columnsFk << fk
      end
    else
      correct.registries = dml_ddl.registries
    end
  end
end
```

Como se puede observar, en las primeras seis líneas realizan una copia desde el modelo DML_DDL al nuevo modelo DML_DDL_Correct. La corrección de la tabla empieza con el *'if'*.
En el *'if'* se llama al método *'hasError'* definido en el decorador de *'Table'*. En dicho método se comprueba si la tabla presenta algún error de cualquier tipo. En caso de que tenga un error se llama al método *'getRegistries'* para obtener los registros que deberá tener la tabla, es decir, de todos los registros que presenta la tabla se eliminan los que presentan algún error. Llama también al método *'getNewsFk'* para obtener las posibles nuevas *'Fk'* en el caso de que exista algún error del tipo *'possible_new_Fk'* asociado a dicha tabla. Las nuevas *'Fk'* se añaden a la lista con las foreign key que contiene la tabla. Si la tabla no presenta ningún error los registros que tenía en el modelo DML_DDL son copiados, sin realizarles modificación alguna, al modelo DML_DDL_Correct.

El decorador asociado a la tabla definido en el fichero *'correct_BD.rb'* del directorio *'helpers'* presenta únicamente los métodos que se han comentado en el párrafo anterior. Contiene los métodos *'hasError'*, *'getRegistries'* y *'getNewsFk'*.

El método *'hasError'* comprueba si una tabla contiene algún error. El método presenta el siguiente aspecto:

```
def hasError
# Comprobamos si la tabla presenta algun error.
# Para ello comprobamos si esta referenciada en ForeginError o CheckError

  @foreignErrors = Errores::ForeignError.all_objects.select{|error|
                                error.tableCont == self && error.apply == true}

  @checkErrors = Errores::CheckError.all_objects.select{|error|
                                    error.table == self && error.apply == true}

  return (!(@foreignErrors.empty?) || !(@checkErrors.empty?))
end
```

Se obtienen todos los *'ForeignError'* del modelo de Errores que tienen su atributo *'apply'* a *"true"* lo cual indica que el error se desea corregir. En el caso de que *'apply'* estuviese a *"false"* el error no se corregiría. Además de que el *'ForeignError'* deba tener *'apply'* a *"true"*, su tabla contenedora debe ser la tabla que ejecuta el decorador, es decir, que la referencia *'tableCont'* debe ser igual a *'self'* (la tabla que ejecuta el decorador). Todos los *'ForeignError'* que presenten las características anteriores se guardan en una variable global al decorador (a la tabla) llamada *'@foreignErrors'*.

Para los *'CheckError'* se realiza la operación análoga. Se seleccionan todos aquellos *'CheckError'* cuyo atributo *'apply'* vale *"true"* y cuya referencia *'table'* es la tabla que ejecuta el decorador. Todos los *'CheckError'* encontrados se guardan en la variable, global al decorador, *'@checkErrors'*.

Como resultado el método devolverá *"true"* si alguna de las variables *'@foreingErrors'* y *'@checkErrors'* contiene algún elemento, al tratarse de dos arrays que almacenan *'ForeignError'* y *'CheckError'* respectivamente. En caso de que ambos arrays estén vacíos se devuelve *"false"*. El método *'empty?'* devolverá *"true"* si el array está vacío.

El método *'getRegistries'* actualiza los registros de una tabla eliminando aquellos registros que no cumplen la '*Ck'* o la '*Fk'* que se quiere habilitar. Presenta el siguiente aspecto:

```
def getRegistries
  registries = self.registries

  if (!@checkErrors.empty?)
    for i in 0..@checkErrors.length-1
      @checkErrors[i].registriesCk.each do |regCk|
        registries.delete(regCk)
      end
    end
  end

  if (!@foreignErrors.empty?)
    for i in 0..@foreignErrors.length-1
      @foreignErrors[i].registriesFk.each do |regFk|
        registries.delete(regFk)
      end
    end
  end

  return registries
end
```

En la fase de detección de errores se obtenían los registros que no cumplía la supuesta *'Ck'* o *'Fk'* que se encontraba deshabilitada o se pretendía crear, es decir, aquellos registros que presentaban algún error. Por lo que éste método lo único que hace es eliminar de todos los registros que presenta la tabla aquellos que se encuentren referenciados en los elementos *'ForeingError'* (referencia *'registriesFk'*) y *'CheckError'* (referencia *'registriesCk'*).

Para ello guardamos en un nuevo array todos los registros de la tabla. Luego se comprueba si en las variables *'@foreignErrors'* y *'@checkErrors'* existe algún elemento. En caso de que presenten algún elemento se recorre el conjunto de registros que presentan algún error y se eliminan del array que contiene todos los registros de la tabla.

Al finalizar se devuelve el array en el que se habían copiado los registros de la tabla al cual se le han eliminado los registros contenidos en los *'CheckError'* y en los '*ForeingError'*.

En el método *'getNewsFk'* se obtiene un array con las nuevas foreign key que se deben crear. Presenta el siguiente aspecto:

```
def getNewsFk
  newsFk = Array.new
  if (!@foreignErrors.empty?)
    for i in 0..@foreignErrors.length-1
      if (@foreignErrors[i].fk == nil)
        fkError = @foreignErrors[i]
        name = "NEW_FK_" + @@id.to_s
        fk = Correct::Fk.new(:nameFk => name)
        @@id.next

        columnName = Array.new
        fkError.fkColumns.each do |column|
          columnName << column.columnName
        end
        fk.columnName = columnName
        fk.references = fkError.tableRef
        fk.columnReference = fkError.tableRef.columnsPk.columnName
        fk.status = "ENABLED"
        newsFk << fk
      end
    end
  end

  return newsFk
end
```

Primero hay que comprobar si el array con todos los errores *'ForeignError'* del modelo Errores presenta algún elemento. En caso de que no esté vacío se recorre dicho array. Para cada uno de los *'ForeignError'* que se encuentre en el array tenemos que comprobar si se trata del tipo *'possible_new_Fk'* y no es del tipo *'Fk_disabled'*. Para ello, lo único que se hace es comprobar si la referencia *'fk'* del elemento *'ForeignError'* es nula. En caso de ser nula se trata de un error del tipo *'possible_new_Fk'*.

Para todos los *'ForeignError'* del tipo *'possible_new_Fk'* se crea un nuevo elemento *'Fk'* del modelo DML_DDL_Correct mediante la sentencia *'Correct::Fk.new'*. El nombre de la nueva foreign key se obtiene mediante la unión del string *"NEW_FK_"* seguido de un identificador que se incrementa cada vez que se crea una nueva foreign key, variable *'@@id'* global a la transformación. Una vez que se tiene la nueva *'Fk'* creada hay que inicializar sus valores. Para ello se le asigna como atributo *'columnName'* todas las columnas que formarán la foreign key, es decir, aquellas columnas contenidas en el conjunto *'fkColumns'*. Como atributo *'references'*

de la *'Fk'*, se le asigna la tabla a la que referencia el elemento *'ForeignError'*, la tabla referenciada en *'tableRef'* (recordemos que se tratará de la tabla que contiene la primary key que obtuvo el error en la fase anterior) y como columnas a las que referenciar, atributo *'columnReference'* de *'Fk'*, aquellas columnas que forman la primary key a la que referencia, se navegará hasta ellas de la siguiente manera: *'foreingError.tableRef.columnsPk.columnName'*. El estado se pone a *"ENABLED"*.

Al finalizar de recorrer todos los errores de *'ForeignError'* se devolverá el array con las nuevas *'Fk'*.

En la regla *'Ck'* lo único que se hace es comprobar si la check constraint que entra en la regla presenta algún error. En caso de presentar algún error y querer corregirse, se cambia su estado de *"DISABLED"* a *"ENABLED"*. La regla *'Ck'* se puede ver a continuación:

```
rule 'Ck' do
  from DML_DDL::Ck
  to Correct::Ck
  mapping do |dml_ddl, correct|
    correct.nameCk = dml_ddl.nameCk
    correct.valuesCk = dml_ddl.valuesCk
    if (dml_ddl.hasError)
      correct.status = "ENABLED"
    else
      correct.status = dml_ddl.status
    end
  end
end
```

Se llama al método *'hasError'* del decorador de las *'Ck'* del modelo DML_DDL para comprobar si la check constraint que entra en la regla presenta algún error. En caso de que tenga un error y quiera ser corregido su estado se pone a *"ENABLED"*. En caso de que la *'Ck'* no presente ningún error o lo presente pero no se desee corregirlo el estado de la *'Ck'* en el modelo DML_DDL_Correct será el mismo que el de la *'Ck'* del modelo DML_DDL. Por lo que el único atributo de la *'Ck'* del modelo DML_DDL_Correct que se modifica con respecto a la *'Ck'* del modelo DML_DDL será su estado.

El decorador de la *'Ck'* es el siguiente:

```
decorator DML_DDL::Ck do
  def hasError
    error = Errores::CheckError.all_objects.select{|error| error.ck == self
                                                    && error.apply == true}

    return !(error.empty?)
  end
end
```

En él se almacenan en una variable todos los *'CheckError'* que tengan su atributo *'apply'* a true y el atributo *'ck'* sea igual a la *'Ck'* que ejecuta el decorador.

Se devolverá *"true"* si el array con todos los *'CheckError'* presenta algún elemento y *"false"* en caso contrario.

En la regla *'Fk'* lo único que se hace es comprobar si la foreign key que entra en la regla presenta algún error. En caso de presentar algún error y querer corregirse, se cambia su estado de *"DISABLED"* a *"ENABLED"*. La regla *'Fk'* se puede ver a continuación:

```
rule 'Fk' do
  from DML_DDL::Fk
  to Correct::Fk
  mapping do |dml_ddl, correct|
    correct.nameFk = dml_ddl.nameFk
    correct.columnName = dml_ddl.columnName
    correct.references = dml_ddl.references
    correct.columnReference = dml_ddl.columnReference
    if (dml_ddl.hasError)
      correct.status = "ENABLED"
    else
      correct.status = dml_ddl.status
    end
  end
end
```

Se llama al método *'hasError'* del decorador de las *'Fk'* del modelo DML_DDL para comprobar si la foreign key que entra en la regla presenta algún error. En caso de que tenga un error y quiera ser corregido su estado se pone a *"ENABLED"*. En caso de que la *'Fk'* no presente ningún error o lo presente pero no se desee corregirlo el estado de la *'Fk'* en el modelo DML_DDL_Correct será el mismo que el de la *'Fk'* del modelo DML_DDL. Por lo que el único atributo de la *'Fk'* del modelo DML_DDL_Correct que se modifique con respecto a la *'Fk'* del modelo DML_DDL será su estado.

El decorador de la *'Fk'* es el siguiente:

```
decorator DML_DDL::Fk do
  def hasError
    error = Errores::ForeignError.all_objects.select{|error| error.fk == self
                                                    && error.apply == true}

    return !(error.empty?)
  end
end
```

En él se almacenan en una variable todos los *'ForeignError'* que tengan su atributo *'apply'* a true y el atributo *'fk'* sea igual a la *'Fk'* que ejecuta el decorador.

Se devolverá *"true"* si el array con todos los *'ForeignError'* presenta algún elemento y *"false"* en caso contrario.

6.2 Residual

Mediante el conjunto de reglas que se han definido lo que se pretende es obtener, en un modelo que conforme el metamodelo DML_DDL, la información residual que quedaría de corregir el modelo DML_DDL.

En dicho modelo residual únicamente se guardaran las tablas que presentan algún error, sus columnas y los registros erróneos. En realidad solamente interesan los registros erróneos que serán los que se eliminen del modelo DML_DDL al corregirlo y obtener el modelo DML_DDL_Correct, pero para dar un poco de sentido a la información guardada se almacenarán también las tablas con sus columnas. De manera que mediante el modelo DML_DDL_Correct y el modelo DML_DDL_Residual se pueda recuperar el modelo DML_DDL inicial.

El conjunto de reglas se encuentran íntegramente en el apartado de "Anexos". A continuación se comentaran únicamente las reglas más relevantes.

En la regla *'DDLDefinition'* se realiza un filtro para asignarle a la referencia con todos los *'Statement'* únicamente aquellas tablas que presentan algún tipo de error. Comentar que en la reglas de cada una de las metaclases hay atributos a los que no se les da valor (no salen en los *mappings*), ya que el asignarles algún valor implicaría crear la regla concreta que se lanzaría en la asignación. Pero como se trata de un modelo con datos residuales nos interesa tener la mínima información, que es la comentada anteriormente. De ahí que se realice este filtro.

Para obtener el conjunto de tablas que presentan algún error se ha hecho uso de un decorador que se ha definido en el fichero *'residual_BD.rb'* del directorio '*helpers'*. La regla tendrá el siguiente aspecto:

```
top_rule 'DDLDefinition' do
  from DML_DDL::DDLDefinition
  to Residual::DDLDefinition
  mapping do |dml_ddl, res|
    res.statements = dml_ddl.getStatements
  end
end
```

Como se puede apreciar será la primera regla que se ejecute ya que es la única del conjunto de reglas que es de tipo *'top_rule'*.
Mediante el método '*getStatments'* del decorador obtenemos únicamente las tablas con algún error. Por ello en el conjunto de las reglas solamente existirán reglas para transformar las tablas, los registros con sus valores y las columnas.

El decorador de '*DDLDefinition'* es el siguiente:

```
decorator DML_DDL::DDLDefinition do
  def getStatements
    # Lo que hacemos es comprobar para cada statements si es de tipo Tabla,
```

```
    # entonces solamente
    # se añadirá al array si dicha tabla presenta algún error, en otro caso
    # no se añade para que así no sea añadida al metamodelo residual
    statements = Array.new
    self.statements.each do |statement|
      if (statement.kind_of? DML_DDL::Table)
        if (statement.hasError)
          statements << statement
        end
      end
    end
    return statements
  end
end
```

Como se puede ver lo que se realiza es lo siguiente: para todas las sentencias del elemento '*DDLDefinition*' se comprueba si dicha sentencia es de tipo '*Table*'. En caso de que sea de tipo '*Table*' se comprueba si presenta algún error. En caso de que presente algún error se añade al array que contendrá las sentencias que se devolverán.

Para cada una de las tablas que presentan algún error se ejecuta la regla '*Table*'. En dicha regla lo único que se hace es asignarle a la tabla del modelo DML_DDL_Residual las columnas y el nombre de la tabla del modelo DML_DDL, así como obtener el conjunto de registros de la tabla del modelo DML_DDL que presentan algún error. La regla se puede ver a continuación:

```
rule 'Table' do
  from DML_DDL::Table
  to Residual::Table
  filter {|dml_ddl| dml_ddl.hasError == true}
  mapping do |dml_ddl, res|
    res.tableName = dml_ddl.tableName
    res.columns = dml_ddl.columns

    res.registries = dml_ddl.getRegistries
  end
end
```

Nos aseguramos que la tabla que pretende ejecutar la regla '*Table*' presenta algún error. Para ello en el apartado '*filter*' de la regla se ejecuta de nuevo el método *'hasError'* del decorador del elemento '*Table*' definido en el fichero *'residual_BD.rb'* del directorio '*helpers*'. Mediante dicho método se sabrá si la tabla presenta algún error o no. También se hace uso del método '*getRegistries*' para obtener el conjunto de registros erróneos que tiene la tabla.

El decorador del elemento '*Table*' tendrá el siguiente aspecto:

```
decorator DML_DDL::Table do
  def hasError
    # Comprobamos si la tabla presenta algun error.
    # Para ello comprobamos si esta referenciada en ForeginError o CheckError

    @foreignErrors = Errores::ForeignError.all_objects.select{|error|
                          error.tableCont == self && error.apply == true &&
                                                !(error.registriesFk.empty?)}

    @checkErrors = Errores::CheckError.all_objects.select{|error|
                                  error.table == self && error.apply == true &&
```

```
                                                    !(error.registriesCk.empty?)}

    return (!(@foreignErrors.empty?) || !(@checkErrors.empty?))
  end

  def getRegistries
    registries = Array.new

    if (!@checkErrors.empty?)
      for i in 0..@checkErrors.length-1
        @checkErrors[i].registriesCk.each do |regCk|
          registries << regCk
        end
      end
    end

    if (!@foreignErrors.empty?)
      for i in 0..@foreignErrors.length-1
        @foreignErrors[i].registriesFk.each do |regFk|
          registries << regFk
        end
      end
    end

    return registries
  end
end
```

El método '*hasError'* es exactamente análogo al método del '*hasError'* del decorador asociado al elemento '*Table'* definido en el fichero *'correct_BD.rb'* del directorio '*helpers'* explicado en la fase de corrección del modelo DML_DDL. Pero con la diferencia de que se añade otro parámetro al filtro. Ya que los errores que se buscan aparte de querer ser corregidos y de referenciar a la tabla que ejecuta el decorador, deben tener algún registro en sus referencias a los registros erróneos. Con esto conseguimos que solamente los errores que tengan presente registros erróneos sean seleccionados.

Una vez que tenemos seleccionados los errores, en el método *'getRegistries'* almacenamos todos los registros en un array que es el que devuelve dicho método.

Las reglas que quedan son simples creaciones de las columnas los registros y los valores de los registros en el nuevo modelo DML_DDL_Residual. Pero con la salvedad de que hay atributos de los elementos a los que no se les da valor alguno ya que no es de nuestro interés dicha información para el modelo residual.

7 Generación de código

En está etapa se pretende generar código a partir del modelo de datos del sistema de información corregido que se obtuvo en la etapa anterior. Para ello se usará RubyTL para realizar una transformación model-to-code a código SQL.

Ésta es la última etapa del proyecto. En ella se pretende mediante RubyTL y teniendo como entrada el modelo DML_DDL_Correct, la obtención de dos scripts con sentencias SQL del ANSI SQL-92 que se correspondan con la base de datos del sistema de información del que se partió pero corregida. En un script se tendrán las sentencias DDL y en otro las sentencias DML.

RubyTL permite realizar transformaciones modelo a código como se comentó en el apartado de RubyTL. Para realizar una transformación modelo a código se tienen que definir dos tipos de ficheros: el fichero '*2code'* y las templates (fichero '*.rtemplate'*).

En la siguiente figura se puede observar un diagrama general de la fase de obtención de código:

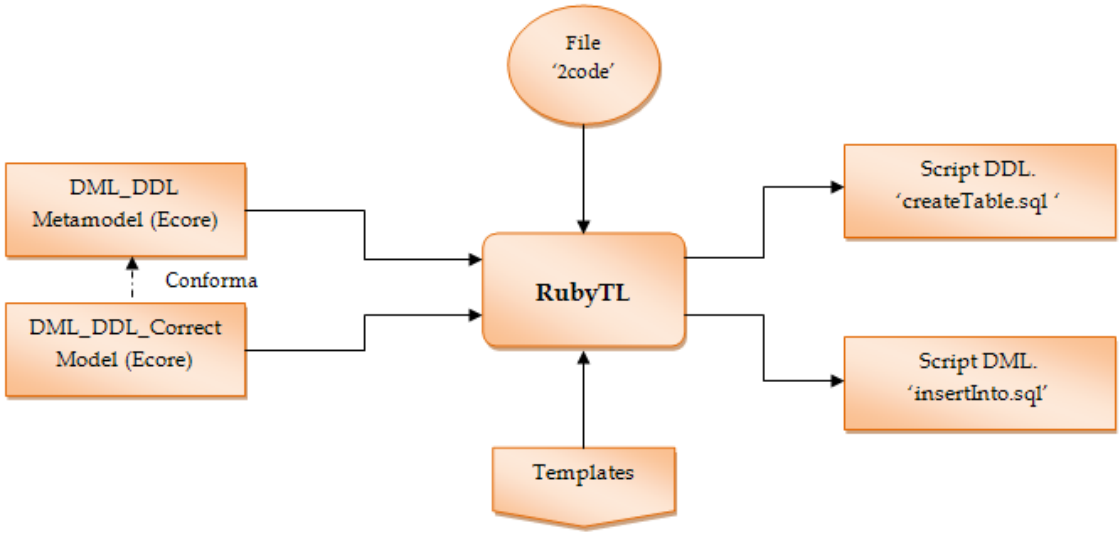

Figura 18: Obtención de los ficheros SQL mediante RubyTL.

La idea es ir recorriendo el modelo DML_DDL_Correct que contiene los datos del sistema de información para componer los scripts DML y DDL. Recordemos que en el script DDL se encuentra la información relacionada con la estructura del sistema de información, es decir, sentencias SQL del tipo *'CREATE TABLE'* con la definición de sus columnas y con sentencias *'ALTER TABLE DROP CONSTRAINT'*. Mientras que en el script DML se encuentra la información relacionada con los datos del sistema de información, es decir, sentencias SQL del tipo *'INSERT INTO'*.

Se ha definido un fichero '*2code*' que teniendo como entrada el modelo DML_DDL_Correct obtiene dos ficheros, uno llamado '*createTable.sql*' que contiene las sentencias DDL y otro llamado '*insertInto.sql*' con las sentencias DML. El contenido del fichero '*modelToSql.2code*' es el siguiente:

```
main do
  compose_file 'createTable.sql' do |file|
    DML_DDL::Table.all_objects.each do |table|
     apply_template 'templates/create_table.rtemplate', :table => table
      end
  end

  compose_file 'insertInto.sql' do |file|
    DML_DDL::Table.all_objects.each do |table|
     apply_template 'templates/insert_into.rtemplate', :table => table
      end
  end
end
```

Como se observa dentro del '*main*' existen dos '*compose_file*' uno para cada uno de los ficheros. Para cada uno de los '*compose_file*' lo que se hace es recorrer todas las tablas que se encuentran en el modelo DML_DDL_Correct aplicándole a cada tabla la plantilla correspondiente, es decir, pasando la tabla que se esta recorriendo como parámetro a la plantilla. El código de las plantillas puede verse íntegramente en el apartado de "Anexos".

Se han definido dos plantillas. Una para obtener las sentencias DDL llamada '*create_table.rtemplate*' y otra para obtener las sentencias DML llamada '*insert_into.rtemplate*'.

Mediante la plantilla '*create_table.rtemplate*' lo que se hace es construir la sentencia para crear cada tabla que se le pasa como parámetro. La estructura de la sentencia SQL '*CREATE TABLE*' del ANSI SQL-92 es la siguiente:

```
CREATE TABLE <nombre_tabla> (
  <nombre_campo> <tipo_datos(tamaño)> [NULL|NOT NULL] [DEFAULT <valor_por_defecto>],
  …
  CONSTRAINT <nombre> PRIMARY KEY (<nombre_campo>[ ,...n ]),
  CONSTRAINT <nombre> FOREIGN KEY (<nombre_campo>[ ,...n ]) REFERENCES
                                    <tabla_referenciada> (<nombre_campo> [ ,...n ] ) ,
  CONSTRAINT <nombre> CHECK (<nombre_columna> [<,>,=,<=,>=,<>] <valor> [AND|OR]
                                    <nombre_columna> IN (<valores_correctos))
) ;
```

Si la tabla presenta alguna check constraint o foreign key desactivada la plantilla '*create_table.rtemplate*' también crea la sentencia SQL '*ALTER TABLE DROP CONSTRAINT*' del ANSI SQL-92. A continuación puede recordarse su sintaxis:

```
ALTER TABLE <nombre_tabla> DROP CONSTRAINT <nombre_constraint>;
```

La plantilla *'create_table.rtemplate'* obtiene en diferentes variables mediante una porción de código escrita en Ruby, por estar encerrado entre *'<% -%>'*, toda la información de la tabla relacionada con la sintaxis de la sentencia SQL según el formato del ANSI SQL-92. Es decir, en el array '*columns'* se guardan cada una de las columnas bajo el formato de la sentencia. En el array '*fk'* se guardan todas las foreign keys con el formato de la sentencia SQL. En el array '*ck'* todas las check constraints de la tabla en el formato de la sentencia y en el string '*pk'* guardamos la sentencia para crear una primary key. A continuación se muestra la porción de código da la plantilla que obtiene todas las columnas en el formato de la sentencia:

```
table.columns.each do |column|
  null = "NOT NULL"
  if (column.columnNull)
       null = "NULL"
  end

  columns << column.columnName + " " + column.columnType.name + " " + null +
","
end
```

Como se puede ver, se recorren todas las columnas de la tabla pasada como parámetro. Para cada columna se comprueba si puede ser nula o no, almacenando en el string '*null'* el resultado *"NOT NULL"* o *"NULL"*. Se crea un string con el nombre de la columna seguido de su tipo y del valor de la variable '*null'* y ese resultado se almacena en el array '*columns'*.

Además de la información almacenada anteriormente, como se ésta recorriendo la tabla se obtienen todas las sentencias del tipo *'ALTER TABLE DROP CONSTRAINT'* para las check constraint y foreign key que estén en estado *"DISABLED"*. A continuación el código para el caso de las foreign keys:

```
table.columnsFk.each do |f|
  if (f.status == "DISABLED")
    fkDisabled << "ALTER TABLE " + tableName + " DROP CONSTRAINT "
                                                          + f.nameFk + ";"
  end
end
```

Para cada una de las foreign keys que presenta la tabla pasada como parámetro se comprueba si su estado es *"DISABLED"*. En caso de estar desactivada se crea un string con la sentencia que indica que la foreign key esta desactivada: *"ALTER TABLE nombreTabla DROP CONSTRAINT foreingKeyName"*. Y lo almacena en el array '*fkDisabled'* que contendrá todas las sentencias de las foreign keys desactivadas.

Una vez que se ha extraído toda la información de la tabla necesaria para crear ambas sentencias DDL. Se recorren cada uno de los arrays anteriores y se imprimen por la salida, es decir, se imprimen en el fichero '*creataTable.sql'*. Esta impresión se lleva a cabo incluyendo entre los delimitadores especiales, *'<%= %>'*, la variable que se quiere mostrar por la salida. A continuación se muestra la porción del fichero '*create_table.rtemplate'* que recorre todas las variables inicializadas anteriormente para mostrar el contenido de cada una por la salida, es decir, para escribir el contenido de cada una de las variables en el fichero:

```
CREATE TABLE <%= tableName %> (
<% columns.each do |column| -%>
  <%= column %>
<% end -%>
<% ck.each do |check| -%>
  <%= check %>
<% end -%>
<% fk.each do |f| -%>
  <%= f %>
<% end -%>
  <%= pk %>
);
<% fkDisabled.each do |disabled| -%>
<%= disabled %>
<% end -%>
<% ckDisabled.each do |disabled| -%>
<%= disabled %>
<% end -%>
```

Primero tenemos las palabras reservadas *"CREATE TABLE"* seguidas del nombre de la tabla a crear. Dentro del paréntesis imprimimos: primero todas las columnas, segundo las check constraint, tercero las foreign key y por último la primary key. Una vez que se ha cerrado el paréntesis, se imprimirán las foreign keys y las check constraint desactivadas que contiene la tabla.

Mediante la plantilla '*insert_into.rtemplate'* lo que se hace es construir la sentencia de inserción de cada uno de los registros de tabla que se le pasa como parámetro. La estructura de la sentencia SQL '*INSERT INTO'* del ANSI SQL-92 es la siguiente:

INSERT INTO <nombre_tabla> (<nombre_columnas>[, ...n]) **VALUES** (<valores>[, …n])

La plantilla '*insert_into.rtemplate'* obtendrá tantas sentencias *'INSERT INTO'* como registros tenga la tabla que se pasa como parámetro a la plantilla. La idea es obtener una especie de cabecera con parte de la información de la sentencia SQL ya que se reutilizará para cada registro. Dicha cabecera contendría la siguiente información: *"INSERT INTO tableName (columns) VALUES "*. Entonces una vez que se tiene esta cabecera se recorren todos los registros de la tabla y se obtiene para cada registro su lista de valores con la sintaxis *"(value$_1$, value$_2$,…,value$_n$)"*, para añadirle como sufijo la cabecera obtenida anteriormente. Así obtendríamos para cada registro su sentencia SQL de inserción, es decir, su sentencia DML.

La obtención de la cabecera se muestra a continuación:

```
cols = table.columns
for i in 0..cols.length-1
  columns += cols[i].columnName
  if (i+1 <= cols.length-1)
    columns += ","
  end
end
columns += ")"

insert = "INSERT INTO " + tableName + " " + columns + " VALUES "
```

Como se puede observar se recorren todas las columnas de la tabla y se almacenan en un string con siguiente el formato: *"(column$_1$, column$_2$, …., column$_n$)"*. Una vez se tienen todas las columnas se crea la cabecera, la cuál se guardará en la variable '*insert*'.

Posteriormente para cada registro se obtiene la lista de sus valores en un string. A dicho string se le añade por delante la cabecera, variable '*insert*', y ese string que resulta se almacena en un array con todos los registros.

Una vez que se tienen todos los registros de la tabla en el array con el formato adecuado, es decir, el de la sentencia SQL, se muestra su contenido por salida. Se escribe en el fichero '*insertInto.sql*'. La porción de código de la plantilla que realiza esta operación es la siguiente:

```
<% registries.each do |registry| -%>
  <%= registry %>
<% end -%>
```

En la variable '*registry*' (cada uno de los registros del array '*registries*') habrá una sentencia de inserción según el formato del ANSI SQL-92.

8 Implementación de un plugin para la integración de las fases del proyecto

8.1 Introducción a PDE

La plataforma Eclipse puede ser extendida mediante plugins. La distribución básica de la plataforma cuenta con el *Plugin Development Environment* (PDE) [PDE]. El PDE es un entorno para el desarrollo de plugins y está diseñado para asistir a los programadores en el desarrollo, prueba, depuración y despliegue del plugin. Está compuesto por herramientas que permiten el desarrollo de los siguientes tipos de proyectos:

- *Plug-in Project*: un plugin normal.
- *Fragment Project*: una adición hecha a un plugin (como puede ser un nuevo idioma).
- *Feature Project*: una proyecto que contiene uno o más plugins.
- *Update Site Project*: un sitio web para la actualización automática de plugins.

El PDE es parte del SDK de Eclipse, y de acuerdo con la filosofía de la plataforma proporciona una gran cantidad de contribuciones (vistas, editores, asistentes, etc.) que se mezclan transparentemente con el resto del *Workbench*.

Un *Plug-in Project* representa un proyecto para la creación de un nuevo plugin. Puede ser de dos tipos distintos: *proyecto Java* o *proyecto simple*. La mayoría de los plugins contienen clases Java y por lo tanto serán de tipo proyecto Java. Si por el contrario el plugin no contiene clases, como podría ser un plugin de documentación, se puede usar un proyecto simple.

El PDE nos facilita la tarea de creación de un plugin proporcionando un asistente que genera parte del código de forma automática. En cada uno de los pasos, el asistente va solicitando información acerca del plugin que se desea crear, como su nombre e identificador. Por defecto, el identificador del plugin será igual al nombre dado al proyecto. Se recomienda usar identificadores únicos para evitar así posibles colisiones con otros plugins. Durante la ejecución del asistente también existe la posibilidad de especificar un directorio para los ficheros fuente y otro para los ficheros compilados. También se dispone de una serie de plantillas estándares que simplifican aún más el trabajo. Existen plantillas para la creación un plugin *Hola Mundo*, un plugin para la creación de un nuevo editor, un plugin que aporte una vista, etc.

En concreto para el plugin creado en este proyecto se ha usado la plantilla *'New File Wizard'*. Esta plantilla crea un asistente capaz de crear nuevos ficheros en el *'Workspace'*. Los ficheros creados por este wizard se pueden abrir mediante un editor Multi-Page.

Una vez terminado el proceso de creación del plugin, el PDE abre automáticamente el fichero de manifiesto *'plugin.xml'* usando la herramienta *'Plugin Manifest Editor'*. La principal función de este editor es hacer más sencillo el trabajo con este fichero, evitando tener que tratar con

en lenguaje XML directamente. Desde este editor es desde donde se controlaran todos los aspectos del plugin. En cualquier momento se puede acceder a este editor haciendo doble clic sobre el fichero *'plugin.xml'* que se habrá creado dentro del proyecto.

8.2 Introducción a EMF

Eclipse Modeling Framework [EMF] es un *framework* de modelado y generación de código para la construcción de herramientas y aplicaciones basadas en un modelo estructurado para Eclipse. Partiendo de la especificación de un modelo descrito en XMI, EMF proporciona herramientas para obtener un conjunto de clases Java para ese modelo.

Usando EMF se pueden transformar modelos en código Java de una manera sencilla, eficiente y fácilmente personalizable. EMF usa XMI como forma canónica para la definición de modelos. Existen varias formas de traducir un modelo a ese formato:

- Exportar el modelo desde una herramienta de modelado como Rational Rose.
- Usar interfaces Java decoradas con propiedades del modelo.
- Usar un esquema XML que describa la estructura de nuestro modelo.

Otra posibilidad es escribir directamente el fichero XMI, pero esta es la opción más complicada ya que este proceso no es muy intuitivo.

Una vez que se ha transformado nuestro modelo en un modelo EMF podemos crear el correspondiente conjunto de clases Java. Además se pueden editar las clases generadas añadiéndoles cualquier nuevo atributo o método y volver a generar el código a partir del modelo sin que se pierdan las modificaciones.

La construcción de una aplicación usando EMF, además de incrementar la productividad, proporciona otros beneficios tales como un mecanismo de notificación de eventos, soporte para persistencia incluyendo serialización XMI basada en esquemas XML y una API para manipular genéricamente los objetos EMF.

8.3 Plugin

Se ha creado un plugin que lanza un asistente llamado *'es.modelum.detErrores'*. Como se ha comentado en el párrafo anterior este plugin ha sido creado mediante la plantilla *'New File Wizard'*. Una vez el asistente ha terminado de crear el plugin, se ha modificado para soportar el wizard que se deseaba.

Resaltar que la estructura, diseño y contenido del plugin no serán explicados debido a que no forman parte de la finalidad de éste proyecto. Ya que la finalidad del plugin es la de facilitar al usuario la modernización de los datos de un sistema de información englobando las etapas 2, 3, 4 y 5 del proceso de modernización. En la siguiente figura se puede ver un esquema de lo que realizaría este plugin o asistente.

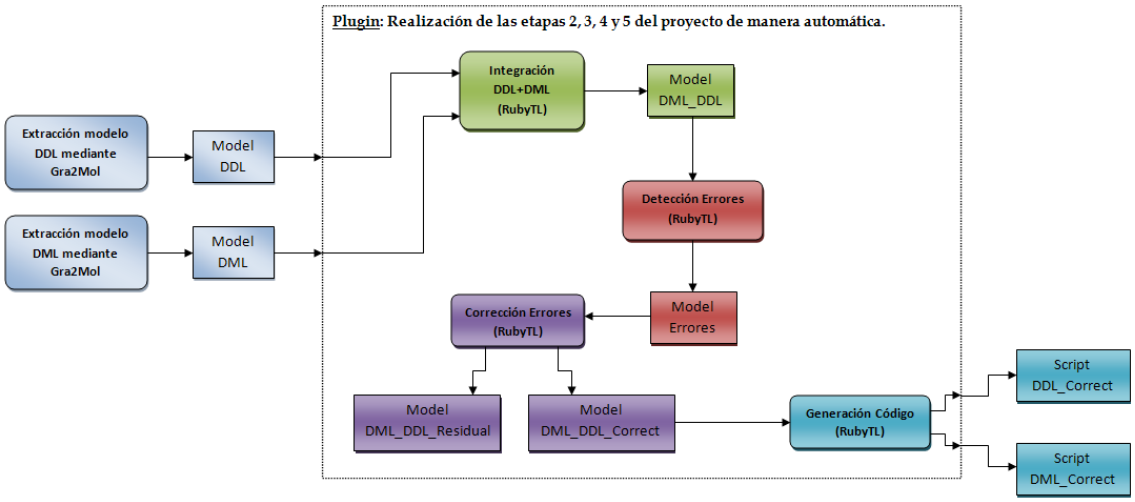

Figura 19: Esquema del plugin.

Se van ha comentar los aspectos más destacados del plugin.

Una vez que se ejecuta el plugin éste comprueba si en el espacio de trabajo de Eclipse existe algún proyecto de naturaleza RubyTL. En caso de que exista un proyecto de dicha naturaleza lo establece como el proyecto en que el asistente trabajará. En el caso de que no exista ningún proyecto de naturaleza RubyTL se creará uno nuevo con el nombre por defecto de *'RequiredDetErrorsProject'*. En la siguiente porción de código se muestra lo comentado:

```
IProject[] projects = ResourcesPlugin.getWorkspace().getRoot().getProjects();
for (int i = 0; i < projects.length; i++) {
  IProject proj = projects[i];

  try {
    IProjectNature projNature = proj.getNature(RubytlCore.NATURE_ID);
    if (projNature != null) {
       exists = true;
       projectName = proj.getName();
       break;
    }
  } catch (CoreException e) {
```

```
        e.printStackTrace();
    }
}

if (!exists) {
  projectName = Activator.PROJECT;
  IProject project =
              ResourcesPlugin.getWorkspace().getRoot().getProject(projectName);
  if (!(project.exists())) {
    IProgressMonitor monitor = null;
    try {
      project.create(monitor);
      project.open(monitor);

      IPath path = new Path("metamodels");
      IFolder customFolder = project.getFolder(path);
      customFolder.create(true, true, null);

      path = new Path("models");
      customFolder = project.getFolder(path);
      customFolder.create(true, true, null);

      path = new Path("tasks");
      customFolder = project.getFolder(path);
      customFolder.create(true, true, null);

      path = new Path("transformations");
      customFolder = project.getFolder(path);
      customFolder.create(true, true, null);

      path = new Path("helpers");
      customFolder = project.getFolder(path);
      customFolder.create(true, true, null);

      path = new Path("templates");
      customFolder = project.getFolder(path);
      customFolder.create(true, true, null);

      path = new Path("generationSQL");
      customFolder = project.getFolder(path);
      customFolder.create(true, true, null);

      setNature(project);
    } catch (CoreException e) {
      e.printStackTrace();
    }
  }
}
else {
  IProject project =
              ResourcesPlugin.getWorkspace().getRoot().getProject(projectName);
  IResource resource = project.findMember("templates");
  if (resource == null) {
    IPath path = new Path("templates");
    IFolder customFolder = project.getFolder(path);
    try {
      customFolder.create(true, true, null);
    } catch (CoreException e) {
      e.printStackTrace();
    }
  }

  resource = project.findMember("generationSQL");
  if (resource == null) {
    IPath path = new Path("generationSQL");
    IFolder customFolder = project.getFolder(path);
    try {
```

```
            customFolder.create(true, true, null);
        } catch (CoreException e) {
            e.printStackTrace();
        }
    }
}
```

Lo primero es obtener todos los proyectos que existan en el *workspace*, esto se consigue mediante la línea:

```
IProject[] projects = ResourcesPlugin.getWorkspace().getRoot().getProjects();
```

Una vez que se tiene un array con todos los proyectos, éste es recorrido para comprobar si existe alguno que presente la naturaleza de RubyTL. Mediante el método '*getNature*' de la clase '*Project*' de Eclipse se obtiene la naturaleza por la que se pregunta (parámetro *'RubytlCore.NATURE_ID'*) o nulo si no contiene dicha naturaleza.

En caso de que exista un proyecto con la naturaleza de RubyTL, como no se sabe que proyecto es ni cuando se creó, se deben crear una serie de directorios. Dichos directorios no se crean por defecto cuando se crea un nuevo proyecto RubyTL. Es por eso que los debe crear el asistente del plugin. Primero comprueba si existen y en caso de que existan los crea. Dichos directorios son '*templates*' y '*generationSQL'.*

Si resulta que no existe ningún proyecto de naturaleza RubyTL el asistente crea uno nuevo. Dicha creación es mediante la línea:

```
projectName = Activator.PROJECT;
IProject project =
            ResourcesPlugin.getWorkspace().getRoot().getProject(projectName);

project.create(monitor);
project.open(monitor);
```

Como se puede ver se crea un proyecto mediante el método '*create*' y posteriormente se abre ya que se tiene que establecer la jerarquía de los directorios del proyecto. La creación de un nuevo directorio dentro de un proyecto se hace de la siguiente manera:

```
IPath path = new Path("metamodels");
IFolder customFolder = project.getFolder(path);
customFolder.create(true, true, null);
```

Primero hay que obtener el path del directorio que se pretende crear. Una vez que se tiene el path del directorio, mediante el proyecto que se acaba de crear obtenemos la carpeta a crear. El método '*getFolder*' establece el path pasado como parámetro relativo al proyecto. Una vez se tiene la carpeta solamente hay que crearla mediante el método '*create*'.
Las carpetas que se crean son: '*metamodels*', '*models*', '*tasks*', '*transformations*', '*helpers*', '*templates*' y '*generationSQL*',

Una vez que se ha creado el proyecto o seleccionado uno existente por medio del asistente, hay que inicializarlo. La inicialización del proyecto es la copia de todos los ficheros necesarios para realizar las etapas 2, 3, 4 y 5 del proceso de modernización. Es decir, son los metamodelos DML, DDL, DML_DDL y Errores, los ficheros de transformación con las reglas de RubyTL, las plantillas y el fichero '*2code'* para obtener el código SQL y los diferentes decoradores. Todos estos ficheros se guardan en el plugin. Por lo que simplemente habrá que copiarlos al nuevo proyecto.

Otro aspecto interesante del asistente es que lanza las transformaciones desde código. Para ello se ha tenido que importar al plugin todas las clases y paquetes incluidos en el paquete *'gts.rubytl.launching'*.

Todas las clases que se ha importado en el plugin *'es.modelum.detErrores'* existen en un fichero '.jar', por lo que se podría haber incluido dicho fichero en vez de todos los paquetes. Pero se producía un error cuando se intentaba usar el fichero *'.jar'* incluido. Se le comentó dicho error a Jesús Sánchez Cuadrado (creador de RubyTL) y la solución a la que se llegó fue la de importar todos los paquetes.

Para crear un fichero '*rakefile'* mediante código Java hay que seguir los siguientes pasos:

1. Creación de la transformación.
2. Ejecución de la transformación creada en el paso 1.

A lo largo de toda la modernización se llevan a cabo dos tipos de transformaciones. Las transformaciones modelo a modelo y las transformaciones modelo a código.

Para obtener una transformación modelo a modelo hay que crearse un objeto de la clase '*TaskM2MConfigurationData'*, al cual en su constructor hay que indicarle cuatro parámetros: el comando para ejecutar Ruby, el path donde está instalado RubyTL, el path del proyecto (que tomará como base para realizar la transformación) y el path donde estará el fichero que contendrá las reglas a ejecutar para realizar la transformación (el fichero de las reglas de RubyTL). Una vez que se ha creado el fichero que contendrá la modernización hay que añadirle los bindings source y target (podrá haber más de un source). Un objeto '*Binding'* establece la correspondencia entre el modelo de entrada con el metamodelo al cuál conforma, en caso de ser un binding source, o la correspondencia entre el modelo que se pretende crear con el metamodelo que debe conformar, en caso de ser un target. Además en el binding se indica el prefijo mediante el cuál se referenciará al metamodelo en la transformación. Un ejemplo de todo esto se puede ver a continuación:

```
sourceDML_DDL = new Binding();
sourceDML_DDL.addModel(data.getDML_DDLModel().toOSString());
sourceDML_DDL.addMetamodel("DML_DDL",data.getDML_DDLMetamodel().toOSString());

targetError = new Binding();
targetError.addModel(data.getErrorModel().toOSString());
targetError.addMetamodel("Errores", data.getErrorMetamodel().toOSString());

TaskM2MConfigurationData m2mConf=new TaskM2MConfigurationData(data.getRuby(),
       data.getRubyTL_Path().toOSString(), data.getProjectPath().toOSString(),
```

```
        data.getTransformationError().toOSString());
m2mConf.addSourceBinding(sourceDML_DDL);
m2mConf.addTargetBinding(targetError);
```

Como se puede observar se crean dos '*Bindings*', un source y un target, a los cuales se le añade un modelo y un metamodelo. A continuación nos creamos la transformación pasándole al constructor toda la información mencionada anteriormente. Por último se añaden ambos bindings a la transformación.

Una vez que se ha creado la transformación se debe crear el fichero del '*rakefile*'. Para eso se llama al método '*write*' de la clase '*TaskM2MWriter*' para obtener en un string todo el contenido que debería tener el '*rakefile*'. A dicho método se le pasan como parámetros la configuración obtenida anteriormente y el nombre que deberá tener la transformación. A continuación se crea un fichero que tendrá como contenido el string obtenido mediante el método '*write*'. Todo esto se puede ver en el siguiente código:

```
public void write() {
  TaskM2MWriter writer = new TaskM2MWriter();
  String result = writer.writeConfiguration(m2mConf, "models2error");

  IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
  IResource resource = root.findMember(new Path(projectName));
  IContainer container = (IContainer) resource;
  final IFile file = container.getFile(rakeFilePath);
  try {
       InputStream stream = new ByteArrayInputStream(result.getBytes());
       if (file.exists()) {
         file.setContents(stream, true, true, null);
       } else {
         file.create(stream, true, null);
       }
       stream.close();
  } catch (IOException e) {
  } catch (CoreException e) {
       e.printStackTrace();
  }
}
```

Con esto se consigue obtener un fichero 'rakefile' mediante el cual poder lanzar la transformación modelo a modelo.

El '*rakefile*' capaz de lanzar una transformación modelo a código se obtiene de manera similar, salvo algunas diferencias. Para obtener una transformación modelo a código hay que crearse un objeto de la clase '*TaskM2TConfigurationData*', al cual en su constructor hay que indicarle cinco parámetros: el comando para ejecutar Ruby, el path donde está instalado RubyTL, el path del proyecto (que tomará como base para realizar la transformación), el path donde estará el fichero '*2code*' y el nombre de la carpeta donde se deberán guardar los ficheros generados por la transformación. En este caso, la transformación únicamente tendrá un binding, que será un source correspondiente al modelo DML_DDL_Correct. Un ejemplo de todo esto se puede ver a continuación:

```
sourceDML_DDL = new Binding();
```

```
sourceDML_DDL.addModel(data.getCorrect().toOSString());
sourceDML_DDL.addMetamodel("DML_DDL",data.getDML_DDLMetamodel().toOSString());

TaskM2TConfigurationData  m2tConf=new TaskM2TConfigurationData(data.getRuby(),
       data.getRubyTL_Path().toOSString(), data.getProjectPath().toOSString(),
     data.getModelToSql().toOSString(), data.getGenerationSQL().toOSString());
m2tConf.addSourceBinding(sourceDML_DDL);
```

El fichero '*rakefile*' se obtiene de la misma manera que en la transformación modelo a modelo, es decir, llamando al método '*write*' pero el de la clase '*TaskM2TWriter*'.

Una vez creado el '*rakefile',* lo único que quedaría sería ejecutar la transformación. Para poder ejecutar un '*rakefile*' obtenido mediante código hay que crearse un objeto de la clase '*RakefileConfigurationData*'. Para crear dicho objeto hay que pasarle en el constructor la siguiente información: el comando para ejecutar Ruby, el path donde está instalado RubyTL, el path del proyecto y el path donde estará el '*rakefile*' que nos hemos creado anteriormente. El objeto '*RakefileConfigurationData*' se le pasa como parámetro al constructor de la clase '*RakefileLauncher*' para obtener el objeto capaz de lanzar una transformación. Éste último objeto presenta un método para ejecutar una transformación llamado '*execute*'. Al método '*execute*' solamente hay que pasarle el nombre que se le asignó a la transformación en el '*rakefile*' que se cargó en el '*RakefileConfigurationData*'. Todo esto se muestra a continuación:

```
RakefileConfigurationData configData =
      new RakefileConfigurationData(data.getRuby(),
      data.getRubyTL_Path().toOSString(),data.getProjectPath().toOSString(),
      rakeFile);
RakefileLauncher launcher = new RakefileLauncher(configData);
launcher.execute(transform);
```

Este plugin desea ofrecerle al usuario la opción de no corregir un error en concreto, por lo que se mostrarán en una de las páginas que contiene el asistente todos los errores que se han obtenido al realizar la detección de errores. Para ello es necesario recorrer el modelo de Errores obtenido en la fase de detección. Para recorrer el modelo de Errores se usa EMF.

Para poder hacer uso de EMF el primer paso es crearse un modelo EMF del metamodelo Errores. Como el metamodelo Errores referencia al metamodelo DML_DDL, es necesario incluir también en la obtención del modelo EMF. Una vez que se obtiene el modelo EMF se procederá a generar el código para dicho modelo. La estructura de paquetes que resultaría de dicha generación de código se puede ver a continuación:

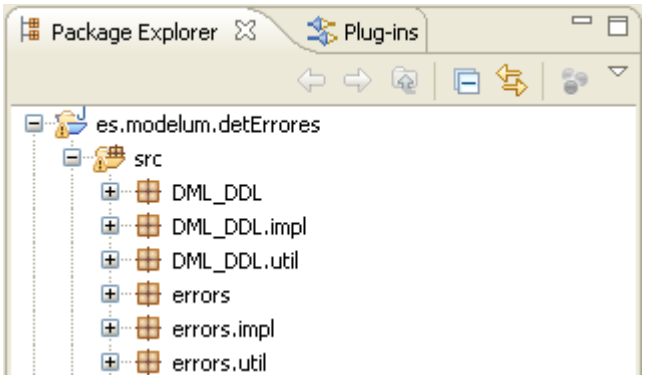

Figura 20: Jerarquía obtenida mediante EMF.

Como se puede ver se obtienen los paquetes para el metamodelo Errores y el DML_DDL con las interfaces, otro con las implementaciones de dichas interfaces y uno extra llamado '*util*'.

Una vez que se obtienen todas las clases del modelo EMF hay que acceder al modelo Errores que contiene todos los errores detectados del sistema de información. La forma de acceder al modelo es la siguiente:

```
public DetErroresEMF(String source) {
  DML_DDLPackage dml_ddlPackage = DML_DDLPackage.eINSTANCE;
  ErrorsPackage errorsPackage = ErrorsPackage.eINSTANCE;

  resourceSet = new ResourceSetImpl();

  resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put
                                  (Resource.Factory.Registry.DEFAULT_EXTENSION,
                                                  new XMIResourceFactoryImpl());

  resourceSet.getPackageRegistry().put
  (ErrorsPackage.eNS_URI, errorsPackage);

  resourceSet.getPackageRegistry().put
  (DML_DDLPackage.eNS_URI, dml_ddlPackage);

  fileURI = URI.createFileURI(source);
  resource = resourceSet.getResource(fileURI, true);
}
```

Como se puede observar, lo primero que hay que realizar es la inicialización del paquete que manejara los recursos de los modelos Errores y DML_DDL. Una vez que se han inicializado se crea un conjunto de recursos, el cuál contendrá todos los recursos que se carguen o se creen, (en nuestro caso contendrá el modelo con los errores). Una vez que se ha creado el manejador de los recursos, se le tiene que asignar un tipo de factoría. En nuestro caso la factoría tiene que ser del tipo XMI, y las extensiones de los ficheros que podrá manejar la factoría serán las de por defecto (en realidad dicha extensión es '*.*'). Una vez inicializado la factoría se añaden los paquetes de los metamodelos al manejador de los recursos. Lo único que resta es obtener el recurso del modelo de errores, para ello le pedimos al manejador el resource mediante

'*getResource*' indicándole la URI del fichero del modelo de errores, y un booleano puesto a "true" que indicará si se desea cargar el resource.

Una vez que se tiene el modelo en un recurso, solamente restaría recorrer el árbol obtenido. Para ello y como el modelo de errores tiene un elemento raíz, la metaclase 'Errores', se obtiene dicho elemento raíz de la siguiente manera:

```
Errores errores = (Errores)resource.getContents().get(0);
```

De todo el contenido del recurso obtenido al cargar el modelo de errores nos quedamos con el primero elemento, que sabemos que será el elemento de la metaclase Errores. Una vez realizado esto ya se puede ir navegando por todo el modelo de errores de una manera muy fácil e intuitiva. Por ejemplo para obtener todos los errores que tiene el modelo dentro del elemento 'Errores' se haría de la siguiente manera:

```
errores.getErrores();
```

Mediante esto obtenemos todos los errores del modelo. Para saber de que error se trata lo único que se tendría que hacer es un '*isInstance*' de Java con cada uno de las tres metaclases.

Para cada uno de los errores del modelo, EMF genera métodos '*Get*' y '*Set*' para cada uno de los atributos y referencias. Por lo que la navegación dentro del error es bastante sencilla. Como se puede ver a continuación:

```
Ck ck = error.getCk();
Iterator<ValuesCk> valuesIt = ck.getValuesCk().iterator();
while (valuesIt.hasNext()) {
  ValuesCk value = valuesIt.next();
  String column = value.getColumnName();
  String comparator = value.getComparator();
  String conjuntion = value.getLogConjuntion();
  EList<String> listValues = value.getValue();

  Iterator<String> valueIt = listValues.iterator();
  while(valueIt.hasNext()) {
  }
}
```

En el código anterior para un elemento '*CheckError*' realizamos lo siguiente:

- Obtenemos la '*Ck*' que referencia del modelo DML_DDL mediante '*error.getCk()*'.
- Se crea un iterador para recorrer cada una de las restricciones que existen en la check constraint obtenida en el paso anterior.
- Obtenemos el elemento '*Value*' y toda la información que contiene dicho elemento del modelo DML_DDL.
- Para cada '*Value*' recorremos la lista de posibles valores de su restricción. Dependiendo del tipo será un único valor o muchos.

Como se puede ver es bastante sencillo y directo el acceso a la información del modelo mediante EMF.

9 Conclusiones y vías Futuras

Se ha obtenido un diseño de una solución basada en modelos para abordar un problema de modernización del modelo de datos al que se enfrentan las empresas y del cual, actualmente, no se dispone de tantas soluciones implementadas como en el caso de la modernización de los aplicativos.

Se ha logrado definir una gramática y un metamodelo para las sentencias DDL de creación de las estructuras básicas de un esquema de datos. Y también se ha logrado definir una gramática y un metamodelo para las sentencias DML de inserción de datos en un esquema de datos.

Se ha hecho uso del lenguaje Gra2Mol para extraer los modelos asociados a las gramáticas DDL y DML de un esquema relacional de datos.

Se ha creado un algoritmo para encontrar e inferir deficiencias en una base de datos relacional, implementado con un lenguaje de transformación M2M. Se han logrado generar mediante una transformación M2C (y una vez que se ha corregido el esquema de datos) las sentencias DDL y DML del mismo esquema pero corregidas.

El último logro ha sido el de integrar en un plugin para Eclipse una herramienta que asista al proceso de modernización, que realice sin supervisión del usuario la inferencia y la detección de errores. Dándole la posibilidad al usuario de seleccionar aquellos errores que desea corregir y una vez corregidos todos los errores, obtener de nuevo las sentencias para la creación del esquema de datos del que se partió pero corregido. Todo esto en dos pasos para el usuario.

Hoy en día debido a la gran proliferación de herramientas, sistemas, lenguajes, etc dentro del mundo de la informática y debido a la constante evolución de la informática, pretender modernizar un software con dependencia a la herramienta mediante la que se creó, del sistema donde se usa o del lenguaje en el que está escrito es poco práctico y aun menos portable . Por lo que si se actualiza, evoluciona o moderniza un software la mejor opción es realizarlo mediante un proceso Dirigido por Modelos, ya que se realiza de forma independiente a la herramienta, sistema, lenguaje, etc.

El proceso de modernización de los datos de un sistema de información aplicando un proceso DSDM como el que se ha comentado a lo largo de este documento ha resultado a veces fácil e intuitivo y otras tedioso y complejo. El empleo de Gra2Mol ha sido sencillo, la pena es que debido a que se trataban de gramáticas sencillas no se ha podido comprobar todo el potencial que puede tener. RubyTL ha sido bastante sencillo de aplicar. La estructura de reglas se parece bastante a la empleada por Gra2Mol, o viceversa, por lo que su manejo resultó bastante rápido. Para poder realizar el algoritmo de inferencia y corrección de errores se tuvo que aprender el lenguaje Ruby. Ruby es un lenguaje orientado a objetos potente y de fácil uso, compresión y sintaxis.

Quizás la peor parte fue la de EMF. Mediante EMF se recorre el modelo de datos una vez que se tiene cargado en memoria. Dicha navegación es bastante sencilla ya que es a través de las

referencias que el propio modelo establece. Comprender el uso y la estructura de EMF ha sido la peor parte de todo el proyecto.

Como vías futuras se pueden destacar el diseñar el algoritmo de inferencia y detección de errores de una base de datos para permitir extensiones mediante la adición de nuevos errores así como considerar la necesidad de ampliar el metamodelo de errores para que pueda soportar los nuevos errores.

Otro trabajo futuro es el de ampliar el número de errores o mejoras que se pueden inferir en una base de datos a partir de sus datos almacenados y la inclusión de un mayor número de criterios en la inferencia de errores (diversidad de datos, coincidencia sintáctica de los nombres de las columnas, posibles relaciones entre más de dos tablas, por ejemplo si una tabla B depende de una tabla A, y otra tabla C tiene una dependencia con B, es probable que además deba estar relacionada con la tabla A,…) mediante la información en [DataBase].

Otra posible mejora es normalizar la BBDD. para determinar si un esquema relacional está acorde a las formas normales diseñadas por Codd. Para ello necesitamos información adicional sobre el mundo real que modelamos con la base de datos y definir las dependencias funcionales entre los campos de las tablas que la componen. Esta tarea aunque en la actualidad está bien definida, no es trivial de comprender además de tediosa de realizar. Además, la identificación de dependencias funcionales puede requerir de la asistencia por parte del diseñador. Las dependencias funcionales se establecen normalmente conociendo el significado y dominio que afecta a cada campo de una tabla, pero si lo que queremos es hacer un proceso automático habremos de utilizar algún tipo de inferencia como las utilizadas en [ELI+08]

Actualmente la carga de los datos de una BB.DD en un modelo no se encuentra optimizada. EMF no soporta la carga de modelos con enormes volúmenes de datos (miles de registros de una BB.DD). Seria conveniente diseñar una infraestructura de soporte para posibilitar el manejo de grandes volúmenes de datos que requieren ser modelizados. Dicha infraestructura debe ofrecer al menos algún DSL de consulta, además de la arquitectura que permita particionar y almacenar de manera eficiente y transparente los mega-modelos de datos.

Por último comentar que sería necesario revisar el algoritmo para la inferencia de mejoras o deficiencias para comprobar si realmente se requiere el uso de los datos de la BB.DD. Anteriormente se ha comentado la alta penalización que sufre el proceso al manejar grandes modelos de datos. Cabría incluso plantearse la viabilidad en la aplicación del proceso cuando los datos a extraer de la BB.DD son enormes. Una alternativa de estudio sería la localización de posibles mejoras o errores analizando la información sobre las relaciones entre los datos que podemos hallar en el código del aplicativo que conforma el sistema de información. Cabe entender que dicha alternativa iría en detrimento de la portabilidad del proyecto (análisis de código y tecnologías específicas para el acceso a los datos). Sin embargo, probablemente esta alternativa sería mas eficiente y viable.

10 Bibliografía

[ADM] **- ADM. Architecture driven modernization. http://adm.omg.org**

[MDA] **- MDA. Model driven architecture. http://www.omg.org/mda**

[Clark, 2004] **- T. Clark, A. Evans, P. Sammut, and J. Willams, Applied metamodelling: A foundation for language driven development, Xactium, 2004.**

[Jouault, 2005] **- Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In MoDELS Satellite Events, pages 128-138, 2005.**

[Sánchez, 2006] **- J. S. Cuadrado, J. G. Molina, and M. Menarguez, RubyTL: A Practical, Extensible Transformation Language, in 2nd European Conference on Model Driven Architecture, vol. 4066. Lecture Notes in Computer Science, June 2006, pp. 158-172.**

[QVT] **- OMG. Revised submission for MOF 2.0 Query/View/Transformation, 2005. http://www.omg.org/cgi-bin/apps/doc?ad/2005-03-02**

[MOF2Text] **- Model to Text Transformation Language. http://www.omg.org/spec/MOFM2T/1.0**

[MOFScript] **- MOFScript. http://www.eclipse.org/gmt/mofscript**

[Comella-Dorda, 2000] **- S. Comella-Dorda K. Wallnau, R. C. Seacord, J. Robert. "A Survey of Legacy System Modernization Approaches". Software Engineering Institute. Carnegie Mellon.**

[ADM06] **- ADM Task Force. Architecture-driven modernization scenarios. http://adm.omg.org/ADMTF_Scenario_White_Paper(pdf).pdf, 2006.**

[Eclipse] **- Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf.**

[Gra2mol] **- Javier Cánovas y J. García-Molina," A Domain Specific Language for ExtractingModels in Software Modernization" 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings, in LNCS 5562, pag. 82-97**

- Página web. http://modelum.es/trac/gra2mol

- "Gra2Mol Lenguage Manual". Autor: **Javier Luis Cánovas Izquierdo.**
http://modelum.es/trac/gra2mol/browser/trunk/Grammar2Model.

doc/manual/gra2molManual.pdf

[KDM] **- Object Management Group, Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), v 1.1, 2009. (accessed 10.09.09).**

[MAZON+07] **- J.N. Mazón, J. Trujillo, A model driven modernization approach for automatically deriving multidimensional models in data warehouses, in: Proceedings of the International Conference on Conceptual Modeling, Auckland, New Zealand, 2007, pages. 56_71.**

[POLO+07] **- Macario Polo, Ignacio García Rodriguez de Guzmán, Mario Piattini: An MDA-based approach for database re-engineering, Journal of Software Maintenance (SMR), 2007**

[CES+09] **- César Domínguez Pérez, Arturo Jaime Elizondo, Tomás A. Pérez Fernández. En XV JENUI, Barcelona 8-10 de julio de 2009.**

[ELI+08] **- Elizabeth Luna Luz. Recuperación de información en Bases de Datos de tipo Bioinformático, 2008.**

[HCMMER08] **- Reiko Heckel, Rui Correia, Carlos Matos, and Luis Andrade Mohammad El-Ramly, Georgios Koutsoukos. Architectural transformations: From legacy to three-tier and services. In Tom Mens and Serge Demeyer, editors, Software Evolution, pages 139_170. Springer, 2008.**

[AGA+06] **- Luis Filipe Andrade, João Gouveia, Miguel Antunes, Mohammad El-Ramly, and Georgios Koutsoukos. Forms2net - migrating oracle forms to Microsoft .net. In GTTSE, pages 261_277, 2006.**

[Bernabe09] - Tesis de Máster: **"Modernización de Bases de Datos Relacionales Basada en Modelos"**. Autor: **Bernabé Nicolás García**. Directores**: Francisco J. Bermúdez Ruiz y Jesús J. García Molina**. 16 de Septiembre de 2009. Universidad de Murcia. Facultad de Informática. Postgrado IMACI.

[OSC+08] **- Óscar Sánchez Ramón. Caso de studio de un proyecto de modernización basada en modelos, 2008**

[Connolly+04] **- Connolly, T.M. and Begg, C.E. Sistemas de bases de datos. Un enfoque práctico para diseño, implementación y gestión. Ed. Person, 2004**

[DataBase] **- Refactoring Databases.** Editorial**: Addison Wesley.** Autores: **Scott W. Ambler, Pramod J.Sodolage.**

[ANTLR] - Página web**. http://www.antlr.org/**

[RubyTL] - "**RubyTL : A practical, extensible transformation language"**. Autores: **Jesús Sánchez Cuadrado, Jesús García Molina, y Marcos Menarguez Tortosa**.

- "**AGE Tutorial Version 0.3.1"**. Autor: **Jesús Sánchez Cuadrado**.

- "**RubyTL a ATC: un caso real de transformación de transformaciones"**. Autores: **Jesús Sánchez Cuadrado y Jesús García Molina de la Universidad de Murcia, Murcia y Eduardo Victor Sánchez Rebull y Antonio Estévez García de Open Canarias S.L, Santa Cruz de Tenerife**.

- "Programming Ruby. The Pragmatic Programmer's Guide". Second Edition. Dave Thomas

[EMF] **- "EMF Eclipse Modeling Framework". Second Edition. Steinberg, Budinsky, Paternostro, Merks.** Editorial: **Addison-Wesley.**

- Proyecto Fin de Carrera: **"Estudio de la plataforma Eclipse".** Autor: **Aurelio Sánchez Serrano.** Febrero 2005. Universidad de Murcia. Facultad de informática.

[PDE] - Proyecto Fin de Carrera: **"Estudio de la plataforma Eclipse".** Autor: **Aurelio Sánchez Serrano.** Febrero 2005. Universidad de Murcia. Facultad de informática.

- **Eclipse Platform Plug-in Developer Guide (PDE).**

11 Anexos

11.1 Gramática DDL

```
grammar DDL;

options {
        backtrack=true;
}

data_definition
        :       (data_definition_statement)*
        ;

data_definition_statement
        :       database
        |       table
        |       comment_table
        |       comment_column
        |       disabled_constraint
        ;

table
        :       'CREATE' (LOG_CONJ 'REPLACE')? 'TABLE' ID '(' (column)* (',')? (ck (',')?)*
(pk)? (',')? (fk (',')?)* ')' ('LOGGING')? ('NOCOMPRESS')? ('NOCACHE')? ('NOPARALLEL')?
('MONITORING')?';'
        ;

comment_table
        :       'COMMENT ON TABLE' ID 'IS' CVALUE ';'
        ;

column
        :       ID type ('(' NUMBER ('BYTE')? ')')? ('CONSTRAINT')? (name_constraint)?
('NULL' | 'NOT NULL')? (defecto)? (',')?
        ;

defecto
        :       'DEFAULT' ('NULL' | 'NOT NULL' | NUMBER | CVALUE)
        ;

comment_column
        :       'COMMENT ON COLUMN'  table_references '\.' ID 'IS'  CVALUE ';'
        ;

disabled_constraint
```

```
        : 'ALTER TABLE' table_references 'DROP CONSTRAINT' name_constraint ';'
        ;

ck
        :       'CONSTRAINT' ID 'CHECK' '(' (log_expresion)+ ')'
        ;

fk
        :       'CONSTRAINT' ID 'FOREIGN KEY' '(' (column_list)* ')' 'REFERENCES'
(schema_references)? ('\.')? table_references '(' (references_column_list)* ')' ('ON')? ('DELETE' |
'UPDATE')? ('CASCADE')? ('DISABLE')?
        ;

pk
        :       'CONSTRAINT' ID 'PRIMARY KEY' '(' (column_list)* ')'
        ;

column_ck
        :       ID
        ;

schema_references
        :       ID
        ;

table_references
        :       ID
        ;

name_constraint
        :       ID
        ;

references_column_list
        :       ID (',')?
        ;

value_list
        :       CVALUE (',')?
        ;

column_list
        :       ID (',')?
        ;

database
        :       'CREATE' ('DATABASE' | 'SCHEMA') ('IF NOT EXISTS')? ID (specification)? ';'
```

```
        ;

specification
        :       ('DEFAULT')? 'CHARACTER' 'SET' (COMPARATOR)? ID
        |       ('DEFAULT')? 'COLLATE' (COMPARATOR)? ID
        ;

log_expresion
        : column_ck (COMPARATOR (NUMBER | CVALUE) LOG_CONJ? | 'IN' '('
(value_list)* ')')
        ;

type
        :       (EXACTO | APROXIMADO | CHARACTERS | BITS | TIMES | INTERVALS
| BINARIES)
        ;

EXACTO
        :       ('NUMBER' | 'NUMERIC' | 'INTEGER' | 'SMALL INTEGER' | 'DECIMAL' |
'INT' | 'SMALL INT')
        ;

APROXIMADO
        : ('DOUBLE PRECISION' | 'LONG' | 'LONG RAW' | 'FLOAT' | 'REAL')
        ;

CHARACTERS
        : ('CHAR' | 'VARCHAR' | 'VARCHAR2' | 'NVARCHAR2' | 'NCHAR' | 'CHAR
VARYING' | 'CHARACTER' | 'CHARACTER VARYING' | 'NATIONAL CHAR' |
'NATIONAL CHAR VARYING' | 'NATIONAL CHARACTER' | 'NATIONAL CHARACTER
VARYING' | 'NCHAR VARYING' | 'CLOB' | 'NCLOB')
        ;

BITS
        : ('BIT' | 'BIT VARYING')
        ;

TIMES
        : ('DATE' | 'TIME' | 'TIMESTAMP')
        ;

INTERVALS
        : ('YEAR-MONTH' | 'DAY-TIME')
        ;

BINARIES
        : ('BFILE' | 'BLOB' | 'BINARY_DOUBLE' | 'BINARY_FLOAT')
```

```
        ;

SQUOTE                                          : '\u0027';
CVALUE                                          : SQUOTE ( options {greedy=false;} : . )*
SQUOTE;
NUMBER                                          : ('0'..'9' | ',')*;
COMPARATOR                                      : '<>' | '>=' | '<=' | '=' | '>' | '<';
LOG_CONJ                                : 'OR' | 'AND';
ID                                              : ('a'..'z' | 'A'..'Z') ( 'a' .. 'z' | 'A' .. 'Z' | '0' .. '9' |
'_' | '$' | '#' | '-' )*;
WS                                              : (' ' | '\t' | '\n' | '\r')+ {$channel=HIDDEN;};
ML_COMMENT:
        '/*' (options {greedy=false;} : .)* '*/' (';')? { $channel=HIDDEN;}
        ;

OL_COMMENT: '--' ( options {greedy=false;} : . )* '\n'+ { $channel=HIDDEN;}
        ;
```

11.2 Reglas transformación DDL

```
-------------------------------------------------------------------------------
-- Gra2MoL transformation definition for extracting DDL models from DDL Scripts
--
-- TODO:
--  * Fill in it!! :D
-------------------------------------------------------------------------------

rule 'mapStatements'
        from data_definition df
        to DDLDefinition
        queries
                stats : //#data_definition_statement;
                types : ///#type[0];
        mappings
                dataType = types;
                statements = stats;
end_rule

rule 'mapDataTypes'
        from type t
        to DataType
        queries
                tps : //#type;
        mappings
```

```
                types = tps;
end_rule

rule 'mapDatabase'
        from data_definition_statement/database cd
        to Database
        queries
                cd : /cd/#database;
        mappings
                databaseName = cd.ID;
end_rule

rule 'mapCommentTable'
        from data_definition_statement/comment_table fcct
        to CommentTable
        queries
                cct : /fcct/#comment_table;
                com : /fcct/comment_table/#comment;
        mappings
                tableName = cct.ID;
                tableComment = cct.CVALUE;
end_rule

rule 'mapCommentColumn'
        from data_definition_statement/comment_column fccc
        to CommentColumn
        queries
                ccc : /fccc/#comment_column;
                tr  : /fccc/comment_column/#table_references;
                com : /fccc/comment_column/#comment;
        mappings
                tableName = tr.ID;
                columnName = ccc.ID;
                columnComment = ccc.CVALUE;
end_rule

rule 'mapTable'
        from data_definition_statement/table sta
        to Table
        queries
                ct : /sta/#table;
                cc : /sta//#column;
                pk : /sta//#pk;
                fk : /sta//#fk;
                ck : /sta//#ck;
        mappings
                tableName = ct.ID;
```

```
                columns   = cc;
                columnsPk = pk;
                columnsFk = fk;
                checks    = ck;
end_rule

rule 'mapColumn'
        from column sta
        to Column
        queries
                type : /sta/#type;
                cc : /#sta;
                nn   :   /#sta{(TOKEN[0].exists   &&   TOKEN[0].eq("NOT   NULL"))   ||
(TOKEN[2].exists && TOKEN[2].eq("NOT NULL"))};
                n  : /#sta{(TOKEN[0].exists && TOKEN[0].eq("NULL")) || (TOKEN[2].exists
&& TOKEN[2].eq("NULL"))};
        mappings
                columnName = cc.ID;
                columnType = type;
                if (nn.hasResults) then
                        columnNull = "false";
                end_if
                if (n.hasResults) then
                        columnNull = "true";
                end_if
end_rule

rule 'mapPk'
        from pk fcpk
        to Pk
        queries
                cpk : /fcpk/#column_list;
        mappings
                namePk = fcpk.ID;
                columnName = cpk.ID;
end_rule

rule 'mapFk'
        from fk fcfk
        to Fk
        queries
                cfk  : /fcfk/#column_list;
                crfk : /fcfk/#references_column_list;
                ctr  : /fcfk/#table_references;
                table : //#data_definition_statement/table{ID.eq(ctr.ID)};
                tab : //data_definition_statement/#table/fk{ID.eq(fcfk.ID)};
                aux : ///#disabled_constraint/table_references{ID.eq(tab.ID)};
```

```
                disabled : /#aux/name_constraint{ID.eq(fcfk.ID)};
        mappings
                nameFk = fcfk.ID;
                columnName = cfk.ID;
                columnReference = crfk.ID;
                references = table;
                if (disabled.hasResults) then
                        status = "DISABLED";
                else
                        status = "ENABLED";
                end_if
end_rule

rule 'mapCk'
        from ck fcck
        to Ck
        queries
                cck : /#fcck;
                leck : /fcck/#log_expresion;
                table : //data_definition_statement/#table/ck{ID.eq(cck.ID)};
                aux : ///#disabled_constraint/table_references{ID.eq(table.ID)};
                disabled : /#aux/name_constraint{ID.eq(cck.ID)};
        mappings
                nameCk = cck.ID;
                valuesCk = leck;

                if (disabled.hasResults) then
                        status = "DISABLED";
                else
                        status = "ENABLED";
                end_if
end_rule

rule 'mapValuesCk'
        from log_expresion le
        to ValuesCk
        queries
                column : /le/#column_ck;
                value_list : /le/#value_list;
                number : /#le{NUMBER.exists};
        mappings
                if (value_list.hasResults) then
                        comparator = "=";
                        value = removeQuotes value_list.CVALUE;
                else
                        comparator = le.COMPARATOR;
                        if (number.hasResults) then
```

```
                                value = le.NUMBER;
                        else
                                value = removeQuotes le.CVALUE;
                        end_if
                end_if

                logConjuntion = le.LOG_CONJ;
                columnName = column.ID;
end_rule

------------------------------------------------------------
---------------------- MAPPINGS TYPES ----------------------
------------------------------------------------------------


------------------------------------------------------------
------------------------- EXACTOS --------------------------
rule 'mapInteger'
        from type[unique]{EXACTO.eq("INTEGER")} t
        to Integer
        queries
        mappings
                name = t.EXACTO;
 end_rule

rule 'mapSmallInteger'
        from type[unique]{EXACTO.eq("SMALL INTEGER")} t
        to SmallInteger
        queries
        mappings
                name = t.EXACTO;
end_rule

rule 'mapInt'
        from type[unique]{EXACTO.eq("INT")} t
        to Int
        queries
        mappings
                name = t.EXACTO;
end_rule

rule 'mapSmallInt'
        from type[unique]{EXACTO.eq("SMALL INT")} t
        to SmallInt
        queries
        mappings
                name = t.EXACTO;
end_rule
```

```
rule 'mapNumeric'
	from type[unique]{EXACTO.eq("NUMERIC")} t
	to Numeric
	queries
	mappings
		name = t.EXACTO;
end_rule

rule 'mapNumber'
	from type[unique]{EXACTO.eq("NUMBER")} t
	to Number
	queries
	mappings
		name = t.EXACTO;
end_rule

rule 'mapDecimal'
	from type[unique]{EXACTO.eq("DECIMAL")} t
	to Decimal
	queries
	mappings
		name = t.EXACTO;
end_rule
------------------------------------------------------------
------------------------------------------------------------
----------------------- APROXIMADOS -----------------------
rule 'mapReal'
	from type[unique]{APROXIMADO.eq("REAL")} t
	to Real
	queries
	mappings
		name = t.APROXIMADO;
end_rule

rule 'mapDoublePrecision'
	from type[unique]{APROXIMADO.eq("DOUBLE PRECISION")} t
	to DoublePrecision
	queries
	mappings
		name = t.APROXIMADO;
end_rule

rule 'mapFloat'
	from type[unique]{APROXIMADO.eq("FLOAT")} t
	to Float
	queries
```

```
	mappings
		name = t.APROXIMADO;
end_rule

rule 'mapLong'
	from type[unique]{APROXIMADO.eq("LONG")} t
	to Long
	queries
	mappings
		name = t.APROXIMADO;
end_rule

rule 'mapLongRaw'
	from type[unique]{APROXIMADO.eq("LONG RAW")} t
	to LongRaw
	queries
	mappings
		name = t.APROXIMADO;
end_rule

------------------------------------------------------------
------------------------------------------------------------
---------------------- CHARACTERS ------------------------
rule 'mapCharacter'
	from type[unique]{CHARACTERS.eq("CHARACTER")} t
	to Character
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapCharacterVarying'
	from type[unique]{CHARACTERS.eq("CHARACTER VARYING")} t
	to CharacterVarying
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapChar'
	from type[unique]{CHARACTERS.eq("CHAR")} t
	to Char
	queries
	mappings
		name = t.CHARACTERS;
end_rule
```

```
rule 'mapVarChar'
	from type[unique]{CHARACTERS.eq("VARCHAR")} t
	to VarChar
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapVarChar2'
	from type[unique]{CHARACTERS.eq("VARCHAR2")} t
	to VarChar2
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapNVarChar2'
	from type[unique]{CHARACTERS.eq("NVARCHAR2")} t
	to NVarChar2
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapNChar'
	from type[unique]{CHARACTERS.eq("NCHAR")} t
	to NChar
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapCharVarying'
	from type[unique]{CHARACTERS.eq("CHAR VARYING")} t
	to CharVarying
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapNationalChar'
	from type[unique]{CHARACTERS.eq("NATIONAL CHAR")} t
	to NationalChar
	queries
	mappings
		name = t.CHARACTERS;
end_rule
```

```
rule 'mapNationalCharVarying'
	from type[unique]{CHARACTERS.eq("NATIONAL CHAR VARYING")} t
	to NationalCharVarying
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapNationalCharacter'
	from type[unique]{CHARACTERS.eq("NATIONAL CHARACATER")} t
	to NationalCharacter
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapNationalCharacterVarying'
	from type[unique]{CHARACTERS.eq("NATIONAL CHARACTER VARYING")} t
	to NationalCharacterVarying
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapNCharVarying'
	from type[unique]{CHARACTERS.eq("NCHAR VARYING")} t
	to NCharVarying
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapClob'
	from type[unique]{CHARACTERS.eq("CLOB")} t
	to Clob
	queries
	mappings
		name = t.CHARACTERS;
end_rule

rule 'mapNClob'
	from type[unique]{CHARACTERS.eq("NCLOB")} t
	to NClob
	queries
	mappings
		name = t.CHARACTERS;
```

```
end_rule
------------------------------------------------------------
------------------------------------------------------------
--------------------------- BITS ---------------------------
rule 'mapBit'
        from type[unique]{BITS.eq("BIT")} t
        to Bit
        queries
        mappings
                name = t.BITS;
end_rule

rule 'mapBitVarying'
        from type[unique]{BITS.eq("BIT VARYING")} t
        to BitVarying
        queries
        mappings
                name = t.BITS;
end_rule
------------------------------------------------------------
------------------------------------------------------------
--------------------------- TIMES ---------------------------
rule 'mapTime'
        from type[unique]{TIMES.eq("TIME")} t
        to Time
        queries
        mappings
                name = t.TIMES;
end_rule

rule 'mapDate'
        from type[unique]{TIMES.eq("DATE")} t
        to Date
        queries
        mappings
                name = t.TIMES;
end_rule

rule 'mapTimestamp'
        from type[unique]{TIMES.eq("TIMESTAMP")} t
        to Timestamp
        queries
        mappings
                name = t.TIMES;
end_rule
------------------------------------------------------------
------------------------------------------------------------
```

```
------------------------ INTERVALS ------------------------
rule 'mapDayTime'
        from type[unique]{INTERVALS.eq("DAY-TIME")} t
        to DayTime
        queries
        mappings
                name = t.INTERVALS;
end_rule

rule 'mapYearMonth'
        from type[unique]{INTERVALS.eq("YEAR-MONTH")} t
        to YearMonth
        queries
        mappings
                name = t.INTERVALS;
end_rule
-------------------------------------------------------------
-------------------------------------------------------------
------------------------ BINARIES -------------------------
rule 'mapBinaryDouble'
        from type[unique]{BINARIES.eq("BINARY_DOUBLE")} t
        to BinaryDouble
        queries
        mappings
                name = t.BINARIES;
end_rule

rule 'mapBinaryFloat'
        from type[unique]{BINARIES.eq("BINARY_FLOAT")} t
        to BinaryFloat
        queries
        mappings
                name = t.BINARIES;
end_rule

rule 'mapBFile'
        from type[unique]{BINARIES.eq("BFILE")} t
        to BFile
        queries
        mappings
                name = t.BINARIES;
end_rule

rule 'mapBlob'
        from type[unique]{BINARIES.eq("BLOB")} t
        to Blob
        queries
```

```
        mappings
                name = t.BINARIES;
end_rule
-------------------------------------------------------------
```

11.3 Gramática DML

```
grammar DML;

insertsStatements
        :       (insertInto)*
        ;

insertInto
        :       'INSERT INTO' ID ('(')? (column)* (')')? 'VALUES' registry ';'
        ;

registry
        :       '(' (value)+ ')'
        ;

value
        :       (ID | CVALUE | NUMBER) (',')?
        ;

column
        :       ID (',')?
        ;

NUMBER
        :       ('0'..'9' | '.')*
        ;

ID
        :       ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '0'..'9' | '_' | '$' | '#' | '-' | '@' | '.')*
        ;

SQUOTE
        :       '\u0027'
        ;

CVALUE
        :       SQUOTE ( options {greedy=false;} : . )* SQUOTE
        ;
```

```
WS
        :       ( ' ' | '\t' | '\n' | '\r' )+ {$channel=HIDDEN;}
        ;
```

11.4 Reglas transformación DML

```
-------------------------------------------------------------------------------
-- Gra2MoL transformation definition for extracting DML models from DML Scripts
--
-- TODO:
--  * Fill in it!! :D
-------------------------------------------------------------------------------

rule 'mapInsertsStatements'
        from insertsStatements is
        to InsertsStatements
        queries
                ins : //#insertInto;
        mappings
                insertsInto = ins;
end_rule

rule 'mapInsertInto'
        from insertInto insertInto
        to InsertInto
        queries
                col : /insertInto//#column;
                re : /insertInto/#registry;
        mappings
                tableName = insertInto.ID;
                columns = col;
                registry = re;
end_rule

rule 'mapColumns'
        from column c
        to Column
        queries
        mappings
                columnName = c.ID;
end_rule

rule 'mapRegistry'
```

```
        from registry re
        to Registry
        queries
                values : /re//#value;
        mappings
                registryValues = values;
end_rule

rule 'mapValue'
        from value v
        to Value
        queries
                id : /#v{ID.exists};
                cvalue : /#v{CVALUE.exists};
                number : /#v{NUMBER.exists};

                insert : //#insertInto//value{this.check(v)};
        mappings
                if (id.hasResults) then
                        value = v.ID;
                end_if
                if (cvalue.hasResults) then
                        value = removeQuotes v.CVALUE;
                end_if
                if (number.hasResults) then
                        value = v.NUMBER;
                end_if

                column = ext InferColumnFromValue(v,insert);
end_rule
```

11.5 Inferir DML

```
import gts.modernization.model.CST.*;
import gts.modernization.extension.*;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class InferColumnFromValue extends MappingExtensionPoint {

    public ExtensionValueReturn execute() {
        Element result = null;
```

```
        // Extraemos el primer parametro, que es el elemento "value"
        List<Element> paramList1 = (List<Element>) getParam(0);
        Object value = null;
        if(paramList1.size() > 0) value = paramList1.get(0);

        // Extraemos el segundo parametro, que es el elemento "insertInto"
        List<Element> paramList2 = (List<Element>) getParam(1);
        Object insertParam = null;
        if(paramList2.size() > 0) insertParam = paramList2.get(0);

        // Comprobamos que ambos elementos de parametro son nodos
        if (insertParam instanceof Node && value instanceof Node) {
                Node insertNode = (Node) insertParam;
                Node valueNode = (Node) value;

                // Extraemos el nodo "registry" del nodo "insertInto"
                Node registry = insertNode.getNode("registry", 0);

                if(registry != null) {
                        List<Node> children = registry.getNodes();
                        int position = 0;
                        // Recorremos sus nodos hijos, que son siempre del tipo "value"
                        for(Node child : children) {
                                // Comparamos el value con el que
                                // recibimos como parametro
                                if(child == valueNode) {
                                        // Si coincide, usamos "position" para
                                        // extraer el elemento "column"
                                        // de "insertNode" de dicha posicion
                                        result = insertNode.getNode("column", position);
                                } else {
                                        // Si no coincide, aumentamos el puntero "position"
                                        position++;
                                }
                        }
                }
        }
        // Devolvemos el nodo encontrado
        // (null si no encontramos nada)
        return returnNode(result);
    }
}
```

11.6 Trasnformaciones DML_DDL

```
transformation 'ModelstoModel'

decorator DML::Column do
  def getInsertInto
    DML::InsertInto.all_objects.find{|dml_ddl| dml_ddl.columns.include?(self)}
  end
end

top_rule 'DDLDefinition2DDLDefinition' do
  from DDL::DDLDefinition
  to DML_DDL::DDLDefinition
  mapping do |ddl, dml_ddl|
    dml_ddl.statements = ddl.statements
    dml_ddl.dataType = ddl.dataType
  end
end

rule 'DataType2DataType' do
  from DDL::DataType
  to DML_DDL::DataType
  mapping do |ddl, dml_ddl|
    dml_ddl.types = ddl.types
  end
end

rule 'Database2Database' do
  from DDL::Database
  to DML_DDL::Database
  mapping do |ddl, dml_ddl|
    dml_ddl.databaseName = ddl.databaseName
  end
end

rule 'Table2Table' do
  from DDL::Table
  to DML_DDL::Table
  mapping do |ddl, dml_ddl|
    dml_ddl.tableName = ddl.tableName
    dml_ddl.columns = ddl.columns
    dml_ddl.columnsPk = ddl.columnsPk
    dml_ddl.columnsFk = ddl.columnsFk
    dml_ddl.checks = ddl.checks
    dml_ddl.commentTable = ddl.commentTable

    dml_ddl.registries = DML::InsertInto.all_objects.select{|dml_ddl|
dml_ddl.tableName == ddl.tableName}.map{|dml_ddl| dml_ddl.registry}.flatten
  end
end

rule 'Pk2Pk' do
  from DDL::Pk
  to DML_DDL::Pk
  mapping do |ddl, dml_ddl|
    dml_ddl.namePk = ddl.namePk
    dml_ddl.columnName = ddl.columnName
  end
end

rule 'Fk2Fk' do
  from DDL::Fk
  to DML_DDL::Fk
  mapping do |ddl, dml_ddl|
```

```
    dml_ddl.nameFk = ddl.nameFk
    dml_ddl.columnName = ddl.columnName
    dml_ddl.references = ddl.references
    dml_ddl.columnReference = ddl.columnReference
    dml_ddl.status = ddl.status
  end
end

rule 'Ck2Ck' do
  from DDL::Ck
  to DML_DDL::Ck
  mapping do |ddl, dml_ddl|
    dml_ddl.nameCk = ddl.nameCk
    dml_ddl.valuesCk = ddl.valuesCk
    dml_ddl.status = ddl.status
  end
end

rule 'ValuesCk2ValuesCk' do
  from DDL::ValuesCk
  to DML_DDL::ValuesCk
  mapping do |ddl, dml_ddl|
    dml_ddl.value = ddl.value
    dml_ddl.comparator = ddl.comparator
    dml_ddl.columnName = ddl.columnName
    dml_ddl.logConjuntion = ddl.logConjuntion
  end
end

rule 'Column2Column' do
  from DDL::Column
  to DML_DDL::Column
  mapping do |ddl, dml_ddl|
    dml_ddl.columnName = ddl.columnName
    dml_ddl.commentColumn = ddl.commentColumn
    dml_ddl.columnNull = ddl.columnNull
    dml_ddl.columnType = ddl.columnType
  end
end

rule 'CommentTable2CommentTable' do
  from DDL::CommentTable
  to DML_DDL::CommentTable
  mapping do |ddl, dml_ddl|
    dml_ddl.tableName = ddl.tableName
    dml_ddl.tableComment = ddl.tableComment
  end
end

rule 'CommentColumn2CommentColumn' do
  from DDL::CommentColumn
  to DML_DDL::CommentColumn
  mapping do |ddl, dml_ddl|
    dml_ddl.tableName = ddl.tableName
    dml_ddl.columnName = ddl.columnName
    dml_ddl.columnComment = ddl.columnComment
  end
end

rule 'Registry2Registry' do
  from DML::Registry
  to DML_DDL::Registry
  mapping do |dml, dml_ddl|
    dml_ddl.registryValues = dml.registryValues
  end
end
```

```
rule 'Value2Value' do
  from DML::Value
  to DML_DDL::Value
  mapping do |dml, dml_ddl|
    dml_ddl.value = dml.value
    dml_ddl.column = DML_DDL::Table.all_objects.select{|dml_ddl|
dml_ddl.tableName == dml.column.getInsertInto.tableName}.map{|dml_ddl|
dml_ddl.columns}.flatten.find{|dml_ddl| dml_ddl.columnName ==
dml.column.columnName}
  end
end

#################################################################
######################### TYPES ##################################
#################################################################

#################################################################
######################### EXACTOS ###############################
rule 'Integer2Integer' do
  from DDL::Integer
  to DML_DDL::Integer
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'SmallInteger2SmallInteger' do
  from DDL::SmallInteger
  to DML_DDL::SmallInteger
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Int2Int' do
  from DDL::Int
  to DML_DDL::Int
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'SmallInt2SmallInt' do
  from DDL::SmallInt
  to DML_DDL::SmallInt
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Numeric2Numeric' do
  from DDL::Numeric
  to DML_DDL::Numeric
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Number2Number' do
  from DDL::Number
  to DML_DDL::Number
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Decimal2Decimal' do
```

```
  from DDL::Decimal
  to DML_DDL::Decimal
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end
################################################################
################################################################
####################### APROXIMADOS ########################
rule 'Real2Real' do
  from DDL::Real
  to DML_DDL::Real
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'DoublePrecision2DoublePrecision' do
  from DDL::DoublePrecision
  to DML_DDL::DoublePrecision
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Float2Float' do
  from DDL::Float
  to DML_DDL::Float
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Long2Long' do
  from DDL::Long
  to DML_DDL::Long
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'LongRaw2LongRaw' do
  from DDL::LongRaw
  to DML_DDL::LongRaw
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

################################################################
################################################################
####################### CHARACTERS #########################
rule 'Character2Character' do
  from DDL::Character
  to DML_DDL::Character
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'CharacterVarying2CharacterVarying' do
  from DDL::CharacterVarying
  to DML_DDL::CharacterVarying
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end
```

```
rule 'Char2Char' do
  from DDL::Char
  to DML_DDL::Char
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'VarChar2VarChar' do
  from DDL::VarChar
  to DML_DDL::VarChar
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'VarChar22VarChar2' do
  from DDL::VarChar2
  to DML_DDL::VarChar2
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'NVarChar22NVarChar2' do
  from DDL::NVarChar2
  to DML_DDL::NVarChar2
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'NChar2NChar' do
  from DDL::NChar
  to DML_DDL::NChar
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'CharVarying2CharVarying' do
  from DDL::CharVarying
  to DML_DDL::CharVarying
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'NationalChar2NationalChar' do
  from DDL::NationalChar
  to DML_DDL::NationalChar
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'NationalCharVarying2NationalCharVarying' do
  from DDL::NationalCharVarying
  to DML_DDL::NationalCharVarying
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'NationalCharacter2NationalCharacter' do
  from DDL::NationalCharacter
```

```
  to DML_DDL::NationalCharacter
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'NationalCharacterVarying2NationalCharacterVarying' do
  from DDL::NationalCharacterVarying
  to DML_DDL::NationalCharacterVarying
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'NCharVarying2NCharVarying' do
  from DDL::NCharVarying
  to DML_DDL::NCharVarying
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Clob2Clob' do
  from DDL::Clob
  to DML_DDL::Clob
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'NClob2NClob' do
  from DDL::NClob
  to DML_DDL::NClob
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end
################################################################
################################################################
######################### BITS #################################
rule 'Bit2Bit' do
  from DDL::Bit
  to DML_DDL::Bit
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'BitVarying2BitVarying' do
  from DDL::BitVarying
  to DML_DDL::BitVarying
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end
################################################################
################################################################
######################### TIMES ################################
rule 'Time2Time' do
  from DDL::Time
  to DML_DDL::Time
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Date2Date' do
```

```
  from DDL::Date
  to DML_DDL::Date
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Timestamp2Timestamp' do
  from DDL::Timestamp
  to DML_DDL::Timestamp
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end
################################################################
################################################################
####################### INTERVALS ##########################
rule 'DayTime2DayTime' do
  from DDL::DayTime
  to DML_DDL::DayTime
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'YearMonth2YearMonth' do
  from DDL::YearMonth
  to DML_DDL::YearMonth
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end
################################################################
################################################################
####################### BINARIES ###########################
rule 'BinaryDouble2BinaryDouble' do
  from DDL::BinaryDouble
  to DML_DDL::BinaryDouble
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'BinaryFloat2BinaryFloat' do
  from DDL::BinaryFloat
  to DML_DDL::BinaryFloat
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'BFile2BFile' do
  from DDL::BFile
  to DML_DDL::BFile
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end

rule 'Blob2Blob' do
  from DDL::Blob
  to DML_DDL::Blob
  mapping do |ddl, dml_ddl|
    dml_ddl.name = ddl.name
  end
end
################################################################
```

```
# Write your rules here. Use templates (CTRL + SPACE) to ease your work.
```

11.7 DetectionErrors.rb

```
transformation 'name'
use_library 'helper://modernizar_BD.rb'

# Contador para llevar un registro de los errores que se van encontrando
counter = Counter.new

top_rule 'error' do
  from DML_DDL::DDLDefinition
  to Errores::Errores
  mapping do |dml_ddl, error|
    dml_ddl.getErrors

    error.errores = dml_ddl.getCks
    error.errores = dml_ddl.getFks
    error.errores = dml_ddl.getPks
  end
end

rule 'ck_Desactivada' do
  from DML_DDL::Ck
  to Errores::CheckError
  mapping do |dml_ddl, error|
    error.porcent = dml_ddl.getPorcent
    error.set('table', dml_ddl.getTable)
    error.set('ck', dml_ddl)
    dml_ddl.getRegistries.each do |reg|
      error.set('registriesCk', reg)
    end
    error.id = counter.next()
    error.apply = true
  end
end

rule 'fk_Desactivada' do
  from DML_DDL::Fk
  to Errores::ForeignError
  mapping do |dml_ddl, error|
    error.porcent = dml_ddl.getPorcent
    error.set('tableCont', dml_ddl.getTable)
    error.set('fk', dml_ddl)
    error.set('tableRef', dml_ddl.references)

    dml_ddl.getRegistries.each do |reg|
      error.set('registriesFk', reg)
    end

    error.apply = true
    error.id = counter.next()
  end
end

rule 'check_Pk' do
  from DML_DDL::Pk
  to Errores::ForeignError
  mapping do |dml_ddl, error|
```

```
    error.porcent = dml_ddl.getPorcent
    error.set('tableRef', dml_ddl.getTable)
    error.set('fkColumns', dml_ddl.getColumnsRef)
    error.set('tableCont', dml_ddl.getTableRef)

    dml_ddl.getRegistries.each do |reg|
      error.set('registriesFk', reg)
    end

    error.apply = true
    error.id = counter.next()
  end
end
```

11.8 CorrectErrors.rb

```
transformation 'name'
use_library 'helper://correct_BD.rb'

top_rule 'DDLDefinition' do
  from DML_DDL::DDLDefinition
  to Correct::DDLDefinition
  mapping do |dml_ddl, correct|
    correct.statements = dml_ddl.statements
    correct.dataType = dml_ddl.dataType
  end
end

rule 'DataType' do
  from DML_DDL::DataType
  to Correct::DataType
  mapping do |dml_ddl, correct|
    correct.types = dml_ddl.types
  end
end

rule 'Database' do
  from DML_DDL::Database
  to Correct::Database
  mapping do |dml_ddl, correct|
    correct.databaseName = dml_ddl.databaseName
  end
end

rule 'Table' do
  from DML_DDL::Table
  to Correct::Table
  mapping do |dml_ddl, correct|
    correct.tableName = dml_ddl.tableName
    correct.columns = dml_ddl.columns
    correct.columnsPk = dml_ddl.columnsPk

    correct.columnsFk = dml_ddl.columnsFk

    correct.checks = dml_ddl.checks
    correct.commentTable = dml_ddl.commentTable

    if (dml_ddl.hasError)
```

```
      correct.registries = dml_ddl.getRegistries
      newsFk = dml_ddl.getNewsFk
      newsFk.each do |fk|
        correct.columnsFk << fk
      end
    else
      correct.registries = dml_ddl.registries
    end
  end
end

rule 'Column' do
  from DML_DDL::Column
  to Correct::Column
  mapping do |dml_ddl, correct|
    correct.columnName = dml_ddl.columnName
    correct.commentColumn = dml_ddl.commentColumn
    correct.columnNull = dml_ddl.columnNull
    correct.columnType = dml_ddl.columnType
  end
end

rule 'Pk' do
  from DML_DDL::Pk
  to Correct::Pk
  mapping do |dml_ddl, correct|
    correct.namePk = dml_ddl.namePk
    correct.columnName = dml_ddl.columnName
  end
end

rule 'Fk' do
  from DML_DDL::Fk
  to Correct::Fk
  mapping do |dml_ddl, correct|
    correct.nameFk = dml_ddl.nameFk
    correct.columnName = dml_ddl.columnName
    correct.references = dml_ddl.references
    correct.columnReference = dml_ddl.columnReference
    if (dml_ddl.hasError)
      correct.status = "ENABLED"
    else
      correct.status = dml_ddl.status
    end
  end
end

rule 'Ck' do
  from DML_DDL::Ck
  to Correct::Ck
  mapping do |dml_ddl, correct|
    correct.nameCk = dml_ddl.nameCk
    correct.valuesCk = dml_ddl.valuesCk
    if (dml_ddl.hasError)
      correct.status = "ENABLED"
    else
      correct.status = dml_ddl.status
    end
  end
end

rule 'ValuesCk' do
  from DML_DDL::ValuesCk
  to Correct::ValuesCk
  mapping do |dml_ddl, correct|
    correct.value = dml_ddl.value
    correct.comparator = dml_ddl.comparator
```

```
    correct.columnName = dml_ddl.columnName
    correct.logConjuntion = dml_ddl.logConjuntion
  end
end

rule 'CommentTable' do
  from DML_DDL::CommentTable
  to Correct::CommentTable
  mapping do |dml_ddl, correct|
    correct.tableName = dml_ddl.tableName
    correct.tableComment = dml_ddl.tableComment
  end
end

rule 'CommentColumn' do
  from DML_DDL::CommentColumn
  to Correct::CommentColumn
  mapping do |dml_ddl, correct|
    correct.tableName = dml_ddl.tableName
    correct.columnName = dml_ddl.columnName
    correct.columnComment = dml_ddl.columnComment
  end
end

rule 'Registry' do
  from DML_DDL::Registry
  to Correct::Registry
  mapping do |dml_ddl, correct|
    correct.registryValues = dml_ddl.registryValues
  end
end

rule 'Value' do
  from DML_DDL::Value
  to Correct::Value
  mapping do |dml_ddl, correct|
    correct.value = dml_ddl.value
    correct.column = dml_ddl.column
  end
end

################################################################
######################### TYPES ################################
################################################################

################################################################
######################### EXACTOS ##############################
rule 'Integer' do
  from DML_DDL::Integer
  to Correct::Integer
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'SmallInteger' do
  from DML_DDL::SmallInteger
  to Correct::SmallInteger
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Int' do
  from DML_DDL::Int
  to Correct::Int
  mapping do |dml_ddl, correct|
```

```
    correct.name = dml_ddl.name
  end
end

rule 'SmallInt' do
  from DML_DDL::SmallInt
  to Correct::SmallInt
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Numeric' do
  from DML_DDL::Numeric
  to Correct::Numeric
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Number' do
  from DML_DDL::Number
  to Correct::Number
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Decimal' do
  from DML_DDL::Decimal
  to Correct::Decimal
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end
################################################################
################################################################
###################### APROXIMADOS ##########################
rule 'Real' do
  from DML_DDL::Real
  to Correct::Real
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'DoublePrecision' do
  from DML_DDL::DoublePrecision
  to Correct::DoublePrecision
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Float' do
  from DML_DDL::Float
  to Correct::Float
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Long' do
  from DML_DDL::Long
  to Correct::Long
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
```

```
  end
end

rule 'LongRaw' do
  from DML_DDL::LongRaw
  to Correct::LongRaw
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

################################################################
################################################################
###################### CHARACTERS ##########################
rule 'Character' do
  from DML_DDL::Character
  to Correct::Character
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'CharacterVarying' do
  from DML_DDL::CharacterVarying
  to Correct::CharacterVarying
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Char' do
  from DML_DDL::Char
  to Correct::Char
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'VarChar' do
  from DML_DDL::VarChar
  to Correct::VarChar
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'VarChar2' do
  from DML_DDL::VarChar2
  to Correct::VarChar2
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'NVarChar2' do
  from DML_DDL::NVarChar2
  to Correct::NVarChar2
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'NChar' do
  from DML_DDL::NChar
  to Correct::NChar
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
```

```
  end
end

rule 'CharVarying' do
  from DML_DDL::CharVarying
  to Correct::CharVarying
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'NationalChar' do
  from DML_DDL::NationalChar
  to Correct::NationalChar
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'NationalCharVarying' do
  from DML_DDL::NationalCharVarying
  to Correct::NationalCharVarying
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'NationalCharacter' do
  from DML_DDL::NationalCharacter
  to Correct::NationalCharacter
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'NationalCharacterVarying' do
  from DML_DDL::NationalCharacterVarying
  to Correct::NationalCharacterVarying
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'NCharVarying' do
  from DML_DDL::NCharVarying
  to Correct::NCharVarying
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Clob' do
  from DML_DDL::Clob
  to Correct::Clob
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'NClob' do
  from DML_DDL::NClob
  to Correct::NClob
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end
#############################################################
```

```
#############################################################
########################## BITS #############################
rule 'Bit' do
  from DML_DDL::Bit
  to Correct::Bit
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'BitVarying' do
  from DML_DDL::BitVarying
  to Correct::BitVarying
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end
#############################################################
#############################################################
########################## TIMES ############################
rule 'Time' do
  from DML_DDL::Time
  to Correct::Time
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Date' do
  from DML_DDL::Date
  to Correct::Date
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Timestamp' do
  from DML_DDL::Timestamp
  to Correct::Timestamp
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end
#############################################################
#############################################################
######################## INTERVALS ##########################
rule 'DayTime' do
  from DML_DDL::DayTime
  to Correct::DayTime
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'YearMonth' do
  from DML_DDL::YearMonth
  to Correct::YearMonth
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end
#############################################################
#############################################################
######################## BINARIES ###########################
rule 'BinaryDouble' do
  from DML_DDL::BinaryDouble
  to Correct::BinaryDouble
```

```
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'BinaryFloat' do
  from DML_DDL::BinaryFloat
  to Correct::BinaryFloat
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'BFile' do
  from DML_DDL::BFile
  to Correct::BFile
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end

rule 'Blob' do
  from DML_DDL::Blob
  to Correct::Blob
  mapping do |dml_ddl, correct|
    correct.name = dml_ddl.name
  end
end
#############################################################
```

11.9 correct_BD.rb

```
decorator DML_DDL::Ck do
  def hasError
    error = Errores::CheckError.all_objects.select{|error| error.ck == self &&
error.apply == true}

    return !(error.empty?)
  end
end

decorator DML_DDL::Fk do
  def hasError
    error = Errores::ForeignError.all_objects.select{|error| error.fk == self
&& error.apply == true}

    return !(error.empty?)
  end
end

decorator DML_DDL::Table do
  # Identficador para la posible creacion de nuevas Fk
  @@id = 0

  def hasError
    # Comprobamos si la tabla presenta algun error.
    # Para ello comprobamos si esta referenciada en ForeginError o CheckError

    @foreignErrors = Errores::ForeignError.all_objects.select{|error|
error.tableCont == self && error.apply == true}
    @checkErrors = Errores::CheckError.all_objects.select{|error| error.table
```

```
== self && error.apply == true}

    return (!(@foreignErrors.empty?) || !(@checkErrors.empty?))
  end

  def getRegistries
    registries = self.registries

    if (!@checkErrors.empty?)
      for i in 0..@checkErrors.length-1
        @checkErrors[i].registriesCk.each do |regCk|
          registries.delete(regCk)
        end
      end
    end

    if (!@foreignErrors.empty?)
      for i in 0..@foreignErrors.length-1
        @foreignErrors[i].registriesFk.each do |regFk|
          registries.delete(regFk)
        end
      end
    end

    return registries
  end

  def getNewsFk
    newsFk = Array.new
    if (!@foreignErrors.empty?)
      for i in 0..@foreignErrors.length-1
        if (@foreignErrors[i].fk == nil)
          fkError = @foreignErrors[i]
          name = "NEW_FK_" + @@id.to_s
          fk = Correct::Fk.new(:nameFk => name)
          @@id.next

          columnName = Array.new
          fkError.fkColumns.each do |column|
            columnName << column.columnName
          end
          fk.columnName = columnName

          fk.references = fkError.tableRef

          fk.columnReference = fkError.tableRef.columnsPk.columnName

          fk.status = "ENABLED"

          newsFk << fk
        end
      end
    end

    return newsFk
  end
end
```

11.10CorrectErrorsResidual.rb

```
transformation 'name'
use_library 'helper://residual_BD.rb'

top_rule 'DDLDefinition' do
  from DML_DDL::DDLDefinition
  to Residual::DDLDefinition
  mapping do |dml_ddl, res|
    res.statements = dml_ddl.getStatements
  end
end

rule 'Table' do
  from DML_DDL::Table
  to Residual::Table
  filter {|dml_ddl| dml_ddl.hasError == true}
  mapping do |dml_ddl, res|
    res.tableName = dml_ddl.tableName
    res.columns = dml_ddl.columns

    res.registries = dml_ddl.getRegistries
  end
end

rule 'Column' do
  from DML_DDL::Column
  to Residual::Column
  mapping do |dml_ddl, res|
    res.columnName = dml_ddl.columnName
  end
end

rule 'Registry' do
  from DML_DDL::Registry
  to Residual::Registry
  mapping do |dml_ddl, res|
    res.registryValues = dml_ddl.registryValues
  end
end

rule 'Value' do
  from DML_DDL::Value
  to Residual::Value
  mapping do |dml_ddl, res|
    res.value = dml_ddl.value
    res.column = dml_ddl.column
  end
end

#####################################################################
```

11.11residual_BD.rb

```
decorator DML_DDL::DDLDefinition do
  def getStatements
    # Lo que hacemos es comprobar para cada statements si es de tipo Tabla,
entonces solamente
    # se añadirá al array si dicha tabla presenta algún error, en otro caso no
se añade
    # para que así no sea añadida al metamodelo residual
    #
    statements = Array.new
    self.statements.each do |statement|
      if (statement.kind_of? DML_DDL::Table)
        if (statement.hasError)
          statements << statement
        end
      end
    end
    return statements
  end
end

decorator DML_DDL::Table do
  def hasError
    # Comprobamos si la tabla presenta algun error.
    # Para ello comprobamos si esta referenciada en ForeginError o CheckError

    @foreignErrors = Errores::ForeignError.all_objects.select{|error|
error.tableCont == self && error.apply == true &&
!(error.registriesFk.empty?)}
    @checkErrors = Errores::CheckError.all_objects.select{|error| error.table
== self && error.apply == true && !(error.registriesCk.empty?)}

    return (!(@foreignErrors.empty?) || !(@checkErrors.empty?))
  end

  def getRegistries
    registries = Array.new

    if (!@checkErrors.empty?)
      for i in 0..@checkErrors.length-1
        @checkErrors[i].registriesCk.each do |regCk|
          registries << regCk
        end
      end
    end

    if (!@foreignErrors.empty?)
      for i in 0..@foreignErrors.length-1
        @foreignErrors[i].registriesFk.each do |regFk|
          registries << regFk
        end
      end
    end

    return registries
  end
end
```

11.12create_table.rtemplate

```
-- Automatic Generation Of TABLE: <%= table.tableName %>
<%
tableName = table.tableName
columns = Array.new
fk = Array.new
fkDisabled = Array.new
ck = Array.new
ckDisabled = Array.new

table.columns.each do |column|
  null = "NOT NULL"
  if (column.columnNull)
       null = "NULL"
  end

  columns << column.columnName + " " + column.columnType.name + " " + null +
","
end

table.checks.each do |check|
  aux = "CONSTRAINT " + check.nameCk + " CHECK ("
  check.valuesCk.each do |val|
       aux += val.columnName
       if (val.value.length > 1)
         aux += " IN ("
         for i in 0..val.value.length-1
              aux += val.value[i]
              if (i+1 <= val.value.length-1)
                aux += ","
              end
         end
         aux += ")"
       else
         aux += " " + val.comparator + " " + val.value[0]
       end
       if (val.logConjuntion != nil)
         aux += " " + val.logConjuntion + " "
       end
  end
  ck << aux + "),"
end

table.columnsFk.each do |f|
  fk << "CONSTRAINT " + f.nameFk + " FOREIGN KEY (" +
f.columnName.map{|colName| colName}.join(',') + ") REFERENCES " +
f.references.tableName + " (" + f.columnReference.map{|colName|
colName}.join(',') + "),"
end

pk = "CONSTRAINT " + table.columnsPk.namePk + " PRIMARY KEY (" +
table.columnsPk.columnName.map{|colName| colName}.join(',') + ")"

table.columnsFk.each do |f|
  if (f.status == "DISABLED")
    fkDisabled << "ALTER TABLE " + tableName + " DROP CONSTRAINT " + f.nameFk
+ ";"
  end
end

table.checks.each do |c|
  if (c.status == "DISABLED")
    ckDisabled << "ALTER TABLE " + tableName + " DROP CONSTRAINT " + c.nameCk
+ ";"
```

```
  end
end

-%>
CREATE TABLE <%= tableName %> (
<% columns.each do |column| -%>
  <%= column %>
<% end -%>
<% ck.each do |check| -%>
  <%= check %>
<% end -%>
<% fk.each do |f| -%>
  <%= f %>
<% end -%>
  <%= pk %>
);
<% fkDisabled.each do |disabled| -%>
<%= disabled %>
<% end -%>
<% ckDisabled.each do |disabled| -%>
<%= disabled %>
<% end -%>
```

11.13insert_into.rtemplate

```
-- Automatic Generation Insert's Of TABLE: <%= table.tableName %>
<%
tableName = table.tableName
registries = Array.new
columns = "("

cols = table.columns
for i in 0..cols.length-1
  columns += cols[i].columnName
  if (i+1 <= cols.length-1)
    columns += ","
  end
end
columns += ")"

insert = "INSERT INTO " + tableName + " " + columns + " VALUES "

table.registries.each do |registry|
  values = "("
  registryValues = registry.registryValues
  for i in 0..registryValues.length-1
    if (registryValues[i].column.columnType.kind_of? DML_DDL::Characters)
      values += "'" + registryValues[i].value + "'"
    else
      values += registryValues[i].value
    end
    if (i+1 <= registryValues.length-1)
      values += ","
    end
  end
  values += ");"

  registries << insert + values
end
-%>
```

```
<% registries.each do |registry| -%>
  <%= registry %>
<% end -%>
```

11.14Instalación y Ejecución

Para poder usar la herramienta del asistente creada para la integración de las etapas del proyecto, así como de los proyectos de Gra2Mol para poder extraer los modelos DDL y DML de una Base de Datos, se tiene que instalar lo siguiente:

- Plataforma Eclipse. Para la elaboración de este proyecto como para sus pruebas se ha usado la versión de Eclipse *"Europa versión 3.3.2"*.
- Máquina virtual de Java. El proyecto ha sido elaborado sobre la máquina virtual *"JDK 1.6.0_18"*.
- EMF. Se ha usado la versión *"emf-sdo-SDK-2.3.2"*.
- Plugin *"AGE versión 0.3.4"* para usar RubyTL en Eclipse.
- El intérprete de Ruby versión *"1.8.6-25"*.
- El intérprete de RubyTL *"rubytl-snapshot.02-Feb-2010.tar"*.
- El plugin con el asistente que se ha creado en este proyecto. *"es.modelum.deteErrores_1.0.0.jar"*.

La única configuración que hay realizar a todo lo instalado anteriormente es la siguiente. Una vez que se ha instalado el plugin de AGE, los intérpretes de Ruby y de RubyTL tienen que ser configurados configurar en Eclipse. En concreto hay que indicarle a Eclipse donde se encuentran ambos intérpretes. Para ello nos vamos a *"Windows -> Preferences"* y seleccionamos *"Ruby -> Installed Interpreters"*. En ese dialogo podemos añadir el intérprete de Ruby que se acaba de instalar. Normalmente el interprete de Ruby suele estar en la ruta *"C:/ruby/bin/ruby.exe"* o *"/usr/bin/ruby"*.

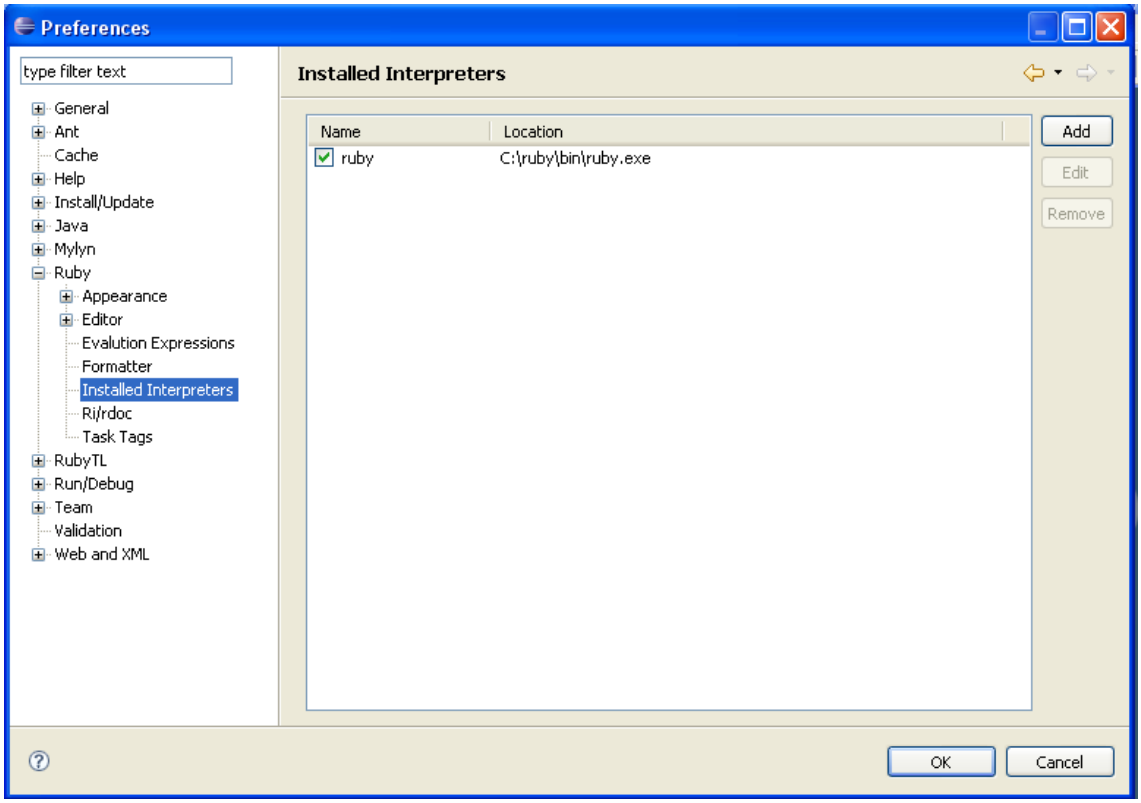

Figura 21: Configuración intérprete Ruby en Eclipse

Para indicarle a Eclipse donde se encuentra el intérprete de RubyTL, nos vamos a *"Windows -> Preferences"* y seleccionamos *"RubyTL -> Path options"*, en el dialogo que te mostrará a la derecha indicamos que queremos usar un path alternativo para RubyTL e indicamos donde se encuentra dicho path alternativo.

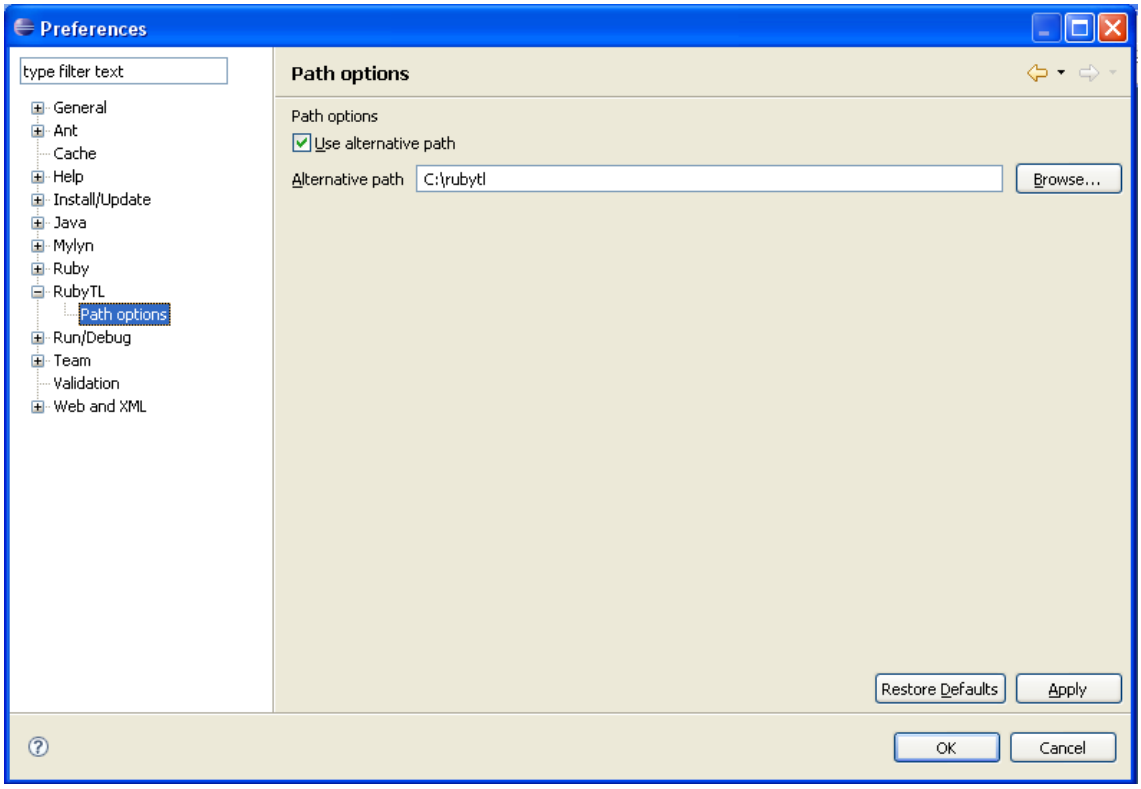

Figura 22: Configuración intérprete RubyTL en Eclipse.

Una vez que se tiene instalada y configurada la plataforma donde poder lanzar la modernización, se comentará muy por encima el uso de los proyectos creados.

Se han creado dos proyectos llamados “DDL” y “DML” para poder realizar la extracción de los modelos DDL y DML de una base de datos, respectivamente. Importamos los dos proyectos a la plataforma y copiamos los scripts obtenidos de la base de datos. Se copia el scripts con las sentencias DDL en el proyecto *‘DDL’* en la ruta *‘files/src’*, y el script con las sentencias DML en el proyecto *‘DML’* en la ruta *‘files/src’*.

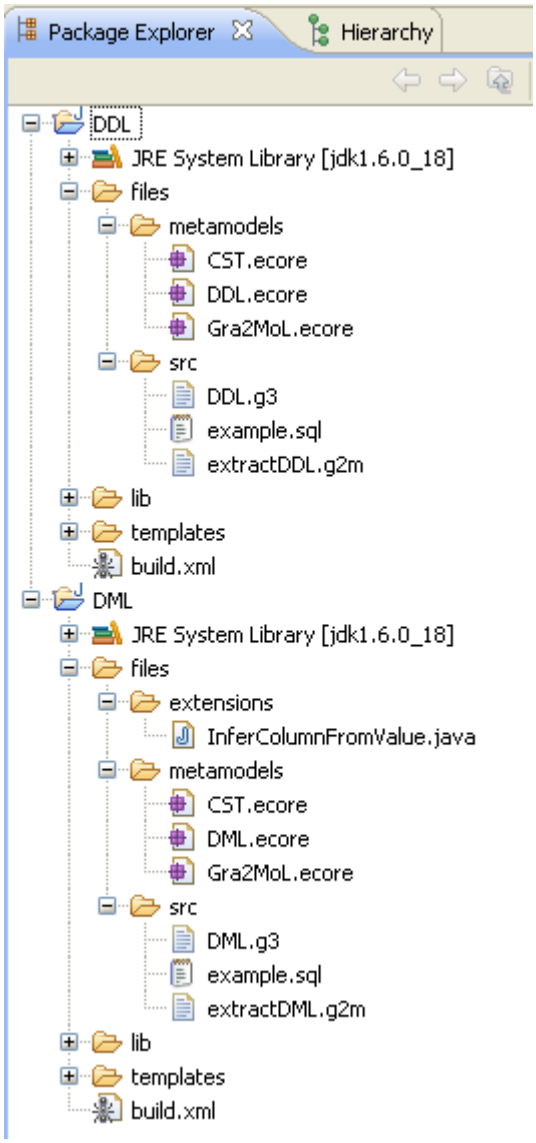

Figura 23: Proyectos DDL y DML.

Una vez que tenemos los scripts copiados se lanzan ambas extracciones. Para ello solamente hay que ejecutar el *'build.xml'* de cada uno de los proyectos. Al finalizar la extracciones se habrán obtenidos, al nivel del proyecto, los modelos *'DDL_Gramol.ecore'* y *'DML_Gramol.ecore'* en los proyectos DDL y DML respectivamente.

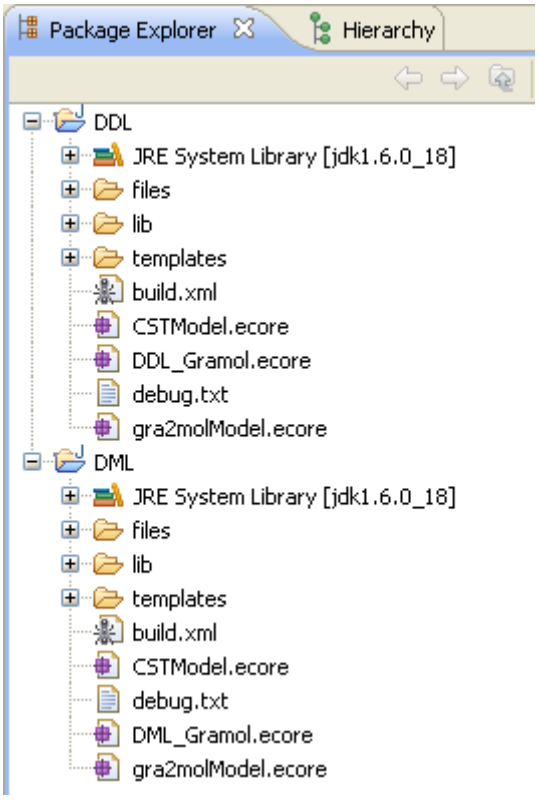

Figura 24: Proyectos DDL y DML después de la extracción.

Un vez que se tienen los modelos de la base de datos se puede lanzar el asistente para la modernización de un sistema de información creado en este proyecto. Para ello nos vamos a *"File -> New -> Deteccion Errors -> Detection Errors"*.

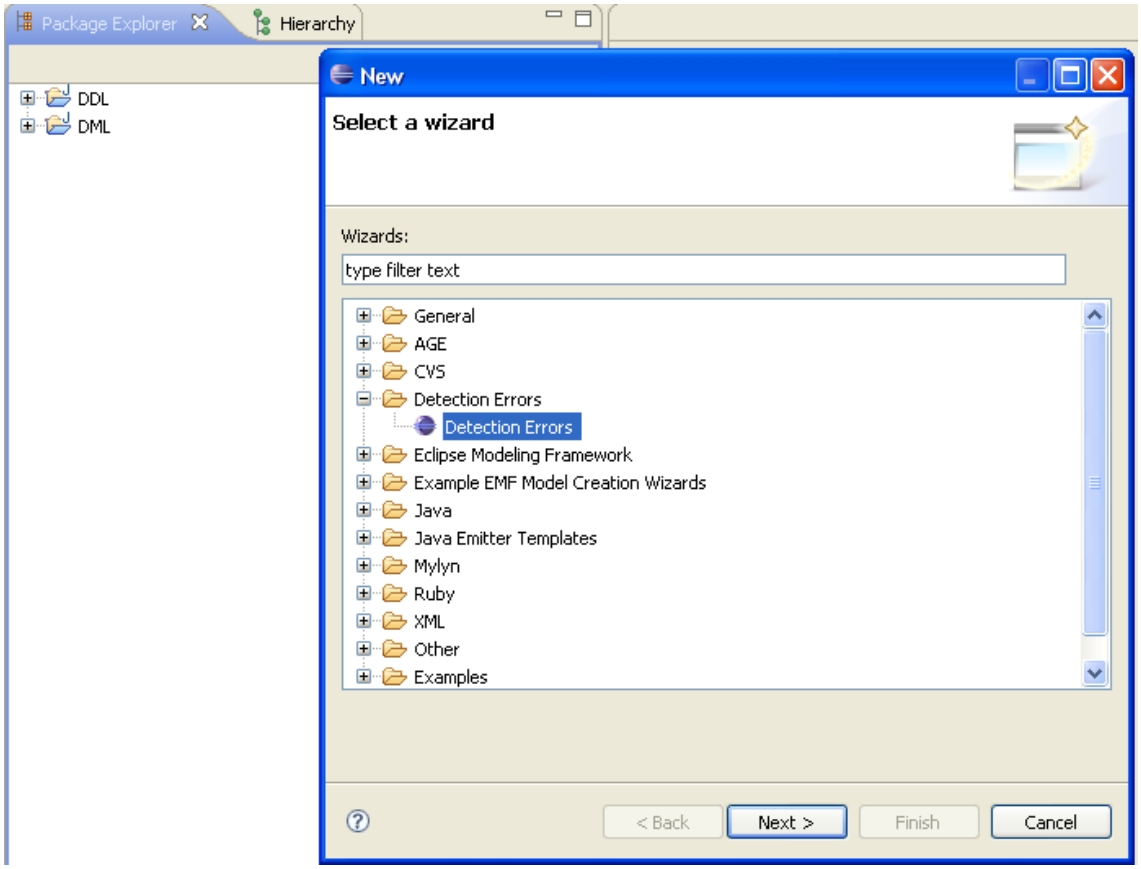

Figura 25: Ejecución del asistente de modernización.

Cómo no existía ningún proyecto de tipo RubyTL, la ejecución del asistente nos ha creado uno nuevo, y a la misma vez nos ha copiado todos los ficheros necesarios al proyecto nuevo para poder lanzar las diferentes transformaciones.

En la primera página hay que importar los modelos DML y DDL obtenidos anteriormente. Una vez que se han importado (se habrán copiado el proyecto), pinchamos en *"Launch"* para lanzar las transformaciones: *"Integración de los modelos DDL y DML"* y *"Detección de Errores"*, que se correspondían a las etapas 2 y 3 de este proyecto.

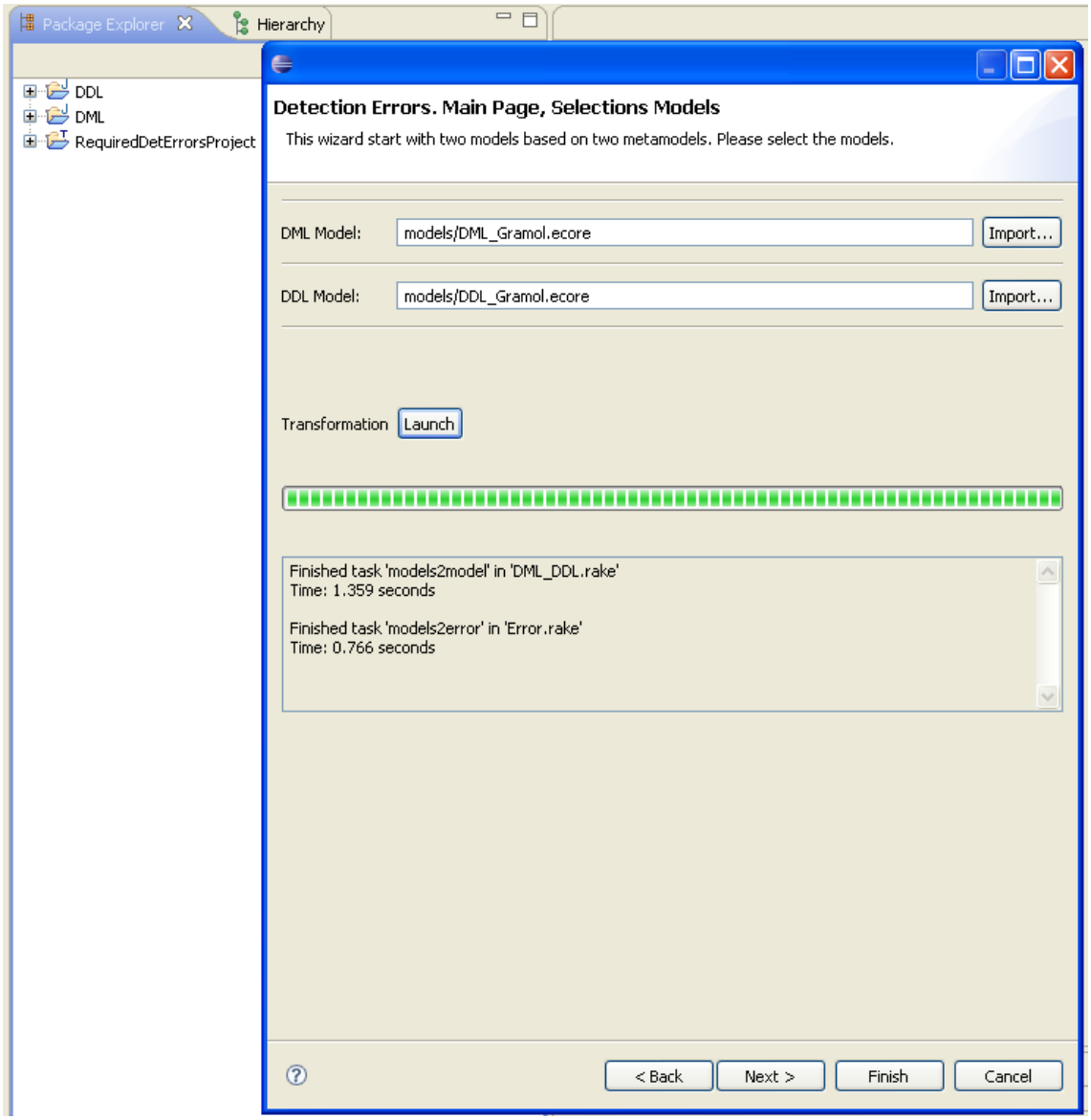

Figura 26: Carga de los modelos DML y DDL y ejecución de las transformaciones mediante el asistente.

Desde esa página ya se pueden corregir todos errores. Si pinchamos en "*Finish*" se corregirán todos los errores encontrados por el algoritmo, ejecutándose las etapas 4 y 5 del proyecto. Si por el contrario queremos comprobar los errores que ha encontrado y quizás lanzar solo unos pocos tendremos que pinchar en "*Next*".

En la siguiente página podemos ver una tabla con tras columnas y dos ventanas de texto a la derecha. Si pinchamos en *"Load Errors"* se cargarán mediante EMF, todos los errores encontrados mediante el algoritmo de detección.

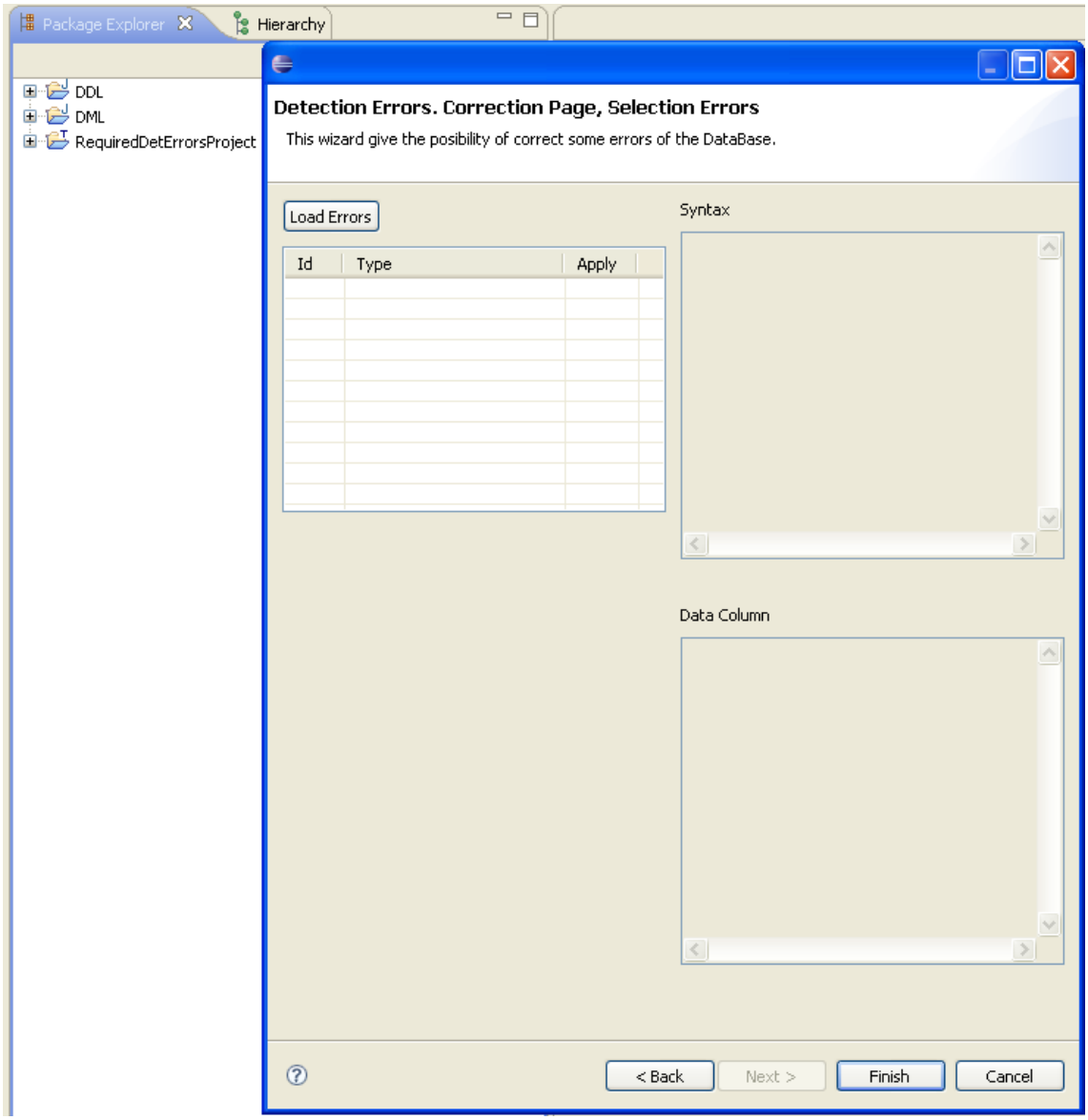

Figura 27: Visualización de los errores y finalización del asistente.

Como se puede ver, se muestra el identificador del error, el tipo de error que es y el booleano *'apply'* que estará por defecto a *"true"*. Si pinchamos encima del booleano se cambiará a *"false"*, si volvemos a pinchar volverá a *"true"* y así sucesivamente. Cuando se pulsa encima de un erro, en los cuadros de la derecha se muestra información relativa al error seleccionado. En la ventana superior se muestra cierta información, que para cada tipo de error será diferente:

- *'Ck_disabled'*: mostrará la tabla que contiene la *'Ck'*, las restricciones asociadas a dicha *'Ck'*.

- *'Fk_disabled'* y *'possible_new_Fk'*: se mostrará la tabla que contiene la *'Fk'* así como la tabla a la que referencia, que será la tabla de la *'Pk'*. Y también la correspondencia de cada una de las columnas dela foreign key con las columnas de la primary key.

En el cuadro inferior se mostrará un conjunto reducido de los registros que presentan algún error. Cada vez que se pinche se mostrarán 10 registros erróneos. La primera vez que se pinche encima del error se mostrarán los 10 primeros, la segunda vez los 10 siguientes y así sucesivamente hasta empezar de nuevo.

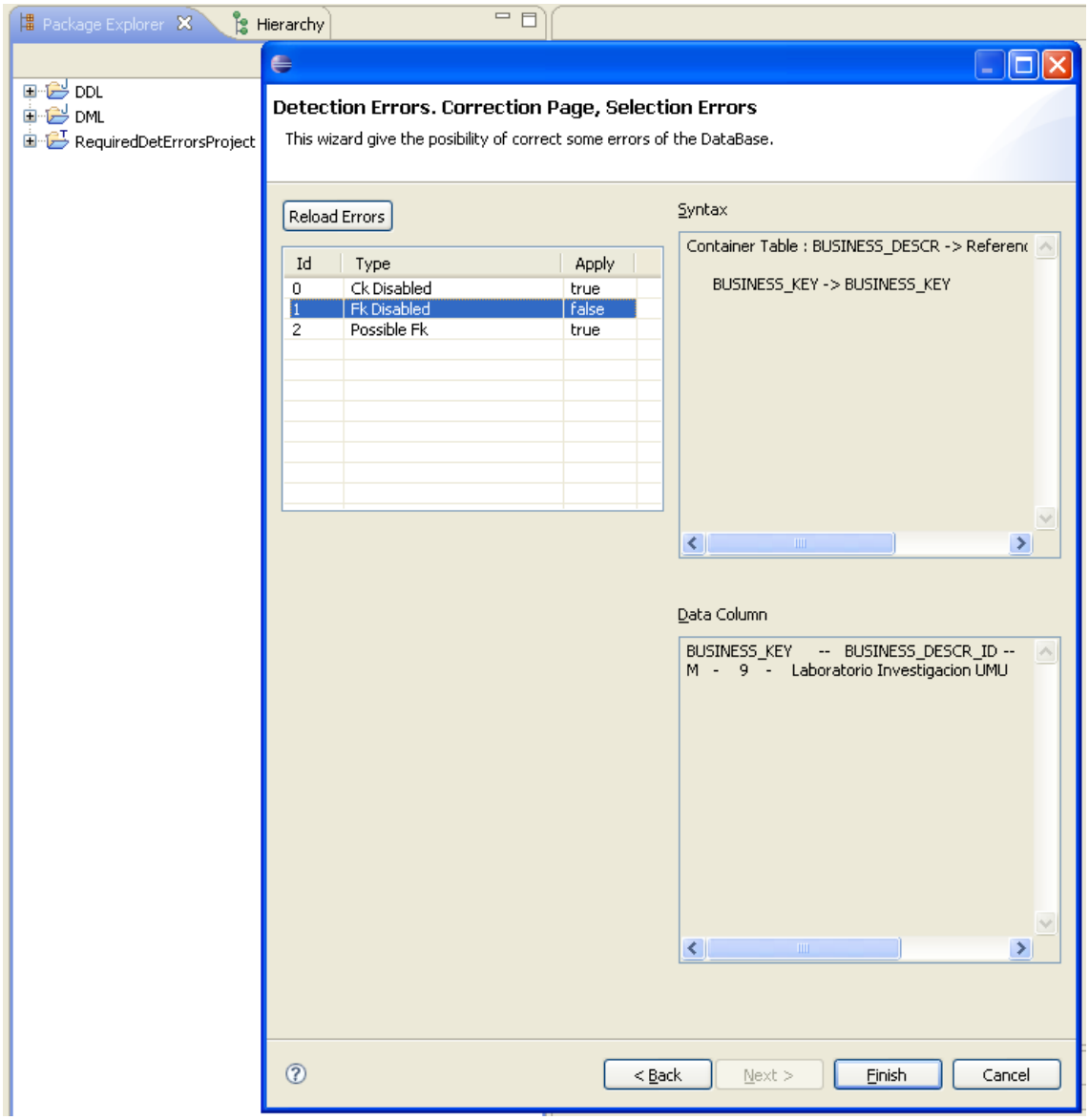

Figura 28: Información de los errores mostrada por el asistente.

Si pulsamos en "*Finish*" terminamos de ejecutar las etapas 4 y 5 del proyecto correspondientes a: *"Corrección de Errores"* y *"Generación de Código"*. Obteniendo todo lo comentado a lo largo de este documento.

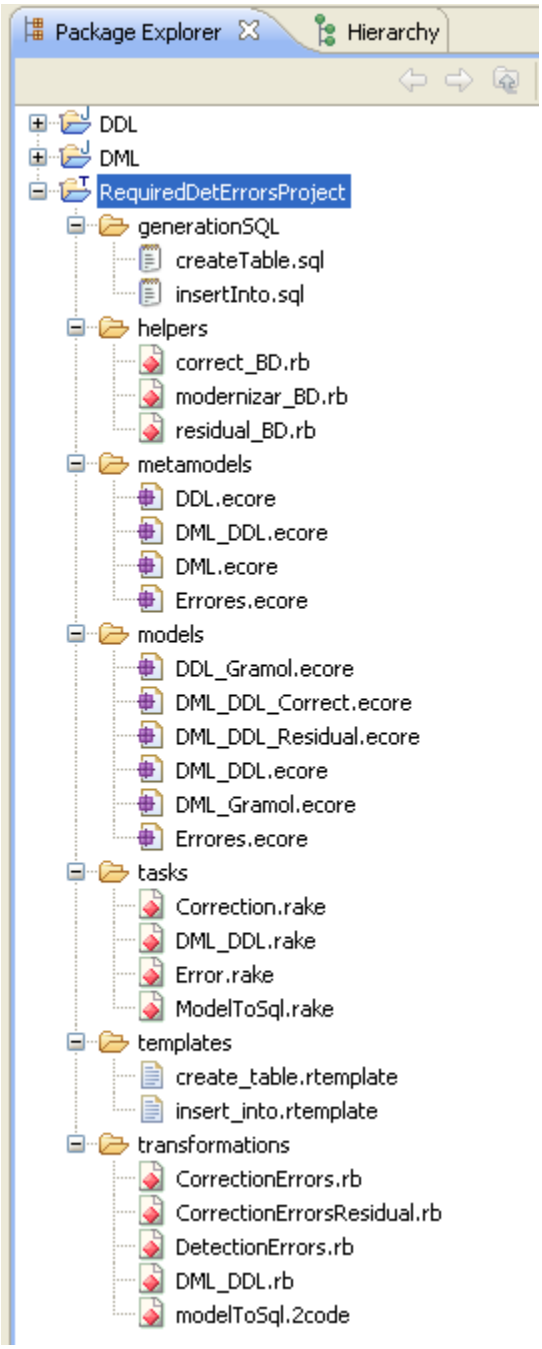

Figura 29: Proyecto final generado por el asistente.

11.15RubyTL Bug

A la hora de lanzar las transformaciones de RubyTL mediante código y no hacerlo mediante un fichero ANT de Eclipse, se detectó un error. El error tenía que ver con el parámetro que contenía el path del proyecto, path que se tomaba como el directorio base para la trasformación, que se le pasaba a la clase '*RakefileConfigurationData'*. Una vez que se lanzaba la transformación el motor de RubyTL incrustaba al final de la cadena que representaba al path los caracteres ***'cmd'***, por lo que siempre mostraba el error de que no encontraba el proyecto que se le había indicado.

Por lo que para poder hacer uso del asistente de integración de las etapas del proyecto, una vez que se ha instalado RubyTL en el ordenador, hay que corregir dicho error de la manera que se relata a continuación.

Recorriendo la traza de error que se mostraba se llegó al fichero *'rubytl.rb'* que se encuentra en ***'rubytl/lib***' (siendo ***'rubytl/'*** el directorio raíz de la instalación de RubyTL). El fichero tenía el siguiente aspecto:

```
RUBYTL_VERSION = '0.3.2'
$LOAD_PATH << File.dirname(__FILE__)
require 'base/platform'
require 'mri_platform'
Platform.impl = MRIPlatform.new
Platform.require_rake

require 'base/base'
Platform.require_model_libraries
Platform.require_components

if $0 == __FILE__
  rakefile = ARGV[0] || 'Rakefile'
  raketask = ARGV[1] || 'default'
  basedir  = ARGV[2] || Dir.pwd
  # This is a horrible hack, I'm not proud of it :-(
  if raketask == "ant_task"
    basedir = basedir.sub(/cmd$/, "")
  end
  config = RubyTL::Base::Configuration.basic(basedir)
  result = RubyTL::Base::RakeBuildSystem.new(config).launch(rakefile,
raketask)
  exit(-1) if not result
end
```

Como se puede observar dentro del '*if*' principal existe otro '*if*' que comprueba si la ejecución ha sido realizada mediante un fichero ANT de Eclipse, entonces le quita los caracteres ***'cmd'*** del final de la cadena del directorio base. En caso de que no sea lanzada la transformación mediante un ANT no la elimina. Por esta razón es por la que al lanzar la transformación desde código no salía el error mencionado anteriormente.

Para solventar esto únicamente hay que comentar el '*if*'. Haciendo que elimine siempre los caracteres ***'cmd'*** de la cadena que representará al directorio base. Quedando como sigue:

```
RUBYTL_VERSION = '0.3.2'
$LOAD_PATH << File.dirname(__FILE__)
require 'base/platform'
require 'mri_platform'
Platform.impl = MRIPlatform.new
Platform.require_rake

require 'base/base'
Platform.require_model_libraries
Platform.require_components

if $0 == __FILE__
  rakefile = ARGV[0] || 'Rakefile'
  raketask = ARGV[1] || 'default'
  basedir  = ARGV[2] || Dir.pwd
  # This is a horrible hack, I'm not proud of it :-(
  #if raketask == "ant_task"
    basedir = basedir.sub(/cmd$/, "")
  #end
  config = RubyTL::Base::Configuration.basic(basedir)
  result = RubyTL::Base::RakeBuildSystem.new(config).launch(rakefile,
raketask)
  exit(-1) if not result
end
```

Una vez realizado este cambio ya se puede usar el asistente que ejecutará las etapas 2, 3, 4 y 5 del proyecto.