Ingeniería Inversa en el Contexto de ADM

Claudia Pereira[1], Liliana Martinez[2], Liliana Favre[3]

[1,2]Universidad Nacional del Centro de la Provincia de Buenos Aires

Paraje Arroyo Seco, (B7000) Tandil, Argentina

[3]Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, La Plata, Argentina

[1]cpereira@exa.unicen.edu.ar, [2]lmartine@exa.unicen.edu.ar, [3]lfavre@exa.unicen.edu.ar

Resumen: *La ingeniería inversa es una etapa crucial en el proceso de modernización de software. La iniciativa Architecture–Driven Modernization (ADM) surgió con el objetivo de definir estándares para soportar el proceso de modernización en el contexto de desarrollos dirigidos por modelos, siendo relevantes para la ingeniería inversa los metamodelos Knowledge Discovery Metamodel y Abstract Syntax Tree Metamodel. Se propone en este artículo aplicar los principios de ADM para proveer vistas de alto nivel de sistemas heredados. Se describe un framework para la ingeniería inversa de código orientado a objetos a fin de extraer modelos UML. Se ejemplifica la propuesta con un caso de estudio que muestra cómo recuperar diagramas de casos de uso y de interacción a partir de código Java. La propuesta fue validada utilizando el framework de modelado Eclipse y la plataforma MoDisco que puede considerarse como la implementación oficial de los estándares ADM.*

Palabras Claves: *Ingeniería Inversa, Modernización Dirigida por la Arquitectura, Metamodelos, Transformaciones.*

Abstract: *Reverse engineering is a crucial stage in the software modernization process. The Architecture-Driven Modernization (ADM) initiative emerged with the aim of defining standards to support the modernization process in the model driven development context being relevant for reverse engineering the Knowledge Discovery Metamodel and the Abstract Syntax Tree Metamodel. In this paper, the ADM principles are applied to provide high-level views of legacy systems. A framework to reverse engineering UML models from object-oriented code is described. The proposal is exemplified by a case study showing how to recover use case and interaction diagrams from Java code. The proposal was validated using the Eclipse Modeling Framework and MoDisco platform that can be considered the official implementation of the ADM standards.*

Keywords: *Reverse Engineering, Architecture-Driven Modernization, Metamodels, Transformations.*

INTRODUCCIÓN

Gran parte de los sistemas de información vitales en organizaciones de nuestro medio fueron implementados hace varios años con tecnologías que hoy pueden considerarse obsoletas y no alineadas con los actuales objetivos estratégicos de las organizaciones. Estos sistemas, conocidos como heredados, involucran software, hardware, procesos de negocio y estrategias organizacionales. En general no están documentados o, si lo están, sus especificaciones no reflejan los cambios de requerimientos que se dieron a través de los años englobando sólo el código la historia de su evolución. La modernización de software se refiere a la transformación de un sistema de software existente en uno nuevo que satisfaga nuevas demandas. Para tratar este problema existen distintas soluciones, típicamente

nuevos desarrollos, wrapping o migración, cada una de las cuales impactan en mayor o menor medida sobre el sistema y como consecuencia sobre la organización. Una buena solución debería permitir recuperar el valor del sistema existente extrayendo la inversión y el conocimiento para transformarlo en uno nuevo que incorpore los nuevos requerimientos y/o tecnologías.

El éxito de la modernización dependerá del grado de automatización de las herramientas CASE que soportan el proceso y de la estandarización de los artefactos heredados. Nuevos enfoques de desarrollo de software dirigidos por modelos enmarcados en lo que se referencia como MDD (Model-Driven Development) podrían dar respuesta a esta demanda. Las técnicas de metamodelado y transformación de modelos en las que se basa MDD pueden ser usadas en este proceso ya que permiten la automatización de muchas actividades básicas incluyendo la manipulación de código (Favre, 2010). En este contexto Architecture-Driven Modernization (ADM) ha surgido como una evolución de Model-Driven Architecture (MDA) y sus estándares con el objetivo de establecer soluciones para la modernización de los sistemas de información (ADM, 2015; MDA, 2015). OMG ADM Task Force (ADMTF) está desarrollando un conjunto de estándares para facilitar la interoperabilidad entre las herramientas de modernización.

ADM lleva a cabo procesos de modernización considerando los principios esenciales de MDA: la representación de artefactos mediante modelos en diferentes niveles de abstracción y las transformaciones entre modelos. MDA distingue al menos los siguientes tipos de modelos: CIM (Computation Independet Model), PIM (Platform Independent Model) y PSM (Platform Specific Model). Una de las características esenciales de MDA es que todos los artefactos involucrados en un proceso de desarrollo son representados a partir del lenguaje de metamodelado MOF (Meta Object Facility) (MOF, 2014) el cual define una forma común de capturar todas las construcciones de los estándares de modelado e intercambio que son usadas en MDA.

El proceso de modernización incluye tres etapas: ingeniería inversa, reestructuración e ingeniería forward. La ingeniería inversa, el proceso de analizar los artefactos de software existentes para extraer información y proveer vistas de alto nivel del sistema es una etapa crucial dentro de la modernización del mismo. En el contexto de ADM la meta de la ingeniería inversa es descubrir el conocimiento del sistema existente y producir modelos en diferentes niveles de abstracción. Estos modelos serán el punto de partida para el proceso de reestructuración y la posterior generación del nuevo sistema.

En este artículo nos enfocamos en la primera de las etapas del proceso de modernización, describimos un framework para la ingeniería inversa en el contexto de ADM y mostramos cómo recuperar modelos que representan una visión abstracta del sistema a partir del código; en particular se analiza la extracción de diagramas de casos de uso y de interacción a partir de código Java. Esta propuesta se validó utilizando el framework de modelado Eclipse (Eclipse Modeling Framework - EMF), tecnología central dentro de Eclipse para el desarrollo dirigido por modelos (Eclipse, 2015). Además se utilizó el lenguaje ATL (Atlas Transformation Language) (ATL, 2015) como lenguaje de transformación de modelos y MoDisco para descubrir modelos intermedios (MoDisco, 2015). Cabe destacar que hasta el momento MoDisco sólo provee soporte para recuperar diagramas de clases a partir de código Java, por lo tanto, nuestro trabajo puede considerarse una contribución a la comunidad MoDisco.

Este artículo está organizado como sigue: la siguiente sección describe los estándares ADM utilizados en nuestra propuesta y el proyecto MoDisco. Luego se presentan trabajos relacionados. A continuación se describe un framework para la inge inge-

niería inversa en el contexto de ADM y se presenta un caso de estudio que muestra cómo recuperar diagramas de casos de uso y de secuencia a partir de código Java. Para finalizar se discuten las conclusiones y trabajos futuros.

MARCO TEÓRICO

Existe un creciente interés tanto en la migración de software como en la modernización de sistemas heredados con el objetivo de extender su vida útil. El éxito de la modernización de sistemas depende de la existencia de frameworks técnicos que permitan la integración de información y la interoperabilidad entre diferentes herramientas.

ADMTF ha establecido un conjunto de estándares (metamodelos) para la modernización con el objetivo de alcanzar interfaces y formatos bien definidos para el intercambio de información sobre modelos de software facilitando la interoperabilidad entre las herramientas y los servicios de modernización a los adherentes de los estándares.

Esto da lugar a una nueva generación de soluciones que beneficiará a la industria del software fomentando la colaboración entre distintos proveedores. Entre los estándares desarrollados cabe mencionar a los metamodelos Knowledge Discovery Metamodel (KDM) y Abstract Syntax Tree Metamodel (ASTM) (KDM, 2014; ASTM, 2011). KDM es la base para la modernización de software y representa no sólo al código sino al sistema completo. ASTM es la especificación de los elementos de modelado para expresar árboles de sintaxis abstracta (Abstract Syntax Tree - AST). KDM y ASTM son dos especificaciones de modelado complementarias. KDM establece una especificación que permite representar información semántica sobre el sistema de software mientras que ASTM establece una especificación para representar la sintaxis del código fuente por medio de AST. ASTM actúa como la base de más bajo nivel para el modelado de software en el ecosistema de estándares de OMG mientras que KDM puede considerarse como una puerta de entrada a los modelos OMG de más alto nivel.

El surgimiento de ADM trae aparejada la necesidad de desarrollar nuevas herramientas que suporten modelado, interoperabilidad y estandarización, transformación automatizada tanto para ingeniería forward como para inversa, acceso a la definición de estas transformaciones y soporte para trazabilidad. Actualmente la tecnología más completa que soporta ADM es MoDisco, un componente Eclipse para la ingeniería inversa dirigida por modelos. Hay distintas formas de extraer modelos a partir de los sistemas heredados debido a su heterogeneidad. Por esta razón MoDisco ofrece un framework extensible y genérico para facilitar el desarrollo de herramientas que permitan extraer modelos a partir de código o sistemas heredados. El proyecto MoDisco está trabajando en estrecha vinculación con ADMTF. MoDisco está organizado en tres niveles para facilitar la reutilización de componentes entre distintas soluciones de modernización. El nivel de Infraestructura contiene componentes genéricos independientes de cualquier tecnología heredada tales como las implementaciones de los metamodelos ASTM y KDM. El nivel de Tecnología contiene componentes dedicados a una tecnología específica tal como el metamodelo del lenguaje Java. El nivel de Casos de Uso contiene componentes que proveen soluciones para casos de modernización específicos.

MoDisco se ha utilizado en aplicaciones reales; en Bruneliere et al., 2014 se describen trabajos de investigación e industrialización realizados utilizando MoDisco en los últimos años ilustrando su uso concreto con escenarios de ingeniería inversa industriales diferentes. Sin embargo los autores sostienen que se debería mejorar el desempeño de la herramienta cuando se trabaja con sistemas grandes.

TRABAJOS RELACIONADOS

Numerosos trabajos han contribuido a la ingeniería inversa de código orientado a objetos. En Tonella y Potrich, 2005 los autores presentan una visión general de las técnicas que han sido recientemente investigadas y aplicadas en el campo de la ingeniería inversa de código orientado a objetos. Los autores describen los algoritmos utilizados para la recuperación de diagramas UML a partir de código tales como diagramas de clases, de estado y de secuencia. Nuestra propuesta puede ser considerada como una formalización de los procesos de ingeniería inversa descriptos por Tonella y Potrich en su trabajo en términos de los estándares involucrados en ADM. En Canfora y Di Penta, 2007 se comparan trabajos existentes en el área de ingeniería inversa, discuten casos exitosos e importantes logros y se provee una guía para posibles desarrollos futuros en el área.

Muchos trabajos están vinculados a la ingeniería inversa basada en MDD. En Fleurey et al., 2007 se describe el uso de MDE como una propuesta eficiente, flexible y confiable para el proceso de migración de software. Este proceso, desarrollado en Sodifrance, incluye análisis automático de código existente, recuperación de modelos abstractos de alto nivel, transformación de modelo-a-modelo y generación de código. Sodifrance ha desarrollado un conjunto de herramientas para la manipulación de modelos llamada Model-In-Action (MIA) utilizada como la base para la automatización de la migración. En Cánovas Izquierdo y García Molina, 2009 los autores analizan las dificultades encontradas al usar soluciones existentes para la extracción de modelos en el contexto de la modernización de software y proponen un lenguaje llamado Gra2MoL (Grammar To Model Transformation Language). Gra2MoL ha sido usado para extraer modelos a partir de código Java y PL/SQL. En Favre et al., 2009 se describe una propuesta de ingeniería inversa basada en MDA que integra técnicas basadas en teoría de compiladores, metamodelado y especificaciones formales. Los autores describen un proceso que combina análisis estático y dinámico para la generación de modelos MDA, muestran cómo los metamodelos MOF pueden ser usados para dirigir el proceso de recuperación de modelos y detallan cómo los metamodelos y transformaciones pueden integrarse con especificaciones formales de una manera interoperable.

Con el surgimiento de ADM se han desarrollado nuevas propuestas y herramientas. En Cánovas Izquierdo y García Molina, 2009 los autores presentan un proceso para extraer modelos que conforman al metamodelo KDM. Primero extraen modelos que conforman al ASTM utilizando Gra2MoL. Luego estos modelos son transformados en modelos KDM utilizando el lenguaje de transformación de modelo-a-modelo RubyTL; el proceso se ejemplifica con una migración de Struts a JSF. En esta propuesta no se recuperan modelos UML. En Barbier et al., 2011 se describe un método de ingeniería inversa dirigida por modelos basado en metamodelos y transformación de modelos y se ilustra la propuesta recuperando modelos UML a partir de sistemas COBOL. El método fue implementado en el módulo BLU AGE® Reverse. Los autores definen un lenguaje textual específico al dominio para transformar el código a modelos KDM que luego son transformados a modelos PIM expresados en UML. En Ulrich y Newcomb, 2010 los autores proveen una visión general de ADM junto con un estudio de distintos casos sobre modernización que cubren una variedad de sectores industriales.

UN FRAMEWORK PARA ADM

En Brambilla et al., 2012 se distinguen tres etapas principales en el proceso de modernización: el descubrimiento del modelo, el entendimiento del

modelo y la regeneración del código. La etapa del descubrimiento del modelo "descubre" un conjunto de modelos iniciales que representan al sistema heredado en el mismo nivel de abstracción. La etapa del entendimiento del modelo transforma los modelos iniciales en otros a un nivel de abstracción más alto, los cuales son el punto de partida de la etapa de regeneración del código.

En este trabajo nos centramos en las dos primeras etapas, describimos cómo recuperar modelos que representan una vista abstracta del sistema existente a partir del código en el contexto de ADM. Describimos un framework para ingeniería inversa dirigida por la arquitectura con el objetivo de recuperar modelos a partir del código orientado a objetos (Figura 1). En el contexto de MDD el proceso de ingeniería inversa extrae elementos a partir del sistema existente y los representa como modelos específicos a la plataforma (PSM) a partir de los cuales se extraen modelos independientes de la plataforma (PIM). En el contexto de ADM, KDM es utilizado como soporte para representar los PSM utilizando árboles de sintaxis abstracta como una representación intermedia del sistema de software.

El nivel de modelos incluye modelos de código (Implementation Specific Model - ISM), modelos KDM (PSM) y modelos UML (PIM). Estos últimos proveen una representación uniforme del sistema en el contexto de ADM e incluyen diagramas de clases, de casos de uso, de actividad, de interacción y de estado.

El nivel de metamodelos incluye metamodelos definidos a través de MOF, siendo éste la base para la definición de las transformaciones a nivel de modelos. El nivel de metamodelos incluye a los siguientes estándares: ASTM que describe los modelos AST, KDM que describe familias de modelos PSM y UML que describe familias de PIM. En este nivel cada transformación se especifica en términos de los metamodelos origen y destino. De esta manera los modelos a nivel PIM se recuperan por la aplicación sucesiva de transformaciones de modelos a partir del código fuente.

El proceso de ingeniería inversa propuesto a nivel de metamodelo consiste de dos pasos fundamentales:

1. Descubrimiento del modelo: el modelo del código se obtiene aplicando transformaciones texto-a-modelo (T2M) a partir del código fuente; el modelo obtenido, de bajo nivel de abstracción, es un árbol de sintaxis abstracta que conforma a ASTM.

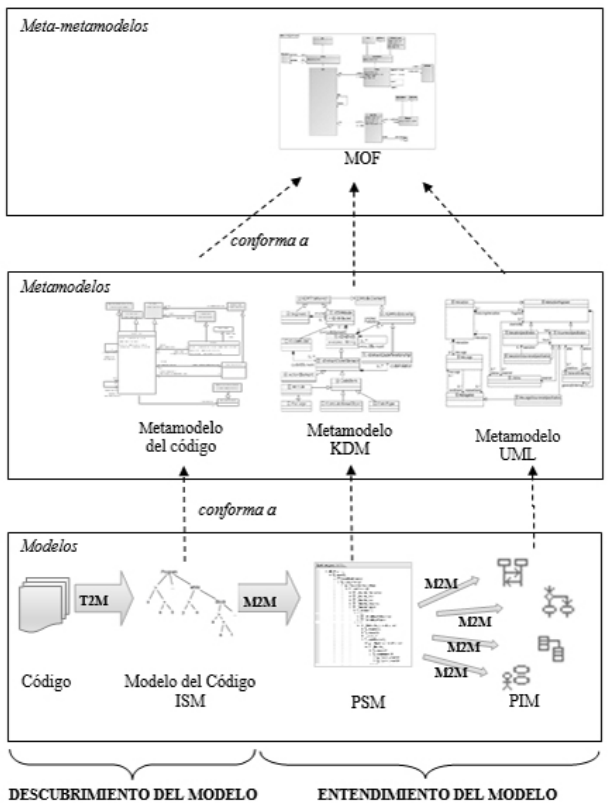


Figura 1 - Framework para ingeniería inversa.

2. Entendimiento del modelo: el objetivo es elevar el nivel de abstracción para obtener vistas de alto nivel de abstracción del sistema heredado que faciliten su análisis, entendimiento y futura regeneración. En esta etapa se obtienen modelos UML a través de transformaciones modelo-a-modelo (M2M) aplicadas a partir de los modelos generados en la primera etapa. Este paso involucra dos transformaciones sucesivas:

2.1. Transformación M2M para descubrir modelos KDM a partir del modelo del código.

2.2. Transformación M2M para descubrir modelos UML a partir de modelos KDM.

Los modelos obtenidos en esta cadena de transformaciones son almacenados en el formato de intercambio XMI, el cual combina XML, MOF y UML permitiendo integrar herramientas, repositorios y aplicaciones en ambientes heterogéneos distribuidos (XMI, 2014).

En este trabajo se muestra cómo recuperar modelos PIM a partir de código orientado a objetos a nivel de transformaciones de metamodelos. Se analiza un proceso para recuperar diagramas de casos de uso y de interacción a partir de código Java y se lo especifica a través de transformaciones basadas en metamodelos. Para la implementación de las transformaciones modelo-a-modelo se optó por utilizar ATL debido a su grado de madurez. ATL es un lenguaje de transformación de modelos híbrido (permite escribir transformaciones declarativas y operacionales) creado por el grupo AtlanMod. Desarrollado sobre la plataforma Eclipse, el ambiente de desarrollo integrado ATL provee un conjunto de herramientas que facilitan el diseño de las transformaciones.

CASO DE ESTUDIO: ESPECIFICANDO INGENIERÍA INVERSA DE CÓDIGO JAVA A DIAGRAMAS UML

Los diagramas UML no sólo son importantes para modelar el sistema durante la ingeniería forward sino también para entender el sistema, tanto su estructura como su conducta durante la etapa de ingeniería inversa. En esta sección ejemplificamos el proceso de ingeniería inversa a nivel de metamodelos para recuperar diagramas de secuencia y diagramas de casos de uso a partir del código. Se utilizó el programa Java eLib, el mismo caso de estudio empleado por Tonella y Potrich (Tonella y Potrich, 2005). El programa eLib soporta las principales funciones de una Biblioteca. El código Java (Figura 2) supone una colección de documentos de diferentes categorías (libros, revistas y reportes técnicos). Cada documento tiene un identificador. Los usuarios de la biblioteca pueden solicitar libros a préstamo. Mientras los libros pueden ser solicitados a préstamo por cualquier usuario, las revistas sólo pueden ser prestadas a usuarios internos y los reportes pueden ser consultados pero no prestados.

Primer Etapa: Descubriendo el modelo

El primer paso del proceso de ingeniería inversa consiste en descubrir modelos a partir del código fuente del sistema existente. El código del programa eLib está escrito en el lenguaje Java, por lo tanto se utilizó el JavaAST Discoverer provisto por MoDisco para obtener el modelo AST correspondiente. Este Discoverer crea modelos Java a partir del código fuente contenido en el proyecto Java. Esta transformación se utilizó tal como la provee MoDisco sin ninguna modificación.

```
class Library {
  Map documents... Map users...Collection loans...
  public boolean addUser(User user) {...}
  private void addLoan(Loan loan) {...}
  public boolean borrowDocument(
                User user,Document doc)
  {if (user.numberOfLoans()<MAX_NUMBER_OF_LOANS&&
     doc.isAvailable()&&doc.authorizedLoan(user)){
      Loan loan = new Loan(user, doc);
      addLoan(loan);
      return true;}
  return false;
}
...}
class Document {
  int documentCode; Loan loan = null;...
  public boolean isAvailable(){return loan==null;}
 public boolean authorizedLoan(User u){
  return true;
}
...}
class Book extends Document {...}
class Journal extends Document {    ...
 public boolean authorizedLoan(User user) {
  return user.authorizedUser();
}
...}
class TechnicalReport extends Document{...
 public boolean authorizedLoan(User user){...}    ...}
class User {  int userCode; Collection loans ...
  public boolean authorizedUser(){return false;}
  public void addLoan(Loan loan){loans.add(loan);} ...}
class InternalUser extends User {   ...
  public boolean authorizedUser() {return true;}  ...}
class Loan {  User user; Document document;
 public Loan(User usr, Document doc) {...}
...}
```

Figura 2 - Programa eLib.

SEGUNDA ETAPA: ENTENDIENDO EL MODELO

El segundo paso del proceso de ingeniería inversa consiste en obtener modelos de alto nivel de abstracción. Para lograr este objetivo el modelo del código obtenido en la etapa previa es transformado en modelos PIM expresados en UML mediante la aplicación de transformaciones sucesivas que se describen a continuación.

RECUPERANDO EL MODELO KDM

La transformación aplicada en esta etapa toma como entrada el modelo del código y devuelve un modelo instancia del metamodelo KDM. En la Figura 3 se muestra parcialmente este metamodelo.

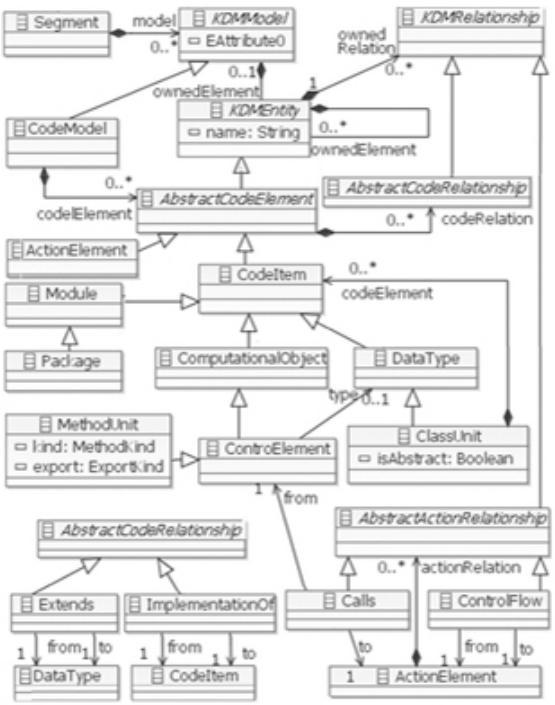


Figura 3 - Metamodelo KDM.

Las metaclases principales del metamodelo KDM son Segment, KDMModel, KDMEntity y KDMRelationship. Un segmento, instancia de Segment, contiene toda la información significativa sobre el sistema de software existente. Un segmento puede incluir instancias de KDMModel que representan vistas arquitecturales del sistema. Una instancia KDMModel posee entidades (instancias de KDMEntity) que representan artefactos del sistema de software tales como paquetes, clases y métodos. Una KDMEntity posee entidades y relaciones propias. Una relación, instancia de KDMRelationship, es una abstracción que especifica relaciones entre entidades. Cada instancia de KDMRelationship, tales como Calls Extends e ImplementationOf, tiene exactamente un origen y un destino.

Para obtener el modelo KDM a partir del modelo del código generado en la primera etapa, se utilizó el Discoverer KDM provisto por MoDisco, pero con modificaciones. Este Discoverer está implementado como una transformación modelo-a-modelo en ATL, crea modelos que conforman al metamodelo KDM a partir del modelo del código Java. Esta transformación, tal cual la provee MoDisco, no está completamente especificada; existen elementos en el modelo del código Java que no son completamente transformados a sus respectivos elementos en el modelo KDM, lo que resulta en una pérdida de información. Por ejemplo:

- En una expresión de invocación a método (Calls) los argumentos del método que son variables simples, tales como variables locales o parámetros, no están presentes en el modelo KDM.

- En una expresión de invocación a método, si el objeto sobre el cual se invoca el método es una variable simple, ésta no está presente en el correspondiente modelo KDM y por lo tanto falta la relación entre esta variable y la invocación al método.

- En expresiones infijas los operandos que son variables simples no están presentes en el modelo KDM.

Para obtener toda la información necesaria que permitiera recuperar los diagramas UML (casos de uso e interacciones) a partir del modelo KDM obtenido en esta etapa fue necesario enriquecer el

Discoverer KDM provisto por MoDisco. Se anexaron nuevos helpers y se modificaron algunas de sus reglas. La nueva transformación se aplicó al modelo del código del programa eLib y se obtuvo como resultado el modelo KDM que se muestra parcialmente en la Figura 4.

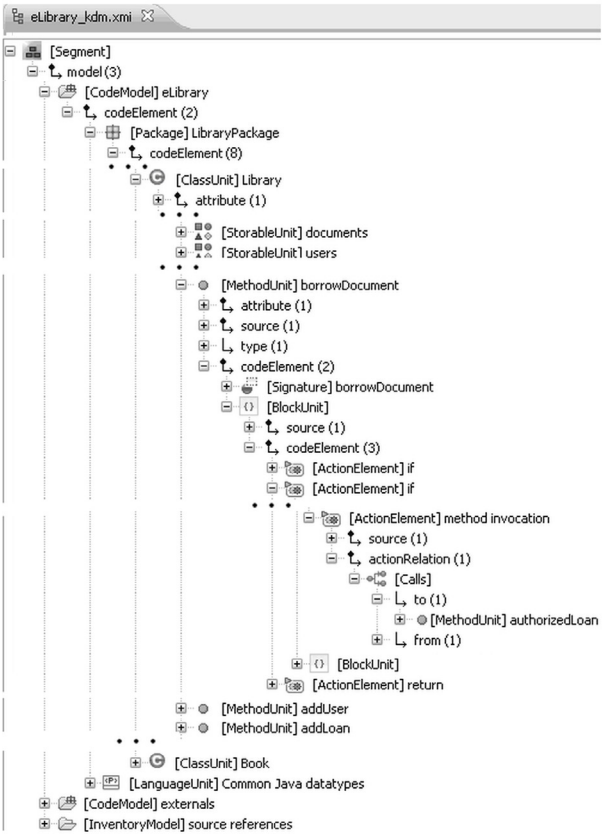


Figura 4 - Modelo KDM del programa eLib.

El modelo KDM obtenido consiste de un Segment que posee tres modelos, cada uno representando una vista arquitectural del sistema. El modelo eLibrary posee una instancia de Package llamada LibraryPackage. Éste contiene ocho instancias de UnitClass que representan a cada una de las clases definidas por el usuario en el programa eLib tales como Library y Book. A modo de ejemplo la ClassUnit Library posee instancias de StorableUnits que representan a las variables miembros de la clase (documents y users) y MethodUnits que representan a los métodos miembros (por ejemplo borrowDocument, addUser y addLoan). En la figura se puede observar el método borrowDocument, el cual posee una signatura (instancia de Signature) que representa la signatura del método y un bloque (instancia de BlockUnit) que contiene un conjunto de ActionElements (unidades básicas de comportamiento) relacionados tanto lógica como físicamente, por ejemplo, sentencias if, invocaciones a métodos (calls) y sentencias return.

RECUPERANDO MODELOS UML

El modelo KDM del programa eLib obtenido en la etapa previa es el punto de partida para recuperar los modelos PIM. Para esto, implementamos transformaciones modelo-a-modelo en ATL que generan modelos UML a partir de un modelo KDM. El metamodelo origen de cada transformación corresponde a KDM (Figura 3) y el metamodelo destino corresponde al metamodelo UML (UML, 2011). Este último se muestra parcialmente en la Figura 5 en particular la parte del metamodelo UML que especifica los diagramas de interacción y los diagramas de casos de uso.

La Figura 5.a muestra el diagrama de interacciones: una Interaction posee líneas de vida (lifeline), mensajes y fragmentos. Una lifeline representa un participante individual en la interacción. Un mensaje define la comunicación entre participantes. Un fragmento puede ser una Interaction, una ExecutionSpecification u OccurrenceSpecification.

Una ExecutionSpecification especifica la ejecución de una unidad de comportamiento o acción en la línea de vida. La duración de una ExecutionSpecification está representada por dos ExecutionOccurrenceSpecifications. Las Occurrence-Specifications, ordenadas a lo largo de una línea de vida, son unidades semánticas básicas de las interacciones. Las secuencias de ocurrencias especificadas por

ellas dan el significado de las interacciones. Un General-Ordering representa una relación binaria entre dos especificaciones de ocurrencias la cual describe que una especificación de ocurrencia debe ocurrir antes que la otra en una secuencia válida.

La Figura 5.b muestra parcialmente el diagrama de casos de uso. Un UseCase es un clasificador con comportamiento que especifica algún comportamiento que un subject puede realizar en colaboración con uno o más actores. Un caso de uso puede estar relacionado con otros a través de relaciones de generalización, extensión o inclusión.

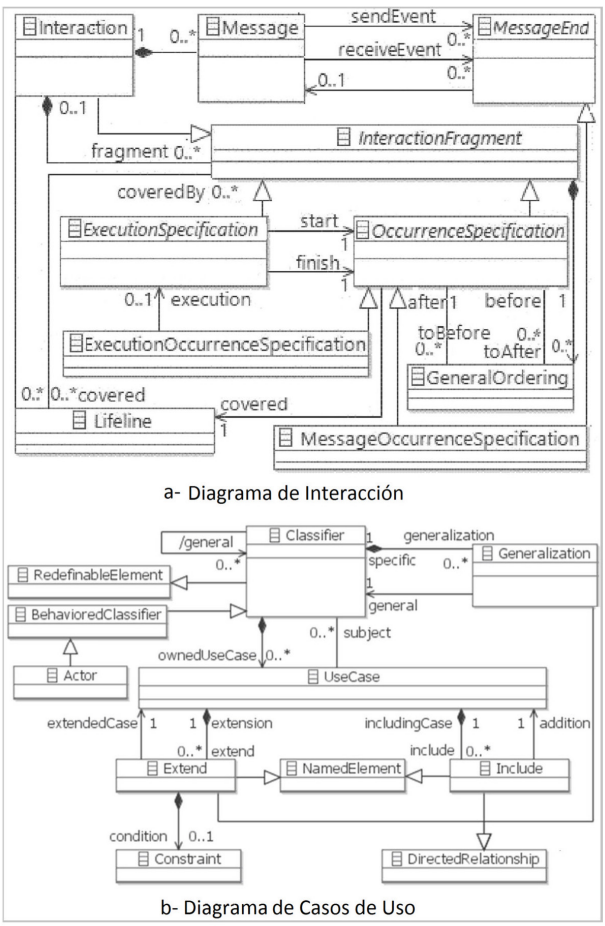


Figura 5 - Metamodelo UML.

La Figura 6 muestra parcialmente las transformaciones KDM2interaction y KDM2UseCases que especifican cómo obtener a partir del modelo KDM diagramas de interacción y diagramas de casos de uso respectivamente. A continuación se describen las transformaciones a través de sus reglas más importantes.

La transformación KDM2interaction (Figura 6.a) especifica la forma de producir diagramas de interacción a partir de modelos KDM:

- La regla method2Interaction transforma cada método público relevante (método complejo con un número significativo de llamadas a métodos) en una interacción. Su nombre se forma con el nombre del método precedido por el nombre de la clase donde el método fue declarado para facilitar su identificación. La primera lifeline de la interacción representa al objeto que es instancia de la clase donde el método fue declarado. El resto de las lifelines se obtienen de las variables locales y de los parámetros del método sobre los cuales un método es invocado. Los mensajes de la interacción se obtienen de las llamadas a métodos. El orden parcial entre los mensajes se establece a través de generalOrderings creados a partir de reglas lazy.

- La regla methodInvocation2Message transforma cada llamada a método, instancia de Calls, en un mensaje. Si el tipo del método invocado es 'constructor', el nombre del mensaje estará formado por el nombre del método precedido por la palabra 'create'. En el resto de los casos el nombre del mensaje se forma con el nombre del método invocado precedido por el nombre de la clase donde el método fue declarado. Esta regla establece tanto el emisor como el receptor del mensaje (sendEvent y receiveEvent).

- La regla CreateGeneralOrdering crea una instancia de GeneralOrdering a partir de una llamada a un método, instancia de Calls. Su nombre se forma por el nombre de los mensajes vinculados separados por '->' indicando el orden entre los mismos. Esta regla establece instancias de MessageOccurrence-Specification, before y after que permiten especificar el orden entre los mensajes.

- La regla createLifeline crea una lifeline a partir de la clase donde se declara el método que dio lugar a la interacción.

- La regla object2Lifeline crea una lifeline a partir de una variable.

- La regla createActionExecutionSpecification crea un foco de control, instancia de Action-Execution-Specification, a partir de un método y un nombre de lifeline. El comienzo y la finalización (start y finish) del foco de control, instancias de Execution-OccurrenceSpecification se crean a partir de reglas lazy.

La Figura 6.b muestra parcialmente la transformación KDM2UseCases que especifica la forma de producir diagramas de casos de uso a partir del modelo KDM:

- La regla KDMPackage2UMLPackage transforma cada paquete KDM en un paquete UML cuyo nombre es igual al nombre del paquete KDM y que contendrá los casos de uso (elementos empaquetados) que son creados a partir de métodos en las reglas correspondientes.

- La regla MethodUnit2UseCase1 transforma cada método público que no redefine métodos heredados en un caso de uso básico que sólo puede tener relaciones de dependencia. Para facilitar su identificación, el nombre del caso de uso se forma con el nombre del método precedido del nombre de la clase que lo contiene.

- La regla MethodUnit2UseCase2 transforma cada método público que redefine métodos heredados en un caso de uso básico que tiene relaciones de generalización y puede tener relaciones de dependencias. El nombre del caso de uso se forma con el nombre del método precedido del nombre de la clase que lo contiene.

- La regla Calls2Dependency transforma cada relación Calls que vincula dos métodos públicos en una dependencia entre los casos de uso creados a partir de esos métodos.

Las Figuras 7 y 8 muestran parcialmente los modelos resultantes de las transformaciones cuando son aplicadas al modelo KDM correspondiente al programa eLib.

La Figura 7 muestra en particular el modelo de interacción que representa el intercambio de mensajes entre los objetos desencadenado por la ejecución del método borrowDocument de la clase Library. En la figura se observa que la interacción está compuesta de lifelines, fragmentos, mensajes y el orden entre los mismos. La Figura 7.a muestra el modelo de interacción en formato XMI mientras que la Figura 7.b muestra el diagrama de secuencia UML correspondiente.

La Figura 8.a muestra el modelo de casos de uso generado a partir del programa eLib en formato XMI mientras que la Figura 8.b muestra el diagrama UML correspondiente. Se visualiza en particular el caso de uso Library_borrowDocument junto con los casos de uso que éste incluye. Si bien corresponde a una vista funcional de bajo nivel se pueden generar vistas de alto nivel mediante transformaciones basadas en heurísticas simples que agrupan un conjunto de casos de uso en un único caso de uso abstracto.

CONCLUSIONES Y FUTUROS TRABAJOS

La ingeniería inversa es una etapa crucial dentro del proceso de modernización. Su rol es analizar el sistema de software existente para entender y extraer modelos de alto nivel de abstracción. En este artículo presentamos un framework para ingeniería inversa en el contexto de ADM que distingue niveles de abstracción vinculados a modelos y metamodelos centrándonos en este último nivel.

Proponemos una técnica de metamodelado basada en metamodelos MOF y transformaciones. Para conformar a ADM utilizamos estándares tales como ASTM y KDM ambos definidos vía MOF. El lenguaje seleccionado para implementar las transformaciones es ATL debido a su grado de madurez. La propuesta se validó usando Eclipse Modeling

```
module KDM2interaction;
create OUT : MM1 from IN : MM;
rule method2Interaction {
  from m:MM!MethodUnit (m.is_relevantPublicMethod())
  to interact:MM1!Interaction (
     name      <- m.get_ContainerClass().name + '::' + m.name
    ,lifeline <- thisModule.createLifeline(m)
    ,lifeline <- m.get_Variables()->collect(v|thisModule.object2Lifeline(v))
    ,message  <- m.get_calls()
    ,generalOrdering <- m.get_pairsOfCalls()->collect ( p | thisModule.CreateGeneralOrdering(p->at(1)))
    ...             )
}
rule methodInvocation2message {
  from call:MM!Calls (call.is_relevantCall())
  to msj:MM1!Message (
    name <- if (call.to.kind= #constructor)then call.get_ContainerClass().name +'::'+'create_' + call.to.name
                                          else call.get_ContainerClass().name + '::' + call.to.name endif
    ,argument <-call.get_Arguments()->collect(name|thisModule.createArgument(name))
    ,sendEvent<- send
    ,receiveEvent <- receive
    )
    ,send: MM1!MessageOccurrenceSpecification (...)
    ,receive: MM1!MessageOccurrenceSpecification(...)
}
unique lazy rule createLifeline {
  from m:MM!MethodUnit
  to obj:MM1!Lifeline (
    name <- m.get_ContainerClass().name.toLower()+': ' +m.get_ContainerClass().name
    ,coveredBy <- thisModule.createActionExecutionSpecification (m,m.get_ContainerClass().name.toLower()))
}
unique lazy rule object2Lifeline {
  from d:MM!DataElement to obj:MM1!Lifeline (     name <- d.name  + ': ' + d.type.name  )
}
unique lazy rule CreateGeneralOrdering {
  from call:M1!Calls   to    genOrdering: MM1!GeneralOrdering(
    name <- call.to.name + '->' + call.get_ContainerMethod().nextTo(call).to.name
    ,before <- ...         , after<- ...
}
unique lazy rule createActionExecutionSpecification {
  from m:MM!MethodUnit ,name:MM!StringType  to exeSpec:MM1!ActionExecutionSpecification (
    name <-   'controlFocus_of_'  + name
    ,start <- ...         , finish <- ...
    ...)
}
```

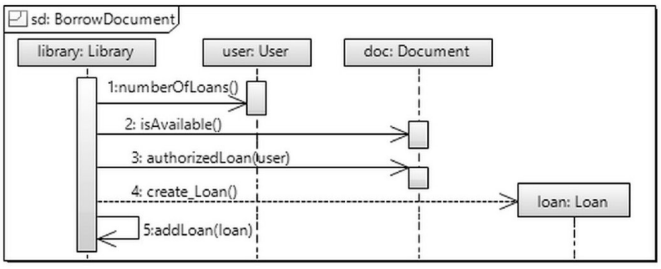
a- Transformación *KDM2interaction*

```
module KDM2useCases;
create OUT : MM1 from IN : MM;
rule KDMPackage2UMLPackage{
from p:MM!Package
to UMLpackage:MM1!Package (
  name <-  p.name ,
  packagedElement <- p.get_MethodUnit()   )
}
rule MethodUnit2UseCase1{
  from method:MM!MethodUnit
     (method.kind = #method and method.export = #public and method.codeRelation.oclIsUndefined() )
  to useCase:MM1!UseCase (
    name <-  method.refImmediateComposite().name+ '_' + method.name        )
}
rule MethodUnit2UseCase2{
  from method:MM!MethodUnit
    (method.kind = #method and method.export = #public and  not method.codeRelation.oclIsUndefined())
  to useCase:MM1!UseCase (
    name <-  method.refImmediateComposite().name + '_' + method.name,
    general <- method.codeRelation ->collect(relation | relation.to )      )
}
rule Calls2Dependency{
  from call:MM!Calls ( call.to.is_publicMethod() and call.in_publicMethod() )
  to use_case_dependency :MM1!Dependency (
    name <- 'from_' + call.get_container().name + '_to_' + call.to.name,
    supplier <- call.to  ,
    client <- call.get_container()        )
}
```

b-Transformación *KDM2useCases*

Figura 6 - Transformaciones KDM a UML.

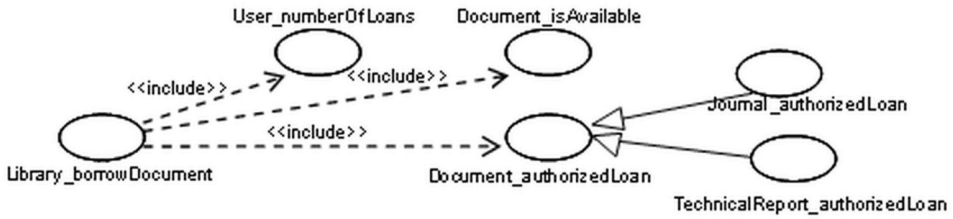
a- SequenceDiagrams.xmi



b- Diagrama UML

Figura 7 - Modelo de Interacción.

a- UseCases.xmi



b- Diagrama UML

Figura 8 - Modelo de casos de uso.

Framework y la plataforma MoDisco. Si bien en este trabajo ejemplificamos nuestra propuesta a través de un caso de estudio de ingeniería inversa a partir de código Java, cabe mencionar que se puede aplicar a cualquier lenguaje orientado a objetos.

Consideramos que nuestra propuesta provee beneficios con respecto a los procesos basados solamente en técnicas de ingeniería inversa tradicionales. La ingeniería inversa dirigida por modelos permite obtener modelos que representan el sistema heredado donde cada modelo se centra en un aspecto diferente del sistema en distintos niveles de abstracción bajo un framework común y bajo los mismos estándares. Esto permite un mejor entendimiento del sistema y facilita la interoperabilidad entre las herramientas y servicios de los usuarios de los estándares alcanzando interfaces y formatos bien definidos para el intercambio de información sobre modelos de software usados por las herramientas de modernización de software.

Los resultados obtenidos pueden ser considerados como una contribución a la comunidad MoDisco. Por un lado proveemos soporte para recuperar diagramas de casos de uso y de interacción complementando el soporte para recuperar diagramas de clases provisto por MoDisco. Por otra parte extendimos el Discoverer KDM de MoDisco para obtener la información necesaria para recuperar los diagramas de interacción.

Actualmente se está trabajando, dentro del framework propuesto, en la recuperación de otros diagramas UML a partir de código Java y su integración con el EMF. Por otra parte el proceso propuesto se está complementando con técnicas de análisis dinámico que permitirán refinar los modelos UML resultantes.

Se prevé especificar el metamodelo para el lenguaje de programación C++ y definir discoverers para recuperar modelos del código C++ (instancias de ASTM) y modelos KDM que nos permitan recuperar modelos UML a partir de sistemas heredados escritos en C++.

REFERENCIAS

Favre, "Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution". (E. S. Reference, Ed.) IGI Global. doi:10.4018/978-1-61520-649-0, (2010).

ADM, "Architecture-Driven Modernization Task Force". http://www.omgwiki.org/admtf/doku.php, (2015).

MDA, "The Model-Driven Architecture". http://www.omg.org/mda/, (2015).

MOF, "Meta Object Facility (MOF) Core Specification", Version 2.4.2, OMG Document Number: formal/2014-04-03. http://www.omg.org/spec/MOF/2.4.2 (2014).

Eclipse, "Eclipse Modeling Framework". Recuperado de http://www.eclipse.org/modeling/emf/, (2015).

ATL, "Atlas Transformation Language Documentation". http://www.eclipse.org/atl/documentation/, (2015).

MoDisco, "Model Discovery". http://www.eclipse.org/ MoDisco, (2015).

KDM, "Knowledge Discovery Meta-Model (KDM)" -OMG Document Number: formal/2011-08-04. http://www.omg.org/spec/KDM/1.3, (2014).

ASTM, "Abstract Syntax Tree Metamodel", version 1.0, OMG Document Number: formal/2011-01-05. http://www.omg.org/spec/ASTM, (2011).

Bruneliere, Cabot, Dupé y Madiot, "MoDisco: a Model Driven Reverse Engineering Framework". Information and Software Technology, 56(8), 1012-1032. doi:10.1016/j.infsof.2014.04.007, (2014).

Tonella y Potrich, "Reverse Engineering of bject-Oriented Code" (Monographs in Computer Science). Springer-Verlag, (2005).

Canfora y Di Penta, "New Frontiers of reverse Engineering". En I. Press (Ed.), Future of Software Engineering (FOSE '07), 326-341, (2007).

Fleurey, Baudry, Nicolas y Jézéquel, "Model-Driven Engineering for Software Migration in a Large Industrial Context". MoDELS 2007- Lecture Notes in Computer Science, 4735, 482-497. doi:10.1007/978-3-540-75209-7_33, (2007).

Cánovas Izquierdo y García Molina, "A domain specific language for extracting models in software modernization". (Springer-Verlag, Ed.) Model Driven Architecture - Foundations and Applications. Lecture Notes in Computer Science, 5562, 82-97, (2009).

Cánovas Izquierdo y García Molina, "Extracción de modelos en una modernización basada en ADM". Actas de los Talleres de las Jornadas de Ingeniería de Software y BBDD, 3, 41-50. http://www.sistedes.es/ficheros/actas-talleres-JISBD/Vol-3/No-2/DSDM09.pdf, (2009).

Favre, Pereira y Martínez, "Foundations for MDA CASE Tools". En M. Khosrow-Pour (Ed.), Encyclopedia of Information Science and Technology, Second Edition, 159 – 166. Hershey, PA: IGI Global. doi:10.4018/978-1-60566-026-4, (2009).

Barbier, Deltombe, Parisy y Youbi, "Model Driven Engineering: Increasing Legacy Technology Independence". Second India Workshop on Reverse Engineering (IWRE, 2011) in The 4th India Software Engineering Conference (pp. 5-10). Thiruvananantpuram, India: CSI ed., (2011).

Ulrich y Newcomb, "Information System Transformation. Architecture Driven Modernization Case Studies". MK/OMG Press, (2010).

Brambilla, Cabot y Wimmer, "Model-Driven Software Engineering in Practice". USA: Morgan & Claypool, (2012).

XMI, "XML Metadata Interchange (XMI) Specification". OMG Document Number: formal/2014-04-04. http://www.omg.org/spec/XMI/2.4.2/PDF/, (2014).

UML, "Unified Modeling Language: Superstructure". Version 2.4.1, OMG Specification: formal/2011-08-06. http://www.omg.org/spec/UML/2.4.1/, (2011).

Fecha de Presentación: 15/05/15
Fecha de Aprobación: 08/06/15