Lenguaje Específico de Dominio para Aplicaciones de Negocios

Claudia Naveda [1], Alberto Cortez[1,2], Germán Montejano[3], Daniel Riesco[3]

[1]Universidad del Aconcagua, [2] Universidad de Mendoza, [3] Universidad Nacional de San Luis.

claudia_naveda@hotmail.com, cortezalberto@gmail.com,

gmonte@unsl.edu.ar, driesco@unsl.edu.ar

Abstract. El desarrollo de software dirigido por modelos surge como respuesta a los principales problemas existentes en las compañías de desarrollo de software. Por un lado, se requiere gestionar la creciente complejidad de los sistemas que se construyen y mantienen, y por otro lado adaptarse a la rápida evolución de las tecnologías de software. En un proceso tradicional los costos de tener actualizado un modelo de negocios, diseñado en lenguaje UML con una herramienta CASE son altos. Esta dificultad causa que los modelos del sistema, a partir de la etapa de mantenimiento, no coincidan con los códigos fuentes. Se viola de este modo el concepto de ingeniería directa. *Los modelos no coinciden con el código fuente.* En virtud de esto surgen las propuestas de los nuevos paradigmas de ingeniería de software enfocados en el modelo como protagonista del proceso de desarrollo. El Modelado específico del dominio DSM (por sus siglas en inglés: Domain-Specific Modelling) produce una solución con mayor abstracción, que expresa las soluciones directamente en términos del dominio. En este trabajo se propone una solución DSL para el modelado de negocios. Se crea un lenguaje específico de dominio que contiene un metamodelo y un editor gráfico. Desde el editor gráfico se pueden producir modelos de negocio. A partir de este modelo se genera código fuente en uno o varios lenguajes de programación. De esta forma las aplicaciones finales se pueden producir a partir de un modelo.

1 Introducción

El Modelado Específico de Dominio (DSM) le otorga independencia de la tecnología al proceso de desarrollo de software. Los programas se especifican a través de los conceptos de un dominio. Los productos finales pueden luego generarse a partir de dichas especificaciones de alto nivel [Wegeler 2013]. Los lenguajes específicos del dominio DSL (por sus siglas en inglés: Domain-Specific Language), proveen a los desarrolladores y usuarios una sintaxis común al área en que trabajan. Los DSL al crearse en colaboración con especialistas de un determinado dominio producen mayor precisión y calidad en sus resultados. En un desarrollo DSM, los DSL están integrados con generadores de código que transforman los modelos en código fuente [Kelly 2008]. Estos generadores se ejecutan dentro de un framework específico del dominio. El principal componente en DSM son los modelos. Estos modelos pueden ser gráficos y textuales, pero sirven al mismo propósito: crear una abstracción del sistema que puede ser fácilmente estandarizado y

fácilmente comunicado a los usuarios. Los diagramas que se utilizan para describir un modelo, no tienen significado para la computadora. Para abordar esta cuestión, se necesita un motor de transformación o generador que pueda ayudarnos a transformar este modelo en un estándar que el ordenador pueda entender. Una solución a este problema es la transformación de modelo a texto; es decir, generación de código en un lenguaje de programación, que a su vez se interpreta o compila y se ejecuta. Esta tesis se centrará en facilitar transformaciones de modelo a texto.

La investigación formula como objetivo construir un DSL para la generación de aplicaciones de negocio. El DSL propuesto contiene un metamodelo con los conceptos necesarios para modelar el diagrama de clases de un sistema. Y además se verifica su buena formación mediante reglas descriptas en OCL. Los modelos se generan a partir de un editor gráfico construido en base al metamodelo. A partir de ellos se puede generar el código fuente automático en diferentes lenguajes. La generación de los lenguajes se produce a partir de plantillas realizadas en XPAND. Dichas plantillas conforman un repositorio abierto a nuevas plantillas en nacientes tecnologías de programación.

La organización del trabajo es la siguiente: en la Sección 2 se presentan los trabajos relacionados. En la Sección 3 se explica la arquitectura de solución DSM donde se da una visión general de DSM, se explican sus componentes, que es el metamodelado y el lenguaje OCL. Además se definen los conceptos de transformación de modelos y plantillas. En la sección 4 se describe cómo se implementó el DSL: análisis, desarrollo y metodología de construcción, En la sección 5 se explica cómo se crea una transformación con el lenguaje XPAND. En la sección 6 se describe su aplicación a un caso de estudio. En la Sección 7 se exponen las conclusiones y líneas de trabajo futuro.

2 Trabajos relacionados

Algunas investigaciones [Lolong 2011], [Freudenthal 2009] han demostrado que un DSM permite crear aplicaciones robustas. Por ejemplo, un desarrollo en Java con acceso a bases de datos MYSQL mediante la utilización de DSL. Es decir, se genera aplicaciones desde el DSL fuente. Teniendo en cuenta esta demostración se evalúa en este trabajo la generación de código en Java como en otros lenguajes. En [Sandven 2012] se plantea un trabajo que tiene lineamientos en común con este trabajo como por ejemplo la generación de código a través de XPAND. En ambos se trata de facilitar la transformación de un modelo a texto, planteando como punto de partida un metamodelo Ecore. El ejemplo en [Sandven 2012] muestra como a partir de la metodología ágil XP se puede construir un metamodelo y generar código. Pero en este trabajo se genera un editor gráfico que puede ser usado por el usuario, y así crear un modelo de negocios a través de este editor.

3 Arquitectura de Solución DSM

En el presente trabajo se ha optado por una forma típica de creación de entornos DSM, donde se parte del desarrollo de modelos a partir de conceptos aplicados por expertos del dominio. Se utilizan los mismos conceptos como input para producir generadores de código según los principios fundamentales de MDE (por sus siglas en inglés, Model-Driven Engineering) [GMF 2012]. Para el desarrollo de una solución DSM en un con-

texto MDE, se utiliza una arquitectura de 3 niveles sobre la plataforma de ejecución: un lenguaje específico de dominio, un generador de código y un framework específico de dominio. Siguiendo este enfoque, la Fig. 1 muestra la arquitectura propuesta para el DSM.

3.1 Visión General de un DSM

En el marco de DSM, el DSL es la herramienta que permite abstraer la complejidad de un dominio dado, a través de la provisión de conceptos y reglas expresados directamente en el dominio del problema. Un DSL se define como "un lenguaje de programación de limitada expresividad focalizado en un dominio en particular" [Freudenthal 2009].

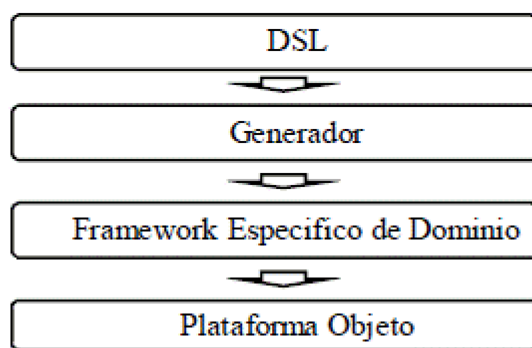

Fig. 1. Arquitectura utilizada para el entorno DSM

Los conceptos principales suelen tener su representación en la notación del lenguaje (gráfica o textual), otros conceptos se representan mediante conexiones o propiedades. En este trabajo se propone una notación gráfica; el lenguaje se formalizó con un meta-modelo que define una sintaxis abstracta. Sobre él se definió una sintaxis concreta con la cual el experto del dominio interactúa. El generador de código especifica cómo se extrae la información de los modelos y se transforma en código ejecutable que no necesita ser modificado para su funcionamiento. El Framework específico de dominio es una interface entre el código generado y la plataforma subyacente. Su objetivo es simplificar la tarea del generador, elimina la duplicación de código y provee de una capa de abstracción sobre la plataforma. Al proveer de componentes previamente testeados, se permite un aseguramiento de la calidad del producto. La plataforma comprende el conjunto de sistema operativo, máquina virtual, lenguajes, librerías y framework's de componentes.

4 Componentes de un DSL

Sintaxis abstracta

La sintaxis abstracta de un lenguaje representa la terminología de los conceptos del lenguaje, las relaciones entre ellos, y las reglas necesarias para construir las sentencias (programas, instrucciones, expresiones o modelos) válidas del lenguaje.

Sintaxis concreta

La **sintaxis concreta** de un lenguaje detalla la notación usada para representar los modelos. Hay dos tipos de sintaxis concretas: textuales y gráficas. La **sintaxis gráfica** (o **visual**) describe los modelos de forma diagramática, usando símbolos para representar sus elementos y las relaciones entre ellos. La **sintaxis textual** permite describir modelos usando sentencias compuestas por cadenas de caracteres, de una forma similar a como hacen la mayoría de los lenguajes de programación.

Semántica

En general, lo que **parece representar** un modelo no es lo mismo que lo que **realmente significa**. La semántica representa el significado.

Relaciones entre sintaxis abstracta, sintaxis concreta y semántica

Mientras que la sintaxis abstracta especifica cuáles son los modelos válidos, la sintaxis concreta permite representar dichos modelos. Por su parte, la semántica les asigna un significado preciso y no ambiguo. La forma natural de definir la sintaxis abstracta de los DSL es mediante metamodelos. La sintaxis concreta de un lenguaje se suele definir a partir de su sintaxis abstracta, mediante una relación (concrete syntax mapping) que asigna un símbolo de la sintaxis concreta a cada concepto de la sintaxis abstracta. Dicha asignación debe respetar la semántica que suelen tener los símbolos.

4.1 Metamodelado

Los metamodelos sirven para representar los conceptos de un lenguaje, las relaciones entre ellos y las reglas estructurales que restringen los posibles elementos de los modelos válidos.

Sintaxis concreta para metamodelos

La sintaxis concreta proporciona una notación que permite a los diseñadores describir los modelos. La sintaxis concreta puede definirse desde cero, o bien haciendo uso de una notación existente, extendiéndola con los conceptos de nuestro lenguaje.

4.2 OCL

OCL (por sus siglas en inglés, Object Constraint Language), es un lenguaje que utiliza la lógica de primer orden con el fin de otorgar formalidad y precisión tanto a modelos como a metamodelos. Un metamodelo se expresa con notación gráfica y con un lenguaje formal, como OCL, para aumentar su precisión y eliminar ambigüedades. Una de sus principales características es su fácil comprensión. OCL es utilizado para definir la semántica de UML, especificando reglas bien formadas sobre el metamodelo. Las reglas bien formadas son definidas como invariantes sobre las metaclases en la sintaxis abstracta.

4.3 Transformación de modelos

Las transformaciones de modelos pueden verse como programas que toman un modelo (o más de uno) como entrada y devuelven otro modelo (o más de uno) como salida. En la Fig. 2 se pueden observar los participantes involucrados en una transformación de modelos. En ella se muestra un escenario de transformación con un modelo origen y uno de destino, los cuales deben conformar a sus respectivos metamodelos.

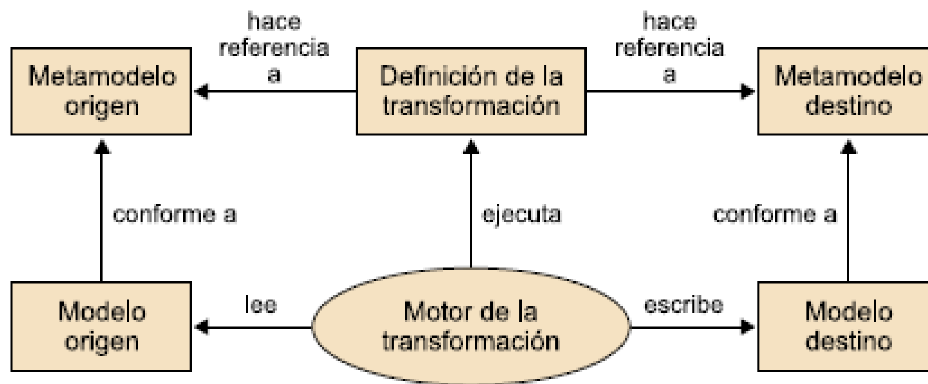

Fig. 2. Participantes de una transformación de modelos

Al definir una transformación entre modelos, se hace respecto de sus metamodelos. En MDE existen varios tipos de transformaciones. En este caso específico se utiliza una transformación vertical, de un nivel abstracto hacia código fuente. Se transforma, en forma descendente, desde un modelo gráfico a texto

4.4 Plantillas

Las transformaciones de modelo-a-texto (M2T) son aquellas que toman un modelo como entrada y devuelven una cadena de texto como salida. El enfoque utilizado en este trabajo está basado en plantillas que soportan la mayoría de las herramientas MDA disponibles. Una plantilla contiene fragmentos de metacódigo para acceder a información del modelo origen y permite la expansión iterativa del texto, produciendo como salida un modelo destino en texto. La herramienta XPAND proporciona un lenguaje independiente del metamodelo que permite usar cualquier tipo de metamodelo y sus instancias para la generación de código. Esta herramienta está basada en el framework EMF como plataforma de metamodelado.

4.5 Lenguaje XPAND

Es un lenguaje de transformación de modelo a texto en forma de plugin de Eclipse con licencia EPL. Se basa en plantillas para la generación de código donde se pueden utilizar variables, bucles y sentencias de control. Una de las características más notables de Xpand consiste en que se distribuye junto con otra herramienta denominada Xtext destinada a la creación de lenguajes textuales de dominio específico. *Xpand* tiene las siguientes características: invocación polimórfica de plantillas, extensiones funcionales, abstracción flexible, validación de modelos. Las plantillas son almacenadas en archivos con la extensión ".xpt". Los archivos de plantilla deben residir en el classpath de Java del proceso generador. Los nombres de propiedades, plantillas, namespaces etc. sólo deben contener letras y números.

Un archivo de plantillas consta de cualquier número de declaraciones de IMPORTACIÓN, seguidas de cualquier número de declaraciones de EXTENSIÓN, seguidas de uno o varios bloques DEFINE (llamados definiciones).

Principales declaraciones

IMPORT: Sirve para importar el modelo. Si la plantilla contiene tal declaración, se pueden usar los nombres cualificados de todos los tipos y archivos de plantilla contenidos en su namespace. Esto es similar a una declaración de importación en Java.

EXTENSION: Los metamodelos son descritos de un modo estructural (gráfico, o jerárquico, etc.). Las extensiones proporcionan un modo flexible y conveniente de definir los rasgos adicionales de metaclases.

DEFINE El concepto central de Xpand es el bloque *Define*. Es la unidad identificable más pequeña en un archivo de plantilla. La etiqueta consiste en un nombre, una lista de parámetros separados por comas, así como el nombre de la metaclase del metamodelo para la cual la plantilla es definida.

FILE: La declaración de *File* remite la salida generada de sus declaraciones de cuerpo al objetivo especificado. El objetivo es un archivo en el sistema de archivos cuyo nombre es especificado por la expresión (relativo al directorio objetivo especificado para ejecutar el generador).

EXPAND: La declaración *Expand* permite expandir el bloque *Define*, inserta su salida en la posición correspondiente y continua con la próxima declaración.

FOREACH Esta declaración acompaña la declaración *Expand* para evaluar una colección de elementos, permite la navegación en el metamodelo.

IF: La declaración apoya la extensión condicional. Permite cualquier número de declaraciones ELSEIF.

REM: La declaración REM sirve para hacer comentarios. Y va seguida de ENDREM.

5 Implementación de un DSM para aplicaciones de negocios

5.1 Análisis del Dominio

En este caso se planteó un refinamiento del modelo UML con un número acotado de elementos. Como resultado se obtuvo un Modelo de Dominio (base para la sintaxis abstracta), donde se representan las principales abstracciones utilizadas en las distintas etapas de estos sistemas, mediante un Diagrama de clases de Dominio UML.

5.2 Desarrollo del DSL propuesto

Eclipse Modeling Framework (EMF)

EMF (por sus siglas en inglés: Eclipse Modeling Framework), es una Estructura de Modelado. EMF que soporta la definición de metamodelos estructurales y el subsecuente uso de modelos conforme a estos metamodelos. También soporta generación de código Java para representar los metamodelos. Adicionalmente el código Java puede ser provisto agregando comportamiento a los metamodelos estructurales.

EcoreMetamodel

Ecore es el metamodelo provisto por EMF (ver Fig. 3). Un metamodelo es un modelo de un lenguaje de modelado, que sirve para describir modelos.

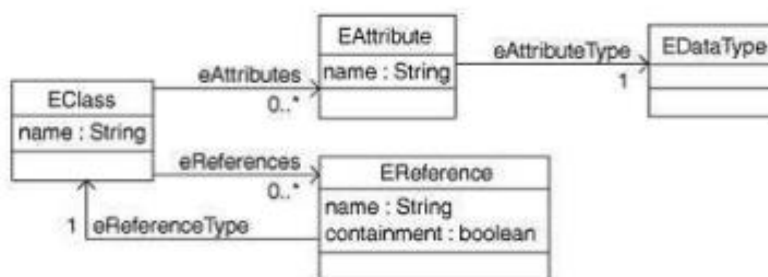

Fig. 3. Metamodelo reducido de Ecore

Formación de la sintaxis abstracta

El Metamodelo de la Fig. 4 representa un fragmento de la sintaxis abstracta del DSL. Las clases del dominio se representan a través de una Eclass de Ecore. El meta-modelo incluye los componentes necesarios (Diagrama, Clase, etc.) para la descripción gráfica del DSL y para la generación de código. Los modelos que se construyan con la herramienta DSM crearán instancias de este meta-modelo.

GMF (Graphical Modelling Framework)

GMF es un framework para describir la sintaxis grafica de un lenguaje. GMF es de código abierto y permite construir editores gráficos, también desarrollado para el entorno Eclipse. La definición tiene como base el metamodelo Ecore.

Metodología de construcción del IDE

El lenguaje visual se construyó por medio de un editor gráfico. Dicho editor contiene una paleta de herramientas y un área de trazado, que es donde se define el modelo. Para la elaboración del editor fue necesario un proceso de refinamiento tanto de los elementos de la paleta como del área gráfica. A continuación se describen los pasos seguidos:

Paso 1: Definir las figuras gráficas: los elementos y las conexiones que aceptará el DSL.

Paso 2: Definir los colores, íconos, formas de diagrama y etiquetas a mostrar, dando a cada una un significado semántico conocido por los expertos del dominio.

Paso 3: Definir la barra de herramientas y el mapeo entre las figuras gráficas y los elementos de la barra de herramientas.

Paso 4: Validaciones. Se definieron un conjunto de validaciones que se aplicarán a cada forma. Una validación puede ser aplicada sobre los elementos de la representación gráfica o sobre el modelo. Las expresiones de validación se escriben utilizando OCL, verificando contra el metamodelo del DSL.

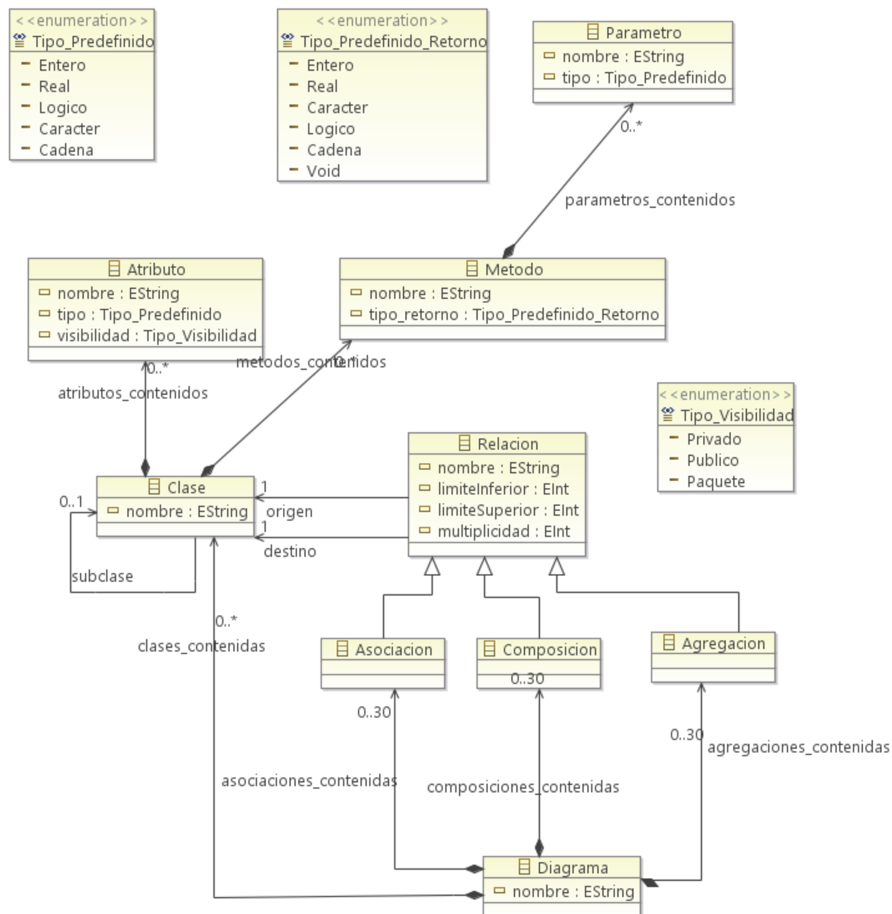

Fig. 4. Fragmento reducido del Metamodelo generado

5.3 Verificación del metamodelo

Implementación de OCL en Ecore

OCL puede ser embebido en Ecore utilizando anotaciones (ver Fig. 5). Esto significa que puedo construir restricciones OCL dentro de un metamodelo construido con EMF. Esta tarea se realiza a partir de la creación de anotaciones y la configuración de sus propiedades. En la primera anotación a crear se especificará la propiedad *Source* con la siguiente URI como sigue: Source = http://www.eclipse.org/emf/2002/Ecore.

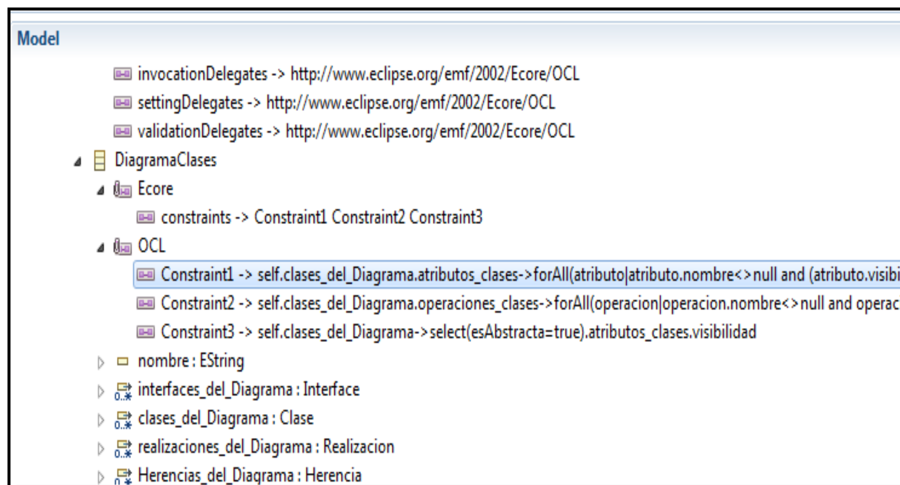

Fig. 5. Restricciones OCL embebidas en el metamodelo

El OCL embebido se activa al especificar la funcionalidad delegate. Deben ser especificadas tres funcionalidades Ecore: invocationDelegate, settingDelegate y validationDelegate. Esta especificación provee a EMF la información de configuración requerida para acceder a la funcionalidad provista por estos delegates que se describe a continuación.

Setting Delegate: La funcionalidad setting delegate permite definir expresiones OCL con valores iniciales (initial) o derivados (derived) de una característica estructural.

Invocation Delegate: La funcionalidad invocation delegate permite que expresiones OCL contengan el cuerpo de una operación. Cuando una operación es invocada, la expresión OCL es evaluada para proveer un valor.

Validation Delegate: La funcionalidad validation delegate permite a la expresión OCL describir una o más invariantes. Cuando se activa, son evaluadas las expresiones OCL.

Una vez definidas las funcionalidades delegates se puede crear las anotaciones para el ocl embebido dentro de un clasificador del metamodelo. Se crea una anotación dentro de Ecore que contenga el o los nombres de las constraints. Por cada una de las constraints se genera un detalle que contenga las propiedades: key y value. En la propiedad key debe aparecer el nombre de la constraint. En la propiedad value se describe la constraint.

A continuación, se explica cómo se formularon algunas ocl's en el metamodelo. En dichas ocl's se verifica la buena formación de los elementos generados por el metamodelo. Para realizar dichas restricciones se navega por el metamodelo. La navegación se produce a partir de una clase como punto de partida para llegar a una clase destino a través de una relación que las une: asociación, composición. Es posible navegar en el sentido de las flechas de la relación.

Aplicación de OCL al metamodelo propuesto

Aquí se muestran algunos ejemplos de ocl's formuladas. El contexto de los ejemplos propuestos, es decir, la clase origen es la clase raíz: Diagrama.

Ejemplo1. Verifica que las clases abstractas contengan el atributo de visibilidad

Como ya se dijo el contexto es el elemento raíz del diagrama Ecore: Diagrama. En ocl se lo refiere como **self**.

self.clases_contenidas->select (esAbstracta=true).atributos_clases.visibilidad

->notEmpty ()

En este ejemplo se navega desde la clase origen *Diagrama* hacia la clase *Clase* a través de la composición *clases_contenidas.* Al obtener las instancias de una clase de esta forma se puede seleccionar un conjunto de instancias que cumplen una propiedad mediante la expresión select. La expresión select en este caso toma todas las instancias de una clase. Para dicho conjunto se navega por *atributos_contenidos*, obteniéndose las instancias de atributos. Y de ellos se verifica que los de tipo visibilidad no están vacíos.

Ejemplo 2. Verifica que las operaciones contenidas en interfaces sean de tipo abstracto. El contexto es el elemento raíz del diagrama Ecore: Diagrama.

self.interfaces_del_Diagrama.operaciones_interfaces->

forAll (operacion|operación.esAbstracto=true)

Por medio de la composición *interfaces_del_Diagrama* se llega a la clase *Interface.*

Se parte desde la clase *Interface* como origen hacia la clase *Operación* a través de la relación de composición *operaciones_interfaces.* A partir de esta navegación se obtienen las instancias de la clase *Operación.* Lo que se quiere comprobar con esta ocl es que las operaciones de las interfaces sean de tipo abstracto. Se accede entonces al atributo de cada una de las instancias de la clase operación con la instrucción forall. Se declara que cada una de ellas contenga el atributo *esAbstracto* en true.

Ejemplo 3. Verifica las propiedades de los atributos de una clase. Se comprueba que el nombre de un atributo no sea nulo y su visibilidad sea protegida.

self.clases_contenidas.atributos_contenidos->

forAll (atributo|atributo.nombre<>null and

(atributo.visibilidad=Tipo_Visibilidad::protected or

atributo.visibilidad=Tipo_Visibilidad::private))

Se navega por *clases_contenidas* y se obtienen las instancias de *Clase*. Con una expresión foral se comprueba que los atributos de tipo nombre de una clase son nulos y su visibilidad es de tipo privada o protegida.

6 Transformación de modelos con Xpand

6.1 Generador de código aplicando el lenguaje Xpand

Para la generación de código se elaboró una serie de plantillas XPAND que conforman un repositorio de lenguajes. A continuación se muestran un par de fragmentos ejemplo de una plantilla realizada para el lenguaje JAVA.

Fragmento 1: Bloque principal de la plantilla

Se enumeran metamodelo, extensiones y definen componentes principales.

```
«IMPORT similuml»
«EXTENSION template::GeneratorExtensions»
«DEFINE main FOR DiagramaClases»
    «EXPAND javaClass FOREACH clases_del_Diagrama»
    «EXPAND javaInterface FOREACH interfaces_del_Diagrama»
«ENDDEFINE»
```

Fragmento 2: Bloque para definir una clase JAVA

Se define el archivo físico, paquetes, librerías y los componentes de una clase: si es abstracta, si contiene herencia o realización, sus atributos, constructores, métodos

```
«DEFINE javaClass FOR Clase»
    «FILE "modelo/" + nombre + ".java"»
        package modelo;
        import java.util.Vector;
    public «EXPAND clase_abstracta FOR this» class «nombre» «EXPAND herencias_clase FOR herenciasClases_contenidas» «EXPAND realizaciones_clase FOR realizacion_contenida» {
            «EXPAND atributos_y_relaciones_clase FOR this»
            «EXPAND constructor_clase FOR this»
            «EXPAND constructor_sobrecargado_clase FOR this»
            «EXPAND set_get_clase FOR this»
            «EXPAND metodos_clase FOR this»
        }
    «ENDFILE»
«ENDDEFINE»
```

Fragmento 3: Bloque para definir atributos de una clase JAVA

Se define un atributo simple que va a contener el tipo, el nombre y será privado para respetar el concepto de encapsulación en la programación orientada a objetos.

```
«DEFINE atributos_clase FOR Atributo»
	private «tipo» «nombre»;
«ENDDEFINE»
```

Fragmento 4: Bloque para definir nombres e inicializar atributos de una clase JAVA

Se define el nombre de un atributo simple y se inicializa el valor del atributo

```
«DEFINE nombre_atributo_constructor FOR Atributo-»
	«nombre-»
«ENDDEFINE-»

«REM» Inicialización de atributos propios «ENDREM»

«DEFINE inicializacion_atributos_constructor FOR Atributo»
	this. «nombre» = «nombre»;
«ENDDEFINE»
```

7 Aplicación del DSL a un caso de estudio

En la Fig. 6 puede observarse el editor. En el costado derecho aparece la plantilla con los elementos de la sintaxis concreta. Está compuesta por Objetos y Referencias. Los objetos son los elementos principales del dominio: Paquete, ClaseConcreta, interface, ClaseAbstracta, etc. En este caso es un dominio con clases refinadas de UML. Las referencias son las posibles relaciones entre los elementos del modelo.

En este es un ejemplo simple donde se utiliza una clase abstracta para representar la clase Persona, de la cual heredan sus propiedades las clases Alumno y Profesor. Pero además tienen sus propias operaciones.

A partir de un modelo se construye una arquitectura MVC (Modelo Vista Controlador) en las plataformas Java, NET, PHP y Python. Se concibe la vista con Bootstrap y JavaScript.

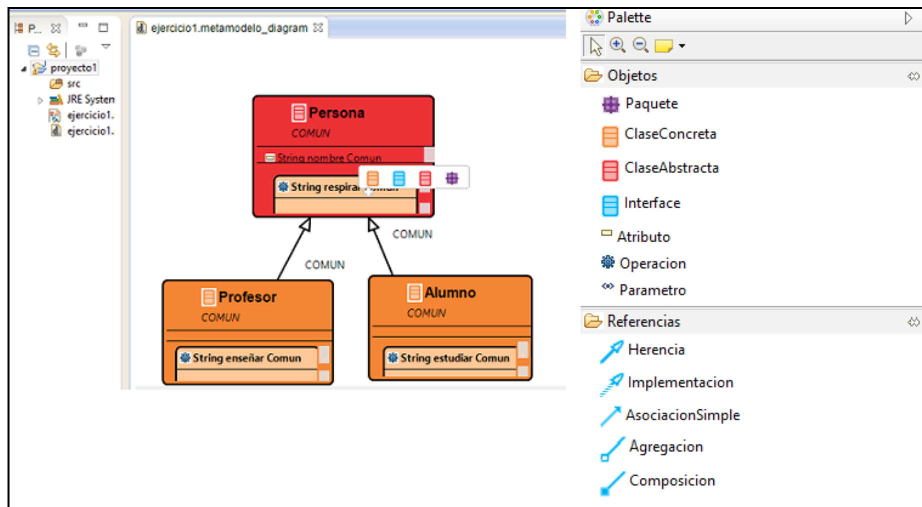

Fig. 6. Ejemplo de modelo Simple

8 Conclusiones

Este trabajo presenta un entorno DSM (por sus siglas en inglés: Domain-Specific Modeling) para el modelado de aplicaciones. Se muestra la técnica de desarrollo de un DSL (por sus siglas en inglés: Domain-Specific Language). Se detallan el diseño y las fases de implementación de un lenguaje para un dominio y como construir el generador para traducir modelos producidos por usuarios a código. Previamente los modelos son definidos mediante el DSL creado. El DSL contiene el lenguaje de dominio de los usuarios.

En el trabajo se introdujo un metamodelo (creado con el plugin Ecore de Eclipse), en el que se verifica su buena formación mediante restricciones OCL. Partiendo de un metamodelo consistente, es posible elaborar un editor gráfico, en el que los usuarios formulan fácilmente modelos de negocio. Estos modelos gráficos se transforman a código fuente de diferentes lenguajes con la ayuda de plantillas XPAND.

Este trabajo se diseña un repositorio de plantillas. Y se plantea como imprescindible el que sea posible el agregar plantillas para las nuevas tecnologías que van surgiendo a través del tiempo. De esta manera se proyecta como idea central : el logro de una herramienta abierta. Entonces este entorno puede ser tomado como modelo para generar también un repositorio de metamodelos y de editores gráficos adaptados para distintos dominios.

La idea esencial del trabajo es construir herramientas que logren la automatización de la mayor cantidad de procesos utilizando la versatilidad de las técnicas DSL, que se proyectan como la tecnología de desarrollo de software del futuro. Se pueden crear de esta manera distintos sistemas en forma más rápida y con buenas prácticas. El metamodelo, al trabajar con conceptos aplicables a varios tipos de negocios, permite concebir distintos tipos de modelos de negocio.

Se propone como trabajo futuro la instauración de procesos de reingeniería dentro del proyecto para construir modelos a partir del código fuente. Esto permitirá también reformular modelos ya construidos para mejorarlos con mejores prácticas.

Referencias

EMF Documentation. .(2008) Disponible enhttp://eclipse.org/modeling/emf/docs/. (2002 -2014)

Steinberg, D., Budinsky, F., Paternostro, M. (2008) EMF: Eclipse Modeling Framework, 2nd Edition Ed MerksEclipse Series.

Wegeler, T., Gutzeit, F., Destailleur, A., Dock, B. (2013) Evaluating the benefits of using Domain-Specific Modeling Languages – an Experience Report. Proceedings of the 2013 ACM workshop on Domain-specific modeling, ACM. New York, USA 7-12

Kelly,S. , Tolvanen, J. (2008) Domain-Specific Modeling Entablan Full Code Generation – IEEEComputer Society

Wiley, J. , Lolong,, S., Achmad, K. (2011) Domain Specific Language (DSL) Development for Desktop-based Database Application Generator. International Conference on Electrical Engineering and Informatics. Bandung, Indonesia.

Sandven, A. (2012) Metamodel based Code Generation in DPF Editor. Master's Thesis in Informatics – Program Development. University of Bergen.

Fowler, M. (2010) Domain-Specific Languages. Addison-Wesley Signature Series Fowler.

DSM Forum. (2011).SM Case Studies and Examples. Disponible en http://www.dsmforum.org/

Steinberg, D., Budinsky ,F., Paternostro, M. y Merks, E. (2011)EMF: Eclipse Modeling Framework, 2nd Edition. Addison Wesley Professional.

EMF. Eclipse Foundation. (2000-2012)Disponible en http://www.eclipse.com/emf/.

Andino, L., Ruiz, G. (2009) Análisis y uso de los frameworks de Eclipse para la definición de DSL's. Universidad Nacional de La Plata.

GMF. Eclipse Foundation. (2000-2012) Disponible en http://www.eclipse.com/gmf/.

Freudenthal, M., (2009) "Domain Specific Languages in a Customs Information System," *Software, IEEE*, vol.PP, no.99, pp.1,1, 0 152