Ingeniería inversa basada en modelos de código PL/SQL en aplicaciones Oracle Forms

Carlos J. Fernández Candel[1], Francisco J. Bermúdez Ruiz[1], Jesús García Molina[1], Jose R. Hoyos Barceló[1], Diego Sevilla Ruiz[1] y Benito J. Cuesta Viera[2]

[1] Grupo Modelum
Facultad de Informática.Universidad de Murcia
{carlosjavier.fernandez1,fjavier,jmolina,jose.hoyos, dsevilla}@um.es
[2] Open Canarias, S.L.
bcuesta@opencanarias.es

Resumen. El alto coste de mantenimiento de las aplicaciones *legacy* promueve en las empresas iniciativas de modernización a nuevas plataformas y tecnologías. La modernización de software, en especial la ingeniería inversa, es uno de los escenarios de aplicación de las técnicas de la Ingeniería del Software Dirigida por Modelos (MDE), con el fin de automatizar las tareas manuales y reducir costes. En este trabajo se presenta una solución MDE para la extracción de modelos del código PL/SQL de aplicaciones Oracle Forms. En concreto, se ha implementado un enfoque propuesto en un trabajo previo del grupo Modelum dentro de una colaboración con la empresa Open Canarias en el marco de un proyecto CDTI destinado a la automatización de aplicaciones Oracle Forms a Java. Los principales retos que se han debido afrontar han sido el uso extensivo del metamodelo KDM, la implementación de transformaciones modelo a modelo complicadas y la validación de estas transformaciones que generan modelos grandes y complejos. A lo largo del trabajo se discutirá sobre estas cuestiones.

Palabras clave: Ingeniería del Software Dirigida por Modelos, Ingeniería Inversa, Modernización, KDM, PL/SQL, Oracle Forms

1 Introducción

El coste de mantenimiento de las aplicaciones legacy es muy alto y las empresas abordan con frecuencia proyectos de modernización (normalmente migraciones a nuevas plataformas). Dado que los procesos de modernización suelen tener un coste elevado existe un alto interés en automatizar el mayor número posible de tareas involucradas. En la última década, las técnicas de la ingeniería del software dirigida por modelos (MDE) han sido utilizadas para este propósito y algunas experiencias han sido publicadas como se discute en [7]. Cabe destacar la iniciativa *ADM* (*Architecture-Driven Modernization*) [3] que lanzó OMG en 2003 y cuya finalidad es ofrecer un conjunto de metamodelos estándares para representar información comúnmente manejada en modernización. *KDM* (*Knowledge*

Discovery Metamodel) [4] es el metamodelo principal que está destinado a representar los artefactos software (código y datos) de una aplicación de forma independiente a las plataformas usadas.

Existen muchas empresas que ofrecen soluciones para soportar la migración de aplicaciones o que tienen líneas de desarrollo en esta área, pero la mayoría no usan técnicas basadas en modelos. En [5] se presenta una lista de algunas empresas que ofrecen herramientas de reingeniería basada en modelos. Open Canarias[3] (en adelante Open) es una empresa que hace más de una década que desarrolla soluciones MDE, en especial el uso de modelos en la modernización de software. En la actualidad una de sus líneas de negocio es la migración de aplicaciones Oracle Forms a Java y está desarrollando una herramienta para la automatización de estas migraciones dentro del proyecto CDTI "MORPHEUS: Modernization of Reports, Procedural Code and the User Interface" [4]. El grupo Modelum colabora en dicho proyecto para el diseño e implementación de la migración a Java del código procedural de los manejadores de eventos (*triggers*) PL/SQL de aplicaciones Oracle Forms. Para esta tarea de migración se ha decidido seguir el enfoque general propuesto en [6] para la extracción de modelos que representen código de aplicaciones RAD (*Rapid Application Development*) a un alto nivel de abstracción. La implementación del proceso descrito en [6] ha supuesto enfrentarse a varias retos importantes entre los que destaca el manejo de modelos KDM en transformaciones modelo a modelo, la implementación de algoritmos de ingeniería inversa complejos y la validación de estos algoritmos que generan modelos grandes y complejos. Por tanto, la principal contribución de este trabajo es exponer una experiencia de ingeniería inversa basada en modelos en un caso real en la industria.

El artículo se organiza del siguiente modo. La sección 2 presenta una visión global del proceso de migración que desea implementar Open. Las siguientes secciones describen en detalle las etapas que tienen que ver con la ingeniería inversa del código procedural. Finalmente, la Sección 6 contrasta el trabajo aquí presentado con el enfoque general descrito en [6] y se presentan las conclusiones extraídas de la experiencia y el trabajo pendiente para completar el proceso.

2 Arquitectura de la solución

Como se puede apreciar en la Figura 1, el proceso de migración tiene como entrada los artefactos del sistema *legacy* Oracle Forms y debe generar los artefactos del nuevo sistema para la plataforma destino que es Java (en concreto, Open ha definido una arquitectura basada en los frameworks Spring, JSF y JPA).

El artículo se centra en los trabajos realizados en la fase de ingeniería inversa. De acuerdo con [6] se ha organizado esta fase en tres etapas que se describen en detalle en las siguientes tres secciones:

1. Primero, el código fuente PL/SQL de los triggers es inyectado en modelos KDM (elementos de los paquetes *Action* y *Code*)

[3] https://www.opencanarias.com/
[4] Referencia: IDI-20150952

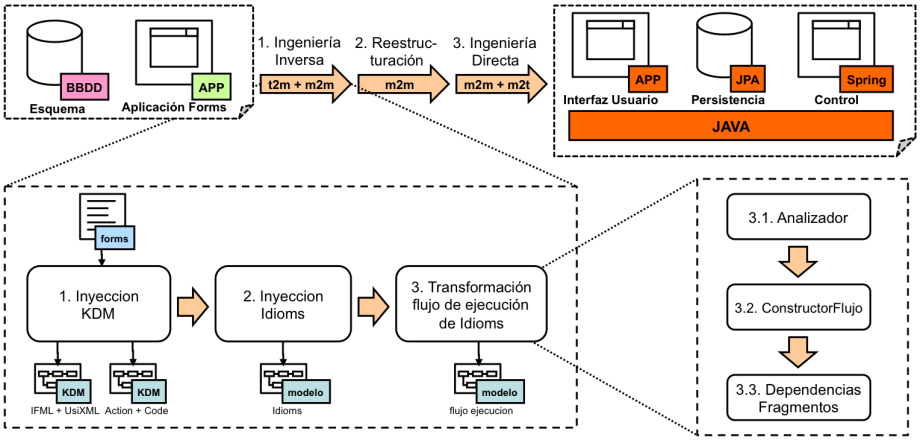


Fig. 1. Ingeniería inversa del proceso de modernización de Oracle Forms a Java

2. En la segunda etapa, los modelos KDM inyectados son analizados para identificar patrones de código (*modelo de Idioms*). Todo el código de un manejador debe expresarse haciendo uso de dichos *idioms*.
3. Finalmente, el modelo de *Idioms* es transformado en un modelo que representa el grafo del flujo de ejecución de los manejadores.

3 Inyección de modelos KDM

Se ha utilizado un inyector de código PL-SQL a modelos KDM desarrollado por Open. Este inyector conecta los elementos KDM relacionados con los triggers a elementos de un modelo que representa las interfaces de usuario por medio de metamodelos basados en UsiXML[5] e IFML[6]. Cabe destacar la complejidad tanto del proceso de inyección como de los modelos obtenidos. Además, el metamodelo KDM es muy grande y complejo (se compone de un total de 12 paquetes interrelacionados). Por ello, en esta sección nos limitamos a describir brevemente los paquetes de KDM que representan el código y a mostrar un ejemplo de inyección de una sentencia PL/SQL.

3.1 Paquetes *Action* y *Code* de KDM

Los paquetes que describen la estructura del código son *Code* y el paquete *Action*. *Code* representa los elementos de la implementación y sus relaciones, tales como llamadas a procedimientos o la declaración y uso de variables, parámetros, constantes y tipos de datos. *Action* representa el comportamiento en forma de sentencias, condiciones, flujos, excepciones, lecturas y escrituras. Ambos paquetes están interrelacionados como muestra la Figura 2.

[5] http://www.usixml.org/en/home.html?IDC=221
[6] http://www.ifml.org/

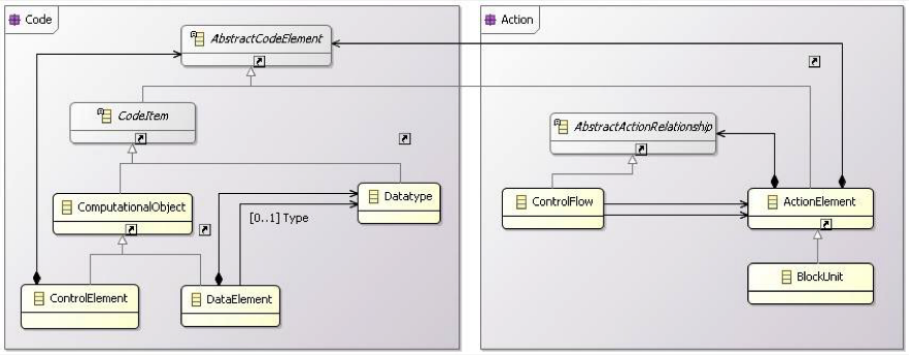
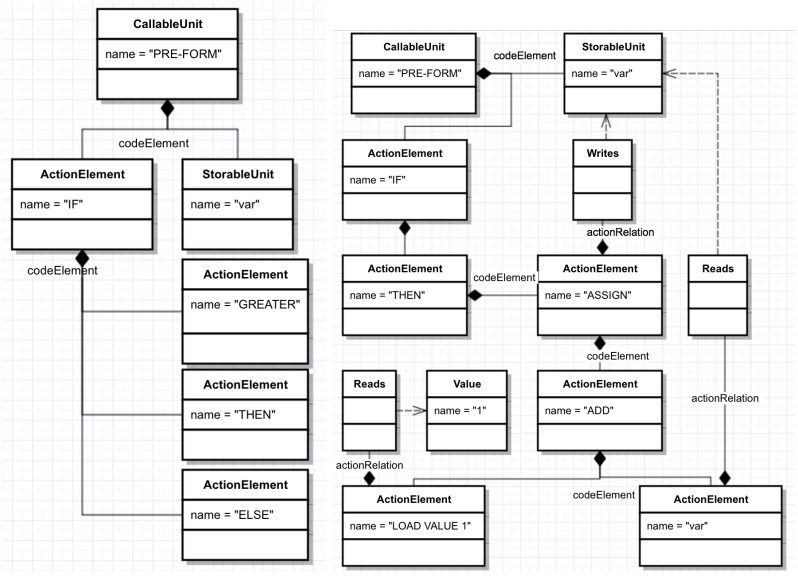


Fig. 2. Metamodelo KDM : Elementos básicos de paquetes *Action* y *Code*

La clase *ActionElement* es el principal elemento para la representación del código. Un *ActionElement* agrega un conjunto de relaciones (*AbstractActionRelationship*) y elementos de código (*AbstractCodeElement*). La clase *AbstractActionRelationship* del paquete *Action* permite representar lecturas, escrituras y flujos. Estos elementos están relacionados con un *ActionElement* que representa el bloque de código por el que debe continuar la ejecución. La clase *ActionElement* tiene varias subclases. Una de ellas es *BlockUnit* que representa un conjunto de instrucciones relacionadas como las instrucciones de un método, trigger o evento. La clase *AbstractCodeElement* del paquete *Code* se especializa en *ActionElement* y *CodeItem* que es el principal elemento del paquete *Code* para representar cualquier elemento de código. *DataType* es un *CodeItem* que representa los tipos de datos. Otra subclase de *AbstractCodeElement* es la clase *ComputationalObject* de la que hereda *CallableUnit* (una de las sublases de *ControlElement*) y *DataElement* que representan llamadas y datos, respectivamente. Un *DataElement* puede representar: i) una unidad de almacenamiento a través de *StorableUnit*, ii) la declaración de los parámetros de un método, procedimiento o función con *ParameterUnit*, iii) un valor establecido con la clase *Value*, iv) una lectura con la clase *Reads* o v) una escritura a través de *Writes*.

3.2 Ejemplo de inyección de una sentencia PL-SQL en KDM

A continuación se expone como ejemplo de inyección una sentencia *IF* que realiza una asignación si la condición se cumple o una llamada a un método si la condición no se cumple (listado 1.1). En la Figura 3a se muestra la inyección de un elemento *CallableUnit* que corresponde al trigger (llamado *PRE-FORM*). El elemento está compuesto por un *StorableUnit: var* y tres elementos *ActionElement* correspondientes a las partes del *IF*: i) la evaluación de la condición, *GREATER*; ii) el código que se ejecuta cuando la condición es cierta, *THEN*; y iii) el código que se ejecuta en otro caso, *ELSE*.



(a) Sentencia IF (b) ActionElement Then

Fig. 3. Inyección de la sentencia IF del ejemplo.

Listing 1.1. Ejemplo de código PL-SQL

```
1 IF var > 0 THEN
2     var := var + 1;
3 ELSE
4     metodo(var);
5 END IF;
```

En la Figura 3b se muestra el modelo del código ejecutado cuando el resultado de la condición del *IF* es verdadero. El elemento *ActionElement: THEN* está compuesto únicamente por una asignación que contiene un *ActionElement* para realizar la suma. La suma se realiza en dos partes, correspondientes a las lecturas de dos valores. Ambas lecturas se realizan del mismo modo: un *ActionElement* contiene *Reads* que referencian donde se encuentra el valor. En este caso, *ActionElement: LOAD VALUE 1* referencia a un elemento *Value* que se corresponde al valor constante, y el elemento *ActionElement: var* referencia a un *StorableUnit* correspondiente a la variable *var*. Una vez se realiza la suma en *ActionElement: ASSIGN*, se procede con la escritura final con *Writes*, que referencia hacia el elemento donde se almacena el valor (*var*).

4 Obtención del Modelo *Idioms*

El modelo KDM es analizado para obtener un modelo más abstracto que representa el código en forma de patrones de código (*idioms*). En esta sección primero se define el metamodelo de *idioms* y se comenta la correspondencia entre idioms

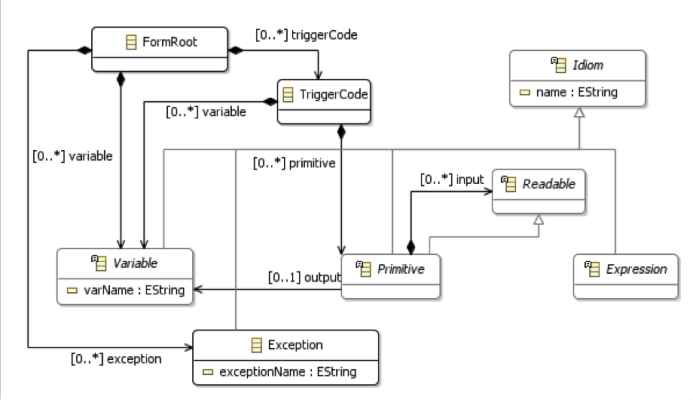


Fig. 4. Metamodelo Idioms

y código, a continuación se describe la arquitectura del componente que realiza la extracción de idioms.

4.1 Metamodelo Idioms

La Figura 4 presenta un extracto del metamodelo *Idioms* que incluye las clases que representan los idioms. El metamodelo presentado en [6] ha sido extendido para incluir 8 nuevos idioms a los 11 ya definidos (por ejemplo, los relacionados con bucles y manejo de excepciones) y 5 nuevos elementos para representar expresiones de comparación (antes había sólo 5 tipos de expresión).

La clase raíz es *FormRoot* y agrega tres elementos: i) el código de un manejador de evento (*TriggerCode*), ii) un conjunto de variables globales o locales (*Variable*), y iii) un conjunto de excepciones (*Exception*) lanzadas por la aplicación. Un *TriggerCode* agrega *Variable*s locales al manejador y elementos *Primitive* que representan las sentencias de código. Un elemento *Primitive* puede tener un valor de salida asociado a una *Variable* y un conjunto de valores de entrada llamados *Readable*. Un *Readable* puede ser una primitiva, una referencia a una variable *VariableRef* o el retorno de una llamada *ReturnValue*. Por otro lado, los elementos *Expression* representan una condición del código y pueden estar anidados. *Variable*, *Primitive* y *Expression* heredan de *Idiom* que es la clase raíz de la jerarquía de idioms definidos. Estas tres clases disponen de una referencia hacia el elemento o elementos KDM que dieron lugar al idiom.

En la Figura 5 se muestra el metamodelo donde se reúnen todas las primitivas necesarias para representar el código a más alto nivel. De la jerarquía de *Primitive* caben destacar las clases *Loop*, *SelectionFlow* y *Try* caracterizadas por estar compuestas de otros elementos. *Loop* está compuesto por una *Expression* correspondiente a la evaluación de la condición del bucle y por primitivas que representan el cuerpo de iteración del bucle. *SelectionFlow* representa la elección de un flujo de ejecución, ya sea con semántica *IF* o *SWITCH*. La primitiva contiene elementos *Case*, que tienen asociados una *Expression* correspondiente a

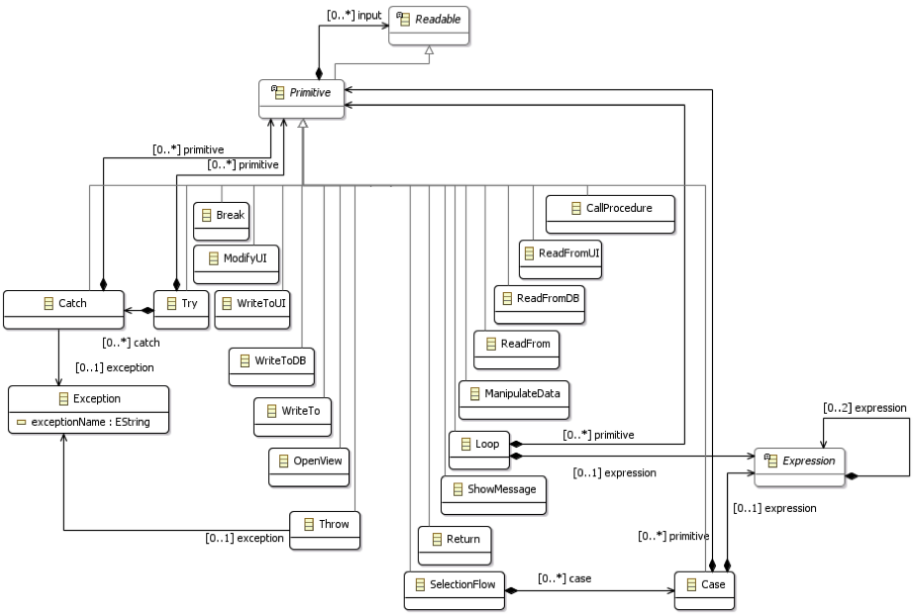


Fig. 5. Jerarquía de Primitive (idioms) del metamodelo

la condición y un conjunto de primitivas que representa el código a ejecutar si la expresión es verdadera. El idiom *Try* representa el código asociado a la captura de una excepción (*TRY-CATCH*). Un *Try* contiene elementos *Variable* de ámbito local y *Catch*, que representa a la captura de una *Exception*. Resaltar también la diferencia entre las primitivas *WriteToUI* y *ModifyUI*. La primera representa la modificación del valor de un widget gráfico mientras la segunda la de una propiedad del widget.

En la Figura 6 se presenta un extracto del metamodelo con los elementos para componer una expresión booleana. Las expresiones se anidan mediante composición sobre la clase *Expression*. De esta clase heredan clases cercanas al código, como por ejemplo las expresiones *And*, *Or*, *Less*. Pero también otras especiales como *VariableRef* que representa el valor de una variable y *ReturnValue* que representa el valor devuelto por una función. El metamodelo define también cuatro tipos de variables: i) *UIVar*, que representa un widget gráfico, ii) *LocalVar*, que representa una variable local, iii) *GlobalVar*, que representa una variable global a toda la aplicación y iv) *Constant*, que representa una constante.

Se ha definido una correspondencia que asocia *idioms* a elementos del modelo KDM. Abajo se expresan tres de estas correspondencias indicando el nombre del *idiom* y los nombres de los elementos KDM (*ActionsElement*) asociados, por ejemplo "LOAD VALUE 5" es el nombre de una acción de lectura. A continuación se describe el componente que se encarga de generar el modelo de *idioms*.

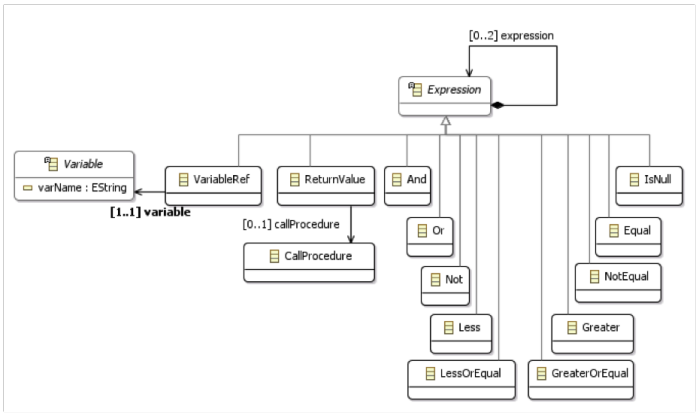


Fig. 6. Conjunto de Expresiones del metamodelo

- `WriteTo` ➟{`ASSIGN`};
- `ReadFrom` ➟{`LOAD VALUE *`, `NombreVariable`};
- `ManipulateData` ➟{`SASTMCONCATENATE`, `ADD`, `SUBTRACT`, `UNARYMINUS`, `DIVIDE`, `MULTIPLY`}.

4.2 Conversión de KDM en Idioms

Se ha creado una transformación modelo a modelo en Java que obtiene un modelo Idioms a partir de un modelo KDM de entrada. Este transformación se ha organizado en cuatro elementos: i) **Analizador** que realiza el recorrido del modelo KDM, ii) **Clasificador** que procesa elementos del modelo KDM y elige qué idiom debe ser creado, iii) **ConstructorIdioms** que realiza la construcción de los idioms, y iv) **ConstructorReferencias** que establece la referencia entre los idioms creados y los ya existentes. Los elementos creados se almacenan como objetos de la clase *PairList*. Existe una clase de este tipo por cada idiom que puede contener primitivas en su interior (por ejemplo, *Loop* o *Case*). Por otro lado, las variables se almacenan en *ControlVariables*.

La transformación se realiza del siguiente modo. El *Analizador* recorre el modelo KDM primero en profundidad. Para cada elemento del modelo se llama al *Clasificador*, el cual identifica el elemento KDM. Este *Clasificador* llama a su vez a *ConstructorIdioms* pasando el elemento KDM correspondiente para crear el nuevo Idiom que contendrá la referencia hacia dicho elemento. El *Clasificador* puede no ser necesario si el *Analizador* ha podido identificar el tipo (los bloques *Try* son uno de estos casos). Finalmente, el *ConstructorReferencias* añade una referencia del nuevo idiom hacia el último idiom creado (según el tipo de idiom creado). En ocasiones es necesario establecer también referencias hacia otros elementos, como *Expression* creados previamente.

El **Analizador** parte de un *Segment* de KDM y analiza los modelos *Model* que contiene hasta encontrar un modelo de código *CodeModel*. Entonces se analizan los *CodeElements* que contiene. Estos elementos se corresponden a *Units*

y se procesan tres de ellos: (1) *DataElements*, que almacena las variables globales; (2) *ExceptionLibrary*, que almacena las excepciones lanzadas en manejador de eventos; y finalmente, (3) *CompilationUnit* que corresponde al código (cuyo análisis es el más complicado). Para realizar el procesamiento del código (de cada manejador de evento) es necesario identificar los *CallableUnit* que componen un *CompilationUnit*. Para cada *CallableUnit* se crea un elemento *TriggerCode* del modelo Idiom y se procesan los *BlockUnit* en cada *CallableUnit*. Finalmente, para cada *BlockUnit* se procesan los *ActionElement* estereotipados que representan el código. Cada vez que se procesa un elemento *BlockUnit* o un *ActionElement* se indica al *ConstructorIdioms* que se ha incrementado el nivel del árbol KDM que se está procesando. Así es fácil crear una pila con los idioms que se van creando y el nivel al que pertenecen dentro del árbol KDM (lo cual posibilita establecer nuevas referencias). Cada *ActionElement* se envía al *Clasificador* para que continúe con el procesamiento. Sin embargo, tal y como se anticipó antes, en el analizador es posible detectar si el nuevo elemento es un *Try* o un *Catch* en cuyo caso se envía directamente al *ConstructorIdioms*. Es necesario identificar directamente estos elementos en el analizador para que se puedan almacenar las variables locales que se declaran en el nuevo bloque.

El **Clasificador** es el componente más sencillo de todos y se encarga de asociar elementos KDM a Idioms. Para ello se identifica el nombre del *ActionElement* y se delega en el *ConstructorIdiom*. la correspondencia se realiza de distintas maneras, dependiendo del *ActionElement*: i) directamente desde el nombre (a veces el propio nombre del *ActionElement* es suficiente para identificar qué Idiom se debe crear); ii) el nombre está compuesto por *LOAD VALUE* seguido de la constante; iii) directamente desde el nombre de la variable (para realizar esta parte se delega en *ControlVariables* donde están almacenadas todas las variables); y iv) directamente desde una *MacroDirective*, donde se almacenan las instrucciones de los cursores implícitos.

El **Constructor de idioms** tiene un método por cada elemento del metamodelo de Idiom. El clasificador invocará al método específico una vez identificado el tipo de elemento de KDM. Cada método realiza la construcción del idiom utilizando la factoría de EMF e inicializa todas las propiedades que se conozcan en el momento en que se crea. El *ConstructorIdioms* se divide en varios constructores especializados en construir elementos de código, actuando de fachada y desacoplando de esta forma el código.

El **Constructor de referencias** es el componente más complejo. Su función es almacenar el nuevo idiom y unirlo al idiom que lo agrega. La complejidad reside en buscar el elemento sobre el que unirlo. Para ello se dispone de varios elementos donde almacenar los idioms que se van creando, independiente del elemento raíz creado anteriormente. Todos los idioms se almacenan en una pila de pares *(nivel, idiom)* que representa la rama hasta el idiom actual.

4.3 Validación

La validación de la transformación de KDM a Idioms se ha realizado manualmente en dos etapas. Primero se validó para triggers muy sencillos que cubrieron

todos los posibles idioms. Luego con 3 formularios proporcionados por la empresa Open, los cuales tenían complejidad baja, media y alta de acuerdo al número de triggers y procedimientos almacenados, que es de 33, 87 y 190, respectivamente. Se han procesado e identificado correctamente el 100% de elementos KDM (*ActionElement*) y se han generado los modelos idioms correctos en cada caso.

En la Tabla 4.3 se exponen las estadísticas completas de la validación. Resumiendo, el 30% de los elementos corresponden a elementos de control de flujo, el 60% de datos y solo el 10% a llamadas.

Nombre		**Porcentaje**
Acceso a variables		
	Globales: 8,96%	
	Locales: 6,70%	15,66%
Condiciones, Switchs e IFs		24,64%
Llamadas a procedimientos		10,70%
Bucles		0,94%
Asignaciones		5,17%
Manipulación de Datos		4,25%
Constantes		16,46%
Base de Datos		
	Lecturas: 1,28%	
	Escrituras: 1,92%	
	Común*: 17,54%	20,74%
Throws		0,91%
Otros (Clasificados pero no importantes)		0,53%
Total		**100%**
***Instrucciones para generar la cláusula WHERE o SELECT.**		

Tabla 1. Estadisticas del proceso de validación

5 Obtención del grafo del flujo de ejecución

A continuación se describe la transformación que obtiene el modelo del flujo de ejecución de un trigger a partir del modelo de Idioms. Se sigue el mismo esquema que en la sección anterior: se presenta el metamodelo destino, luego la transformación, y finalmente la validación.

5.1 Metamodelo de flujo de ejecución de Idioms

El metamodelo de flujo de ejecución se muestra en la Figura 7 y es el presentado en [6] pero con algunas nuevas referencias que facilitan el manejo de los modelos. El flujo de ejecución se representa como un grafo y se utiliza el algoritmo descrito en [1] para su identificación. Este algoritmo se basa en la identificación de *bloques básicos de fragmentos*. Los fragmentos se han etiquetado en tres categorías: presentación, lógica de negocio y datos.

El metamodelo tiene como raíz la clase *FormFlowRoot*, que está compuesta por *EventHandler* y representa bloques de código de manejadores de eventos.

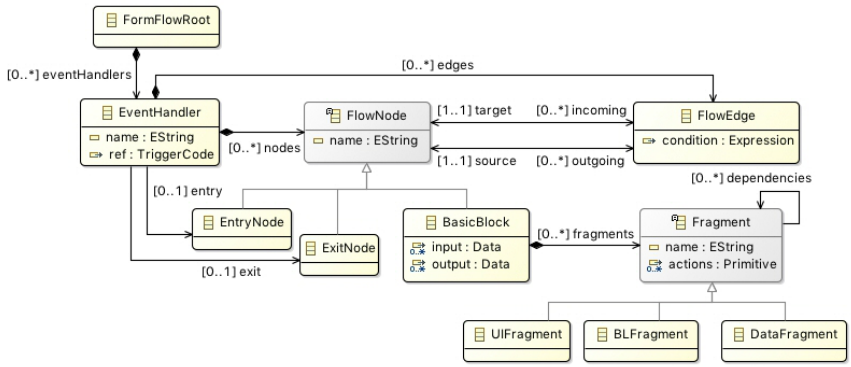


Fig. 7. Metamodelo de flujo de ejecución de Idioms

Mediante el atributo *ref* se referencia al manejador en el metamodelo de Idioms. *EventHandler* tiene nodos representados como elementos *FlowNode*. Existen tres tipos: (1) *EntryNode*, el primer nodo del subgrafo; (2) *ExitNode*, el último nodo y (3) *BasicBlock*, un bloque de código. Esta última clase presenta referencias a las variables utilizadas del metamodelo de Idioms. *BasicBlock* contiene fragmentos que pueden ser de tres tipos: *UIFragment*, *BLFragment* y *DataFragment*. Estos fragmentos tienen referencias hacia las primitivas del metamodelo de Idioms incluidas en el bloque de código. Los fragmentos pueden ser dependientes entre sí en caso de que los valores de salida de un nodo sean los valores de entrada de otro. Por último, los nodos se relacionan utilizando las aristas representadas con *FlowEdge*. Las aristas pueden mantener una referencia hacia una expresión del metamodelo Idioms, significando que la ejecución de la arista está condicionada a dicha expresión.

5.2 Arquitectura del transformador

La transformación (también implementada en Java) se implementa por medio de tres componentes: (1) Un *Analizador* del código de los elementos *TriggerCode* del modelo de Idioms; (2) *ConstructorFlujo* que dirige la construcción del flujo de ejecución para un manejador específico, y (3) *DependenciasFragmentos* que actualiza las entradas y salidas (*Input* y *Output*) de cada *BasicBlock* con las variables encontradas en el bloque que representa. Además, establece las dependencias entre los fragmentos de un mismo *BasicBlock*. El **Analizador** aplica un recorrido por el modelo de Idioms de entrada y para cada *TriggerCode* (1) construye el grafo utilizando *ConstructorFlujo* y almacena el subgrafo correspondiente al manejador actual y (2) delega en *FragmentDependencies* el establecimiento de las variables de entrada/salida y las relaciones entre fragmentos del modelo de Idioms.

El **ContructorFlujo** implementa los pasos del algoritmo descritos en [1]. Genera el subgrafo correspondiente a un manejador de eventos construyendo los

elementos *EventHandler*, *EntryNode* y *ExitNode*. Una vez realizada esta fase inicial, se itera sobre todos los elementos de código del modelo KDM y se construye un *BasicBlock* por cada *Primitive* que sea la primera primitiva de un bloque de código. Se construye, entonces, un fragmento dependiendo del tipo de *Primitive* del modelo de Idioms delegando en una clase *Clasificador* que clasifica de forma estática la primitiva en una sentencia: (a) de interfaz de usuario (*UI*), (b) de lógica de negocio (*Business Logic*) o (c) de datos (*Data*). El criterio de clasificación considera como *UI* las siguientes primitivas : *ReadFromUI*, *WriteToUI*, *ModifyUI*. Las primitivas que modifican datos en la base de datos, como *ReadnFromDB* o *WriteToDB*, se clasifican como *Data*. El resto se almacenan como *Business Logic*. La clasificación permite distribuir los fragmentos de código en las diferentes capas de la aplicación resultante de la modernización.

El proceso continúa y se repite, salvo que la nueva primitiva pertenezca al mismo bloque de código, en cuyo caso no se crearía un nuevo bloque. Tampoco si la nueva primitiva pertenece a la misma clasificación, asignándola al fragmento creado en la iteración anterior.

Una vez creado el subgrafo completo, el algoritmo delega en el componente **DependenciasFragmentos**. Este recorre de nuevo todo el subgrafo y para cada *BasicBlock* examina sus distintos fragmentos, añadiendo las variables utilizadas en ellos a los atributos *Input* y *Output* de *BasicBlock*. Una variable en un fragmento se establece como entrada si se realiza una lectura de ella y como salida si la variable ha sido escrita. *BasicBlock* tendrá como entradas y salidas las variables utilizadas por los elementos *Primitive* de sus fragmentos. Una vez realizado el establecimiento de las variables en *BasicBlock*, se procede a comprobar las dependencias entre fragmentos. Se comprueban las variables usadas en dos fragmentos de un mismo *BasicBlock*. Si una variable de la que lee un fragmento es escrita en otro fragmento anterior, entonces existe una dependencia. Las dependencias entre fragmentos se construyen siguiendo el orden de aparición en código. El posicionamiento entre fragmentos refleja por tanto el orden de ejecución.

5.3 Validación del algoritmo del flujo de ejecución

Aplicar un proceso de validación al algoritmo de generación de flujo resulta complejo. Esto se debe, principalmente, al hecho de que los modelos generados son grafos complejos y su validación visual requiere recorrer las aristas. Esta tarea es muy engorrosa y consumiría mucho esfuerzo con el visor de modelos proporcionado por Eclipse que muestra la lista de nodos y aristas de manera secuencial ocultando la propia naturaleza del grafo. Incluso EMF to GraphViz[7] tampoco mostraría los grafos de forma apropiada a nuestros intereses (de hecho, no se pudieron cargar los modelos por su tamaño).

Se ha optado por representar los modelos mediante el visor de Neo4J [8]. Mediante una transformación modelo a texto se genera un script de inserción de

[7] https://marketplace.eclipse.org/content/emf-graphviz-emf2gv
[8] https://neo4j.com/

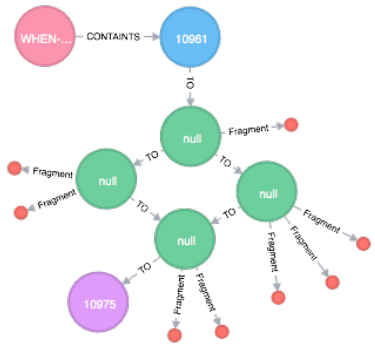


Fig. 8. Grafo en Neo4J del Listado 1.2.

elementos en Neo4j a partir del modelo de flujo. La transformación resulta sencilla puesto que el propio metamodelo de flujo posee estructura de grafo. El grafo de Neo4J incluye referencias a los fragmentos como nodos (en rojo) mediante aristas desde un *BasicBlock*. Para una sentencia condicional de ejemplo como la mostrada en el listado 1.2 se construye el grafo presentado en la Figura 8. El nodo rosa representa el trigger siendo evaluado. Los nodos azul y violeta representan los nodos inicial y final del grafo. Los nodos verdes representan sentencias de código: el primero se corresponde con la primera asignación (inicialización del salario), tras la cual se produce una bifurcación en dos nodos para la sentencia condicional, los cuales acaban uniéndose en el nodo de la última sentencia (actualización del salario). La validación ha resultado más simple y menos costosa que la realizada para los idioms, que exigía ver el código PL/SQL, su modelo KDM y el modelo idiom generado.

Listing 1.2. Ejemplo usado para la validación del grafo

```
1 cota := 1000;
2 IF salario > cota THEN
3         bonificacion := salario - cota;
4 ELSE
5         bonificacion := 100 + cota - salario;
6 END IF;
7 salario := salario + bonificacion;
```

6 Trabajo relacionado y Conclusiones

En [6] se propone un enfoque basado en modelos para la ingeniería inversa de manejadores de eventos en aplicaciones RAD, el cual se aplica a un caso de estudio de un formulario Forms muy simple. El código se inyecta en un modelo del árbol de sintaxis abstracta del lenguaje. En nuestro caso se ha utilizado el metamodelo estándard KDM que es independiente de la plataforma, lo cual hace que la solución sea más reutilizable y permite manejar modelos a un nivel más abstracto. Por otra parte, ha sido necesario extender el conjunto de idioms y expresiones (y su correspondencia con KDM) como se indicó en 4.1 para poder

representar en KDM todas las sentencias PL/SQL. Y lo que es más importante, se han implementado las dos transformaciones de modelos que obtienen los modelos Idioms y el de flujo de ejecución. Se trata de transformaciones complejas debido a la complejidad de los modelos involucrados y al propio proceso de ingeniería inversa. Se han implementado en Java en vez de usar lenguajes de transformación de modelos debido a sus limitaciones para expresar algoritmos complejos y manejar estructuras de datos intermedias como listas y tablas [7]. Además, las transformaciones han sido aplicadas y validadas a un proyecto real. La validación de los modelos de flujo se ha realizado utilizando Neo4J para su visualización lo que, además de original, podría ser útil en otros casos dado que los modelos de grafos son muy frecuentes. Tras aplicar esta estrategia, se ha pensado que podría ser más sencillo definir test unitarios que validen modelos de idioms muy simples que cubran todos los casos y usar el visor de grafos para comprobar que esos tests son correctos, en vez de validar grafos grandes.

La comprensión de cómo representar en KDM código de un determinado lenguaje no es nada trivial y existe poca información al respecto [2]. Como parte del proyecto se ha elaborado un tutorial que explica cómo representar sentencias PL-SQL en KDM (paquetes Action y Code). Cabe destacar que no hemos encontrado nuevo trabajo relacionado al considerado en [6]. En cuanto al trabajo pendiente, se está abordando la fase de generación de código Java, en la que jugarán un papel importante las etiquetas asignadas a los fragmentos (control, datos, e interfaz de usuario) con el fin de distribuir el código en una arquitectura MVC. Siguiendo un proceso de reingeniería clásico (ingeniería inversa, reestructuración y generación del nuevo sistema) [8], antes de la generación de código se realizará una etapa de *reestructuración* que obtendrá un modelo de la arquitectura destino a partir del modelo de flujo de ejecución. Una vez se complete la generación de código, se aplicará un proceso de validación empírica mediante casos de estudio reales para verificar la consecución de los objetivos. Por último, señalar que se ha detectado la conveniencia de cambiar el inyector de KDM para denotar los tipos de sentencias PL/SQL en micro-KDM en vez de usar estereotipos propios de Open.

Referencias

1. Aho, A.V., et al.: Compilers: Principles, Techniques and Tools. Addison-W. (1986)
2. Izquierdo, J.L.C., Molina, J.G.: An architecture-driven modernization tool for calculating metrics. IEEE Software 27(4), 37–43 (2010)
3. OMG: Architecture-Driven Modernization (ADM). http://adm.omg.org/ (2007)
4. OMG: Knowledge Discovery Metamodel. http://www.omg.org/spec/KDM (2008)
5. Pérez-Castillo, R., de Guzmán, I.G.R., Piattini, M., Ebert, C.: Reengineering technologies. IEEE Software 28(6), 13–17 (2011)
6. Sánchez Ramón, O., et al.: Reverse engineering of event handlers of rad-based applications. In: WCRE. pp. 293–302 (2011)
7. Sánchez Ramón, O., et al.: Una valoración de la modernización de software dirigida por modelos. In: XIII JISBD. pp. 288–301 (2013)
8. Tilley, S.R., Smith, D.B.: Perspectives on legacy system reengineering. Tech. rep., Software Engineering Institute, Carnegie Mellon University (1995)