Acceso a la información mediante un modelo conceptual

Information access through a conceptual model

Jorge Landrian García

Universidad de las Ciencias Informáticas

jlandrian@uci.cu

Resumen

La variante de acceso a la información descrita en este trabajo permite acceder a los datos de cualquier sistema a través de un lenguaje general, que posibilita un nivel de abstracción superior al utilizado por el mecanismo específico que implemente la solución del sistema al cual se quiere acceder. La descripción de un sistema mediante un mapa conceptual, donde se representen los conceptos y relaciones entre estos, permite una visión descontextualizada de la implementación específica del sistema informático. En este trabajo se describe un mecanismo de acceso que utiliza como base estos mapas conceptuales, además de una sintaxis de expresiones definibles según el problema específico para el cual se utilice el lenguaje, siendo acoplable así a cualquier mecanismo de persistencia de la información y al negocio concreto que lo utiliza.

Palabras clave: Acceso a datos, base de datos, datos, información, lenguaje, ontología, reportes, sintaxis.

Abstract

The information access option described in this paper provides access to data from any system through a general language, which enables a higher level of abstraction than that used by the specific mechanism implemented in the system to which access is desired. The description of a system through a conceptual map, which represents concepts and relationships between them, allow a vision out of the implementation context of a specific computer system. This paper describes a mechanism for data access, which uses as a basis these concept maps and expression syntax definable according to the specific problem in which the language is used, being well suitable to any information persistence mechanism and the specific business that uses it.

Key words*: Data access, data base, data, information, language, ontology, reports, syntax.*

Introducción

Existe gran variedad de mecanismos para acceder y persistir la información en sistemas informáticos. Dependiendo de la información a manejar por el sistema y variables como: disponibilidad de los datos, seguridad, velocidad de acceso, entre otras, se selecciona la variante de almacenamiento más adecuada para cada caso, en un mismo sistema pueden estar implementadas diversas formas de persistencia y acceso de la información.

Los métodos de almacenamiento más utilizados son:

- Ficheros estructurados.
- Ficheros planos.
- Ficheros XML.
- Base de datos relacional.
- Base de datos orientada a objetos.
- Base de datos jerárquica.

En dependencia de la variante de persistencia que se seleccione será el mecanismo a utilizar para el acceso y la actualización de la información. Existe gran cantidad de librerías, gestores de base de datos, formatos de ficheros, estándares en general que permiten

la utilización de estas formas de almacenamiento, y aunque hay estándares para cada tipo de persistencia y muchos sistemas los implementan, siempre hay particularidades a tener en cuenta cuando se usa un mecanismo específico.

Por ejemplo, para acceder a ficheros XML, se puede utilizar diferentes *parsers*:

- MSXML de Microsoft
- Oracle's XML parser.
- Saxon
- Xalan
- Xerces

Cada uno de estos *parsers* tiene una interfaz específica que hay que dominar para poder usarlos como intermediarios para acceder a la información de los ficheros XML.

Algunos de los gestores de base de datos relacionales más utilizados son:

- Microsoft SQL Server
- Oracle
- Postgree SQL
- SQL Lite
- DB2
- Sysbase

Aunque existen estándares SQL para el acceso a datos en base de datos relacionales, cada uno de los gestores e incluso versiones de un mismo gestor posee implementaciones específicas de estos estándares, generándose así estándares específicos para muchos gestores de base de datos.

Esta diversidad de mecanismos de persistencia y al mismo tiempo la diversidad de implementaciones de estos mecanismos provoca que los sistemas informáticos estén muy ligados a las variantes de almacenamiento que se seleccionaron en la etapa de diseño del sistema.

El problema anterior sería solucionado con la creación de un lenguaje de acceso a la información que no sea dependiente del mecanismo utilizado para persistir la información y la creación de traductores específicos para las variantes de almacenamiento utilizadas.

Requisitos del lenguaje de acceso a la información

SQL: En los gestores de base de datos relacionales el lenguaje estándar utilizado es el SQL, el SQL es un lenguaje potente, pero al mismo tiempo la creación de una consulta SQL para acceder a cierta información puede resultar muy compleja y muchas veces es necesario tener un amplio conocimiento del álgebra relacional.

XPATH: Xpath es el lenguaje de acceso a los datos de un XML, es un lenguaje jerárquico y relativamente más comprensible que el SQL aunque no posee algunas de las potencialidades del SQL, pero de una forma u otra siempre es posible acceder a la información.

La mayoría de los lenguajes de acceso a datos han sido diseñados de acuerdo a la estructura de almacenamiento de la información. El objetivo es crear un lenguaje que sea independiente de la estructura final de almacenamiento, por lo que es necesario crear una estructura de la información que sea independiente de cómo está persistida.

Librería CONCEPTUALMAP.DLL

Mapa conceptual

Los mapas conceptuales son herramientas muy utilizadas para la representación del conocimiento, la información cuando se estructura en conceptos y relaciones entre estos, es mucho más comprensible y utilizable tanto por las personas como por los sistemas informáticos.

En la librería **ConceptualMap.dll** se utiliza una representación conceptual de la información, manteniendo una total independencia de los conceptos y relaciones, de la implementación específica que se utilice para almacenar la información. Al documento que describe los conceptos y las relaciones se le llama Ontología y es persistido en un fichero XML que cumple con el siguiente esquema:

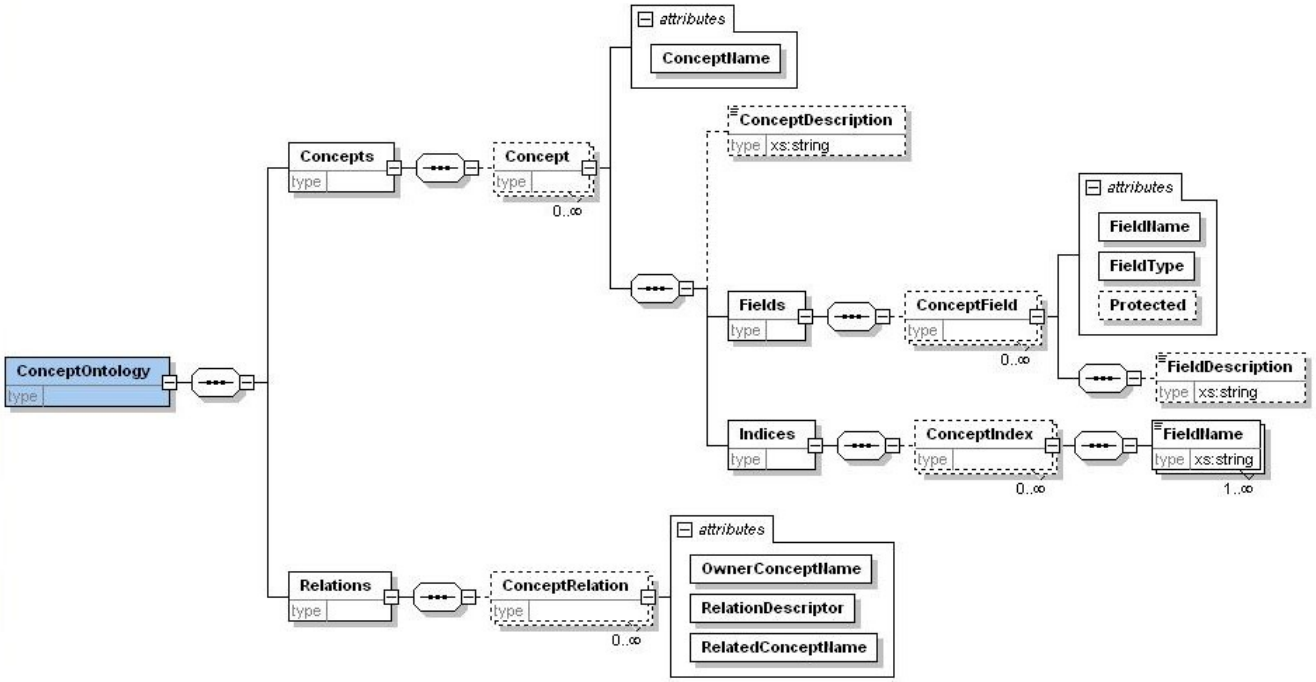


Fig. 1. Esquema de una ontología

Una ontología está compuesta por conceptos y relaciones entre estos, cada concepto está compuesto por campos que son los atributos del concepto, y además un concepto puede tener índices, un índice es un conjunto de campos que hace único a una instancia específica de un concepto.

Lenguaje

Tomando como representación de la información una ontología que describe la estructura de la información mediante conceptos y relaciones, el lenguaje de acceso a datos se basa en describir el camino hacia el concepto a navegar (localizador) y describir cuál es la información a mostrar (vista) en esa localización.

Gramática del lenguaje representada en Backus-Naur Form (BNF)

<vista-consulta-concepto> ::= <consulta-concepto> “(” <colección-expresión> “)”

<consulta-concepto> ::= <concepto>|<consulta-concepto> <separador> <relación-concepto>[1]|<consulta-concepto> “[”<filtro>“]”

<relación-concepto> ::= <texto> <concepto> | <concepto> <texto>

<concepto> ::= “(” <texto> “)”

<texto> ::= <identificador> | <identificador> <espacios-blanco> <texto>

<espacios-blanco> ::= “ ” | “ ” <espacios-blanco>

<filtro>[2] ::= <expresión>

<colección-expresión> ::= <expresión> | <colección-expresión> “,” <expresión>

<expresión> ::= <valor-literal> | <campo-concepto> | <variable> | <operación> | <llamada-función> | “(” <expresión> “)”

<valor-literal> ::= <número> | <cadena>

<campo-concepto>[3] ::= <texto> | “$” <entero> “.” <texto>

<variable>[4] ::= “@” <texto>

<llamada-función> ::= <identificador> “(” ”)” | <identificador> “(” <colección-expresión> “)”

<operación> ::= <expresión> <operador>[5] <expresión>

<identificador> ::= ((<letra> | _) (<letra> | <dig> | _)*) | ” ([^”] | ””)* ”

<número> ::= <entero>(\.<dig>*)?

<entero> ::= <dig>+

<cadena> ::= ’([^’]* | ‘’)*’

<dig> ::= [0..9]

<letra> ::= [a..zA..ZñáéíóúÑÁÉÍÓÚ]

Nota: Los superíndices en las etiquetas son descritos a continuación en la semántica.

Semántica

1- En el momento que se hace referencia a una relación, esta relación tiene que estar definida en la ontología, además que el concepto anterior donde se represente la relación debe contenerla.

Ejemplo:

- (Persona) / tiene (Perro)
- (Perro) / (Persona) tiene

Para que estas consultas sean válidas debe existir la relación “Persona tiene Perro”. Al definirse en la ontología una relación “(Concepto 1) relación x (Concepto 2)” es posible expresar dos localizadores

- (Concepto 1) / relación x (Concepto 2)
- (Concepto 2) / (Concepto 1) relación x

En el momento de crear una ontología es importante tener en cuenta que no es necesario definir las relaciones inversas.

2- Un <filtro> es una expresión lógica que retornan *true* o *false*.

3- El acceso a los campos de un concepto se describe mediante la profundidad que tiene el concepto al cual hacemos referencia partiendo del punto en donde se localiza la expresión y el nombre del campo al cual estamos accediendo. En caso que el campo pertenezca al concepto en el cual está localizada la expresión, se puede omitir la profundidad, es decir, la profundidad 0 se puede suprimir, y sólo se colocaría el nombre del campo al cual se quiere acceder. Analicemos la siguiente consulta:

(Persona) / conduce (Carro) [Color = ‘Azul’] ($1.Nombre, $1.Fecha de nacimiento, Marca, Año)

El localizador (Persona)/conduce(Carro)[Color=’Azul’] hace alusión a todos los carros de color azul que son conducidos por personas. La vista de esta consulta es ($1.Nombre, $1.Fecha de nacimiento, Marca, Año). En una expresión, la profundidad de cada concepto es calculada de atrás hacia delante, siendo “0” la posición de donde se parte. Teniendo en cuenta esto, la vista de esta consulta mostraría los campos “Nombre” y “Fecha de nacimiento” del concepto “Persona” y los campos “Marca” y “Año” del concepto “Carro”. En el filtro “[Color=’Azul’]” se hace referencia al campo “Color” del concepto “Carro”, otra forma de escribir este mismo filtro es “[$0.Color=‘Azul’]” la referencia “$0” se puede omitir.

Supongamos que se quiere obtener ahora la misma vista de la consulta anterior, pero de las personas que tienen como nombre Pedro y el carro que conducen es azul. Esta consulta puede se puede escribir de las siguientes formas:

- (Persona) [Nombre = ‘Pedro’] / conduce (Carro) [Color = ‘Azul’] ($1.Nombre, $1.Fecha de nacimiento, Marca, Año)

- (Persona) / conduce (Carro) [Color = ‘Azul’ y $1.Nombre = ‘Pedro’] ($1.Nombre, $1.Fecha de nacimiento, Marca, Año)

Las dos opciones son válidas, aunque es más recomendable la primera variante, debido a que es relativamente más fácil de comprender. La referencia a un concepto siempre es con números positivos o cero, lo que imposibilita hacer referencia a un concepto que quede a la derecha de la expresión donde se utilice la referencia.

4- Una variable es un mecanismo que permite asignar posteriormente cualquier expresión a esta variable. El valor de las variables es suministrado por el VariableProvider que se le asigne a la variable.

5- Los operadores y las funciones son definidos en un documento sintaxis, creando así un lenguaje dinámico que es diseñado de acuerdo a los requerimientos del sistema en el cual se va a utilizar.

Documento Sintaxis

El documento sintaxis es un XML que cumple con el siguiente esquema, en el se definen todos las funciones y operadores y la precedencia de estos. La sintaxis es utilizada por el *parser* para generar los *Tokens* correspondientes, según la gramática del lenguaje y la sintaxis expresada en el documento.

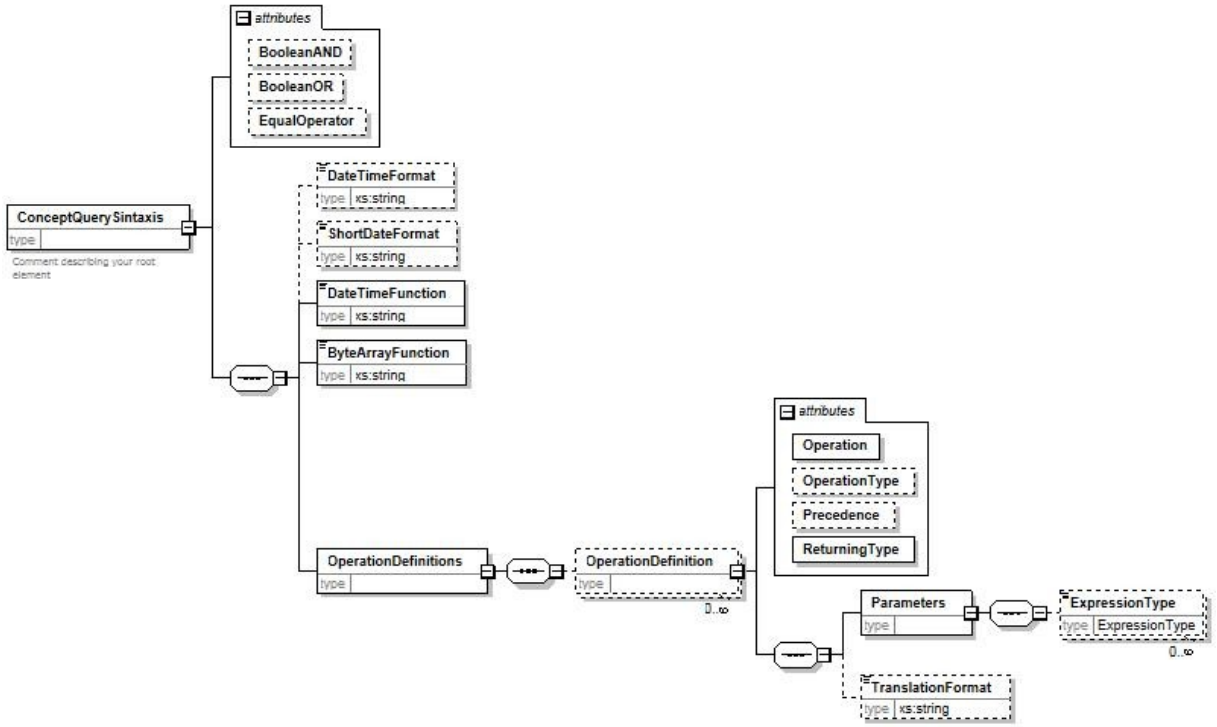


Fig. 2. Esquema de la sintaxis

El objeto sintaxis contiene todas las definiciones de operaciones, y también es independiente de la implementación final que se esté utilizando, aunque en la sintaxis se definen operaciones que pueden ser específicas de la problemática a la cual se le esté dando solución.

En la sintaxis, se hacen especificaciones de los operadores lógicos AND, OR, además del operador de igualdad. Esto es necesario para poder generar expresiones lógicas de forma dinámica.

Expresiones

Una expresión es modelada utilizando un patrón composición reduciendo así la complejidad del problema. La semántica del problema no es predefinida en el diseño de la jerarquía, cada uno de los componentes de una expresión conoce cómo validar las operaciones a realizar, de esta forma se simplifica el problema, y brinda gran flexibilidad a la jerarquía. Las operaciones, tanto funciones como operadores son definidas en la sintaxis, y cada invocación a una operación conoce la definición de la operación a realizar.

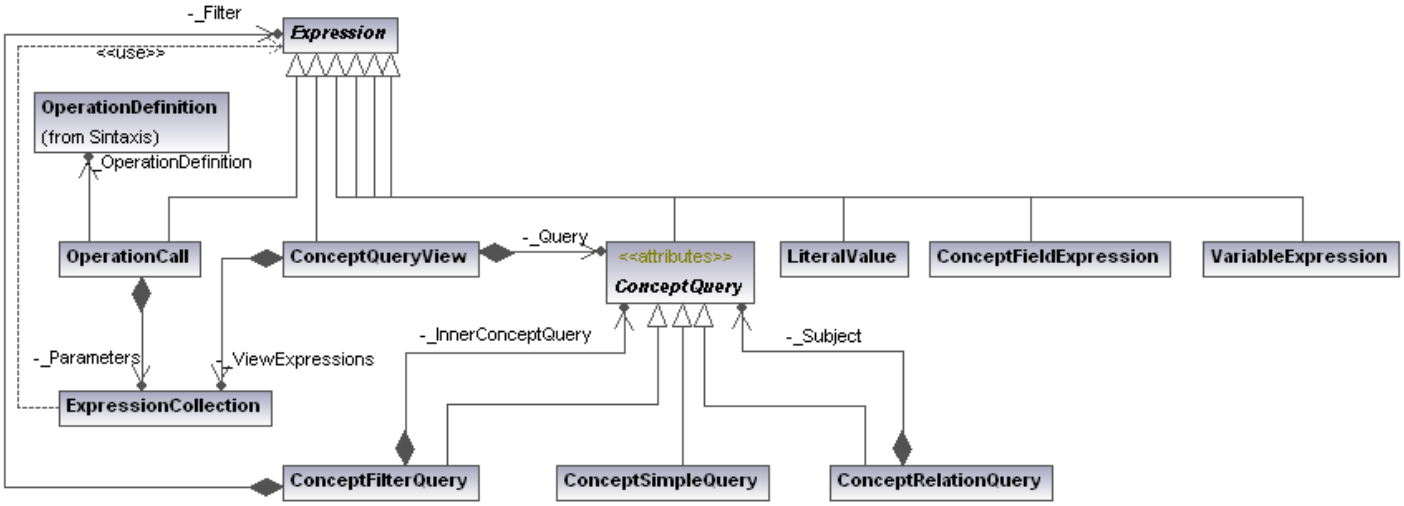


Fig. 3. Jerarquía de expresiones

La clase base de un localizador es ConceptualMap.Query.ConceptQuery, de la cual hereda ConceptSimpleQuery, ConceptFilterQuery y ConceptRelationQuery; esta jerarquía permite crear cualquier localizador que cumpla con la gramática del lenguaje. Al mismo tiempo la clase ConceptQuery hereda de Expression, permitiendo esto cualquier combinación en las expresiones.

ConceptQueryView es la clase encargada de manejar las vistas de un localizador, los campos a mostrar en la vista son derivados de Expression, según los requerimientos del sistema en que se vayan a utilizar las expresiones, se validarían qué tipos de datos son permitidos en las vistas, esto se verá más detallado en el ejemplo del traductor a SQL.

Esta jerarquía permite representar cualquier operación matemática con números reales, y cualquier operación con cadenas de caracteres. Además es posible incorporar nuevos conceptos a la herencia sin afectar el resto de los conceptos existentes, dando la posibilidad de poder utilizar este diseño para representar las expresiones que sean requeridas en el sistema que se esté implementando. Una de las mayores potencialidades que tiene esta estructura es, que sin tener que agregar nuevos componentes al diseño es posible utilizar cualquier función y operador con sólo definirlos en la sintaxis.

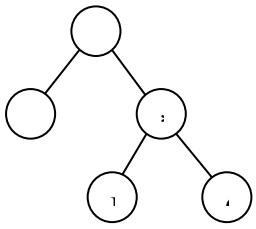
Construcción de expresiones

Para la creación de una Expression a partir de la expresión representada en texto, se utiliza la clase ConceptualMap.Query.ExpressionBuilder y un algoritmo interesante es el utilizado para crear el árbol de una Ecuación. La clase que encapsula este algoritmo es ConceptualMap.Query.Sintaxis.ExpressionEquation, esta es la encargada de reducir una expresión formada por subexpresiones y operadores a una sola expresión.

Un algoritmo tradicionalmente utilizado para dar solución a este problema es la conversión de la expresión a una notación postfija y luego reducirla utilizando una pila, en la cual al terminar el proceso queda la expresión final; esta solución requiere conocer la precedencia de los operadores de antemano, porque es necesario en el proceso de conversión a postfija. En la solución implementada en ConceptualMap.dll, los operadores están definidos en la sintaxis, y la definición de un operador no está dada

sólo por el símbolo que utilice, sino por los tipos de dato de los operandos. En la sintaxis el identificador de un operador se representa por “símbolo (tipo del operando izquierdo, tipo del operando derecho)” debido a esto en una expresión textual el operador únicamente puede ser identificado de acuerdo a los valores sobre los que está operando. Por ejemplo en la siguiente expresión: A+B*C.

Según la precedencia utilizada normalmente para estos operadores, primero se realizaría la operación B*C y al resultado se le sumaría A, el árbol sería el siguiente:



Lo particular de tener los operadores en la sintaxis, es que para analizar cuál operador es el de mayor precedencia para realizar la operación, es necesario buscar en la sintaxis cada operador utilizando como operando los valores adyacentes a él. En el ejemplo anterior habría que buscar en la sintaxis los identificadores “+(tipo(A), tipo(B))“ y “*(tipo(B), tipo(C))” y el operador que mayor precedencia tenga es el que se selecciona para agruparlo en una OperationCall y así se va realizando consecutivamente hasta reducir a una sola expresión, o si es imposible reducir, retornar un error debido a que se está utilizando un operador no definido en la sintaxis.

Traductores

Hasta este punto ya está descrito el modelo utilizado para representar las expresiones, se ha creado un lenguaje independiente del contexto para el cual se quiera utilizar, y lo suficientemente flexible para poder ser utilizado en diversos tipos de soluciones. Ahora es imprescindible crear una estructura que permita la traducción de este lenguaje a las necesidades específicas que se requieran.

Es evidente que los requerimientos de cada sistema específico pueden ser disímiles, por lo que solamente se definen dos interfaces básicas a implementar por los traductores.

Fig. 4. Interfaces para la traducción

El objetivo es definir dos conceptos fundamentales, el traductor y la traducción, para que puedan ser implementados luego, de acuerdo a los requerimientos específicos del problema.

Es importante tener en cuenta que una traducción no tiene por qué ser sólo texto, puede ser un programa compilado que sea capaz de extraer los datos de un fichero, una tabla de datos, o cualquier variante que se desee.

En **ConceptualMap.dll** se han implementado dos traductores, un TextTranslator que retorna un TextTranslation y un SQLExpressionTranslator que retorna un TextTranslation o un ConceptQueryTranslation para los localizadores. Con TextTranslator es posible convertir un objeto Expression en el texto correspondiente a la representación de la expresión.

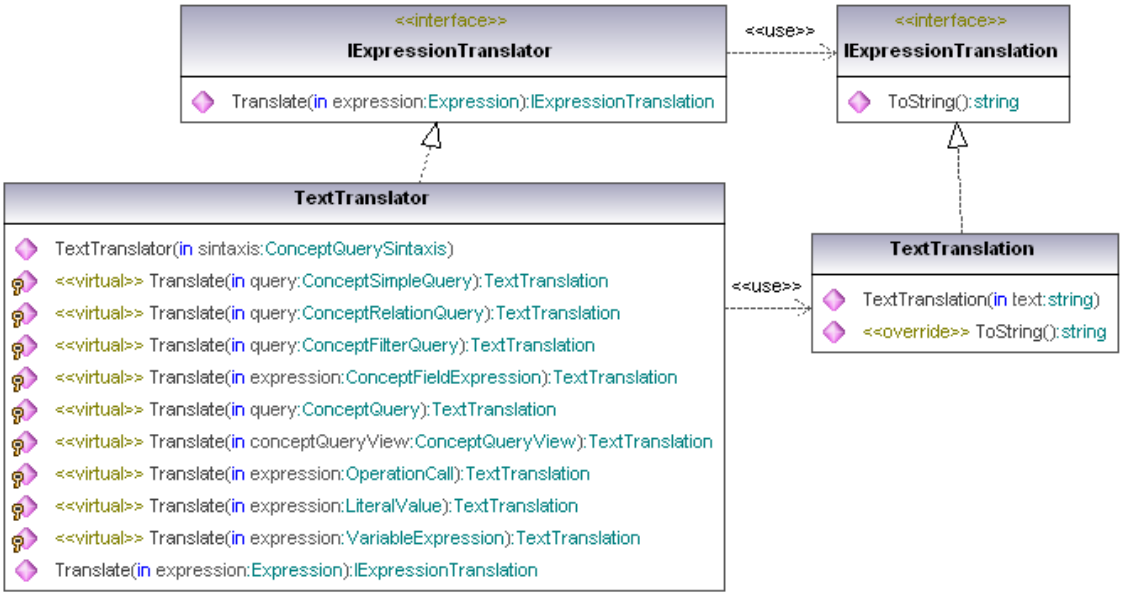


Fig. 5. TextTranslator

Traductor para base de datos relacionales

Con los componentes descritos hasta el momento ya es posible interpretar una expresión escrita en texto plano y crear un objeto derivado de la clase Expression que contenga toda la información en un modelo de objetos.

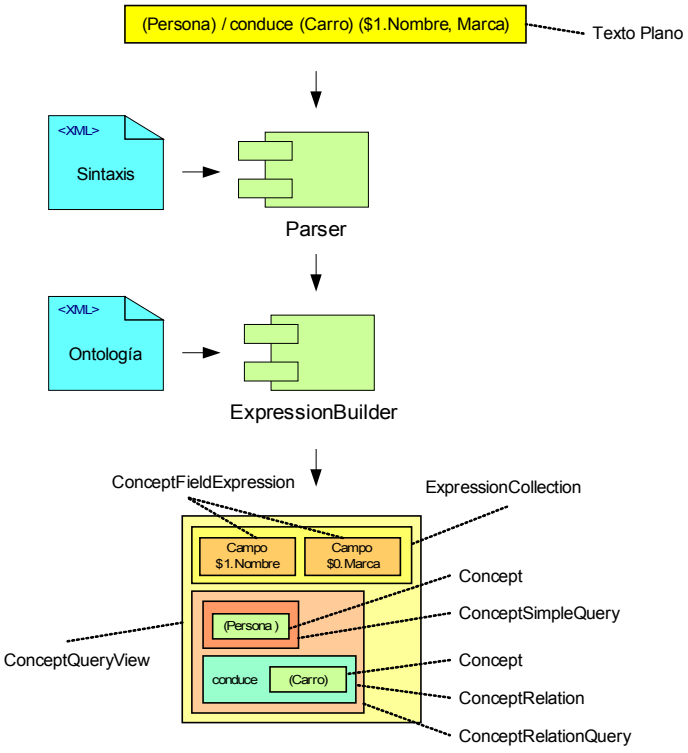


Fig. 6. Proceso de interpretación de una expresión

Un traductor indispensable es el traductor hacia sistemas de base de datos relacionales, debido a que son los sistemas de almacenamiento de datos más comúnmente utilizados por las aplicaciones. El traductor implementado en **ConceptualMap.dll** es capaz de traducir una expresión a una consulta SQL que es el estándar utilizado para el acceso a datos en las base de datos relacionales.

Es importarte tener en cuenta lo expresado anteriormente, donde se hace alusión a las diferencias existentes entre los gestores de base de datos. Una posible solución sería crear un traductor para cada gestor de base de datos, pero esto implicaría desarrollar varios traductores que prácticamente realizan la misma función. Siguiendo la filosofía de desarrollo llevada hasta el momento, la información particular de cada gestor es encapsulada en un objeto el cual será utilizado para generar el SQL de acuerdo con las necesidades específicas que se requieran.

Otra información necesaria para el traductor, es cómo se traduce cada concepto y relación al SQL concreto a ejecutar en la base de datos particular que se vaya a utilizar. Toda esta información es descrita en el documento de descripción conceptual de la base de datos (DCBD), el cual posee la siguiente estructura:

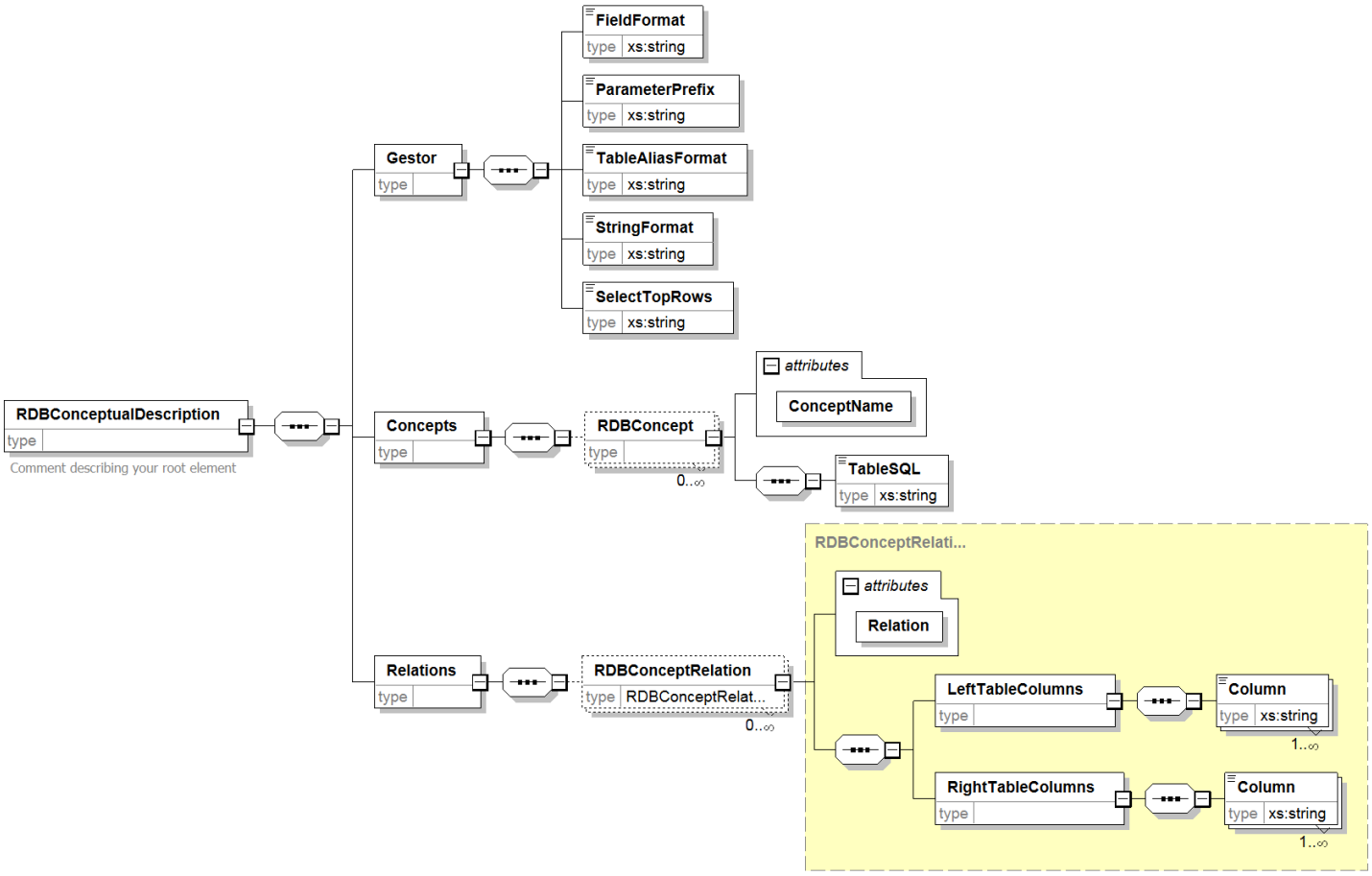


Fig. 7. Esquema de la descripción concptual de una base de datos

Un ejemplo de gestor de base de datos es el Oracle y la representación sería:

```
<Gestor>
        <FieldFormat>"{0}"</FieldFormat>
        <ParameterPrefix>:</ParameterPrefix>
        <TableAliasFormat>{0} {1}</TableAliasFormat>
        <StringFormat>'{0}'</StringFormat>
        <SelectTopRows>select * from ({0}) tbl where rownum &lt;= {1}</SelectTopRows>
</Gestor>
```

Cada concepto de la ontología debe poseer una descripción en el documento de descripción conceptual de la base de datos. Por cada concepto se coloca el nombre del concepto que es único entre todos los conceptos de la ontología, y la consulta SQL a la que corresponde en la base de datos, si un concepto corresponde a una sola tabla en la base de datos, y todos los campos tienen el

mismo nombre que la columna correspondiente en la tabla, no es necesario poner una consulta SQL, se puede colocar solamente el nombre de la tabla.

Ejemplo de descripción de un concepto para una base de datos:

```
<RDBConcept ConceptName="Estudiante">
        <TableSQL>
select
   p.IDPersona as [IDPersona],  p.Nombres as [Nombres],  p.Apellidos as [Apellidos], e.Número_Solapin as [Solapin]
   from DEstudiante as e inner join DPersona as p on e.IDPersona = p.IDPersona
        </TableSQL>
</RDBConcept>
```

Las relaciones entre los conceptos cuando son descritas en el documento DCBD pueden ser de dos tipos, las relaciones simples que son aquellas en las que uno de los dos conceptos posee un identificador del otro, una relación 1-N en la base de datos; y las relaciones con una tabla intermedia, las relaciones M-N. Estas relaciones son RDBConceptRelation y RDBInnerTableConceptRelation, la segunda es una extensión de la primera.

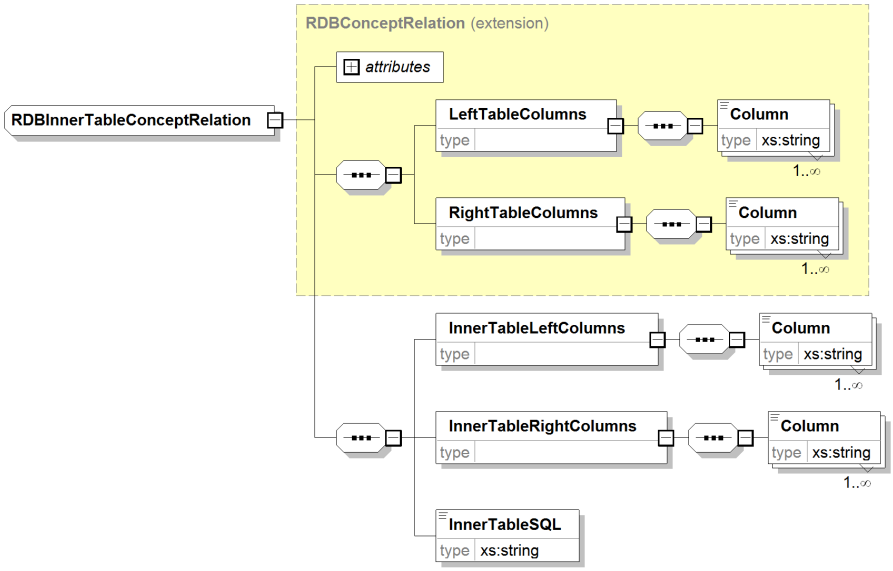


Fig. 8. Elemento RDBInnerTableConceptRelation

Con el componente SQLExpressionTranslator es posible utilizar las expresiones escritas en el lenguaje definido para acceder a cualquier base de datos relacional.

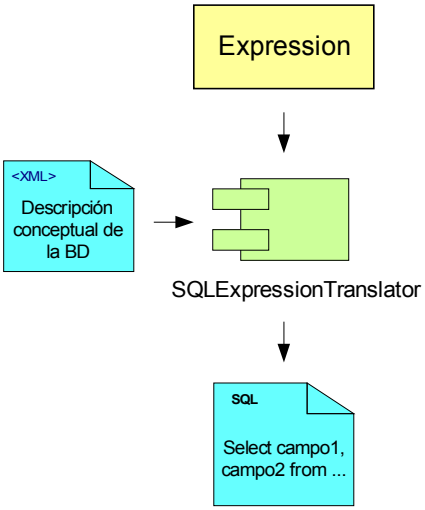


Fig. 9. SQLExpressionTranslator

Esta variante de acceso a datos implementada sobre las base de datos relacionales nos permite crear independencia entre el modelo de conceptos y el modelo relacional de la base de datos, orientando el acceso a datos a la conceptualización del negocio y no a la implementación de la base de datos.

Conclusiones

Este mecanismo de acceso a la información está orientado a reducir la complejidad inducida por la gran cantidad de variantes de acceso a datos existentes. Introduciendo además una organización conceptual de la información, que facilita el manejo de la misma tanto por las personas como por los sistemas informáticos. Una de las grandes potencialidades que brinda, es el permitir organizar y relacionar conceptualmente la información de un sistema, independientemente de donde esté persistida, desacoplando de esta forma los sistemas del mecanismo final de persistencia.

Bibliografía Consultada

Hermann Helbig. Knowledge Representation and the Semantic of Natural Languages. Alemania, Springer, 2006. 652.

XML Path Language (XPath) Version 1.0. 1999. http://www.w3.org/TR/xpath

XML Schema Part 0: Primer Second Edition. 2004. http://www.w3.org/TR/xmlschema-0/