# Sistema Manejador de Versiones

# para

# Esquemas Entidad-Relación

*Carla Demarchi, Valeria Irazabal, Alicia Revello, Raul Ruggia*
Instituto de Computación - Facultad de Ingenieria - Uruguay
email: ≅fing.edu.uy

*La modelarización conceptual de bases de datos no es un proceso lineal. Su naturaleza de prueba y error genera múltiples versiones antes de llegar a la solución definitiva del Esquema Conceptual. Un Manejador de Versiones es un sistema que permite organizar los conjuntos de versiones intermedios durante el desarrollo del producto.*

*En otras áreas tales como CAD (Computed-Aided Design) y SE (Software Engeneering) la administración de versiones ha sido ampliamente aplicada. No así en Base de Datos. Por lo tanto la motivación para construir un Manejador de Versiones para esquemas Entidad-Relación (ER) surge de esta carencia así como del interés por explotar propiedades de los esquemas ER que no se cumplen en el software general, por ejemplo las que permiten integrar y comparar esquemas ER.*

*Este artículo describe un sistema Manejador de Versiones para esquemas ER desarrollado en el contexto de un proyecto de grado. El sistema propuesto aplica los conceptos básicos del manejo de versiones (versión, derivación) e introduce otros, nuevos, orientados a los esquemas ER. Un ejemplo de éstos es la relación de inclusión entre esquemas, la cual organiza los conjuntos de versiones de acuerdo a las características de su estructura.*

*Dentro de un ambiente CASE, el Manejador de Versiones exporta primitivas para el manejo de versiones. Por lo tanto, centraliza la organización de los esquemas ER y unifica el desarrollo de los productos, inclusive con aquellos que fueron desarrollados con otras metodologías.*

## 1. INTRODUCCION

La Modelización Conceptual de Bases de Datos no es un proceso lineal sino de naturaleza exploratoria, el cual da origen a múltiples versiones de trabajo antes de alcanzar el esquema conceptual final de una base de datos. Cuando desarrolla un esquema conceptual, el diseñador explora diferentes alternativas de diseño y retiene algunas en estado intermedio. Esto le permite fijar puntos de diseño significativos así como retornar a estados anteriores si los subsiguientes no condujeron a un diseño mejor. Más aun, el diseñador habitualmente desea mantener la *traza* de las decisiones que tomó a lo largo del proceso de diseño, esta *traza* constituye información valiosa acerca del conocimiento aplicado en la resolución del problema.

Los Manejadores de Versiones son sistemas que permiten organizar el conjunto de versiones intermedias que se obtienen en el desarrollo de un producto. En la Modelización de Bases de Datos, el producto a ser desarrollado es un *esquema conceptual*, y las versiones intermedias serán esquemas que resultan de diferentes opciones de diseño y diferentes estados en su historia de diseño.

Este artículo propone un Manejador de Versiones para esquemas conceptuales Entidad-Relación (ER). Este sistema se inserta en un ambiente CASE para diseño de bases de datos, permitiendo que las diferentes herramientas hagan uso del mecanismo de versiones.

**Panorama del estado del arte.**

A pesar de las motivaciones existentes, los ambientes para Diseño de Bases de Datos carecen de mecanismos de manejo de versiones. Este problema ha sido ampliamente reconocido en la literatura [Jark 92][Roll 92] pero ha sido parcialmente resuelto a través de propuestas que se focalizan en aspectos de *traceado* mas eque en la organización de las versiones [Jark 92].

Por otro lado, en otras áreas como CAD (Computed-Aided Design) e Ingeniería de Software, el manejo de versiones ha sido considerado a través de Modelos de Versiones (Version Models) y de Sistemas [Katz 90][Tych 82][Estu 94]. Brevemente, un *Modelo de Versiones* define que es una *versión* en un contexto de trabajo, como se organizan y qué operaciones son aplicables sobre ellas. Estos aspectos son fuertemente dependientes del tipo de producto a ser desarrollado (p. Ej., circuitos VLSI, software, esquemas conceptuales). Un *Mecanismo Manejador de Versiones* implementa un *Modelo de Versiones*.

Los mecanismos de manejo de versiones también ha sido aplicados con otros objetivos diferentes de la organización de productos de diseño. Otros dos son: la optmización de espacio de almacenamiento a través de objetos *delta*, y mecanismos para asegurar la coherencia de una base de datos frente a operaciones de evolución.

En Ingeniería de Software los estudios realizados indican que el 30 % del tiempo de diseño es utilizado para desarrollar nuevo software y la mayor parte del tiempo modificar código existente.

En este contexto, la mayoría de los sistemas aplican conceptos introducidos en los mecanismos SCCS y RCS. En SCCS [Rock 75], un objeto (archivo) es una secuencia de *revisiones*, na para cada cambio en objeto. El sistema RCS [Tych 82] es una extensión de SCCS y considera que la evolución no es una simple secuencia de revisiones sino un árbol donde cada rama (*variante*) sigue su propia evolución como una sucesión de *revisiones*. Casi todos los productos comerciales siguen la solución propuesta por SCCS y RCS. Estos pueden clasificarse en generaciones.

**1ª generación: Basada en Archivos**

La primera generación de herramientas proveía lo que comúnmente se conoce como Control de Versiones. Estas eran herramientas que estaban basadas en archivos (meta archivos), en los que almacenaban los cambios realizados en archivos individuales. Los meta archivos guardaban los contenidos y los meta datos (nombre de usuario, etiquetas, etc.) por cada versión de archivo.

Los productos que se incluyen en esta primera generación son: SUN Microsystem Team Ware, Intersolv PVCS, MKS Source Integrity

Los dos últimos, agregaron sobre los meta archivos una capa de meta datos de proyectos y funcionalidades adicionales.

**2ª generación: Basada en Repositorios de Proyectos**

Son fáciles de identificar porque almacenan todo el proyecto y el archivo de metadatos en una base de datos (repositorio) que esta separada de los meta archivos.

Esta arquitectura traslada el foco del nivel de archivos al nivel del proyecto. Esto resulta en una mejora en el soporte del desarrollo en paralelo, la coordinación del equipo y provee la infraestructura necesaria para llevar a cabo el manejo de procesos.

Algunos de los productos involucrados: Microsft Visual SourceSafe, IBM CMVC, TrueSoftware, Aide-de-Camp/Pro, Platinum CCC, SQL Software PCMS

**3ª generación: Basada en la transparencia de archivos**

Una restricción de las herramientas que pertenecen a la segunda generación es que los archivos que controlan no pueden ser accedidos desde otras herramientas, a menos de ser copiadas desde el repositorio. Esto puede traer como consecuencia la proliferación de copias locales y el riesgo de sobrescribir el repositorio. La tercera generación de herramientas precisamente agrega la "transparencia de archivos".

Los productos son: Aria ClearCase, Continuus Continuus/CM

ClearCase implementa la transparencia de archivos con un sistema de archivos (MVFS) que intercepta las llamadas de acceso a los archivos (abrir(), leer(), etc.) y las re dirige al repositorio.

Continuus/CM provee un acceso directo a los archivos creando uniones desde el área de trabajo del usuario a sus directorios ocultos.

En los sistemas de CAD, el énfasis se pone sobre la estructura de los objetos compuestos y en su versionamiento desde una perspectiva de base de datos. Un objeto de diseño es un agrupamiento coherente de componentes manejado como una unidad. Un survey sobre estos sistemas puede encontrarse en [Katz 90]. Además [Katz 90] propone tres aspectos hoy por hoy clásicos: la historia de versiones, configuraciones y equivalencia entre versiones.

Más recientemente, [Ahme 91] propone un mecanismo de manejo de versiones para bases de datos CAD. Las principales características son: (i) la distinción entre objetos *genéricos, versionados y no versionados*, donde los objetos genéricos tienen asociados un conjunto de versiones; (ii) la clasificación de los atributos de los objetos en *invariantes, significativos y no significativos*. Esta clasificación es la base para la creación automática de versiones: la modificación de un atributo invariante lleva a la creación de un nuevo objeto, mientras que la modificación de un atributo *significativo* lleva a crear una versión. En [Tale 93] se sigue un encare similar pero definiendo *atributos sensibles*, cuya modificación lleva a la derivación de una nueva versión.

Resumiendo, el manejo de versiones en Ingeniería de Software y CAD difiere por la naturaleza de los objetos modelados. En Ingeniería de Software, la granularidad más pequeña es la de archivo o procedimiento, y en general no hay una semántica real asociada a estos. Mientras que en CAD se trabaja con objetos estructurados que tienen una clara semántica asociada.

*Cuales de estás técnicas pueden ser aplicadas para Manejo de Versiones en la Modelización Conceptual de Bases de Datos ?*

A pesar que varios conceptos generales y modelos pueden ser tomados de las técnicas existetes, algunos no pueden ser aplicados sobre esquemas conceptuales (p. Ej., la noción de objeto compuesto). Por otro lado, interesa poder aplicar técnicas específicas al manejo de esquemas conceptuales como ser mecanismos de comparación e integración de esquemas.

El Sistema Manejador de Versiones propuesto en este artículo incluye un Modelo de Versiones y su implementación. El Modelo de Versiones adopta varios conceptos clásicos como ser las nociones de *derivación de versiones, y alternativas*, e introduce otros nuevos que son específicos a esquemas ER: una relación de *inclusión* para organizar las versiones según las características de las estructuras ER que permite definir criterios automáticas para derivación automática de versiones, y la aplicación de operaciones de integración de esquemas para resolver la fusión de versiones.

El Modelo de Versiones subyacente al sistema, se basa en dos conceptos: Conjunto de Versiones y Relaciones entre ellas. Una versión es un esquema ER. Las relaciones son dos, derivación e inclusión. Cada relación define un grafo dentro del conjunto de versiones. Existen operaciones sobre el conjunto de versiones, sobre los grafos y sobre las versiones. Por ejemplo: crear un conjunto de versiones, derivar una versión, borrar un subgrafo, crear una nueva versión, etc.

El sistema Manejador de Versiones esta implementado en C++ y su arquitectura es C/S/S (Servidor de Versiones y Servidor de operaciones sobre esquemas Entidad Relación).

El Servidor de Versiones provee de las primitivas para la creación, inserción y borrado de versiones. El mismo ofrece estas primitivas a través de una interface Remote Procedure Call (RPC). Se construyó una herramienta para manejo interactivo de versiones, la cual es un cliente del Servidor de Versiones. La misma se conecta al Servidor de Versiones como un cliente especializado. La comunicación RPC entre los clientes y los dos servidores se realizó de forma de separar cada proceso e independizar de la herramienta con que se conecte.

Finalmente, el sistema Manejador de Versiones formará parte de un ambiente CASE desarrollado en un ambiente universitario, para el diseño de sistemas de información.

El resto del artículo se estructura de la forma siguiente. La sección 2 presenta el Modelo de Versiones. La Sección 3 presenta el diseño e implementación del sistema. La Sección 4 presenta algunas conclusiones y perspectivas de trabajo futuro.

# 2. EL MODELO DE VERSIONES

El problema de administración del diseño es que los objetos compuestos pueden tener muchas alternativas si las componentes también tienen versiones.

Tradicionalmente los sistemas de datos administran una estructura regular de datos sin versiones de sus objetos componentes.

A pesar que la relación de composición es importante no es adecuada para modelar esquemas ER en un ambiente dinámico de desarrollo.

**Los objetos en el Modelo**

Un Modelo de Versiones esta basado en dos conceptos principales: Proyecto de Modelación Conceptual (CMP), y versiones pertenecientes al CMP. Mientras que las versiones modelan un objeto dado (esquema entidad relación, fuente, etc.) en un determinado paso del proceso de diseño, el CMP modela el conjunto de versiones cuyos elementos representan una aplicación del Sistema de Información.

Este Modelo de Versiones define que es una versión en el proceso de diseño, como se relacionan y organizan las mismas y que operaciones podemos aplicar sobre una versión o sobre el conjunto de versiones.

Una versión consiste de un identificador de versiones (dentro de un CMP), un nombre, el tipo de la versión, un estado de desarrollo y una referencia al ambiente de trabajo al que pertenece.

***Definición***: **VERSION**

Una versión es una quíntupla: <id, nombre, tipo, estado, ambiente>

donde:

- id: identifica a la versión en el conjunto de versiones
- nombre: es una etiqueta que nombra a la versión
- tipo: establece el tipo de la versión (ej: esquemas ER, biblioteca, fuente, etc.)
- estado: es el estado de la versión y puede ser {borrador, analizado, completo, validado}. El mismo indica si la versión es un borrador, si ha pasado un análisis sintáctico, si incluye todas las declaraciones requeridas para la integridad ó si ha sido validada de acuerdo a un conjunto de criterios de calidad.
- ambiente: pertenece a {público} U Nombre_Usuarios. La referencia al ambiente de trabajo establece si la versión pertenece a un ambiente privado o a uno público.

**Relación de derivación**

Una versión vd deriva de otra vs si ha sido creada como un producto de diseño subsecuente de vs. En un conjunto de versiones, cada versión se relaciona con las demás a través de esta relación.

La relación de derivación representa si una versión modela una nueva solución o si deriva de una ya existente, además de mantener la evolución histórica de los objetos diseñados.

La relación de derivación define un árbol, llamado ARBOL DE DERIVACION, donde los nodos son las versiones del CMP. El árbol de derivación también es denominado ARBOL HISTORICO DE VERSIONES porque mantiene un histórico de la evolución de las versiones.

Es importante remarcar que la relación de derivación no implica que exista alguna relación semántica entre las versiones.

Si vd deriva de vs significa que:

1. Ambas pertenecen al mismo conjunto de versiones (CMP)
2. La versión derivada vd ha sido construida luego de su origen vs.
3. El diseñador ha decidido unir vd como el siguiente a vs en la secuencia de diseño en el modelo conceptual.

***Definición***: **relación de derivación**

La relación de derivación en un conjunto de versiones, se define de la siguiente manera:

$$\overset{d}{\rightarrow} \subseteq (V \times V)$$

$$v1 \overset{d}{\rightarrow} v2 \Leftrightarrow \text{v2 es derivada a partir de v1}$$

La relación $\overset{d}{\rightarrow}$ satisface las siguientes propiedades:

$$- (\forall\ v1, v2 \in V)\ (v1 \overset{d}{\rightarrow} v2 \Rightarrow \neg\ v2 \overset{d}{\rightarrow} v1)$$ Anti-Simétrica y Anti-Reflexiva

$$- (\forall\ v1, v2, v3 \in V)\ (v1 \overset{d}{\rightarrow} v2 \wedge v3 \overset{d}{\rightarrow} v2 \Rightarrow v3 = v1)$$ Único Ancestro

La relación de derivación define un árbol donde el conjunto de nodos es *V* (el conjunto de versiones) y los arcos corresponden a los pares de versiones relacionadas a través de

la relación $\overset{d}{\rightarrow}$.

**Relación de Inclusión**

Diremos que *v´ incluye a v* si las entidades y relaciones definidas en v tiene un correspondiente en v´

Para determinar la correspondencia entre dos versiones definimos la función φ.

Sea los esquemas ER S, S´ que tienen asociados un conjunto de entidades E y un conjunto de relaciones R

La relación de inclusión entre los dos esquemas ER se define:

$$S' \supseteq S \Leftrightarrow ((\forall\ E \in E)(\exists\ E' \in E)(\varphi(E,E')) \wedge (\forall\ R \in R)(\exists\ R' \in R)(\varphi(R,R')))$$

De esta forma la función φ queda definida a partir de los nombres de las entidades y relaciones de los esquemas ER.

Para lo cual deberíamos seguir un padrón de nomenclatura para los objetos que pertenezcan al mismo, de forma de que esta función pueda determinar las relaciones existentes

***Definición***: **relación de inclusión**

Dado un esquema ER la relación de inclusión $\supseteq$ en un conjunto de versiones $\sqrt{}$ se define:

$$\rightarrow \subseteq ( \sqrt{} \times \sqrt{} )$$

donde:

$$v2 \rightarrow v1 \quad \Leftrightarrow \quad (v2.esquema \ \ v1.esquema \wedge v1.id \neq v2.id \ \wedge$$
$$\neg (\exists v \in \sqrt{}) \ (v.id \neq v1.id \ \wedge v.id \neq v2.id \wedge \ v2 \rightarrow v \ \wedge v \rightarrow v1 ) )$$

La $\rightarrow$ relación define un grafo dirigido $<\sqrt{}, A>$ donde $\sqrt{}$ es el conjunto de nodos y A es el conjunto de nodos relacionados a través de $\rightarrow$.

La relación de inclusión define un grafo denominado GRAFO DE INCLUSION. Dicho grafo nos brinda una visión acerca de las características de los esquemas ER incluidos en las versiones. Permite al diseñador identificar las versiones cuyos esquemas tienen características en común.

La relación de inclusión se calcula cada vez que el conjunto de versiones (CMP) es modificado.

*Example:* *A CMP and its Inclusion Graph.*

The figure shows the **`Professor CMP`** (cf. Section 5.2). Its version set is viewed in the *Derivation Tree* form. The *derivation relations* that hold are:

**`{Profs_Teachers`**
**`Profs_Department,`**

**`Profs_Projects_Emps`**
**`Profs_Projects,`**

**`Profs_Projects_Emps`** $\xrightarrow{\supseteq}$
**`Profs_Projects }`**

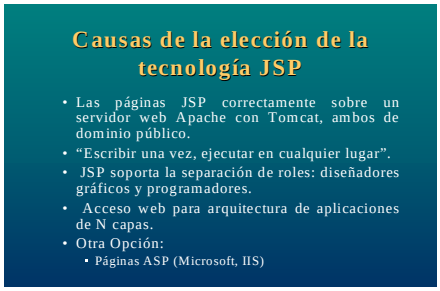


*Figure 7.2. The inclusion relationship in* `Professor` *CMP.*

**Relación de Construcción**

Modela como las versiones han sido construidas a partir de otras versiones. Una relación de construcción entre dos versiones v1 y v2 a través de una expresión de tipo <exp> significa que la versión v2 ha sido construida a partir de v1 aplicando <exp>. Esta relación se aplica en la reutilizacion de componentes.

***Definición***: **relación de construcción**

Consideramos un conjunto de CMP y un conjunto de componentes reusables *RSC*.

$\rightarrow \subseteq ((\ \surd ids \cup\ RSCids\ )\ x\ Esq\_exp\ x\ \surd ids\ )$

$o \rightarrow v\ \Leftrightarrow (v.esquema = exec(exp)\ \Rightarrow\ o \in ref\_schema\ (exp)\ )$

donde:

- √ides es la union del CMP-name y version_id que identifican de forma única a una versión en el conjunto de CMP.
- *RSCids* es el conjunto de identificadores de los esquemas reusables.
- *ref_schema (exp)* es una función que devuelve el conjunto de esquemas referenciados por exp.

Dado un conjunto de CMP una restricción de construcción es un predicado de la siguiente forma:

$$construccction\_constraint(o,exp,v)\ \Rightarrow\ (\text{Para todo de Vi})\ o \rightarrow v$$

La relación de construcción se declara sobre versiones por lo que todas las versiones que se deriven a partir de esta no heredan esta relación de construcción. Por lo que la creación y la derivación de nuevas versiones no afecta esta relación.

Las tres relaciones definen un espacio de visualización y organización de las versiones:

- la relación de derivación representa las ramas de diseño y la evolución histórica de las versiones
- la relación de construcción representa como las versiones han sido construidas.
- la relación de inclusión relaciona versiones con esquemas ER.

**CMP**

Un CMP es el resultado de modelar una situación del mundo real a través de un modelo conceptual de esquemas ER

CMP = <nombre,V,Vc,$\rightarrow$,$\Rightarrow$)

Donde

- nombre:etiqueta que lo identifica
- V: conjunto de versiones del CMP que modelan la situacion.
- Vc: versión corriente del conjunto.

- $\rightarrow$: relación de derivación entre las versiones de V.
- $\Rightarrow$:relación de inclusión entre las versiones de V.

**Operaciones**

Las operaciones en el Modelo de Versiones permiten manipular versiones y conjuntos de versiones. A continuación presentamos las posibles operaciones que se podrían aplicar a un Modelo Conceptual Genérico de Versiones. Las agrupamos en tres categorías: operaciones sobre un CMP, operaciones sobre versiones y operaciones sobre grafos de versiones.

**Operaciones sobre un CMP**

**`create_CMP ( string nombre_CMP):`**

Un CMP es creado a través del operador *create_CMP*, el cual crea el conjunto de versiones como un conjunto vacío.

**`delete_CMP (string nombre_CMP)`**

Elimina un Modelo Conceptual Genérico incluyendo su conjunto de versiones.

**`open_CMP`**

Se utiliza para abrir un CMP

**`save_CMP (string nombre_CMP)`**

Salva un CMP

**`close_CMP (string nombre_CMP)`**

Cierra un modelo conceptual generico.

**Operaciones sobre versiones**

**`create_version (CMP c, tipo_version t, string nombre)` $\rightarrow$ `id_versión`**

La creación de una versión en un conjunto de versiones se logra por medio del operador *create_version*. La nueva versión será la raíz de un nuevo árbol de derivación. El operador retorna el identificador asignado a la nueva versión.

**`derive (CMP c, tipo_version t, string nombre)`$\rightarrow$ `id_versión`**

La creación de una nueva versión derivada de una ya existente se realiza a través del operador *derive*. El operador retorna el identificador asignado a la nueva versión.

**`delete_version (CMP c, Version v)`**

El operador *delete_version* borra versiones individuales. Las versiones derivadas de la misma se "cuelgan" de la fuente de derivación de la versión borrada.

**`rename_version (string nombre, Version v)`**

Para renombrar la versión corriente se utiliza la función *rename_version*.

**`read_version ()` $\rightarrow$ `datos_versión`**

Para leer datos de la versión corriente se utiliza la función *read_version*.

**`ancestors (id_version) → lista_versiones`**

Para obtener los ancestros de una versión, en el árbol de derivación, se utiliza el operador *ancestors*.

**Operaciones sobre grafos de versiones**

**`path (id_version) → lista_versiones`**

Para obtener el camino que hay desde la raíz a una versión dada se utiliza el operador *path*.

**`sub_tree (id_version) → lista_versiones`**

El operador *sub_tree* permite ver la sub_rama que "cuelga" de una determinada versión.

**`sub_branch (id_version_from, id_version_to) → lista_versiones`**

Para obtener la sub_rama que hay entre dos versiones existe el operador *sub_branch*.

**`del_sub_tree ()`**

El operador *del_sub_tree* borra las versiones en el árbol de derivación con raíz la versión corriente. La versión corriente no es borrada.

**`del_path ()`**

El operador *del_path* borra los ancestros de la versión corriente en el árbol de derivación. La versión corriente no es borrada.

Existen dos problemas concernientes a las operaciones de modificación: la estrategia para la derivación automática y la preservación de las restricciones de construcción.

La estrategia para la derivación automática de versiones establece que hacer cuando el esquema esquemas ER de una versión es modificado: si la versión es actualizada o si una nueva versión es derivada. La elección de la estrategia se basa en dos criterios fundamentales:

- Evitar la pérdida de información. La modificación de esquemas ER en una versión podría implicar el borrado de estructuras de esquemas ER. Para evitar la pérdida de información una nueva versión debería ser derivada incluyendo el nuevo esquema.
- Evitar la proliferación de versiones. Derivar una nueva versión por cada operación conduciría a una proliferación de versiones no deseada.

En nuestro Modelo de Versiones aplicamos la siguiente estrategia: si el esquema modificado no incluye al esquema original, entonces se deriva una nueva versión.

Con esta estrategia si se borra una estructura de esquemas ER entonces se deriva una nueva versión, y el esquema inicial que incluye la estructura borrada se preserva en el ambiente.

Controlar la proliferación de versiones implica el borrado de versiones que no son más esenciales en el conjunto de versiones. En el contexto de esquemas ER es posible definir un criterio para implementar este control: borrar las versiones que están incluidas en alguna otra.

## 3. DISEÑO E IMPLEMENTACION

# 1. Arquitectura

La conexión se realiza de la siguiente forma:

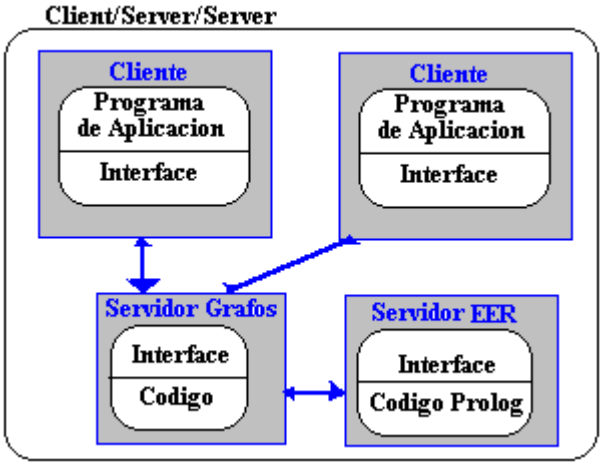


Figura ...

La herramienta cliente es el módulo donde se implementa toda la interface gráfica. Todas las operaciones invocadas acceden al servidor de grafos. La implementación realizada es la siguiente: primeramente al comenzar el Cliente se comunica vía RPC al Servidor de Grafos y este se conecta al Servidor de esquemas ER. Una vez establecida la comunicación con el Servidor de Esquemas esta no finaliza y permanece abierta para optimizar las llamadas futuras a cualquier otra operación exportada por el mismo. Cuando el cliente finaliza el Servidor de Grafos cierra la conexión con el Servidor de esquemas ER.

# 2. Los Servidores

### Servidor de Grafos

El Servidor de Grafos esta implementado en C++. Fue implementado a través de la clase **`CMProjects`** que mantiene información de las versiones.
La estructura y las relaciones entre las versiones se implemento utilizando una biblioteca para Tipos Abstractos de Datos LEDA la cual nos permitió definir grafos paramétricos. Las versiones son los nodos de dichos grafos y las relaciones de Derivación e Inclusión son los arcos. El conjunto de nodos se encuentra almacenado una sola vez, mientras que el tipo de aristas se instancia en dos: derivación e inclusión. Las operaciones de administración de los distintos grafos están implementadas en C++, las operaciones aplicadas a la estructura de cada nodo (operaciones especificas de los esquemas ER) y la relación de inclusión están implementadas en Eclipse-Prolog.
La clase **`CMProjects`** se encarga de todas las operaciones de administración de los grafos y se comunica con las rutinas de manejo de Esquemas Entidad Relación a través de la función **`Oper`** vía el mecanismo de comunicación RPC**.**
Cada versión se representa por medio de la clase **`Version`** la cual permite la creación y modificación de los nodos. La información almacenada de cada versión es: estado, identificador, nombre, valor. El significado de cada uno de ellos fue explicado anteriormente.

La clase **Arc** permite la creación y administración de las relaciones de los distintos Grafos (Derivación, Inclusión) Por eficiencia en los algoritmos de recorrida del grafo tanto en la relación de Derivación como en la de Inclusión los arcos son dirigidos.

La clase **V_System** es la clase que administra el conjunto de grafos que son accedidos por los distintos clientes. En esta clase se implementaron las funciones de carga de los grafos que serán enviados en estructura de datos a los distintos clientes cada vez que estos lo soliciten para su posterior visualización.

```
class V_SYSTEM (List(CMProjects))

class CMProjects {GRAPH(Version,Arc)}

class Version (identif,valor,name,status)

class Arc (identif,Tipo)

class Derivation_From,Derivation_to

class Inclusion_from,Inclusion_to
```

**Servidor de Manejo de esquemas ER**

La estructura de los esquemas ER se encuentran almacenados en *facts* Prolog.Por lo que la comunicación entre los servidores se realiza a través de RPC. El servidor de manejo de esquemas ER describe las operaciones exportadas, que son el conjunto de operaciones que tiene que acceder el Servidor de Grafos. El Servidor se conecta con los esquemas a través de un ***pipe UNIX*.**
De la especificación que se partió se tenía una comunicación a pedido entre el Cliente y el módulo administrador de los esquemas ER. Es decir que cada vez que el cliente tenia que efectuar una operación sobre un esquema ER, llamaba a una función que ejecutaba el predicado (PROLOG). Para potencializar la modularidad de la herramienta se decide que la conexión debe ser independiente del modulo que administre un nivel inferior del grafo (esquemas ER) por lo que vía RPC se decide la implementación de un Servidor de esquemas ER. Además el Servidor de grafos será un cliente de este servidor por lo que tendríamos una comunicación Clientes –Servidor de Grafos –Servidor de Esquemas. Además esta comunicación ya no es mas a pedido sino que el canal de comunicación esta siempre establecido por una eventual consulta y performance de la misma

## 3. El Cliente

Una de las principales funciones del Cliente es la interface gráfica para los grafos de versiones, que permiten a los usuarios visualizar y manipular los nodos de los grafos de derivación e inclusión de forma fácil y amigable.
Se buscó una herramienta amigable, esto es de fácil manejo para el usuario y el programador. Como ya se dijo anteriormente cada CMP define dos árboles: el árbol de derivación y el árbol de inclusión. Estos grafos son desplegados por la interface gráfica de cada cliente del *Manejador de Versiones*. Se pretendía una herramienta que permita modificar la estructura de los grafos

(administrar la organización de los nodos en la pantalla y grabar dicha organización para una posterior utilización).
Uno de los principales requerimientos de la aplicación fue que la organización de los nodos de los diferentes grafos pudiese ser administrada por el usuario. Esto implica que los nodos tengan la propiedad de poder "arrastrarse" con el ratón hacia diferentes posiciones. Una vez elegida la distribución de los mismos, esta se almacena en un archivo de forma tal que la próxima vez que se acceda a dichos grafos la distribución sea la misma que la de la última vez que se trabajó con él. Esta distribución depende de cada cliente, o sea existe un archivo con las posiciones de los nodos de cada grafo por cada cliente que se conecte al Servidor de Grafos.
Las estructuras implementadas en sus distintas capas se comunican de la siguiente manera:

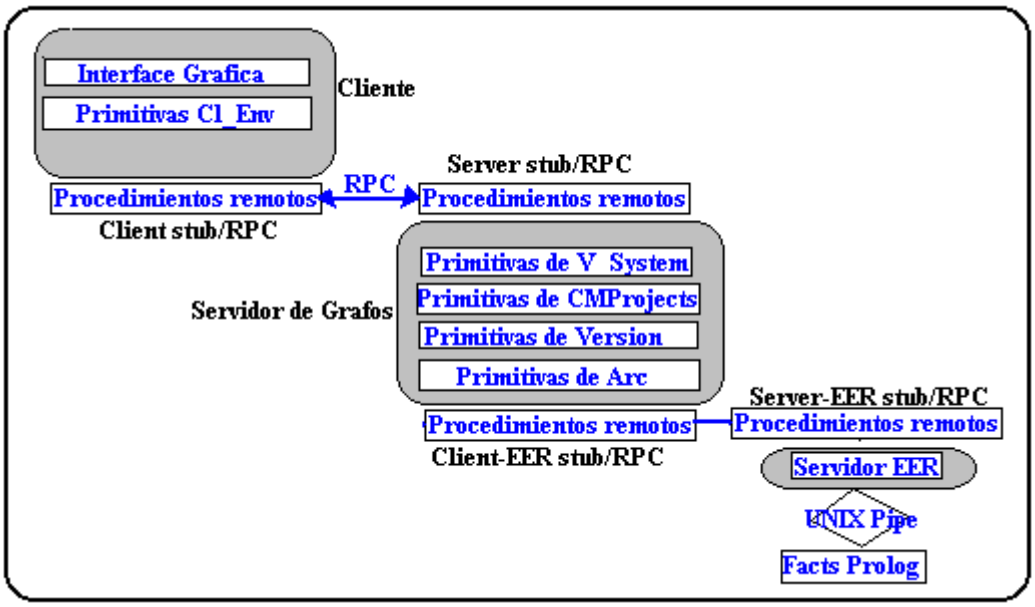


*Figura ...*

Al tener uno o mas procedimientos en una máquina remota es necesario agregar código entre la llamada al procedimiento y el procedimiento remoto. Del lado del cliente, el nuevo código deberá convertir los argumentos y traducirlos a la representación independiente de la máquina, crear el mensaje de llamada, enviar el mensaje al procedimiento remoto, esperar los resultados y traducir los valores resultantes a la representación nativa de la maquina del cliente. Del lado de los servidores, el nuevo código deberá aceptar un pedido RPC entrante, traducir los argumentos a la representación nativa del servidor, realizar el despacho del mensaje al procedimiento adecuado, crear un mensaje de respuesta traduciendo los valores a la representación independiente de la máquina y enviar el resultado al cliente. Para mantener la estructura del programa y aislar el código de manejo de RPC se agregan dos nuevos procedimientos que encapsulen completamente los detalles de comunicación. Estos procedimientos son llamados *stub procedures.*

Debido a la arquitectura de la aplicación el desarrollo fue modular y esto trajo como consecuencia un testeo modular. Luego de esto se realizaron pruebas de integración.
Los cambios realizados en cada módulo fueron los siguientes:
Servidor de Versiones: rediseñamos las clases y los miembros de las mismas, así como reprogramamos funciones que no estaban implementadas(Up, Ancestors, Del_all_sp_links, Del_includes, Del_included, etc.). Testeo de módulo.
Servidor de Esquemas: diseñamos e implementamos la conexión con el Servidor de Versiones. Testeo de la misma.

Para adecuarnos a la especificación dada y a las modificaciones realizadas fue necesario la unificación de clases (C++) existentes así como su renombre, definición de nuevas funciones miembros etc.
Debido a que primariamente estabamos trabajando en un ambiente C/S y era necesario probar las nuevas funcionalidades implementadas, las etapas de testeo realizadas fueron las siguientes:

1. Debug del código sin introducir problemas de red.
2. Agregamos transporte RPC para testear Cliente/Servidor
3. Pruebas sobre la Red

El propósito fue realizar un testeo incremental:

1. Se probaron funcionalidades del servidor en un ambiente sin conexión RPC. En un espacio de direcciones local, para lo cual compilamos el main() del cliente junto con el servidor de grafos.
2. Una vez que se asegura que el cliente y el servidor están correctos en un "modo" local, agregamos raw-functions de comunicación. Estas funciones ejecutan RPC en la máquina local en el espacio de direcciones asignado.
3. Finalmente se separa el código, probando las funcionalidades en un ambiente remoto.
Estas etapas se volvieron a realizar una vez que se finalizó el servidor de esquemas ER. Por último se testeo la comunicación C/S/S

## 4. CONCLUSIONES

El *Manejador de Versiones* propuesto provee un Modelo de Versiones basado en versiones de tipo esquemas ER, las cuales se agrupan en el Proyecto de Modelación Conceptual y se relacionan entre ellas a través de las relaciones de derivación, inclusión y construcción.
Este *Manejador de Versiones* permite organizar los esquemas ER y centralizar su organización independientemente de las técnicas que se utilicen para desarrollarlos.

Su estructura consiste de un Servidor de Versiones, el cual exporta primitivas del CMP y almacena la estructura de los grafos y los datos de las versiones, un Servidor de Esquemas que exporta primitivas de administración de esquemas ER y una Herramienta para el manejo de Versiones, la cual permite manejar los grafos de versiones dentro de un ambiente gráfico amigable.

El Manejador ya se encontraba especificado así como sus principales funcionalidades. Se realizaron varios cambios en cada uno de sus módulos.
Dentro del Servidor de Versiones se realizó un rediseñamiento de clases y miembros de las mismas, y se implementaron funciones que estaban especificadas pero no implementadas.
En el módulo del Servidor de Esquemas se diseñó e implementó la conexión con el Servidor de Versiones.
Para el manejo de las versiones se buscó una herramienta amigable tanto para el usuario como para el programador, eligiéndose la biblioteca Ilog Views. Con la misma se diseñó la interface gráfica en su totalidad.
Todos estos cambios fueron testeados en forma modular.

Dentro de un ambiente CASE el *Manejador de Versiones* exporta primitivas para el manejo de versiones como por ej. creación, modificación, lectura, consultas, las cuales pueden ser llamadas

por las demás herramientas. Dentro de este ambiente se comunica con el resto de las herramientas por medio de llamadas remotas, utilizando RPC y archivos asccii.

Las principales contribuciones de este proyecto son las aplicaciones de la comparación de esquemas ER y las técnicas de manejo para manipular versiones. La relación de inclusión se basa en la comparación entre esquemas ER al igual que el criterio para la derivación automática de versiones, la cual permite decidir cuando derivar una versión o cuando modificarla.

## Trabajo Futuro

### Concurrencia

Las operaciones realizadas vía RPC son sincrónicas es decir el proceso cliente esta bloqueado hasta que el proceso del servidor halla finalizado. En alguno casos esto puede no ser aplicable por lo que seria necesario incluir procesos threads.Un proceso normal puede incluir varios threads cada uno comportándose como un proceso normal desde el punto de vista de uso de CPU, todos los procesos threads del mismo proceso comparten el mismo espacio de memoria.
Para implementar esta solución la aplicación cliente inicia las llamadas RPC en un proceso threads y luego continua su ejecución.

### Proliferación de Versiones

La estrategia utilizada en este Manejador para la proliferación de versiones es derivar una nueva versión cuando el esquema modificado no incluye al inicial. Con esto evitamos la pérdida de información.
Consideramos que se podrían definir nuevas estrategias para este problema y realizar un estudio de las mismas para poder determinar cual es la mas adecuada. Por ejemplo se podría considerar determinadas características de las versiones como significantes, y si alguna de ellas es modificada entonces se debería derivar una nueva versión.

### Relación de Construcción

La relación de construcción modela cómo las versiones han sido construidas a partir de otras aplicando operaciones de esquema. En esta implementación del *Manejador de Versiones* no se trabaja en el ambiente gráfico con el árbol de construcción, por lo cual en el futuro sería interesante poder implementarlo y que el usuario pueda trabajar con el mismo de la misma manera que lo hace con el árbol de derivación y el árbol de inclusión.