Instituto de Computación
Facultad de Ingeniería
Universidad de la República Oriental del Uruguay

PROYECTO DE Taller 5
de la Carrera Ingeniería en Computación

Titulo del Proyecto:

Manejador de Versiones
de Esquemas Entidad-Relación

Presentado por:

Carla Demarchi
Valeria Irazábal
Alicia Revello

Defendido en 1997

El proyecto fue supervisado por:

Raul Ruggia

Manejador de Versiones de Esquemas Entidad Relación

Carla Demarchi
Valeria Irazábal
Alicia Revello

Contenido

Tabla de Ejemplos

Tabla de Figuras

1. INTRODUCCIÓN

1.1. Idea General

La naturaleza iterativa e indagatoria del proceso de diseño conduce a dos aspectos importantes en el diseño de objetos: los objetos son usualmente complejos por ser el resultado de unir componentes; pueden haber muchos diseños alternativos, llamados versiones, de un determinado objeto.
El problema de administrar el diseño surge a partir del hecho de que los objetos compuestos pueden tener múltiples alternativas de configuración si se toman en cuenta las versiones de cada componente.
Necesitamos un sistema unificado el cual incorpore estos requerimientos de diseño y provea una herramienta para manejar objetos compuestos de forma eficiente, además de administrar las versiones.

¿ Qué es la administración de versiones?

Es un conjunto de programas y procedimientos que gestiona ordenadamente el cambio de circuitos, datos, programas, etc.
Proporciona las siguientes características:

- Es un método para mantener información
- La habilidad de realizar sobre la información, cambios controlados
- Un camino para rastrear las modificaciones y saber quien las llevó a acabo
- Uno o más métodos para comparar entre dos versiones
- Provee un método de trabajo dentro de un equipo de desarrollo
- Las modificaciones se realizan en forma ordenada, ahorra tiempo de desarrollo
- La propiedad de mantener un histórico de los cambios
- Se puede generar de forma rápida y sencilla una lista de los cambios efectuados
- Es sencillo restaurar un mal diseño de un objeto, volviendo a una versión anterior
- Construir nuevos objetos a partir de otros ya existentes

En áreas tales como CAD (Computer-Aided Design) y SE (Software Engineering) la administración de versiones ha sido ampliamente aplicada. En los últimos años existió un auge en la investigación y desarrollo de sistemas para ayudar y controlar los esfuerzos de diseño de una gran variedad de productos dentro de las áreas que abarca la ingeniería, tales como circuitos VLSI, partes mecánicas, sistemas de software, etc.
El control de versiones ha evolucionando ampliamente en los últimos 20 años. Las herramientas han ido desde el versionamiento orientado a archivos hasta sistemas basados en repositorios que manejan proyectos y soportan ambientes de desarrollo en equipo.

El *Manejador de Versiones* implementado permite organizar conjuntos intermedios de modelos conceptuales que reflejan diferentes opciones de diseño según un modelo llamado Modelo de Versiones (Ver Sección 3). Nuestro proyecto consiste en la construcción de un *Manejador de Versiones* que permite organizar los Esquema Entidad Relación (esquemas ER).

El proceso de diseño del Esquema Entidad Relación de un problema se le conoce como Modelado Conceptual. A medida que se avanza en un proyecto (aplicando los paradigmas de la Ingeniería del Software), se experimentan con diferentes instancias de esquemas Entidad Relación que reflejan diferentes opciones y estados del ciclo de producción, creándose múltiples versiones de trabajos, de los cuales una será la versión final (ver Figura 1).

La Modelización Conceptual es la primer fase en el desarrollo de una aplicación de base de datos. Este es un proceso complejo que consiste principalmente en tres actividades:

- Adquisición del conocimiento de la aplicación
- Conceptualización de dicho conocimiento en forma de entidades y relaciones para construir el esquema conceptual
- Validación del esquema conceptual con respecto a los requerimientos del usuario y con respecto a criterios de calidad dependientes del modelo de datos

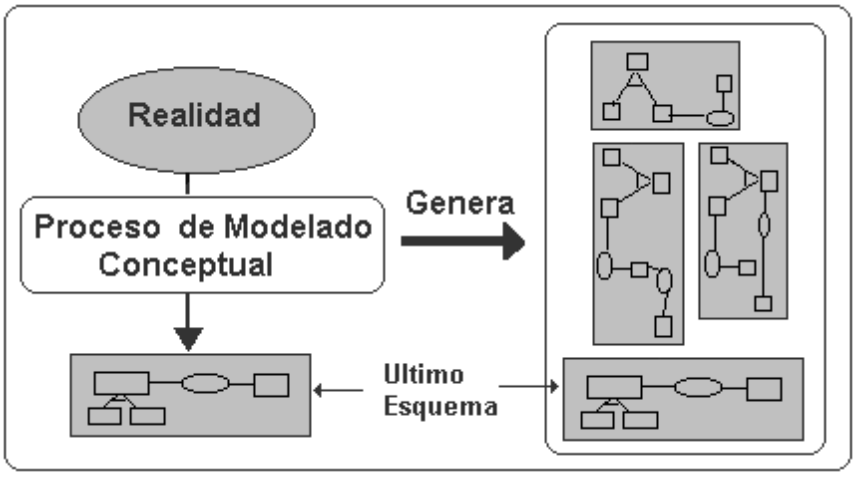


Figure 1: Manejador de Versiones

En este proyecto se implementó un *Manejador de Versiones* que aplica conceptos básicos del manejo de versiones (versión, derivación) e introduce otros nuevos que son específicos de 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. Una aplicación de esta relación es la implementación de mecanismos de control de proliferación 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 tres: derivación, inclusión y construcción.

La relación de construcción es la que define a nuestro *Manejador de Versiones* como una

herramienta orientada al desarrollo basado en la reutilización de componentes.

La reutilización ha sido ampliamente aplicada en otras áreas de la computación, en las cuales ha mostrado que contribuye a reducir los problemas de costo y efectividad en el desarrollo de software. En el contexto de Modelización Conceptual, a pesar de que ha sido reconocido como una estrategia prometedora, la reutilización ha sido poco explorada.

Las relaciones de Derivación e Inclusión definen dos grafos 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 las primitivas para la creación, inserción y borrado de versiones, a través de una interfaz Remote Procedure Call (RPC).
Se construyó otro servidor el cual se encarga del manejo interactivo de versiones. Es un cliente del Servidor de Versiones, 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.

Formará parte de un ambiente CASE (KHEOPS) desarrollado en un ambiente universitario, para el diseño de sistemas de información. El *Manejador de Versiones* integrado a KHEOPS a través de otra herramienta del mismo (Construcción Tool), permite incorporar al conjunto de versiones, esquemas desarrollados para la reutilización y trabajar con ellos de la misma forma que con otros esquemas (ver Figura 2).

Las herramientas que forman parte de la ingeniería del software suministran un soporte automático y semiautomático para los métodos: planificación y estimación de proyectos, análisis de requisitos, diseño de estructura de datos, codificación, prueba, mantenimiento, etc. Cuando se integran las herramientas de forma que la información creada por ellas pueda ser usada por otra, se establece un sistema para el soporte del desarrollo del software llamado CASE.

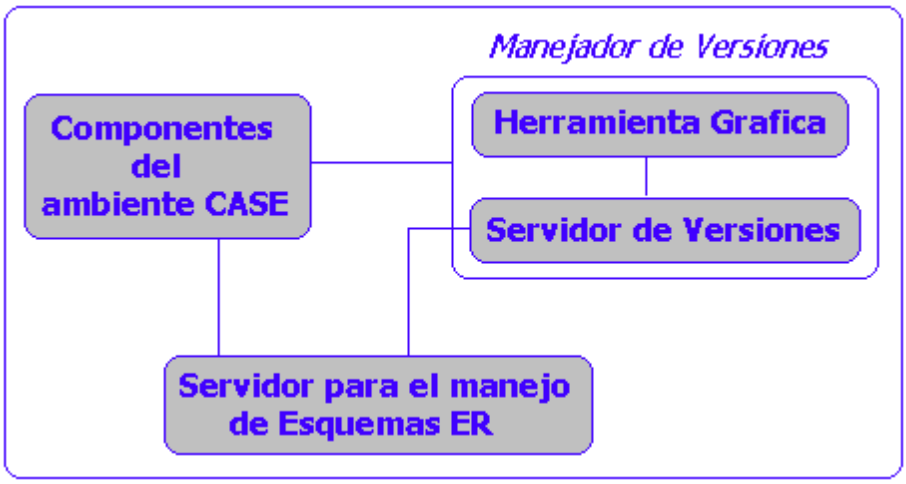


Figure 2: Integración al CASE

Constantemente se resuelven problemas semejantes, lo que nos lleva a plantearnos la reutilización de componentes.

Lo que básicamente se busca es:

- Construir aplicaciones extensibles y fácilmente modificables
- Construir software reusable
- Disponer de un modelo natural para representar al mundo

Para lo cual es necesario disponer de herramientas que administren los componentes utilizados en el desarrollo.

Gracias a la integración con el ambiente CASE, el *Manejador de Versiones* centraliza la organización de los esquemas ER y unifica el desarrollo de los productos, inclusive con aquellos que fueron desarrollados con otras metodologías. Este es un aspecto importante debido a la diversidad de técnicas y herramientas existentes y que, podrían llegar a usarse, en una aplicación de gran tamaño.

1.2. Trabajo realizado

Una primera versión del *Manejador de Versiones* se encontraba especificadoa así como las principales funcionalidades. La prototipación base estaba realizada en C++ en un contexto de SO UNIX ejecutando en Sun Sparc Workstation. Utilizaba bibliotecas y utilitarios tales como LEDA (representación de Tipos Abstractos de Datos) y RPC (procedimientos remotos).

El desarrollo del proyecto pasó por diferentes etapas. La lista básica de requerimientos era: una interfaz gráfica amigable y la implementación de una nueva arquitectura. Por lo tanto fue necesario realizar una investigación de las diferentes herramientas gráficas así como también del diseño y la codificación de la nueva arquitectura.

Las etapas en el proyecto fueron:

1. Análisis de Requerimientos
2. Búsqueda de herramientas
3. Diseño de arquitectura
4. Codificación y testeo de módulos
5. Integración y testeo de todo el sistema

Basándose en los requerimientos del proyecto se identificaron las diferentes tareas y se realizó una traza de trabajo.
Los requerimientos eran: una interfaz que permitiera la organización de los nodos en forma automática, esto es que no fuera necesario programar los movimientos; el renombre de las clases de forma que quedara coherente con la especificación; una arquitectura que permitiera la modularización de los diferentes componentes.

La segunda etapa involucró la búsqueda de una interfaz gráfica. Las restricciones para la búsqueda fueron: fácil uso para el programador y para el usuario así como una buena documentación. Del conjunto de herramientas investigadas seleccionamos ILOG VIEWS.

El diseño de la arquitectura se realizó de forma de lograr independencia entre los módulos del sistema. Para la comunicación entre ellos se utilizó RPC.

Como ya se explicó, los módulos que componen el *Manejador de Versiones* son: Servidor de Versiones, Servidor para el manejo de esquemas ER y el módulo cliente.
La codificación del módulo administrador del conjunto de versiones (o Servidor de Versiones) consistió en: definir nuevamente las clases y funciones para que quedara coherente con la especificación dada, además de las funciones que no estaban implementadas. Se unieron las diferentes clases, se renombraron las mismas así como sus métodos. El siguiente paso fue el testeo de sus funcionalidades en forma independiente.

Por otra parte, en el módulo cliente, se llevó a cabo la implementación de la interfaz y el testeo de sus funcionalidades.

El proceso de integración de los tres módulos se realizó en etapas. En una primera instancia se integraron el Servidor de Versiones con el Servidor para el manejo de esquemas ER (el cliente era simplemente un menú ASCCII). Luego de esto, se integro al sistema, el módulo del cliente con su interfaz gráfica. Una vez finalizada esta etapa se realizó un testeo de todo el sistema.

1.3. Aportes

El sistema implementado ofrece funcionalidades para el manejo de versiones en un ambiente CASE, permitiendo integrar los resultados obtenidos por diferentes herramientas de diseño.
A diferencia de otros manejadores de versiones, el aquí implementado debido a la semántica de los esquemas ER, permite definir la relación de inclusión lo cual evita la proliferación de versiones (ver pág. 23). Por lo cual esta herramienta se podría utilizar con todos aquellos objetos en los cuales se pueda definir formalmente una relación de inclusión.

El *Manejador de Versiones* es una herramienta Cliente/Servidor, aportando todas las ventajas de esta arquitectura como ser: presentar la información a través de interfaces gráficas, acceso simultáneo a varias fuentes de información en forma modular, integración adaptable y portable con otras herramientas y el entorno. Pero el punto mas fuerte de esta arquitectura es que permite integrar herramientas que hacen uso del mecanismo a través de una arquitectura Cliente/Servidor, accediendo de esta manera a los servidores. Por esta razón decimos que su arquitectura permite organizar los esquemas ER en forma independiente de la herramienta con la que fueron desarrollados.

Como vimos, el *Manejador de Versiones* se integrará a un CASE en el ámbito universitario, pero puede intregrarse con cualquier otra herramienta, debido a su arquitectura portable, a través de RPC y de un conjunto de primitivas públicas.

Otro de los aportes es la herramienta gráfica, que permite una rápida y amigable visualización y organización de los grafos de derivación e inclusión.

1.4. Estructura del documento

El documento esta organizado de la siguiente manera: en la sección 2: Descripción General, se describen las herramientas existentes, además de una comparación entre ellas y la solución planteada. A través de un ejemplo práctico se muestran las funcionalidades del *Manejador de Versiones*, la integración con el ambiente CASE y su arquitectura.

En la sección 3: Modelo de Versiones, se establece la base teórica de todos los términos y definiciones que lo componen.

En la siguiente sección: Diseño e Implementación, se detallan los tres módulos que componen el *Manejador de Versiones*: Servidor de Versiones, Servidor de esquemas ER y el módulo cliente; cuales son sus funcionalidades e implementación y como se realiza la comunicación entre ellos.

En la quinta sección se presenta una breve descripción de cada una de las herramientas utilizadas.

Finalizamos con las conclusiones y trabajo futuro.

El Apéndice muestra el código implementado que define algunas de las clases que componen los diferentes módulos del *Manejador de Versiones*. Luego de ésto, a través de un ejemplo, se muestra como se arma el árbol de llamadas.

2. SISTEMAS MANEJADORES DE VERSIONES

2.1. Visión general del estado del arte

Cuando se produce software, el cambio es una constante. 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. Esto implica que se pueden obtener varias versiones del software totalmente operativas a través de la vida útil del producto. Para permitir la solución de problemas, sin que sea una experiencia traumática, es necesario guardar todas las versiones operativas.

En el contexto de Ingeniería de software la mayoría de los sistemas aplican los términos Control de Versiones (VC) y Manejo de Configuraciones de Software (SCM).

El Control de Versiones es el proceso de administrar y mantener múltiples *revisions* o modificaciones de un mismo archivo. Contiene información acerca de cada *revision* del archivo, permitiendo a los usuarios de desarrollo obtener, modificar, o agregar una *revision* en una forma segura, organizada y consistente.

El Manejo de Configuraciones de Software es el proceso de administración de todo el desarrollo del proyecto.

No es sencillo clasificar las herramientas que soportan el VC y SCM, en términos de sus aspectos o lineamientos técnicos. Sin embargo es mucho más fácil si las observamos desde el punto de vista de la arquitectura subordinada y la orientación de sus metodologías.

2.2. Evolución de la arquitectura

1ª generación: Basada en Archivos

La primera generación de herramientas proveía lo que comúnmente se conoce como Control de Versiones. La unidad mínima de trabajo para estas herramientas es un archivo, sin importar su contenido. Estaban basadas en meta archivos (datos de datos), en los que se almacenaban los cambios. Los meta archivos guardaban los contenidos y los meta datos (nombre de usuario, etiquetas, etc.) por cada versión de archivo. Un meta archivo es un conjunto de *revisions* uno para cada modificación realizada al 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. Considera que la evolución no es sólo una secuencia de *revisions* en un único archivo sino que dos versiones de un archivo se distinguen por medio de números, indicando cual es el origen de cada archivo, en una estructura arborescente. Por ejemplo puede existir un árbol de versiones de la siguiente manera:

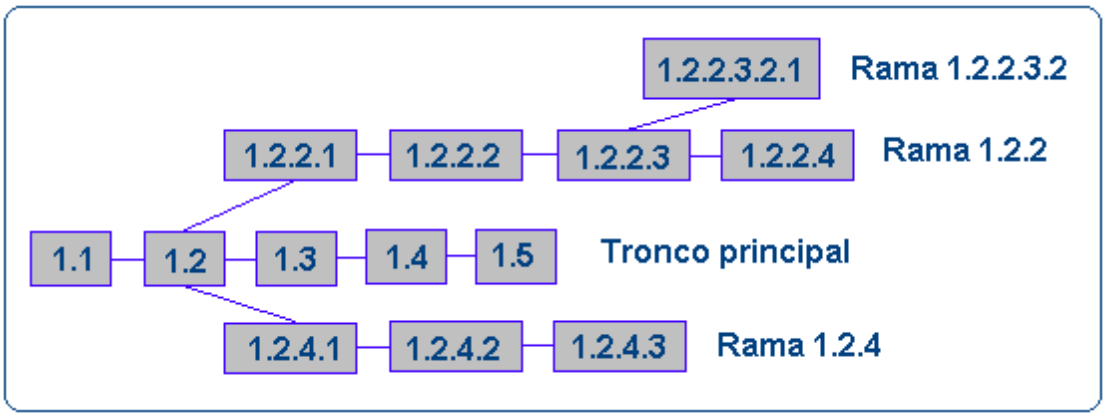


Figure 3: Árbol de versiones

En este caso la primera versión del archivo es la 1.1 y de ella derivan 12 versiones, 3 de ellas de la versión 1.2 (las cuales dan origen a la rama 1.2.4), 1 de la 1.2.2.3 y así sucesivamente.
Esto resulta en una mejora en el soporte del desarrollo en paralelo y la coordinación del equipo.

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
Continous Continous/CM
ClearCase implementa la transparencia de archivos con un sistema propietario de archivos (MVFS) que intercepta las llamadas del sistema operativo de acceso a los archivos (abrir(), leer(), etc.) y las re-dirige al repositorio.
Continous/CM provee un acceso directo a los archivos, creando uniones desde el área de trabajo del usuario, a sus directorios ocultos en el repositorio.

En áreas tales como CAD (Computed-Aided Design) y SE (Software Ingeneering) la administración de versiones puede basarse en Modelos de Versiones (Version Model) o Mecanismos de Administración (Managment Mechanisms).

Modelo de Versiones: definen que es una versión en el contexto de un ambiente de diseño, como están organizadas las versiones y las operaciones aplicables sobre las mismas. Esto depende fuertemente del tipo de objeto que se esta desarrollando (código fuente, circuitos VLSI, esquemas conceptuales, etc.)

Mecanismos de Administración: incluye el Modelo de versiones y el conjunto de primitivas que son aplicables. También implementan operaciones para optimización del espacio (archivos *delta*). Por ejemplo *forward deltas* que almacenan la primera versión completa mas las diferencias para crear versiones posteriores.

En sistemas CAD lo importante es la estructura de los objetos es decir como estos están compuestos. Las principales características de este modelo son: i) distinción entre: objetos genéricos, objetos de los cuales se necesita una administración de versiones y los que no la necesitan, ii) la clasificación de los atributos de un objeto en: invariantes, importantes para el versionamiento y no importantes para el versionamiento.
Esta clasificación es sumamente importante en la creación de una nueva versión: i) la modificación de un atributo invariante lleva a la creación de un objeto genérico ii) la modificación de un atributo importante para el versionamiento lleva a la creación de una nueva versión y por último iii) la modificación de un atributo que no es importante para el versionamiento es un modificación del objeto que no produce la creación de ningún otro objeto.

La administración de versiones en Ingeniería de Software difiere de la de ambientes CAD en la naturaleza de como se modelan los objetos. En Ingeniería de Software el menor nivel de granularidad de un objeto es el archivo, en los que generalmente no existe una semántica asociada entre los mismos. En ambientes CAD se trabajan con objetos compuestos por otros en los que claramente existe una semántica asociada (circuitos, unidades lógicas, etc.).

De todas las técnicas y herramientas investigadas no existe ninguna aplicada a administrar el proceso de Modelado Conceptual.
Aunque muchos conceptos y modelos de los antes presentados son utilizados en este *Manejador de Versiones*, algunos no son aplicables a esquemas conceptuales. Por otro lado en la Administración de Versiones de Esquemas ER, es interesante incorporar nuevos conceptos como ser la relación de construcción, comparación de esquemas, integración, etc.

2.3. El sistema desarrollado

El *Manejador de Versiones* se basa en dos conceptos básicos: CMP (Conceptual Modelling Projects) y Versiones que pertenecen a dichos CMP.
Una versión representa un esquema ER correspondiente a un paso en el proceso de diseño.
Los CMP modelan el conjunto de las versiones cuyos elementos representan el sistema de información de la aplicación.
El conjunto de versiones pertenecientes a un CMP están relacionadas entre si, a través de las siguientes relaciones: relación de derivación, relación de inclusión y relación de construcción.

En la siguiente sección mostraremos, a través de un ejemplo, las principales características del *Manejador de Versiones* y su conexión con el ambiente de reutilización de componentes.

Consideremos el diseño de un esquema ER que modela el sistema de facturación a clientes y proveedores de una empresa. El diseñador comienza creando un CMP llamado **`Clientes`,** utilizando la operación **Create_CMP** (ver sección 3.2). Luego crea una primera versión llamada **Clientes_Facturas** (versión corriente), con un esquema asociado generado con alguna herramienta del ambiente CASE (por ejemplo un Editor Gráfico de Esquemas ER).

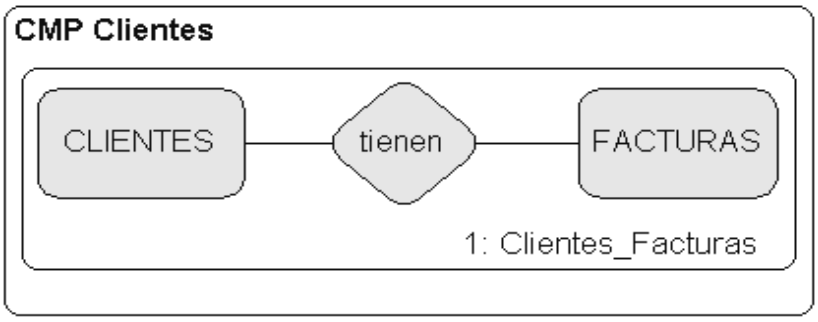
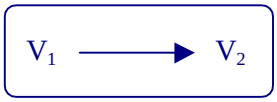


Figure 4: Clientes_Facturas

En un siguiente paso, el diseñador agrega dos nuevas entidades, **Personas** y **Proveedores** a su única versión, la cual era hasta ese momento la versión corriente. El esquema resultante es salvado con el nombre de **Clientes_Personas**.
Como el esquema **Clientes_Personas** es considerado por el diseñador como un paso subsecuente luego de **Clientes_Facturas**, es modelada como derivada desde **Clientes_Personas**. Esto es posible ejecutando la función **write_version** la cual crea una nueva versión y la deriva de la versión corriente. La versión creada, será la versión corriente.

Mas formalmente decimos que v_2 deriva de v_1 si (Ver Sección 3.2):



1. Ambas pertenecen al mismo conjunto de versiones (CMP).
2. La versión derivada v2 ha sido construida luego de su origen v1.
3. El diseñador ha decidido unir v2 como el siguiente a v1 en la secuencia de diseño en el modelo conceptual.

El Grafo de derivación correspondiente al ejemplo sería:

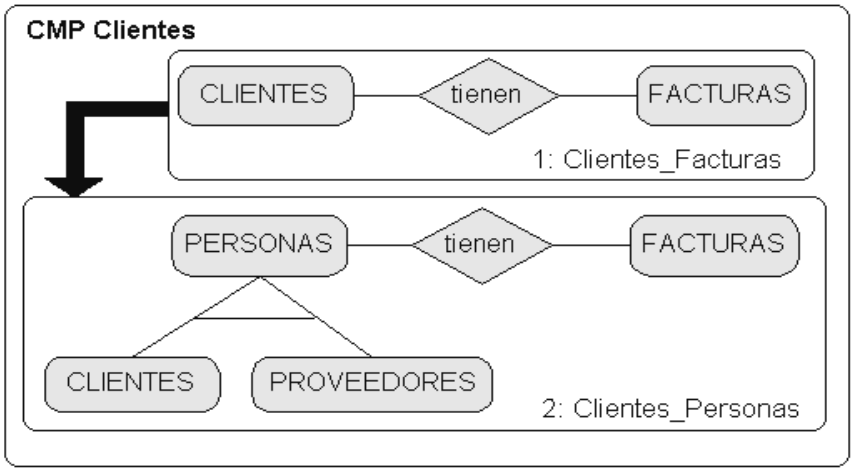


Figure 5: CMP con dos versiones

Supongamos que luego de sucesivas decisiones de diseño se llega a el siguiente árbol de derivación en el **CMP Clientes**:

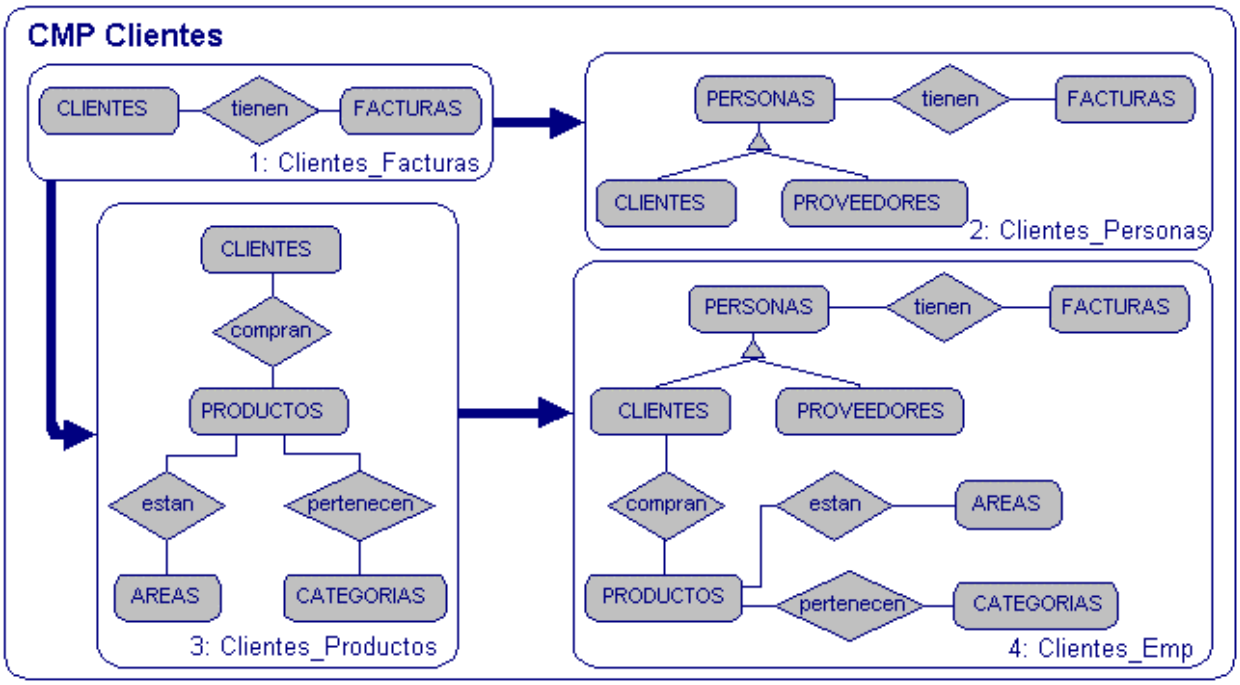


Figure 6: CMP final

Ahora bien, hasta ahora hemos presentado el ejemplo basándonos en la relación de derivación, pero ¿Qué pasa con la relación de construcción y la relación de inclusión?.

La relación de inclusión refleja cuando la estructura de un esquema ER incluye la misma estructura de otro esquema ER. La Relación de Inclusión esta basada en características de los esquemas ER.

> Se dice que v_1 incluye a v_2 si todas las entidades y relaciones definidas en v_2 tienen un correspondiente en v_1 a través de la función de correspondencia φ (Ver Sección 3.3).

El grafo de inclusión nos brinda una visión acerca de las propiedades de los esquemas ER incluidos en una determinada versión. Permite al diseñador identificar las versiones cuyos esquemas tienen características en común.

El grafo de inclusión del ejemplo se observa a continuación:

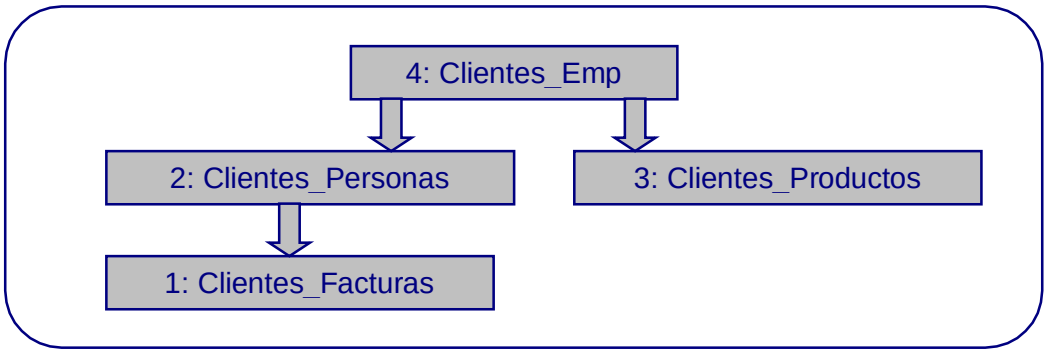


Figure 7: Árbol de Inclusión

La relación de construcción nos muestra como han sido construidas las versiones. Por ejemplo, el esquema asociado a **Clientes_Productos** se obtuvo como resultado de la unión asimétrica entre dos componentes reusables de esquemas ER, llamémosle **R1** y **R2**. Estos esquemas no pertenecen al **CMP Clientes**. La selección de **R1** y **R2** se ejecuta en el Seleccion Tool mientras que la operación de unión de esquemas, se efectúa importando las primitivas del Construction Tool.
La relación de construcción modela, como las diferentes versiones han sido construidas a partir de otras aplicando operaciones de esquema. Esta relación asocia versiones no sólo del mismo conjunto de versiones (CMP) sino también con versiones de otros conjuntos e inclusive con componentes de esquema reusable (Ver Sección 3.4).

> Se dice que v_1 fue construida a partir de v_2 a través de una expresión <*exp*> si v_1 se obtuvo a partir de v_2 aplicando <*exp*>.

El Construction Tool lee las versiones de esquemas ER (utilizando otras técnicas) y las compone con esquemas reusables.
La interconexión entre el Construction Tool y el *Manejador de Versiones* es lo que posibilita el manejo de esquemas ER sin considerar la herramienta con la cual fueron desarrollados.
El *Manejador de Versiones* permite al diseñador la visualización y el manejo del conjunto de versiones. Consiste en un Servidor para el Manejo de Esquemas ER al que le concierne el calculo de la relación de inclusión y un Servidor de Versiones que exporta el conjunto de primitivas implementadas en el Modelo de Versiones. La comunicación entre el *Manejador de Versiones* y el Construction Tool se realiza vía RPC.

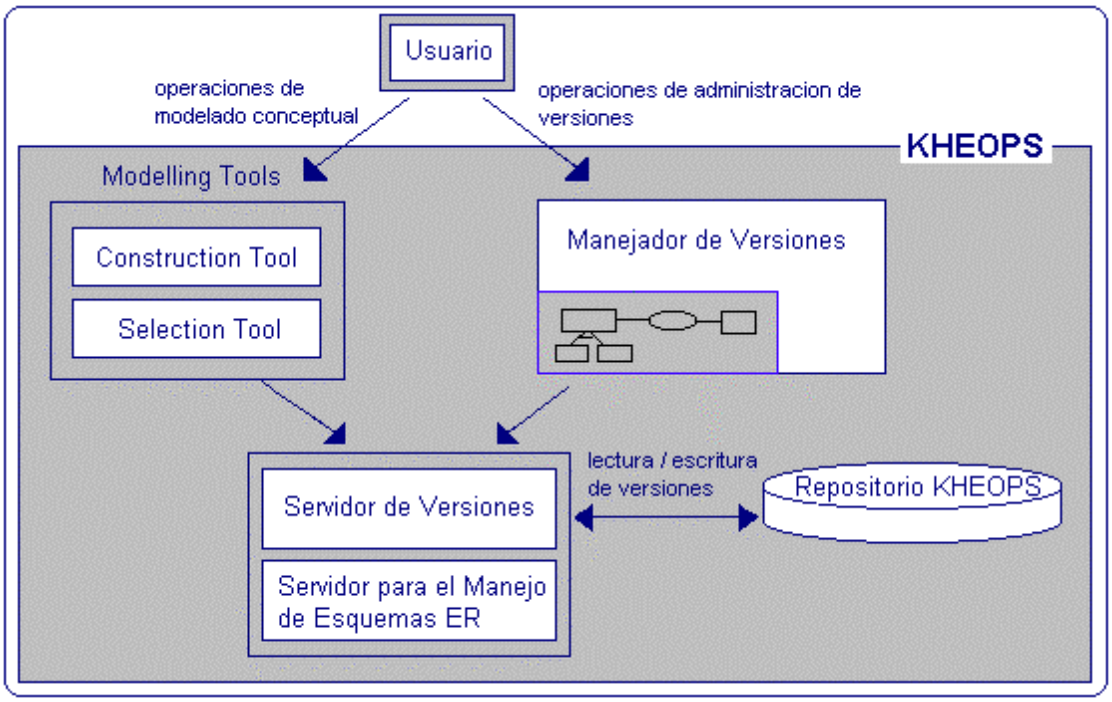


Figure 8: Conexión con el CASE

La arquitectura del *Manejador de Versiones* es Cliente/Servidor/Servidor. Las razones que llevan a diseñar una arquitectura de este tipo fueron: la necesidad de presentar la información a través de Interfaces Gráficas (GUI), el acceso simultáneo a varias fuentes de información en forma modular y la integración modular, adaptable y portable con otras herramientas y el entorno.

El *Manejador de Versiones* esta compuesto por un módulo cliente donde se implementó la interfaz gráfica y por los dos servidores. La comunicación entre los módulos e inclusive entre las demás herramientas que pertenecen al ambiente CASE se realizan mediante llamadas remotas (RPC) y archivos. El *Manejador de Versiones* exporta primitivas de manejo de versiones y de conjunto de versiones (CMP): creación, modificación, consulta, borrado, etc. Los grafos de versiones son almacenados en archivos ASCCII (*.vpl).

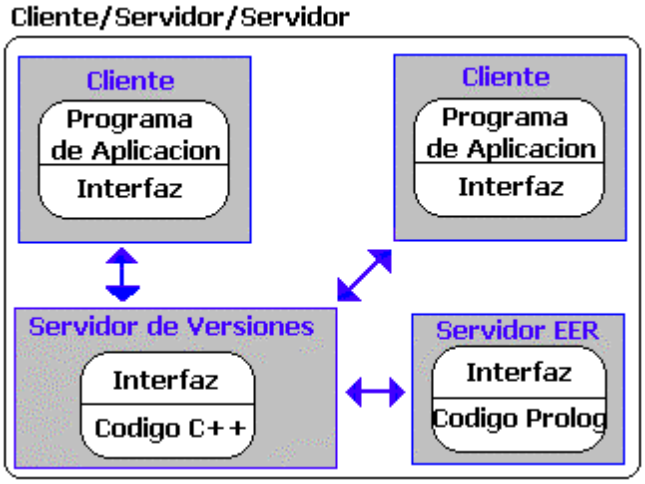


Figure 9: Arquitectura

Su diseño además de ser modular debe tener una conectividad tal que le permita su integración a la herramienta CASE. Para lo cual es deseable que cada componente sea independiente del otro con una capa de comunicación pública a todas las demás. Esto asegura que una modificación en cualquier componente no tiene un impacto mayor en las restantes.

Para independizar de la plataforma y el lenguaje, el Servidor de manejo de esquemas ER exporta las funciones invocadas desde el Servidor de Versiones. La comunicación del Cliente con el Servidor de manejo de esquemas ER es a través del Servidor de Versiones.

Como ya se mencionó, las operaciones en el *Manejador de Versiones* nos permiten: administrar versiones y los conjunto de versiones.
Ya vimos algunas de ellas (**`Create_CMP`** y **`Write_version`**) pero existen otras. Se pueden agrupar dependiendo de la clase de objeto que involucren: operaciones sobre un CMP, operaciones sobre versiones y operaciones sobre los grafos de versiones.
Las operaciones de manejo de Proyectos de Modelado Conceptual, son básicamente: abrir, borrar, crear, salvar y cerrar un determinado CMP.
También es posible borrar, crear, modificar o renombrar versiones, entre otras.
El tipo de operaciones que se aplican a los grafos de versiones son de consulta –obtener el camino de un nodo a la raíz del grafo- así como también el borrado de un conjunto de nodos que pertenecen a una rama del árbol (Ver sección 3.5).

3. MODELO DE VERSIONES

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.

3.1. Versión

Una versión consiste de un identificador de versiones (dentro de un CMP), un nombre y un valor.

Definición

Una versión es una tupla: <id, nombre, valor >

donde:

- id: identifica a la versión en el conjunto de versiones
- nombre: es una etiqueta que nombra a la versión
- valor: es el nombre del archivo donde se encuentra la especificación del esquema ER.

3.2. Relación de derivación

Una versión v_d deriva de otra v_s si ha sido creada como un producto de diseño subsecuente de v_s. 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 ÁRBOL DE DERIVACIÓN, donde los nodos son las versiones del CMP. El árbol de derivación también es denominado ÁRBOL HISTÓRICO 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.

Definición

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

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

$v_1 \overset{d}{\rightarrow} v_2 \Leftrightarrow$ v_2 es derivada a partir de v_1

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

Anti-Simétrica y Anti-Reflexiva: $(\forall\ v_1, v_2 \in V)\ (\ v_1 \overset{d}{\rightarrow} v_2 \Rightarrow \neg\ v_2 \overset{d}{\rightarrow} v_1\ )$

Único Ancestro: $(\forall\ v_1, v_2, v_3 \in V)\ (\ v_1 \overset{d}{\rightarrow} v_2 \wedge v_3 \overset{d}{\rightarrow} v_2 \ \Rightarrow\ v_3 = v_1\ )$

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

3.3. 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:

Sean E, E' dos entidades y R, R' dos relaciones cualesquiera, entonces:

φ(E,E') $\Leftrightarrow$ (E.nombre = E'.nombre)
φ(R,R') $\Leftrightarrow$ (R.nombre = R'.nombre)

Sea un esquema ER S = <***E***,***R***> y un esquema S' = <***E'***,***R'***> donde ***E***, ***E'*** son el conjunto de entidades y ***R***, ***R'*** son el conjunto de relaciones respectivamente.

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

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

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:

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

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

donde:

$$v_2 \overset{\supseteq}{\rightarrow} v_1 \quad \Leftrightarrow \quad (v_2.\text{esquema} \supseteq v_1.\text{esquema} \wedge v_1.id \neq v_2.id \wedge \neg (\exists\, v \in \sqrt{}) \; (v.id \neq v_1.id \wedge v.id \neq v_2.id \wedge v_2 \overset{\supseteq}{\rightarrow} v \wedge v \overset{\supseteq}{\rightarrow} v_1))$$

La $\overset{\supseteq}{\rightarrow}$ 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 $\overset{\supseteq}{\rightarrow}$.

La relación de inclusión determina el GRAFO DE INCLUSIÓN. 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.

3.4. Proyecto de Modelado Conceptual (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 situación.
- Vc: versión corriente del conjunto.
- $\rightarrow$: relación de derivación entre las versiones de V.
- $\Rightarrow$:relación de inclusión entre las versiones de V.

3.5. Relación de Construcción

Modela como las versiones han sido construidas a partir de otras versiones. Una relación de construcción entre dos versiones v_1 y v_2 a través de una expresión de tipo <*exp*> significa que la versión v_2 ha sido construida a partir de v_1 aplicando <*exp*>. Esta relación se aplica en la reutilización de componentes.

Definición

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

$\rightarrow \subseteq ((\surd ids \cup RSCids) \times Esq\_exp \times \surd ids)$
$o \rightarrow v \Leftrightarrow (v.esquema = exec(exp) \wedge o \in ref\_schema(exp))$

donde:

- √ides es la unión 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 el conjunto de esquemas referenciados por *exp*.

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 permite identificar las versiones cuyos esquemas tienen características en común.

3.6. Operaciones

Las operaciones en el Modelo de Versiones permiten manipular versiones y conjuntos de versiones. A continuación presentamos algunas de 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.

3.6.1. Operaciones sobre un CMP

`create_CMP (string nombre_CMP)`
Un CMP es creado a través del operador create_CMP, el cual crea una versión root que será la versión corriente.

`open_CMP (string nombre_CMP)`
Se utiliza para abrir un CMP y cargar en memoria el conjunto de nodos y grafos que estan almacenados en el archivo de nombre "nombre_CMP.vpl". El nombre del CMP es la concatenación de dos substrings: nombre_aplicacion y nombre_modulo.

`save_CMP (string nombre_CMP)`
Salva un CMP, almacenándolo en el archivo de nombre: "nombre_CMP.vpl".

close_CMP ()
Cierra el CMP con el que se este trabajando.

Give_information ()
Devuelve el nombre del Modelo Conceptual que está siendo consultado.

3.6.2. Operaciones sobre versiones

write_version (CMP c, valor_version t, string nombre) → datos_versión
La creación de una versión en un conjunto de versiones se logra por medio del operador *write_version*. Esta versión será la raíz de un nuevo árbol de derivación y tendrá como padre a la versión que era hasta ese momento la corriente. La versión creada pasa a ser la nueva versión actual. El operador retorna la estructura de la versión creada.

delete_version (CMP c, Version v)
El operador *delete_version* borra una determinada versión v que pertenece a un CMP. Las versiones derivadas de la misma se "cuelgan" de la fuente de derivación (en el árbol de derivación) de la versión borrada. En el árbol de inclusión esta relación sería recalculada.

rename_version (string nombre, Version v)
Para renombrar una versión dada se utiliza la función *rename_version*.

read_version () → datos_versión
Para leer datos de la versión corriente se utiliza la función *read_version*. Esta operación devuelve la estructura de la versión leída.

update_version (int Version_id, string valor)
Permite modificar el valor de una versión.

Set_current (int Version_id)
La versión con el identificador indicado será la versión corriente en el CMP.

Give_id (string Version_name) → id_versión
Devuelve el identificador correspondiente al nombre de versión ingresado.

Give_curr_version ()→ id_versión
Devuelve el identificador de la versión corriente.

3.6.3. Operaciones sobre grafos de versiones

ancestors (id_version) → lista_versiones
Esta operación devuelve el conjunto de versiones que fueron fuente de derivación de la versión indicada (el camino desde la raiz hasta la versión dada, en el árbol de derivación).

path (id_version) → lista_versiones
Devuelve el conjunto de versiones en las cuales esta incluida la versión indicada (el camino desde la raiz hasta la versión dada, en el árbol de inclusión).

sub_tree (id_version) → lista_versiones
Esta operación devuelve el conjunto de versiones que fueron derivadas de la versión indicada, en distintos pasos del modelado conceptual.

sub_branch (id_version_from, id_version_to) → lista_versiones
Esta operación devuelve el conjunto de versiones que fueron derivadas desde la versión indicada como inicio, hasta la version indicada como fin.

del_sub_tree (id_version)
El operador del_sub_tree borra las versiones pertenecientes al árbol de derivación cuya raíz es la versión dada.

del_branch (id_version_from, id_version_to)
El operador del_branch borra las versiones pertenecientes a la sub-rama determinada por las versiones pasadas como parámetros.

del_path (id_version)
El operador del_path borra los ancestros de la versión indicada, en el árbol de derivación. La versión queda como hija de la raiz.

del_includes(id_version)
Esta operación borra las versiones incluidas en la versión ingresada.

def_super_version (id_sub_version, id_super-version)
Define a la versión id_super_version como la versión padre de la versión id_sub_version, en el árbol de inclusión.

La estrategia para la derivación automática de versiones establece que hacer cuando el esquema 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 el Modelo de Versiones se aplica 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.

4. DISEÑO E IMPLEMENTACIÓN

La conexión se realiza de la siguiente forma:

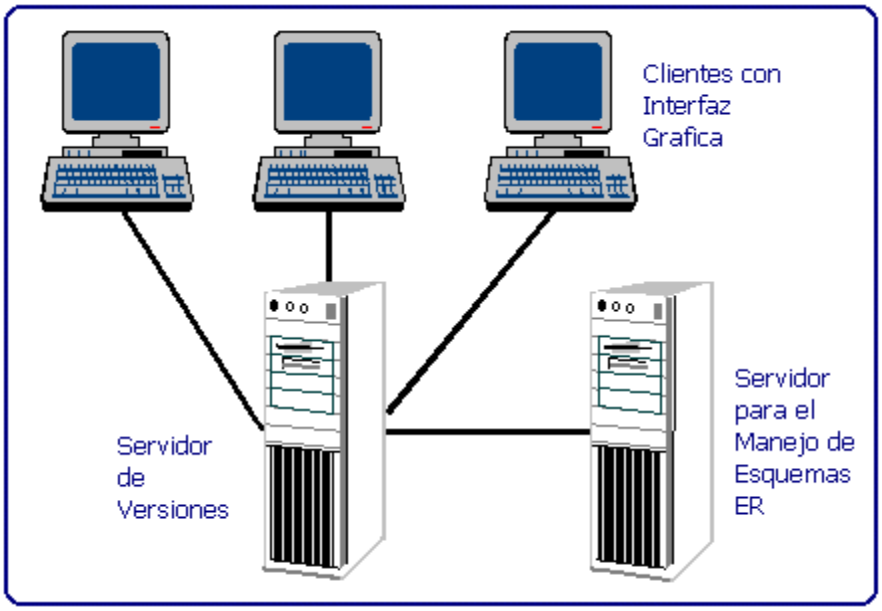


Figure 10: Arquitectura

La herramienta cliente es el módulo donde se implementa toda la interfaz gráfica. Todas las operaciones invocadas acceden al Servidor de Versiones.

La implementación realizada es la siguiente: al comenzar, el Cliente se comunica vía RPC al Servidor de Versiones y este se conecta al Servidor de esquemas ER.
Una vez establecida la comunicación con el Servidor de Esquemas, ésta no finaliza y permanece abierta para optimizar las llamadas futuras.

RPC nos permite que procesos distintos en distintos nodos de la red puedan comunicarse, cooperar y coordinar actividades.
La mayor diferencia entre una llamada local y una llamada remota es que, mientras la primera tiene lugar entre procedimientos del mismo proceso, en el mismo espacio de memoria del sistema, la llamada mediante RPC tiene lugar entre un proceso cliente en un determinado sistema y un servidor en otro (ambos están comunicados a través de la red).

Asume la existencia de mecanismos de red de bajo nivel como TCP/IP1o UDP/IP2 (en nuestro caso para TCP/IP) y sobre ellos se implementa un sistema lógico de comunicaciones Cliente/Servidor.

1 Transmision Control Protocol
2 User Datagram Protocol

En contraposición con servicios que corren bajo el control de inetd3 como Telnet o FTP, RPC implementa una interfaz funcional y de esta manera los servicios remotos pueden embeberse como llamadas dentro de una aplicación.
El paradigma RPC extiende el concepto de llamada a una subrutina escondiendo al programador los detalles relativos a las comunicaciones. El modelo usado por RPC es un enfoque orientado a la aplicación, lo que nos permitió seguir los principios de un buen diseño, es decir, aplicaciones modulares y fáciles de mantener.
A continuación describiremos en forma detallada la implementacion de los 3 módulos que componen el *Manejador de Versiones*.

4.1. Servidor de Versiones

El Servidor de Versiones esta implementado en C++. La clase **`V_System`** es quien administra el conjunto de grafos (CMP) que son accedidos por los distintos clientes. Cada cliente se identifica por su número de conexión (**`cnn_number`**) y se almacena en la estructura **`versions`**.
En esta clase se implementaron las funciones de carga de los grafos que serán enviados a través de una estructura de datos, **`vg_arr`**, a los distintos clientes, cada vez que éstos lo soliciten.
La relación de construcción es la que establece como se construyeron nuevas versiones a partir de otras ya existentes que podrían pertenecer a distintos CMP, por lo cual esta relación debería almacenarse en esta clase.

```
class V_SYSTEM  {
public:
     CMProjects * versions[50];
     vg_arrays vg_arr;

     V_SYSTEM();

     CMProjects *give_graph(int cnn_number);
     int give_id_curr_version( int cnn_nunber );
     int give_cnn_number(char*, char*, char*);
     void save_version_plane(int cnn_number);
     int open_version_plane(char*,char*,char*);
     int write_version(int cnn_n,char *v_n,char *v_v, char
               *drv_frm_v);
     . . .
};
```

Ejemplo 1 : Clase V_SYSTEM

3 Internet services Deamond

Las rutinas del Modelo de Versiones fueron implementadas a través de la clase **CMProjects.**

```
class CMProjects {
public:
     char application_name[20];
     char module_name[20];
     GRAPH<Version,Arc> structure;
     node root, current_version;

     CMProjects(char*, char*);
     ~CMProjects();
     Version* give_version(node);
     GRAPH<Version,Arc>* give_structure();
     int give_last_id_version();
     node search_version(int st_id);
     edge search_edge(int id_arc);
…};
```

Ejemplo 2: Codigo de la clase CMProyect

El módulo que maneja el CMP se representa a través de la clase **CMProjects** la cual 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**, función exportada por el servidor para el manejo de esquemas ER.
La estructura y las relaciones entre las versiones se implementó utilizando una biblioteca para Tipos Abstractos de Datos LEDA la cual nos permitió definir grafos paramétricos (**GRAPH<Version,Arc> structure**). Las versiones son los nodos de dichos grafos y las relaciones de Derivación e Inclusión son los arcos.

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: identificador, nombre y valor. El significado de cada uno de ellos fue explicado anteriormente.

La clase **Arc** permite la creación y administración de las relaciones de los dos grafos.

En la relación de derivación se mantienen dos tipos de lazos: **Derivation_from** y **Derivation_to,** análogamente existen dos tipos de aristas de inclusión: **Inclusion_from** e **Inclusion_to.**
De lo anterior se desprenden las clases: **Derivation_from, Derivation_to, Inclusion_from, Inclusion_to** para la creación y mejorar la performance en los algoritmos de recorrido sobre los grafos. El conjunto de nodos se encuentra almacenado una sola vez.

La jerarquía dentro del Servidor de Versiones es la siguiente:

```
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,
            Inclusion_from,Inclusion_to:public Arc;
```

4.2. Servidor de Manejo de esquemas ER

Para potencializar la modularidad de la herramienta se decide que la conexión debe ser independiente del módulo que administre un nivel inferior del grafo (esquemas ER), por lo que vía RPC se decidió la implementación de un Servidor para el manejo de esquemas ER. Por otro lado, el Servidor de Versiones será un cliente de este servidor por lo que tendríamos una comunicación Cliente – Servidor de Versiones – 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

Los esquemas ER se encuentran almacenados en *facts* o predicados Prolog.
El Servidor de manejo de esquemas ER esta implementado en C++ y necesita acceder a operaciones que consultan la fuente Prolog , dicha comunicación no es trivial, pues no existe un standard de comunicación entre C++ y predicados Prolog. Para establecer esta comunicación se diseña un “pipe” Unix.
Es sabido que Prolog consume muchos recursos, por lo tanto no es deseable la ejecución en paralelo de varios núcleos, por tanto es conveniente unificar las ejecuciones.
Es decir, si varios clientes consultan esta fuente de información se estaría creando dinámicamente un pipe unix por cada conexión. Para resolver este problema se implementa una solución híbrida: el Servidor de Manejo de esquemas ER combina y encapsula el pipe Unix y RPC.

El servidor de manejo de esquemas ER define la operación **`Oper`**. El argumento de esta operación, invocada desde el Servidor de Versiones, es un string que contiene la operación y el conjunto de parámetros que se necesitan para acceder a la fuente Prolog, es decir el predicado Prolog.

El diagrama de esta comunicación sería:

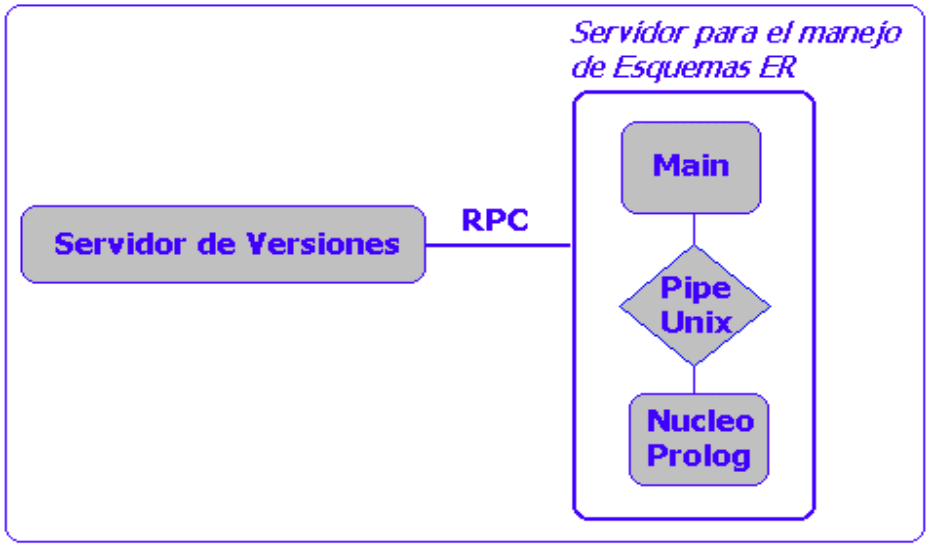


Figure 11 :Comunicación

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 tenía que efectuar una operación sobre un esquema ER, llamaba a una función que ejecutaba el predicado Prolog.

4.3. Cliente

Una de las principales funciones del Cliente es la interfaz gráfica para los grafos de versiones, que permiten a los usuarios visualizar y manipular los nodos de los grafos de forma fácil y amigable.
Como ya se explicó, se realizó una búsqueda de herramientas para el diseño de GUI (Graphical User Interfaz).
Se pretendía una herramienta que permitiera 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 en pantalla 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, ésta 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 Versiones.
Además la herramienta elegida debía tener buena documentación.

Los criterios seguidos para seleccionar una buena herramienta de desarrollo de GUI fueron:

Escalable

Es deseable tener grafos, gráficas, grilla, mapas, etc. en la interfaz, no teniendo sentido implementar estos objetos gráficos existiendo productos que se dedican especialmente a este desarrollo. Existen también un conjunto de bibliotecas C++, OCX que solucionan distintos aspectos de la interfaz. Por ejemplo puede existir un OCX que solucione los grafos, otro las grillas, etc., por lo que el desarrollador tiene que aprender distintas APIs y conceptos para integrarlos. Por todo esto la solución a un desarrollo GUI no es la combinación de distintas herramientas, la solución es una herramienta GUI que sea lo suficientemente poderosa para reducir el uso de otras herramientas. Consecuentemente el código de la aplicación será homogéneo y compartirán recursos eficientemente.

Extensible

Esto significa que la herramienta se basa en una arquitectura orientada a objetos (OO).
Buscamos una herramienta con la estructura de jerarquías y herencia, bien documentada.
La mayoría de estas herramientas están implementadas en C++ y algunas, mas nuevas, en Java.

Interacciones

Las interacciones con objetos gráficos son un standard que requiere generalmente mucho desarrollo. Rechazamos aquellas herramientas en las que era necesario implementar eventos directamente. Básicamente existen dos tipos de interacciones: aquellas que se heredan del objeto (por ejemplo: un objeto se comporta como un botón cuando es seleccionado), las que se limitan a la visualización: seleccionar, mover, drag-and-drop, etc. En ambos casos la interacción debe estar separada del objeto gráfico para que esta pueda tener cualquier comportamiento. La herramienta debe incluir un conjunto completo de interacciones.

Portable

Dado que nuestra aplicación se ejecuta en un ambiente UNIX la herramienta seleccionada debía soportarlo. Como es sabido Microsoft ha cambiado dramáticamente el mercado, y proyectos que son comenzados en ambientes Unix, luego pueden querer extenderse a ambientes MS-Windows y más aún con aquellos proyectos GUI. Aquellas herramientas que no son portables generalmente se implementan directamente con las API. Pero existen herramientas que proveen un motor en las que, todas las operaciones básicas están implementadas, además de ofrecer el “look-and-feel” para todas las plataformas.

Las bibliotecas estudiadas fueron las siguientes: Xwindows, Interviews, Gina++ e IlogViews. El resultado de nuestra evaluación es al siguiente:

Xwindows
Ventajas: interfaz amigable.
Desventajas: no era sensitivo al ratón. Había que programar cada movimiento de los nodos.

Interviews
Desventajas: no era posible mover los nodos de los grafos.

Gina++
Desventajas: mala documentación.
IlogViews
Ventajas: buena documentación. Buen soporte. Generador de interfaz. Clase de grafos con funciones de manejo propias. Clase Interactor, no fue necesario programar cada movimiento del ratón.
Utilizando esta última generamos la interfaz en su totalidad.

El módulo Cliente esta implementado en C++ y utiliza funcionalidades de ILOG Views.
En la clase **`Client_Env`** se implementa el conjunto de funciones solicitadas por los usuarios que posibilitan la comunicación con el Servidor de Versiones (**`open_module`, `Close_version_plane`, `save_version_plane`**, etc.). Además se definen las estructuras que permiten tener en memoria a los distintos grafos y las posiciones de cada nodo por cliente conectado a la herramienta. La información propia que contiene esta clase es el número de conexión de cada cliente e información relativa al mismo.
La clase **`Client_Env`** es quien importa vía RPC las funciones implementadas en la clase **`V_System`**.

```
class Client_Env {
     char      *server_VM;
CLIENT    *cl_VM;
     char      *server_OPS;
     CLIENT    *cl_OPS;
public:
     int conn_number;
     table_vg  vg_t;
     char obj_names[MAX_LENGHT_ID][MAX_QTTY_OBJECTS];
     objs_in_cmp obj_cmp;
     Client_Env(int cn, void (*f)(char *a[],
          Client_Env *),Suit_Gr *gr);
     void conn_server_VM(char *server_name);
     int open_module(char *a,char *m,char *W);
     int close_version_plane();
     int save_version_plane();
     void give_vg_arrays();
     version_rec *give_curr_version();
     list_VIds *ancestors(int v_id); // |-> status
     vp_rec *write_version(char *,char *,char *);
     ..........
};
```

Ejemplo 3: Código de la clase CL_ENV

En este módulo además de la implementación de todas las funcionalidades de la interfaz gráfica (lectura del grafo desde el Servidor de Versiones, carga en la estructura de grafos de ILOG, etc.) se almacena información de la distribución de los nodos de cada cliente. Esta información se almacena

en archivos ASCII (*.psc) y es accedida al comenzar una sección, a pedido del usuario, y al finalizar la misma.

4.3.1. Interfaz gráfica

Para la implementación de la Interfaz gráfica utilizamos Ilog Views Studio que es una herramienta que compone Ilog Views (ver sección 5.3).

Las clases mas importantes generadas por el Studio fueron:

- Aplicación, en donde están definidas y especificadas todas las ventanas de interacción con el usuario
- Menú, en donde se define el Menú del *Manejador de Versiones*.

En general, en las herramientas investigadas las interacciones con el usuario requerían mucho desarrollo, en Ilog Views únicamente tuvimos que implementar los c*allbacks* específicos de nuestra aplicación, sin resolver eventos de mouse o de ventanas.

Esta facilidad de Ilog se debe a la arquitectura orientada a objetos con una jerarquía entre ellos que permite que las aplicaciones sean fácilmente extensible.

Los objetos gráficos exportan un conjunto de primitivas que ofrecen todo lo que se necesita para crear la interfaz. En Ilog todos los gráficos tienen el mismo conjunto de primitivas, lo que significa que no se tiene en cuenta el tipo de objeto al aplicarle una operación.
Por ejemplo: modificar el tamaño de un objeto gráfico (*resizing*) es una operación que se aplica tanto a círculos, líneas, rectángulos; cuando en un caso estamos modificando el diámetro y en otro el largo de los lados. En una arquitectura orientada a objetos, existe una única función "*resize*" que actúa dependiendo del objeto al que es aplicada.

Para representar los distintos grafos utilizamos el objeto **`Grapher`** de IlogViews. Cada nodo y lazo son objetos gráficos separados que tienen propiedades de despliegue y conducta propias. Esto significa que los nodos pueden tener bitmaps, labels, etc., y tener interacciones como si fuesen botones, editores, etc. Ilog Views representa separadamente para cada objeto, sus aspectos de despliegue y conducta, por lo que es posible combinar todas estas opciones.

Además para simplificar la construcción del grafo existe un conjunto predefinido de tipos de lazos como: straight links, spline links, etc. Modificando la interacción de cada lazo se puede: seleccionar, modificar tamaño, moverlos, etc. Además las operaciones aplicadas a los nodos son automáticamente aplicadas a los lazos del grafo y se recomputan acorde con el desplazamiento de los nodos.

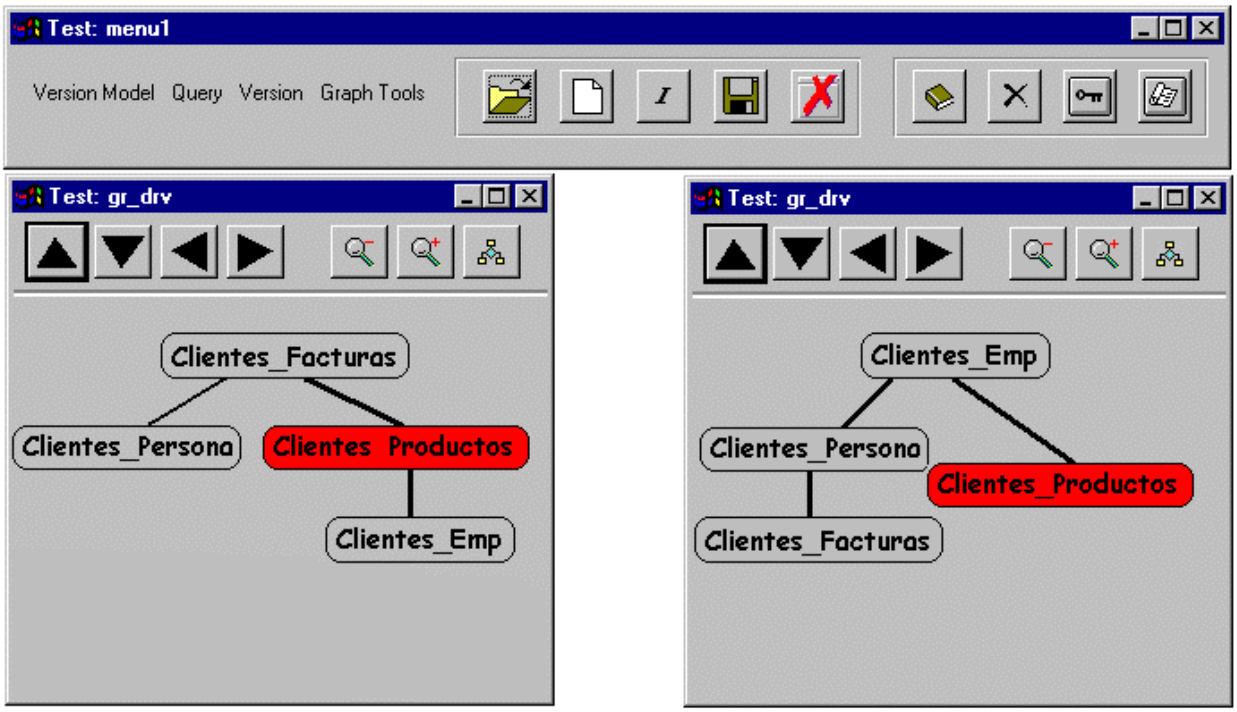


Figure 12: Interfaz

4.4. Conexión y Testeo

La siguiente figura muestra como se conecta los tres módulos explicados anteriormente.

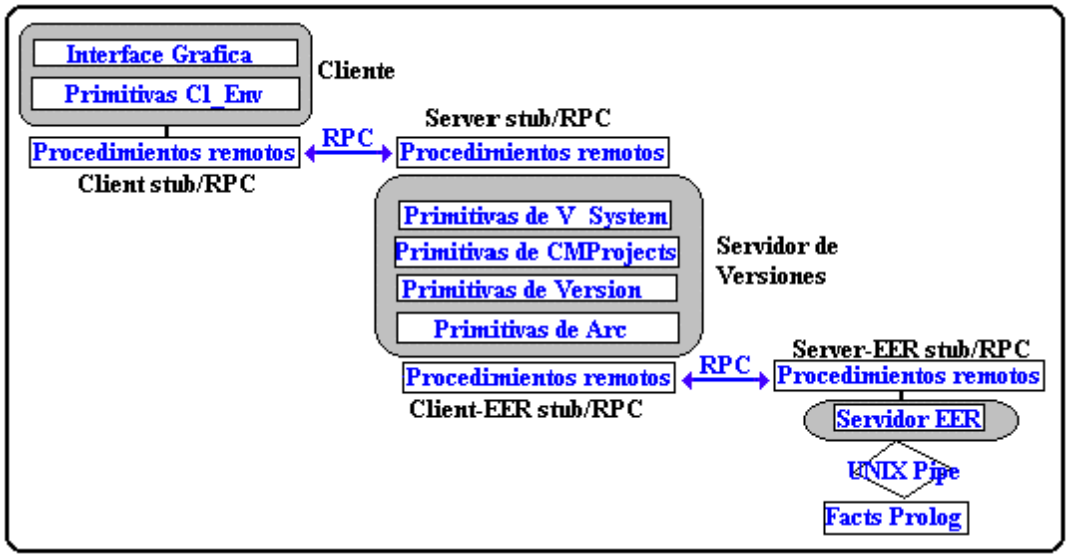


Figure 13: Integración de los 3 módulos.

Las estructuras implementadas en sus distintas capas se comunican de la siguiente manera: 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 máquina 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 ésto 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_includes, 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 Versiones.
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

4.5. Como se realizarían nuevas conexiones.

La característica del *Manejador de Versiones* implementado, es que a partir de una fuente de información se establece la relación de inclusión entre los objetos. Como vimos esta relación no se puede establecer con cualquier objeto, es una característica de los ER y también se ve en componentes mecánicas, circuitos, etc.
La herramienta que se conecte al *Manejador de Versiones* debe ser un servidor que tenga por ejemplo, las siguientes funciones públicas:

`Write(nueva_version Version)->Información`

Esta función obtiene la especificación del esquema ER de la nueva_versión, y determina si este nuevo esquema no verifica la relación φ (definida en la sección 3.3), es decir si no está incluido en otro esquema ER. En este caso devuelve el padre de la nueva versión, en el grafo de inclusión.

`Update(Version_modificar Version)->Lista of Version`

Esta función obtiene el nuevo valor de la versión. Si este valor implica la pérdida de un esquema existente, entonces una nueva versión deberá crearse, en este caso se devolverá el nuevo grafo de inclusión.

Estas funciones son invocadas desde el Servidor de Versiones vía RPC y actualizan el grafo de inclusión.

El primer paso, en la creación de una nueva conexión, es generar la interfaz entre el servidor de versiones y el nuevo servidor. Esto se realiza utilizando un compilador de protocolo, en nuestro caso ONC RPCGEN.
Deberá especificarse un archivo, por ejemplo Nuevo_Server.x (Ver sección 5.1), que incluya las funciones detalladas anteriormente. Este archivo es llamado “Archivo de Definición de Protocolo” y esta especificado con RPCL (ONC RPC language).
El generador RPCGen generará, mediante el comando **`RPCGen Nuevo_Server.x`**, un conjunto de archivos (Nuevo_Server.h, Nuevo_Server _clnt.c, Nuevo_Server_svc.c).
Para determinar que el Servidor de Versiones será un cliente de este nuevo servidor, es necesario compilar y linkeditar los fuentes del Servidor de Versiones (design_tree.c) con Nuevo_Server.h y Nuevo_Server _clnt.c**.**
El módulo que define las funcionalidades de este nuevo servidor, debe ser compilado y linkeditado con algunos de los archivos generados por RPCGen (Nuevo_Server.h y Nuevo_Server_svc.c).

5. HERRAMIENTAS UTILIZADAS

5.1. RPC (Remote Procedure Call)

RPC es un conjunto de software que nos permite distribuir partes de un programa en otra computadora de la red. RPC es un paradigma de comunicaciones de alto nivel usado con el sistema operativo.

Definición de Sun Microsystems de la llamada a procedimiento remoto

RPC de Sun define el formato de los mensajes que el llamador (cliente) envía para invocar un procedimiento remoto en un servidor, el formato de los argumentos, y el formato de los resultados que el procedimiento llamado devuelve al llamador. Permite usar al procedimiento llamador los protocolos UDP4 como TCP5, para el transporte de mensajes, y utiliza XDR6 para representar los argumentos así como otros ítems en la cabecera de un mensaje RPC.

Además de la especificación del protocolo, Sun RPC incluye un compilador que permite al programador la construcción de programas distribuidos en forma automática (RPCGen).

Programas y Procedimientos Remotos

Sun Rpc define un programa remoto como la unidad básica de software que ejecuta en una máquina remota. Cada programa remoto constituye lo que llamamos un servidor y contiene uno o más procedimientos remotos además de datos globales. Los distintos procedimientos dentro de un programa remoto comparten el acceso a los datos globales.

Semántica de Comunicaciones

El estándar de Sun RPC no especifica protocolos o mecanismos adicionales para lograr envíos confiables. Por un lado, para conseguir que una llamada remota a procedimiento se comporte lo más parecido posible a una llamada local, RPC debería usar un protocolo de transporte confiable como es TCP y garantizarle al programador confiabilidad: la llamada remota a procedimiento debería, ya sea, transferir dicha llamada al procedimiento remoto y retornar una respuesta, o, reportar que ha sido imposible lograr la comunicación. Por otra parte, para permitir usar al programador un protocolo de transporte eficiente, no orientado a la conexión, RPC debería poder usar comunicaciones vía un protocolo de datagramas como UDP.

4 User Datagram Protocol
5 Transmicion Control Protocol
6 External Data Representation

Por estas razones es que la semántica de RPC se define en función de la semántica del protocolo de transporte subyacente.

Por ejemplo, al usar UDP, tanto un mensaje de pedido de servicio, como uno de respuesta, podría perderse o duplicarse. En consecuencia, si una llamada remota a procedimiento no retorna, el procedimiento llamador, no podrá concluir que el procedimiento remoto no fue llamado, ya que podría haberse perdido el mensaje de respuesta, aún cuando el de requerimiento de servicio haya llegado en forma correcta. La otra cara de la moneda indica, que el hecho de recibir un mensaje de respuesta, permite asegurar al procedimiento llamador que el procedimiento remoto fue llamado por lo menos una vez. El llamador no podrá concluir que el procedimiento remoto fue llamado sólo una vez, ya que el mensaje de pedido de servicio podría haberse duplicado, o, algún mensaje de respuesta podría haberse perdido.

El estándar RPC utiliza el término ***semántica de por lo menos una vez*** para describir la ejecución RPC cuando el procedimiento llamador recibe una respuesta, y ***semántica de cero o más veces*** para describir el comportamiento de una llamada remota a procedimiento cuando el llamador no recibe respuesta.

El programador que elija UDP como protocolo de transporte con Sun RPC, deberá construir sus aplicaciones de modo que toleren la semántica de ejecución de cero o más veces. Esto significa que deberá efectuar sus llamadas remotas en forma idempotente, o sea, la ejecución en forma repetida de una misma operación deberá devolver siempre el mismo resultado.

¿Cómo dialoga un cliente con un servidor? - El Portmapper

Los servicios TCP/IP en general, se anuncian en *ports7* bien conocidos: un servidor crea un *socket8* pasivo, une el socket a un port bien conocido (*binding*), y espera por los programas clientes que lo contacten. Estamos asumiendo entonces, que cada servicio está vinculado con un único número de port de protocolo y que dichos vínculos son bien conocidos. De esta manera, el cliente y el servidor pueden acordar el port de protocolo en el que el server aceptará pedidos, ya que ambos disponen de una tabla pública con las asignaciones de ports.

Los protocolos de transporte UDP y TCP usan números de port de 16 bits para identificar terminales de comunicación. Por otra parte, Sun RPC usa números de 32 bits para numerar los programas remotos, con lo que se sobrepasan la cantidad de ports de protocolo. Por lo tanto, dado que no es posible asignar un único port de protocolo a cada programa RPC, el programador no podrá usar un esquema que dependa de asignaciones de ports bien conocidas como se describió anteriormente. Si un servicio RPC no usa un port de protocolo reservado y bien conocido, los clientes no podrán contactarlo directamente.

Para resolver el problema de identificación del port, un cliente debe poder obtener, dado un número de programa RPC y una dirección de máquina, el port de protocolo que dicho programa (servicio) obtuvo del sistema operativo cuando comenzó a ejecutar. Esta operación (mapping) debe ser resuelta

7 Canal logico de comunicaión dentro de una red.
8 Canal de IPC.

en forma dinámica, ya que podrían cambiar los vínculos, si por ejemplo la máquina remota se reinicia, o si el programa RPC comienza a ejecutar nuevamente.

Para que los clientes logren contactar a los servidores remotos, el paquete de Sun RPC incluye un servicio de vinculación dinámico9 llamado ***Portmapper*** que se anuncia siempre en el port bien conocido 111.

Un servidor portmapper deberá correr en cada máquina que implemente el lado servidor de un programa RPC.

El algoritmo que sigue el port mapper es como sigue:

1. Crear un socket pasivo vinculado al port bien conocido del servidor portmapper de Sun (port 111).
2. Aceptar en forma iterativa pedidos de registración de programas RPC y/o de consulta (dado un número de servicio RPC, qué port de protocolo le corresponde).

Los pedidos de registración provienen de programas RPC que residen en la misma máquina que el Portmapper. Cada pedido de registración especifica el par compuesto de un número de programa RPC y el port de protocolo que está en este momento vinculado para contactar ese programa. Cuando llega el pedido de registración, el Portmapper agrega el par a la tabla de vínculos10.

Los pedidos de consulta provienen de máquinas arbitrarias. Cada uno proporcionará el número de programa que desea contactar y pedirá el número de port de protocolo en que se está anunciando dicho programa. El Portmapper se fija entonces en su tabla de programas remotos y responde retornando el correspondiente port de protocolo de ese programa. Una vez que el cliente obtuvo el número de port que está usando el servicio remoto, podrá contactarlo en el futuro en forma directa.

¿Cómo se implementa una herramienta C/S?

La herramienta RPC lee un archivo de especificaciones proporcionado por el programador y produce como salida archivos fuentes de código C++.
El archivo de especificaciones contiene la declaración de las constantes, tipos de datos globales, datos globales y los procedimiento remotos (incluyendo los argumentos y los tipos de resultado).
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 (RPC CALL), enviar el mensaje al programa remoto, esperar por los resultados y traducir los valores resultantes a la representación nativa de la máquina del cliente. Del lado del servidor, 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.

9 Dynamic mapping service.
10 Mappings.

El código adicional requerido para RPC se puede agregar en la forma de dos nuevos procedimientos que encapsulen completamente los detalles de comunicación. Estos, agregarán la funcionalidad necesaria sin cambiar la interfaz original entre los procedimientos llamador y llamado.
Los procedimientos adicionales agregados a un programa para implementar RPC, reciben el nombre de *stub procedures*.
Cada stub se divide en dos partes, rutina de interfaz y rutina de comunicación, tal que rpcgen resuelve qué convención de llamadas usar para la rutina de comunicaciones, mientras el programador puede elegir cuál usar para el procedimiento remoto. El programador entonces, crea los stubs de interfaz para hacer coincidir las convenciones de llamadas entre el procedimiento remoto y las convenciones suministradas por los stubs de comunicaciones que genera RPCGen.
rpcgen toma como entrada un archivo de especificaciones de un programa remoto en nuestro caso:

Versions.x

```
program VERSIONS
{
    version VERSIONS_1
    {
         int SAVE(int) = 1;
         int OPEN(open_rec) = 2;
         int CREATE(open_rec) = 3;
         int CLOSE(int) = 4;
         version_rec GIVE_CURR_VERSION(int) = 5;
         table_vg GIVE_VGA(give_vg_rec) = 6;
         int GIVE_VERSION_ID(give_version_id_rec) = 7;
         int WRITE(version_write_rec) = 8;
         res_read_v_rec READ_VERSION(set_ver_rec) = 9 ;
         int DEL_VERSION(set_version_rec) = 10;
         int DEL_VERSION_TREE(set_version_rec) = 11;
         int RENAME_VERSION(ren_v_rec) = 12;
         int UPDATE_VERSION(upd_v_rec) =  13;
         …
    } = 1;
} = 0x20000001;
```

Ejemplo 4: Versions.x

Se especifica el número del programa servidor, de forma tal de poder ubicarlo en el host remoto. En nuestro caso es: **`= 0x20000001;`**. Con este número el programa cliente consultara al portmapper del host remoto para conocer el port que esta utilizando el servidor.

El número de versión: **`= 1;`**, permitirá determinar si el cliente se esta conectando con un servidor de la misma versión o no, ya que dos versiones distintas pueden ser incompatibles, por lo cual podría ser deseable que el servidor rechace la llamada.

El número de procedimiento identifica el procedimiento remoto. Por cada servicio o procedimiento del programa servidor, vamos a tener una entrada que especifica el tipo del valor de retorno, el tipo del parámetro y un número diferente para cada procedimiento.
Además se establecen que operaciones pueden acceder los clientes en el servidor y los tipos de datos que se pasan como parámetro.

El archivo **`version.x`** será la entrada para el comando rpcgen, el cual generara tres archivos:

`version.h`
Contiene las definiciones utilizadas en los programas, por ejemplo: el número de programa.
`version_svc.c`
Incluye las funciones del *stub* del servidor.
`version_clnt.c`
Aqui se definen las funciones del *stub* del cliente.

De igual forma tenemos un **`Server_ops.x`, `Server_ops.h`, `Server_ops_clnt.C`, `Server_ops_svc.C`** para la especificacion del Servidor de EER.

En la siguiente de muestra como se deben compilar los archivos generados por RPC para construir un cliente y un servidor:

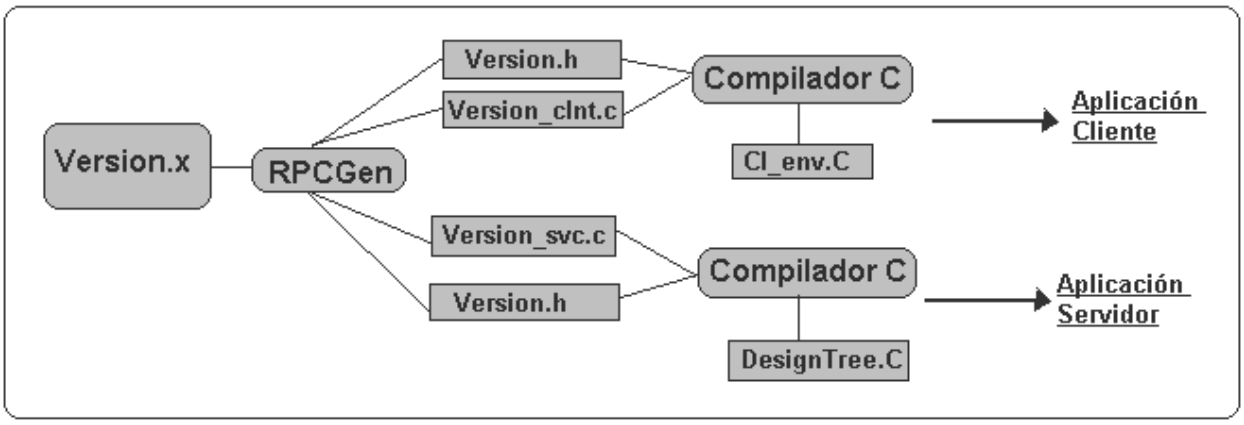


Figure 14: Como se compila

5.2. LEDA (Library of Efficient Data types and Algorithms)

LEDA es una biblioteca de tipos de datos y algoritmos que los manipulan. Provee en forma breve abstracta y precisa una especificación por cada uno de los tipos de datos y algoritmos que define. LEDA contiene un tipo de datos grafo, el cual ofrece las iteraciones estándar como por ejemplo: para todos los nodos del grafo G hacer ... . Permite agregar y borrar vértices y aristas y ofrece arrays y matrices indexadas por nodos y aristas. Esta biblioteca esta implementada en C++. En esta sección

vamos a ver algunas de las definiciones y funciones que utilizamos para el desarrollo del *Manejador de Versiones*.

Definición: **Grafo**
Una instancia del tipo de datos **`Graph`** consiste de una lista *V* de nodos y una lista *E* de aristas. Tanto nodo como arista son tipos de datos

```
graph G;
```

El valor de una variable de tipo nodo es: el nodo de algún grafo ó el valor especial nil ó es indefinido (antes de la asignación inicial a la variable). Lo mismo se aplica para una variable de tipo arista: *E*.
Un grafo con la lista de nodos vacía es llamado vacío.
Un par de nodos (v,w) e *V* esta asociado con una arista e $\in E$, v es llamado la fuente de e y w es llamado el destino de e. v y w son considerados puntos terminales de e. La arista e se dice que es incidente a sus puntos terminales.
Un grafo es o bien directo o bien indirecto. La diferencia es la manera en que las aristas incidentes en un nodo son almacenados y como el concepto de adyacencia es definido.

Implementación de Grafos

Los grafos son implementados por una doble lista de nodos y aristas. La mayoría de las operaciones son de tiempo constante a excepción de **`for_all_nodes`**, **`all_edges`**, **`del_all_nodes`**, **`del_all_edges`**, **`make_map`**, **`make_planar_map`**, **`compute_faces`**, **`all_faces`**, **`clear`**, **`read`** y **`write`**, las cuales tienen un orden n+m, donde n es el número de nodos y m es el número de aristas.

Definición: **Grafo Parametrizado**

Un grafo parametrizado G, es un grafo cuyos nodos y aristas contienen datos adicionales. Cada nodo contiene un elemento de tipo *Vtype* llamado el tipo de nodo de G y cada arista contiene un elemento de tipo *Etype*, llamado el tipo arista de G. *Vtype* y *Etype* son definidas por el usuario.

```
GRAPH < Vtype , Etype > G;
```

Usamos la notación <*v,w,y*> para denotar una arista *(v,w)* con información *y*, <*x*> denota un nodo con información *x*.
Todas las operaciones definidas para el tipo básico grafo, son también definidas en instancias de cualquier tipo de grafo parametrizado. Para los grafos parametrizados existen operaciones adicionales para acceder o actualizar la información asociado con sus nodos y aristas.
Instancias de un tipo de grafo parametrizado pueden ser usadas dondequiera que una instancia del tipo de datos grafo pueda ser usado, por ejemplo en asignaciones o como argumento de funciones con parámetros formales del tipo grafo.

Si una función **f(graph &G)** es llamada con un argumento **Q** del tipo **GRAPH(Vtype, Etype)**, entonces en el cuerpo de **f** solo se accede a la estructura básica de **Q**, **graph**. Esto permite el diseño de algoritmos de grafos genéricos.

Implementación de Grafos Parametrizados

Los grafos parametrizados son derivados de los grafos dirigidos. Todas las operaciones adicionales para el manejo de los nodos y aristas tienen tiempo constante.
En nuestro modelo de versiones implementamos el grafo de inclusión y el grafo de derivación con grafos paramétricos de LEDA, cuyos nodos son las versiones y cuyas aristas corresponden a las relaciones de inclusión y derivación.
Algunas de las funciones que utilizamos para recorrer los grafos son las siguientes:

- **forall_nodes (version v, Graph G)**

Los nodos de G son sucesivamente asignados a v.

- **forall_edges (edge e, Graph G)**

Las aristas de G son sucesivamente asignadas a e.

5.3. ILOG VIEWS

Ilog Views es una biblioteca C++ que permite desarrollar interfaces gráficas en ambientes: XWindow y Microsoft Window. Esta herramienta tiene un diseño orientado a objetos con clases bien definidas y documentadas, por lo tanto promueve al desarrollo de aplicaciones modulares y encapsuladas en clases jerárquicas. Además permite que se realicen extensiones a las clases predefinidas, brindando el comportamiento deseado.
Otro aspecto importante de esta herramienta es que incluye una clase de objetos para la creación de grafos (**IlvGrapher**).

El motor gráfico de Ilog (**IlvGraphic**) incluye un conjunto de primitivas con todo lo que se necesita para crear una interfaz. Los objetos gráficos contienen toda la especificación necesaria para describir los aspectos visuales. Los métodos que se incluyen en la clase son:

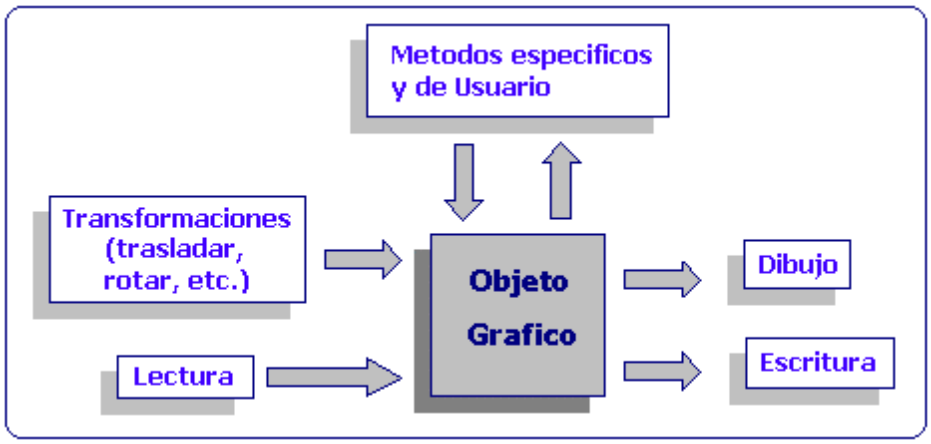


Figure 15: Motor Gráfico de ILOG Views

Otra consecuencia del diseño orientado a objetos, es que todos los objetos gráficos tiene el mismo conjunto de operaciones sin importar su tipo.
Por ejemplo: si se desea modificar el tamaño de un círculo y de un rectángulo, la única operación aplicable es *resize* sin importar la estructura interna de estos objetos.

Una interfaz gráfica no se compone solamente de objetos gráficos sino de la conducta que se le asignan a los mismos, es así que ILOG Views separa los aspectos gráficos de su *"behavior"*. Esto es una ventaja, pues si la presentación de los objetos esta definida conjuntamente con su comportamiento, agregar un nuevo comportamiento provocaría la creación de un nuevo objeto gráfico. Además existe un conjunto standard de *interactors* que permite realizar cualquier combinación de ellos en un objeto gráfico.

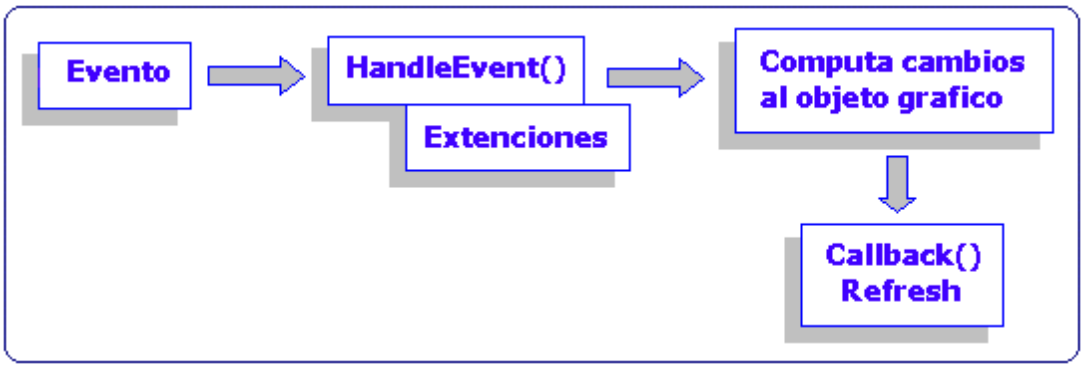


Figure 16: ILOG VIews

Detallaremos los componentes mas utilizados en nuestra aplicación:

IlvDisplay

Es la clase más importante de toda la biblioteca pues es la que maneja todos los aspectos de conexión del sistema gráfico.

Las tareas más importantes de esta clase son:

- Comandos de dibujo: se usan para dibujar líneas, rectángulos, etc. Estos objetos son dibujados en memoria o en la pantalla dentro de la región definida en la constructora de esta clase.
- Recursos gráficos: colores, estilos, patterns, fonts, etc.

IlvContainer

La clase **`IlvView`** representa una ventana, básicamente esta clase no brinda ningún otro servicio. De ella deriva **`IlvContainer`** que puede contener un conjunto de objetos gráficos. Esta clase es la que administra la mayor parte de los controles sobre los objetos por ejemplo: dibuja los objetos gráficos, controla cuales serán visibles o invisibles, la conducta que tendrán asociadas, etc. Es aconsejable que cada objeto gráfico tenga un nombre en el container para que sea mas claro su acceso.
Los containers tienen operaciones que permiten leer un conjunto de objetos gráficos desde un archivo (.ilv) y desplegarlos en la ventana del container. El **`container`** es la clase que capta los eventos realizados por los usuarios y dispara los **`callbacks`** asociados. Por ejemplo: el usuario con el mouse desea mover un objeto gráfico, la clase container es la que determina el objeto que es seleccionado por el mouse y despacha la acción que se llevará a cabo.

Accelerators

La clase **`IlvContainer`** representa los objetos de Ilog Views que también participan de las interacciones del usuario. Estos, para simplificar la traducción de eventos a acciones, possen **`accelerators`**. Cada **`accelerator`** tiene un tipo de evento asociado, por ejemplo: double-clicked, left mouse, Ctrl Q, Alt X, click, etc. Cuando ocurren "disparan" el código del **`callback`** asociado al **`acelerator`**. Para asociar una acción con un evento dado, se tiene que agregar un **`accelerator`** al **`container`** (**`addAccelerator`**).

Un **`accelerator`** administra un único evento de usuario que puede ocurrir en un **`container`**. Este es una conexión directa entre el evento realizado por el usuario y el llamado a una función.
Por ejemplo: Cuando se oprime la tecla "Q" se dispara la función Quit.

```
static void ILVCALLBACK
Quit (IlvContainer* cont,IlvEvent &,IlvAny)
{
     ……..
}
Cont->addAccelerator(Quit,IlvKeyUp,'Q´,0);
```

Ejemplo 5: Como se realizaria un Callback

Interactors

Los interactors se asocian a objetos gráficos. Filtran los eventos de usuario a través del gráfico al cual están asociados. Si el evento apropiado ocurre, entonces el interactor es el responsable de "disparar" la función asociada al evento. La respuesta es la conducta del objeto.
Por lo tanto permiten: mover, realizar scroll, comportamiento de botón, sliders, toggle, etc. Extender esta clase es simple, cada clase interactor tiene una función miembro llamada **`handleEvent`** que es llamada cuando el evento asociado al **`interactor`** ocurre. Un nuevo **`interactor`** tiene que únicamente redefinir esta función miembro.
Los **`interactors`** son similares a los **`callbacks`** en Motif y Microsoft Windows, pero son más fáciles de usar. Los **`callbacks`** son llamados con un solo tipo de objetos y de la cual toda la información para realizar el evento se debe derivar, por lo que se debe obtener el objeto y realizarle un cast al tipo de datos apropiado al **`callback`**.

IlvManager

Mientras un container atiende los eventos del conjunto de objetos gráficos que contiene, un Manager maneja la información y el despliegue en una variedad de contextos (IlvView). Mejoran la performance, la funcionalidad y el control global del grupo de objetos gráficos.
Posibilitan el uso de transformaciones geométricas sobre el conjunto de objetos gráficos pertenecientes a cualquier vista (IlvView).

¿ Qué es ILOG Views Studio ?

Ilog Views Studio es una herramienta para desarrollar interfaces GUI. Permite editar y generar aplicaciones con múltiples ventanas y paneles.
El editor de GUI, posibilita la combinación de los componentes standard de interfaz con objetos gráficos. Luego de esto, simplemente seleccionando la plataforma, se genera el código C++ necesario. Es esto lo que asegura la portabilidad de las diferentes aplicaciones GUI.
Cada ventana que compone una aplicación (llamadas paneles) tiene asociada una clase que hereda las propiedades de la clase IlvContainer. El nombre de esta clase es definido por el usuario del Studio.

La descripción gráfica de este panel se almacena en un archivo (*.ilv). La descripción global de la aplicación se almacena en un archivo (*.iva).

Por lo tanto, Ilog Views Studio genera para cada aplicación:

- Un archivo fuente C++ y header de la clase aplicación.
- Un archivo fuente C++ y header para cada panel que forma parte de la aplicación.
- Un archivo *.ilv con la representación visual de cada uno de los paneles.
- Un archivo *.iva con la descripción global de la aplicación.
- Un Makefile (*.mak) para la compilación de toda la aplicación.

Esta separación entre lo visual y la codificación es lo que hace realmente potente a la herramienta, pues es posible modificar el color, posición, tamaño de cualquier objeto de un panel, sin que sea necesario volver a compilar la aplicación.

El fuente C++ contiene funciones que leen estos archivos de especificaciones visuales (*.ilv) y que cargan los paneles.

A continuación mostraremos como extendimos las clases generadas por Ilog Views Studio para la implementación del *Manejador de Versiones*.

El *Manejador de Versiones* esta compuesto por un conjunto de ventanas y una menu bar.

Para ello definimos en Ilog Views Studio los paneles realizando “drag-and drop” desde la ventana de Gadgets al panel principal de Ilog Views Studio. Algunos de los paneles creados fueron: Open, Create, Read, Write, Delete, etc. Por ejemplo: la ventana de Open es la ventana que solicita el nombre de la aplicación y el nombre del módulo, para luego llamar a la función Open_CMP, con estos valores como parámetro.

En la generación del código de estos paneles, se puede especificar que el código de los callbacks sea generado o no. Si se especifica que no se generen los callbacks, estos pueden ser implementados en archivos separados.

Para poder acceder y asociarles callbacks e interacciones a los diferentes objetos gráficos dentro de cada panel es aconsejable asignarle a cada uno un nombre. Estos nombres se utilizarán luego para agregar a la clase de cada panel, las funciones miembros que devuelven dichos objetos gráficos.
Por ejemplo: OK_Open_button al botón de OK en el panel de Open, esto se hace ingresando el nombre en la casilla Name dentro del Generic Inspector.

De igual forma le asociamos a cada botón un callback. Por ejemplo en la ventana de Open existen dos botones, uno de OK con el nombre del callback asociado: OK_Open y Cancel para el callback asociado al botón de Cancel.
Luego se salva el panel y se generan los *.ilv para cada panel.

Luego en el panel Application Editor de ILOG Views Studio, indicamos los paneles que van a formar parte del *Manejador de Versiones*. El código generado para esta aplicación será el encargado de crear y desplegar los paneles. Al salvar se almacena el archivo aplicacion.iva y generamos el conjunto de paneles y la clase aplicación.

Extensión del código generado por ILOG Views Studio

Paneles

Los paneles son una instancia de la clase IlvContainer de Ilog Views por lo que heredan todas las propiedades antes vistas.
En el archivo Panel.h generado define la forma en que el panel será cargado una vez que la función constructora de esta clase sea invocada. Esta información puede ser un string que se genera en el Panel.h o un archivo, esto depende de como se especificó la propiedad Panel Class Files en el momento que se generó el panel.
Si esta propiedad fue seleccionada se generará un string que será accedido por la función *istrstream* en la función ***initialize*** de dicho panel.

Por ejemplo una extensión que le realizamos a los paneles generados es la definición de los callbacks de Quit en cada uno. Si especificamos que los callbacks no se almacenen en el momento en que se definió el panel se puede crear un nuevo archivo que incluya:

```
#include “panel1.h”
     void  Panel1::quit(IlvGraphic* g)
     {
          IlvContainer*              container          =
          IlvContainer::getContainer(g);
          IlvDisplay* display=container->getDisplay();
          Delete container;
          Delete display;
          IlvExit(0);
     }
```

Ejemplo 6: Archivo extendido con los Callback

Como ya se mencionó anteriormente, si los objetos gráficos que componen el panel, fueron nombrados, se generaran miembros que devuelven los objetos gráficos nombrados. Por ejemplo: OK_Open_button al boton de OK en el panel de Open:

```
IlvButton* getButton() const
     {return (IlvButton *)getObject(“OK_Open_button”)}
```

Ejemplo 7: Obtener un objeto gráfico en un panel

Para el almacenamiento de las posiciones de los nodos, existen dos funciones: Load y Save que guardan en archivos *.psc la distribución de las versiones. Hay un archivo por grafo. Se pueden llamar desde el menú o también con <Ctrl>L y <Ctrl>S.

Aplicación

ILOG Views tiene una clase C++ llamada: **`IlvApplication`** para administrar el conjunto de paneles definidos en una aplicación. Algunas de los miembros mas usados:

```
IlvApplication(const char* name,const char * displayName,int argc =
0,char** argv =0 )

GetName:devuelve el nombre de la aplicación.

GetDisplay:devuelve el Display de la aplicación.
```

- Inicialización de la Aplicación:

La aplicación se inicializa (***initialize***) una sola vez y al comienzo de la ejecución. Básicamente consiste e la creación de las clases de los paneles que la componen, para lo cual se ejecuta el método ***makePanels*** Éste método es definido en la clase IlvApplication y tiene el control de:

- creación de las clases de los paneles, pasándole como parámetro, entre otras cosas, la posición el tamaño de la ventana.
- Agregar cada panel a la aplicación (función ***addPanel***).
- Mostrar el panel (***show***).

Los paneles de una aplicación están almacenados en una lista interna y son agregados automáticamente en la función **`initialize`** a través de la función **`addPanel`**. Consecuentemente se utilizará esta función únicamente para aquellos paneles que no fueron creados con Ilog Views Studio.

La función miembro **`getPanel`** permite acceder a un panel por su nombre.

Es así que en el *Manejador de Versiones* por ejemplo para mostrar la ventana de Open realizamos:

```
IlvApplication* apli = getApplication();
IlvDisplay* display = apli->getDisplay();
IlvContainer* cont_drv = apli->getPanel("Open");
cont_drv->show;
```

Ejemplo 8: Mostrar un Panel

Las funciones **`showPanel, hidePanel`** las utilizamos para mostrar y ocultar los paneles.

Las propiedades de la aplicación y el conjunto de paneles que forman la aplicación están almacenadas en un archivo “aplicacion.iva”. No es aconsejable que los desarrolladores modifiquen este archivo sin utilizar el Application editor.

El *Manejador de Versiones* esta compuesto de una Menú Bar la cual es otro objeto gráfico que esta contenido en un panel cuyo nombre de clase asociado es: Menu1.
Es en este fuente (Menu1.cc) donde se agregaron Callbacks para la realización de las distintas operaciones de nuestra aplicación.
Estos Callbacks en un primer paso llaman a los paneles de ingreso de datos y luego a la función correspondiente pasándole como parámetro los datos ingresados.

En la función initialize de la clase Menu1, se registran todos los Callbacks que se ejecutarán al seleccionar una opción del menú.

```
registerCallback("open_CMP", _open_CMP);
```

Grafos

Además del Menú Bar, existen dos ventanas, una para cada grafo. Fueron creadas con el Studio, pero se agregaron en la aplicación en forma manual.
A continuación vemos un ejemplo, para la ventana del grafo de derivación:

```
IlvContainer* cont_drv = new IlvContainer(display,
     "Cont_drv","Cont_drv",IlvRect(9,114,470,514), IlvTrue,
     IlvFalse);
strstream stream((char*)_datadrv);
cont_drv->read(stream);
cont_drv->draw();
IlvManagerRectangle* rectmgr_drv =
     (IlvManagerRectangle*)cont_drv->getObject("mgr");
IlvGrapher* grapher_drv= new IlvGrapher(display);
rectmgr_drv->setManager(grapher_drv);
IlvView* managerview_drv = rectmgr_drv->getView();
IlvGraphSelectInteractor(grapher_drv, managerview_drv);
IlvManagerViewInteractor* select_drv = new
     IlvGraphSelectInteractor(grapher_drv,managerview_drv);
grapher_drv->setInteractor(select_drv);
```

Ejemplo 9: Paneles de los Grafos

Luego de haber creado el container y de haber leído la especificación de la pantalla (generada con el Studio), el container se castea a IlvManagerRectangle para luego poder asociarle a través del método setManager el grafo correspondiente.
La función SetManager establece el manager que maneja la vista (IlvView) en la cual esta almacenada el objeto.
También creamos una instancia de la clase IlvGraphSelectInteractor. Esta clase es un interactor especifico de los grafos que permite seleccionar, modificar el tamaño y mover los objetos (nodos). Los lazos se recalculan automáticamente, cada vez que se modifica la posición de un nodo.

6. CONCLUSIONES

A lo largo de todo el documento hemos trabajado sobre la idea general de un *Manejador de Versiones*, las características especiales del Manejador de Esquemas ER, su arquitectura, implementación y herramientas utilizadas para implementarlo.

Hemos visto la importancia del *Manejador de Versiones* para el manejo ordenado de un proyecto y su utilidad para poder tener un histórico de los cambios realizados. En especial, el Manejador de Esquemas ER, al permitir definir la relación de inclusión, evita la proliferación de versiones.
Por otra parte, debido a su arquitectura portable, puede integrarse con cualquier otra herramienta, ofreciendo una interfaz amigable.

Aunque el Manejador ya se encontraba implementado, así como muchas de sus funcionalidades, se tuvieron que realizar múltiples cambios en sus módulos, re-programar código y codificar nuevos módulos para poder implementar la nueva arquitectura C/S e integrar la nueva interfaz gráfica. Esta última le da gran potencialidad a la herramienta, debido a su fácil manejo y amigabilidad.

Algunos de los puntos mas remarcables del proyecto, con aportes importantes, fueron:

- Nos introdujo en el tema relacionado a la administración de versiones, del cual no teníamos mucho conocimiento, y que sin duda es de gran ayuda durante el procesos de desarrollo.
- Manejo de RPC. Al comenzar con la implementación tuvimos que realizar un estudio exhaustivo de esta herramienta, para poder implementar la conexión entre el Cliente y el Manejador de Esquemas ER, a través del Servidor de Versiones.
- Estudio de ILOG VIEWS. Para implementar la interfaz gráfica así como sus funcionalidades, tuvimos que aprender a utilizar esta herramienta. Lo importante no es en si, el haber aprendido a manejarla, sino que dado que la misma esta orientada a objetos, esto nos permitió conocer como trabajan este tipo de utilitarios, la potencialidad de los mismos y poder trabajar fluidamente con ellos.

Hoy en día, controlar las versiones de software ya no es suficiente. La situación actual implica:

- los equipos de desarrollo están distribuidos a lo largo de diferentes lugares físicos
- el desarrollo en paralelo es cada vez mayor como consecuencia de: planificación de tiempo y mercado, mantener versiones previas y soportar las variaciones de las aplicaciones para distintas plataformas.

Si combinamos esto con:

- ambientes distribuidos en múltiples plataformas
- aplicaciones Cliente/Servidor
- eficientes herramientas de desarrollo

- código compartido entre diferentes aplicaciones
- Internet/Intranet

Lo que se esta requiriendo, hoy en día, es una forma de manejar los complejos proyectos de software, además de dar poder a los miembros del equipo de desarrollo de forma tal que, juntos, trabajen de manera efectiva.
En pocas palabras, la solución que se busca debería:

- Mantener una traza de los cambios de todos los componentes del proyecto.
- Soportar el desarrollo en paralelo (ramificaciones, diferencias, uniones, equipos distribuidos)
- Controlar el proyecto entero y su evolución en el tiempo.
- Administrar los cambios.
- Manejar las nuevas construcciones y dependencias.

De todas las técnicas y herramientas investigadas no existe ninguna aplicada a administrar el proceso de Modelado Conceptual.
Aunque muchos conceptos y modelos de los antes presentados son utilizados en este *Manejador de Versiones*, algunas no son aplicables a esquemas conceptuales. Por otro lado en la Administración de Versiones de Esquemas ER, es interesante incorporar nuevos conceptos como ser la relación de construcción, comparación de esquemas, integración, etc.

6.1. Trabajo Futuro

6.1.1. Concurrencia

Las operaciones realizadas vía RPC son sincrónicas es decir el proceso cliente esta bloqueado hasta que el proceso del servidor halla finalizado.
En algunos casos esto puede no ser aplicable por lo que sería 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 thread y luego continúa su ejecución.

6.1.2. Relación de Construcción

Como vimos la relación de construcción involucra esquemas ER reusables.
Los esquemas reusables son definidos dentro del Construction Tool, que como ya se explicó, forma parte de la herramienta CASE.

Los esquemas se pueden crear:

- Aplicando funciones de esquemas ER (unión, diferencia, etc.) a esquemas existentes y creados en este CASE.
- Integrando nuevos esquemas que fueron creados con otras herramientas y satisfacen un Control de Calidad de esquemas ER.
- Aplicando funciones de esquemas ER en la combinación de los antes mencionados.

Estos esquemas así creados en el Modelo de Versiones pueden pertenecer a distintos CMP, por lo que la relación de construcción debería implementarse en la clase **`V_system,`** que es la que almacena el universo de esquemas conceptuales.

Además, al igual que los otros grafos implementados, estos serían por cliente que se encuentre trabajando.
Su estructura sería un vector de grafos paramétricos (similar a los ya implementados), donde cada posición en el vector representaría el numero de conexión de cada cliente.

Su implementación:

```
grafo_of_construcc = array ( Cnn_number ) of grafo_cons;
grafo_of_cons = GRAPH (  Arc , Versions_Reusables );
```

donde **`Versions_Reusables`** es una instancia de la clase **`Versions.`**

¿ Cómo se cargaría esta estructura ?

El Construction Tool se comunica con el *Manejador de Versiones* de dos formas: vía RPC y a través de archivos.

Por otra parte el *Manejador de Versiones* al comenzar la conexión de un nuevo cliente y al abrir un CMP, se tendría que conectar al Servidor de Versiones y de igual forma que carga la estructura de la relación de inclusión y derivación, cargaría esta nueva estructura.

Luego esta nueva relación se tiene que reflejar en la interfaz gráfica, por lo que se debe extender la misma para hacer posible su visualización.

Se debería de desplegar el grafo de construcción para la versión corriente del CMP.

En el *Manejador de Versiones* esta relación es visual, es decir que ninguna operación que se realize en esta herramienta modificara el grafo de construcción, por lo que unicamente se calculará al comenzar la sesión.

7. APÉNDICE

La clase del cliente

```
class Client_Env {
      char  *server_VM;
      CLIENT     *cl_VM;
      char  *server_OPS;
      CLIENT     *cl_OPS;
public:
      int conn_number;// num. de conexión cliente.
      table_vg  vg_t;       // Estructura donde se almacenan
                              los dos grafos.
      char obj_names[MAX_LENGHT_ID][MAX_QTTY_OBJECTS];
      objs_in_cmp obj_cmp;
      Client_Env(int cn, void (*f)(char *a[],
            Client_Env *));

// Funciones de acceso al Version Server
      void conn_server_VM(char *server_name);
      int create_module(char *a,char *m,char *w);
      int open_module(char *a,char *m,char *w);
      int close_version_plane();
      int save_version_plane();
      void give_vg_arrays(); //carga en vg_t los dos grafos
      vp_rec *give_vp_info();
      version_rec *give_curr_version();
      int give_version_id(char *ver_name);
      void get_vg_arrays(display_pkg *dp);
      list_VIds *ancestors(int v_id);
      version_rec *read_current_version(int version_id);
      vp_rec *write_version(char *v_n,char *v_v,char *o_v);
      int def_super_version(int sub_v_id, int super_v_id);
      int delete_version(int v_id);
      int update_version(int vers_id, char* new_val);
      int rename_version(int vers_id, char* new_name);
      int update_drv_frm(int op_id, char* new_op_v);
      result_read_v_rec *set_current_version(int vers_id);
      int up() ;
      int del_branch(int from,int to);
      int del_includes(int );
      list_VIds *op_reperc_path(char *, int v_begin);
      list_VIds *op_reperc_tree(char *, int root);
      list_VIds *op_reperc_sub_branch(char *,int from,
            int to);
      int op_reperc_del_path(char *, int root);
      CLIENT *give_client();
};
```

Clases de la aplicación gráfica que se integra al Manejador de Veriones.

```
class Versiones: public IlvApplication {
public:
      Versiones (
      const char* appName,
      const char* displayName = 0,
      int argc = 0,
      char** argv = 0
      );
    Versiones (
        IlvDisplay* display,
        const char* appName
    );
    ~Versiones();
    virtual void makePanels();
    virtual void beforeRunning();
};
```

Clase del Menú de la aplicación gráfica

```
class Menu1 : public IlvGadgetContainer {
public:

// Extensión de la clase
      char* server;
      int conectado;
      IlvGraphicSet* Set_drv;
      IlvGraphicSet* Set_incl;
      int cnt_nodes_drv;
      int cnt_links_drv;
      int primera_vez;
      int cnt_nodes_incl;
      int cnt_links_incl;
      gr_pos gr_incl_pos[200];
      gr_pos gr_drv_pos[200];

    Menu1(IlvDisplay* display,
          const char* name,
          const char* title,
          IlvRect*    size             = 0,
          IlvBoolean  useAccelerators = IlvFalse,
          IlvBoolean  visible          = IlvFalse,
          IlvUInt     properties       = 0,
          IlvSystemView transientFor  = 0,
          char* serv,
          int conectado)
```

```
      : IlvGadgetContainer(display,
                           name,
                           title,
                           size ? *size: IlvRect(0,0,900,65),
                           properties,
                           useAccelerators,
                           visible,
                           transientFor)
          { initialize(serv,conectado); }
    Menu1(IlvAbstractView* parent,
          IlvRect* size = 0,
          IlvBoolean useacc = IlvFalse,
          IlvBoolean visible = IlvTrue,
          char* serv,
          int conectado)
      : IlvGadgetContainer(parent,
                           size ? *size : IlvRect(0,0,900,65),
                           useacc,
                           visible)
        {initialize(serv,conectado);}

    virtual void sub_tree_sch_op(IlvGraphic*);
    virtual void delete_version(IlvGraphic*);
    virtual void write_version(IlvGraphic*);
    virtual void open_module(IlvGraphic*);
    virtual void up(IlvGraphic*);
    virtual void create_gt(IlvGraphic*);
    virtual void give_info_module(IlvGraphic*);
    virtual void path_sch_op(IlvGraphic*);
    virtual void new_module(IlvGraphic*);
    virtual void del_tree(IlvGraphic*);
    virtual void upd_drv_link(IlvGraphic*);
    virtual void update_version(IlvGraphic*);
    virtual void save_module(IlvGraphic*);
    virtual void del_incld_gt(IlvGraphic*);
    virtual void del_br_gt(IlvGraphic*);
    virtual void del_all_gt(IlvGraphic*);
    virtual void ancestors(IlvGraphic*);
    virtual void Quit(IlvGraphic*);
    virtual void del_path(IlvGraphic*);
    virtual void give_id(IlvGraphic*);
    virtual void rename_version(IlvGraphic*);
    virtual void branch_sch_op(IlvGraphic*);
    virtual void del_incls_gt(IlvGraphic*);
    virtual void close_module(IlvGraphic*);
    virtual void read_version(IlvGraphic*);

    // Actualiza ambos grafos en pantalla
    virtual void Refresh(IlvGraphic*,IlvAny);
    virtual void AddTree(IlvGrapher*,IlvDisplay*,int);
    virtual void DeleteNodes(IlvGrapher*,IlvDisplay*,int);
```

```
     virtual void Current_1(IlvContainer *, IlvEvent&,
           IlvAny);

 // Guarda las posiciones de los nodos en un archivo (gr_drv.psc
y gr_incl.psc).
      virtual void Save();

 // Carga las posiciones de los nodos desde los dos
archivos: gr_drv.psc y gr_incl.psc.
      virtual void Load();

// Funciones generadas por ILOG Views Studio
    IlvButton* getbtn_open() const
      {return (IlvButton*)getObject("btn_open"); }
    IlvButton* getbtn_save() const
      {return (IlvButton*)getObject("btn_save"); }
    IlvButton* getbtn_delete() const
      {return (IlvButton*)getObject("btn_delete"); }
    IlvButton* getbtn_read() const
      {return (IlvButton*)getObject("btn_read"); }
    IlvButton* getbtn_quit() const
      {return (IlvButton*)getObject("btn_quit"); }
    IlvButton* getbtn_give_id() const
      {return (IlvButton*)getObject("btn_give_id"); }
    IlvButton* getbtn_new() const
      {return (IlvButton*)getObject("btn_new"); }
    IlvButton* getbtn_give_info_mod() const
      {return (IlvButton*)getObject("btn_give_info_mod"); }
    IlvButton* getbtn_write() const
      {return (IlvButton*)getObject("btn_write"); }

      protected:
           void initialize(char* serv, int conectado);
};
```

Clases del Servidor de Versiones

```
class Version {
      int id;
      char name[20];
      char *value;
      int status ; /* 0: Deleted, >0: Active */
public:
      Version(char* name, char* val);
      Version(int id, char* name, char* val, int status);
      int give_id();
      char* give_name();
      char* give_value();
      int give_status();
      char* give_all();
      void set_name(char* st_name);
      void set_id(int st_id);
      void set_value(char* value);
      void set_status(int status);
};
```

```
class Arc {
protected:
      int id;
public:
      Arc();
      Arc(int);
      void set_id(int new_id);
      virtual int give_id();
      virtual char* give_all();
      virtual char* give_all_formated();
      virtual void set_value(char *new_val) { ; };
      virtual char* give_value() { return 0; };
      virtual char* type() { return("Arc"); };
};
```

```
class Derivation_from: public Arc {
      char *value;
public:
       Derivation_from( char* value);
       virtual void set_id(int new_id);
       virtual void set_value(char *new_val);
       virtual char* give_value();
       virtual char* type() { return("Operation"); };
       virtual char* give_all();
       virtual char* give_all_formated();
};
```

```
class Derivation_to: public Arc {
public:
      Derivation_to();
      virtual char* type()
            {return("Previous_state");};
};
```

```
class Inclusion_from: public Arc {
public:
      Inclusion_from();
      virtual char* type()
            {return("Super_version");};
};
```

```
class Inclusion_to: public Arc {
public:
      Inclusion_to();
      virtual char* type()
            {return("Sub_version");};
};
```

```
typedef Version* pVersion;
typedef Derivation_from* pDerivation_from;
typedef Arc* pArc;

class CMProjects {
friend class Version;
public:
            char application_name[20];
            char module_name[20];
            char workspace_name[20];
           CMProjects *version_graph;
            char name[20];
            GRAPH<pVersion,pArc> structure;
            node root, current_version;
            int last_id_version;
            int last_id_drv_frm ;

            CMProjects(char*, char*, char*);
           ~CMProjects();
```

```
        node give_root();
        Version* give_version(node);
        Arc* give_arc(edge);
        GRAPH<pVersion,pArc>* give_structure();
        int give_last_id_version();
        int give_last_id_drv_frm();
        int give_id_version(node);
        int give_id_drv_frm(edge);
        int give_curr_id_version();
        int give_curr_id_drv_frm();
        node add_version(node parent, Derivation_from*,
             Version*);
        node add_version(Derivation_from *, Version*);
        int  delete_version(int id_version);
        void delete_sub_tree(node root, int id_drv_frm);
        int del_sp_link(edge e);
        int def_inclusion_from(int sub_v_id,
             int super_v_id);
        int update_version(int id_v, char* new_val);
        int rename_version(int id_v, char* new_name);
        int update_drv_frm(int drv_frm_id, char*
             new_drv_frm_v);
        node search_version(char* st_name);
        node search_version(int st_id);
        edge search_edge(int id_arc);
        edge search_OpBetweenNodes(node source,
             node target);
        edge search_edge_OpToState(int id_drv_frm,
             node source);
        edge to_drv_to(node version);
        edge to_descendent(node source, node target,
             char *t_link);
        node previous_version(node);
        int ancestors(node version_id,
             char* type_of_link, node l_ancs[]);
        int recorro_Inclusion(int v_id,char *,int r[]);
        node inclusion(node n,char * link);
    int path(int st_id, int result[] );
        int sub_tree(int root_st_id,char *type_link,
             int r[], int &tope );
        int sub_branch(int st_id_from,int st_id_to,
             int res[]);
        int ancestors_trans(int st_id, char*
             type_of_link, int result[] );
        int qtty_ancestors(node st);
        int ordinal_in_brother_list(node st);
        int number_versions();
        int number_arcs();
```

```
// storage
          void save(char * file_name);
          int open(char * file_name);
};
```

```
class V_SYSTEM {
public:
     CMProjects * versions[50];

     V_SYSTEM();
     int add_version_plane(CMProjects *);
     CMProjects *give_graph(int cnn_number);
     int give_id_curr_version( int cnn_nunber );
     int give_cnn_number(char*, char*, char*);
     table_vg *give_vg_arrays(int cnn_nunber);
     int find_version(vg_node ver_n[MAX_QTTY_VERS ],
          int ver_id,int tope);
     void load_graph(vg_arrays *vg, GRAPH<pVersion,pArc>
          *tree, char *type_of_link, int tope,vg_node
          ver_nodes [MAX_QTTY_VERSIONS] );
     void save_VM_env(char * file_name);
     int open_VM_env(char * file_name);
     void save_version_plane(int cnn_number);
     int open_version_plane(char*,char*,char*);
     int open_version_plane(char* filename);
     int new_version_plane(char*,char*,char*);
     int close_version_plane(int cnn_number);
     int write_version(int cnn_n,char *v_n,char *v_v,
          char *drv_frm_v);
     Version *read_version(int cnn_nb, int v_id);
     Derivation_from *read_drv_frm(int c_n,int v_id);
     int def_inclusion_from(int ,int sub_v,int super_v);
     int delete_version_subtree(int, int version_id);
     int delete_version(int cnn_nb, int version_id);
     int update_version(int, int vers_id, char* new_val);
     int rename_version(int, int vers_id, char* name);
     int update_drv_frm(int, int drv_f, char* new_drv_f);
     int del_all_sp_links(int cnn_nb, int v_id);
     int set_current_version(int cnn_nb, int version_id);
     int set_current_to_parent_version(int cnn_nb);
     int give_version_id(int cnn_nb, char *version_name);
     vp_rec *give_vp_info(int cnn_nb);
     int path(int cnn_nb,int st_id, int result[] );
```

```
      int del_path(int cnn_nb,int st_id);
      int del_branch(int cnn_nb,int from,int to);
      int delete_included(int cnn_nb,int version_id);
      int delete_includes(int cnn_nb,int version_id);
      int sub_tree(int,int root_st_id,char *type_link,
           int res[], int tope );
      int sub_branch(int cnn_nb,int id_from,int id_to,
           int res[]);
      int ancestors_trans(int c_n,int v_id, char*
           type_of_link, int result[]);
};
```

Ahora que hemos visto la definición de las clases mas importantes, mostraremos a través de un ejemplo como sería el árbol de llamadas (incluyendo el nombre de los archivos fuente) que se generaría cuando un usuario efectúa una operación (en este caso Write) desde el módulo del cliente.

MENU1.CC

```
static void ILVCALLBACK
_write_version(IlvGraphic* g, IlvAny)
{
    Menu1* o = (Menu1*)IlvContainer::getContainer(g);
    o->write_version(g);
}
……
registerCallback("write_version", _write_version);
……
struct vp_rec vp_r;  // Estructura de datos utlizada por RPC para el pasaje de
                        parametros entre los procedimientos remotos. Los
                        campos que la componen se pueden observar en el
                        archivo: Versions.x

CL_ENV = new Client_Env(cn, menu_suit, new Suit_Gr());
……
static void
ILVCALLBACK Ok_write(IlvGraphic* g,IlvAny d)
{
      ……
      vp_r=*CL_ENV->write_version(n1,v1,o1);
      ……
}
```

CL_ENV.C

```
vp_rec *
Client_Env::write_version(char   *v_name,char   *v_value,   char
*op_value)
{
      ……
      if ( (k = write_1(&p, cl_VM)) == NULL ) {
            clnt_perror(cl_VM,server_VM);
            exit(4);
      };
      ……
};
```

VERSION_SV.C

```
static V_SYSTEM *V_M = new V_SYSTEM();
……
int *
write_1(version_write_rec *params, CLIENT*)
{
      ……
      curr_version_id = V_M->write_version(cnn_nb,
            params->version_name,
            params->version_value, params->op_value);
      return(&curr_version_id);
};
```

DESIGN_TREE.C

```
int
V_SYSTEM::write_version(int cnn_nb, char* v_n, char* v_v, char*
op_v)
{
      ……
      CMProjects *dt = give_graph(cnn_nb);
      n = dt->add_version(
            dt->current_version,
            newDerivation_from(op_val),new
      Version(v_name,v_val));
      ……
};
```

DESIGN_TREE.C

```
node
CMProjects::add_version(node root, Derivation_from* oper,
Version* new_version)
{
     edge e, e1;
     node n;

     ++last_id_version;
     new_version->set_id(last_id_version);
     n = structure.new_node(new_version);
     current_version = n;

     ++last_id_drv_frm;
     oper->set_id(last_id_drv_frm);
     e = structure.new_edge(root,n,oper);
     e1 = structure.new_edge(n,root,new Derivation_to() );
     return(n);
};
```

8. GLOSARIO

API - Application Programming Interfaz
Procedimientos, funciones y llamadas provistas por una aplicación o sistema operativo. El desarrollador de aplicaciones utiliza la API como interfaz con la aplicación padre o sistema operativo, en vez de tener que escribir procedimientos personalizados.

Aplicación cliente
Aplicación escrita por un usuario que solicita operaciones remotas que serán llevadas a cabo por un servidor.

Aplicación Servidor
Aplicación escrita por un usuario que lleva a cabo las respuestas a las preguntas solicitadas por los procesos clientes.

Callback
Es una función especifica que el usuario define y se ejecuta al activar el objeto (por ejemplo, hacer click en un botón).

Circuitos VLSI:Very Large Scale Integrated
Integración a muy grande escala.

Facts
Predicado prolog

IP - Internet Protocol - Protocolo de Internet
Le permite a un paquete cruzar varias redes en su camino al destino.

TCP/IP - Transmission Control Protocol
Conjunto de protocolos que permite la conexión a Internet

Operación idempotente
Una operación que si es llevada a cabo no afecta futuras llamadas a cualquier otra operación.

Panel
Es una instancia de una clase IlvContainer utilizada por el ILOG Views Studio. Representa una ventana de la aplicación.

Pipe
Un canal de interconexión entre procesos, es un buffer FIFO (first-in,first-out) con archivos separados de lectura y escritura.

Port
Una red lógica de comunicación. El Portmapper se encuentra en el port 111. Todos los demás servicios son asignados estáticos o dinámicos.

Portmapper
Un utilitario de Sun que permite a otros servicios identificar ports en otras máquinas.

Proceso sincrónico
Un modo de procesar la ejecución. La ejecución queda suspendida hasta que el llamado remoto es respondido.

Procesos threads - LWP light weight processes.
La idea es que un proceso esta compuesto por varias partes: código, datos, I/O, stack, etc. Los threads reducen el "overhead" pues comparten partes fundamentales del proceso, como consecuencia es mucho mas eficiente.

Puerto bien conocido
Es un puerto cuyo número es conocido como una parte de la definición de la red.

Rpcgen
Compilador de protocolos de Sun ONC, utiliza una especificación similar a el lenguaje C

Socket
Un canal IP. Una vez conectado, los procesos para comunicarse pueden leer y escribir.

UDP/IP - User Datagram Protocol
Es un transporte de red que utiliza la conexión a través de datagramas sokets.

XDR - External Data Representation
Es un standard de especificación para la transmisión de datos.

9. BIBLIOGRAFÍA

Libros

- *Power Programming with RPC*

Autor: Bloommer, John
Editor: O'Reilly & Associates, Inc.
Año: 1992

- *Practical Internetworking with TCP/IP and UNIX*

Autor: Carl Mitchell
Editor: Addison Wesley Publishing Company
Año: 1993

Revistas

- *Version-Control Software*

Revista: PC Tech
Fecha: March 1997
Autor: Gabrielle Gagnon

- *R.H Katz and E.Chang "Managing Change in a Computer-Aided Design Database"*

13 th Internacional Conference on Very Large Databes,Brighton UK (September 1987)

Manuales

- *Sun Microsystems*

1988 RFC1057 RPC :Remote Procedure Call Protocol

- *ILOG Views*

Reference Manual 2.3

- *ILOG Views User's Manual*

R 2.3

- *The Leda User Manual*

Version R 3.4.

Referencias en Internet

- *Silicon Beach Communications*
  http://www.silcom.com

- *FortNet*
  http://www.fortnet.org

- *University of Strathclyde - CAD Centre*
  http://www.cad.strath.ac.uk

- *Technical University Berlin - Department of Computer Science*
  http://wwwwbs.cs.tu-berlin.de

Trabajos Académicos

- *Modelado Conceptual*

Autor: Raul Ruggia
ruggia@fing.edu.uy

- *Version Management of Composite Objects*

Autor: Rafi Ahmed
ahmed@hplabs.hp.com

- *Introducción al Sun RPC*

Autores: Federico Wagner, Juan Schaffner
Fwagner@artech.com.uy