UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

U.A.C.P. y P. del C.C.H.

CATALOGO DE COMPONENTES REUSABLES DE SOFTWARE -PROTOTIPO EN ALGRES-

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

ARMANDO HERNANDEZ SOLIS

Directora: Dra. Hanna Oktaba

MEXICO, D. F. 1991

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE

INTRODUCCION

Esta tesis tiene como objeto mostrar el diseño e implantación de un prototipo, para un catálogo de componentes reusables de software (chips de software). Es inspirado en el trabajo de Hanna Oktaba, Concepción Pérez de Celis, y Roberto Zicari [OkCe90].

Cualquier establecimiento que se dedique a comerciar con hardware, tiene un catálogo acerca de sus artículos. El catálogo es una herramienta útil en la organización de sus productos. Contiene información acerca de las características de funcionamiento de cada componente, entre otra. Un catálogo análogo es de gran utilidad para componentes reusables de software, que permite considerar a cada componente como una caja negra, lista para usarse. El catálogo para este caso, debe tener una descripción abstracta de cada componente. Las especificaciones algebraicas son útiles para tal descripción.

El diseño e implantación del catálogo requiere tocar las áreas de especificaciones formales, y bases de datos. Estos temas se tratan en este documento.

El prototipo maneja un diccionario de tópicos y dos bibliotecas:

a) biblioteca de especificaciones abstractas.

b) biblioteca de implantaciones en lenguajes de programación, por ejemplo Modula-2, Ada, C++, ML.

La biblioteca de especificaciones maneja textos como elementos, y la de implantaciones contiene información de reusabilidad de cada implantación, así como el lenguaje usado para su

programación. Tanto en el diccionario de tópicos, como en las bibliotecas, se pueden efectuar las operaciones de crear, incluir y extraer un elemento especifico de una manera organizada.

Las especificaciones abstractas son simples o estructuradas (dependen de otras especificaciones). Cada componente tiene asociada una especificación escrita en el lenguaje de especificación LESPAL [Okta89], y por lo menos una implantación, como se muestra en la figura 0.1.

Figura 0.1. Un componente, su especificación y sus implantaciones.

Tomadas dos implantaciones I_i e I_j de un componente, pueden suceder los siguientes casos:

1. Programadas en lenguajes diferentes, y usan el mismo algoritmo
2. Programadas en lenguajes diferentes, y usan diferente algoritmo.
3. Programadas en lenguajes iguales, y usan diferente algoritmo.

La especificación abstracta de un componente nos sirve para indicar sus propiedades, debiendo existir para todo componente una demostración matemática que garantice que la implantación del componente satisface su especificación.

Los componentes de las librerias que existen comercialmente o que vienen con los compiladores de lenguajes como C, Pascal, Fortran, etcétera, no han sido formalmente probados. En el libro de Lins [Lins89], se hace una reunión de componentes en Modula-2, cuyas especificaciones no son muy precisas. El prototipo supone que las especificaciones y las implantaciones son correctas.

El prototipo brinda, visto de una forma general, el siguiente servicio al usuario:

1. Una lista de tópicos.
2. El usuario selecciona un tópico.
3. Una lista de componentes del tópico seleccionado.
4. El usuario selecciona un componente.
5. Del componente seleccionado se puede elegir entre información de su especificación, o de sus implantaciones.
6. Desplegar la especificación de un componente.
7. Listar los ancestros o descendientes de una especificación, así como las especificaciones que la importan.
8. Mostrar los lenguajes de implantación del componente.
9. Listar la información de reusabilidad de una implantación (como por ejemplo, enseñar la forma de parametrizarlo).

La herramienta que se usó para realizar el prototipo es Algres [Ceri88] (que es un manejador de base de datos basado en un modelo de relaciones anidadas), ya que uno de los objetivos del proyecto fué también utilizar a Algres para una aplicación real. Debido a que el modelo de datos de Algres es una combinación de los modelos jerárquico y relacional, con la primera forma normal relajada, se intuyó que la naturaleza de la estructura de las relaciones que existen entre las importaciones, exportaciones e implementaciones de las especificaciones, se podían expresar apropiadamente en Algres. Algres es un proyecto del Politécnico de Milán, de Italia, el cual ha servido como enlace académico entre la

comunidad de la maestría y esa institución. Los fundamentos teóricos de Algres se tratan en [Guch88].

El catálogo que se propone pretende fomentar la creación y uso de componentes reusables, organizando todos los componentes reusables que se realicen en los diversos lenguajes de programación, que contengan conceptos de reusabilidad. Los componentes tendrán garantizada matemáticamente su funcionalidad, lo cual no sucede con las librerías comerciales y las publicadas hasta el momento. Debe agilizar el desarrollo de componentes complejos de software.

En el capítulo 1, se tocan aspectos históricos sobre el nacimiento del concepto de componente reusable. Se resumen dos sistemas, que tienen un objetivo semejante al catálogo de componentes reusables: el sistema de Prieto y Freeman, y el sistema PARIS ("partially interpreted schemas").

Los componentes reusables que se ordenan en el catálogo tienen asociada una especificación. En el capítulo 2, se tratan las ventajas de las especificaciones formales. Una de ellas es que las implantaciones de un componente son suceptibles a una prueba de correctés, por lo que se supone que cada implantación de un componente, existente en el catalogo es corrrecta. En este capítulo, tambien se dan las características de los métodos formales, que proporcionan infraestructuras para efectuar las etapas en el desarrollo de software; desde la especificación de un programa, hasta la prueba de correctés de la implantación.

El área de bases de datos tomó auge desde que surgieron los primeros modelos clásicos de datos. En el capítulo 3 se resumen los modelos clásicos, y se dan las perspectivas de investigación de esta rama de la ciencia de la computacion. La herramienta usada en la implantación del prototipo de catálogo, fué el ambiente de programación Algres, el cual utiliza el modelo de datos de

relaciones anidadas, también conocido originalmente como NF2. Tanto el modelo como Algres se analizan en el capítulo 4.

En el capítulo 5 se da el diseño del catálogo, como una base de datos en Algres, que utiliza básicamente tres objetos complejos: diccionario de tópicos, especificaciones, e implantaciones. En el conjunto de especificaciones se definen las relaciones H (herencia), e I (importación), las cuales crean tres nuevos conjuntos: ancestros, descendientes, e importaciones. Tales conjuntos se pueden construir como objetos en Algres, a partir de los objetos básicos, y el operador de punto fijo FP (Fix Point). También se indica como interactuar con el sistema.

En los apéndices A, B, y C, se muestran los detalles de implantación de los objetos ancestros, descendientes, e importaciones, respectivamente. En el apéndice D, se enseña como instalar la versión que se tiene de Algres, en otro máquina con sistema operativo Unix.

CAPITULO 1

SISTEMAS ANALOGOS AL PROTOTIPO

En este capítulo, empezaremos dando una perspectiva general de la reusabilidad de software. Posteriormente, se trataran los sistemas que son análogos al prototipo de catálogo de componentes reusables.

1.1 Panorama de la reusabilidad

El gran avance de la computación en los últimos años se debe a que la tecnología empleada en el desarrollo de hardware es cada vez más poderosa, reduciendo en gran medida el costo de la producción de nuevos equipos. No sucede lo mismo en la producción de software, que se realiza lentamente, es muy cara, y no explota al máximo la capacidad del nuevo equipo. Se tiene la creencia de que el atraso del desarrollo de software, se deriva de la falta de reusabilidad de los componentes de programas; ya que todo nuevo sistema se comienza a programar partiendo de casi nada. Un sistema complejo de hardware se hace en gran parte, ensamblando componentes cuyas características se encuentran descritas en catálogos, por lo que el esfuerzo se reduce a encontrar los componentes adecuados que satisfagan las necesidades del nuevo sistema, y a construir lo que no se haya hecho nunca.

Entendemos por *reuso* el uso de conceptos y objetos usados anteriormente en una nueva situación. *Reusabilidad* es una medida de lo fácil que es poder usar esos conceptos y objetos previos en una nueva situación. Al desarrollar software, una de las propiedades que determinan su calidad, es la reusabilidad.

Otro concepto que es importante aclarar es el de componente de software reusable. Un *componente* se define como una parte

simple, o una entidad relativamente compleja considerada como una parte de un sistema. Un *componente de software reusable* puede entonces ser definido como una parte simple, o una entidad relativamente compleja considerada como una parte de un sistema de software, que puede ser rápidamente empleada dentro de otros sistemas.

La idea de catálogos de software reusable ha sido tratada desde 1967 por M. D. McIlroy [McI167] en el articulo ''Mass-Produced Software Componentes''. Entre otros muchos beneficios que proporciona el software reusable, podemos mencionar que incrementa notablemente la productividad de software y su calidad. Las ventajas resultantes del reuso han sido grandemente reconocidas por los ingenieros de software, pero la promesa más brillante ha sido casi irrealizable: ensamblar cualquier sistema con ''componentes reusables'', cuya funcionalidad ha sido formalmente probada. Se ha estimado que al menos 15% de los billones de lineas de código fuente creado a lo largo de los años en el mundo, pueden ser adecuadamente generalizadas para reuso. La escasez de ingenieros de software, la gran variedad de capacitación y productividad entre individuos, y la carestía de programadores altamente productivos, hacen que la práctica del reuso de software no se haya extendido a la industria del software.

Recientemente, los japoneses han reportado en sus fábricas de software gran mejoramiento en la productividad de programadores a través de la reusabilidad. Ellos integran técnicas conocidas de diferentes disciplinas como manejo de recursos, ingeniería de producción, control de calidad, ingeniería de software, y psicología industrial [TaMa84]. Aún más, inventarios viejos de software - los cuales son reconocidos como posesiones muy valiosas - se han modificado para convertirlos en reusables por medio del mejoramiento de su

estructura, documentación, mantenimiento, y actualización [Lyon84].

En los años pasados, varios investigadores han encontrado los factores responsables de la práctica limitada del reuso de software. Estos estudios implican varios problemas técnicos y administrativos. Los investigadores están de acuerdo sobre los factores que originan el problema, y se han clasificado en tres grupos principales: *económico, administrativo y técnico.*

Económico.- se tiene que invertir en la creación de software reusable y no hay quien se interese. Las empresas en las que se acumula el capital, se interesan más en realizar proyectos particulares, por lo que no le dedican ningún recurso al dearrollo de componentes reusables.

Administrativo.- la empresas que se dedican a la creación de programas, no exigen ni premian la creación de software reusable.

Técnico.- los principales problemas técnicos incluyen la organización y recuperación de componentes reusables, variaciones en los niveles de reuso (código objeto, código fuente, diseño), ausencia de técnicas adecuadas de diseño aplicables al reuso de software, pocos son los lenguajes de programación que soportan el reuso, y las dificultades encontradas en el intento de escribir software reusable.

Los dos primeros factores son consecuencia del tercero, el cual se debe a la falta de fundamentos teóricos y experiencia en el campo de la reusabilidad. Comparando esto con los componentes de hardware, se nota claramente la diferencia. Los fundamentos matemáticos y físicos utilizados en la creación de un componente electrónico están perfectamente determinados y estudiados, por lo que se puede dar una descripción clara y precisa de las características de cada componente, que los especialistas entienden sin ninguna dificultad.

No hay suficiente experiencia en el área de programación para dar el soporte teórico y práctico necesario para producir componentes reusables. Aunque ya existen métodos formales para el diseño de programas (por ejemplo: [Hoar89], [Dijk88], [Grie81]), y lenguajes de programación con sintaxis y semántica precisamente definidas (por ejemplo: Standar ML [Harp86]), su aplicación en la práctica requiere de un esfuerzo educativo muy fuerte, principalmente en matemáticas, además de que estos métodos y lenguajes, no tienen como objetivo fundamental soportar el diseño de software reusable.

En la práctica existen muy pocas librerias de componentes de software reusable, las más conocidas son las de paquetes numéricos escritos en FORTRAN. Su éxito se debe al reducido campo de aplicación, a las bases matemáticas que soportan los algoritmos, y a la gran experiencia que permite cierto tipo de estandarización. Los cálculos numéricos fueron los primeros trabajos realizados en la historia de la programación.

Una forma de describir las caracteristicas de un componente reusable es por medio de las especificaciones abstractas, que se estudian con más detalle en el siguiente capitulo. Una abstracción proporciona una entidad conceptual, vacia de substancia tangible, y por lo tanto elimina complejidad innecesaria. Por ejemplo, las estructuras de datos son objetos muy tangibles. Las estructuras de datos abstractas son una clase de objetos y un conjunto de operaciones aplicables a un objeto de aquella clase. Por lo tanto, debe haber un vehiculo para describir al objeto y el comportamiento de las operaciones aplicadas al objeto. Una especificacion tiene el proposito de definir el comportamiento de una abstracción.

1.2 Sistemas que soportan reusabilidad

La falta de soporte técnico como ya hemos mencionado, contribuye a que no se practique el uso de componentes reusables. Se han hecho varios sistemas por especialistas con el fin de contribuir a solucionar este problema. En esta sección se analizan los que pueden compararse a nuestro catálogo de componentes reusables

Uno de los principales problemas en el reuso de componentes de software, es localizarlos y recuperarlos en una grande colección, por lo que Prieto y Freeman proponen un sistema de clasificación de software, cuyas ideas se estudian a continuación.

1.2.1 El sistema de Prieto y Freeman (SPF)

En este sistema se tiene como objetivo hacer del reuso de programas un trabajo atractivo, por lo que se considera que el esfuerzo para reusar debe ser menor que el esfuerzo para crear nuevamente código.

Hay tres pasos fundamentales a seguir para reusar software:

(1) accesar el código
(2) entenderlo
(3) adaptarlo

Consideran que un esquema de clasificación es esencial para la accesibilidad del código. El entendimiento depende de la experiencia del usario y las caracteristicas del programa, como tamaño, complejidad, documentación, y lenguaje de implantación. La adaptación del código depende de las diferencias entre los

requerimientos y las características ofrecidas por los componentes existentes y la destreza del usario.

Una manera de reducir el esfuerzo requerido para entender y adaptar el código es una clasificación apropiada de una colección de componentes. Si la colección es organizada por atributos que definen requerimientos de software, la probabilidad de recuperar componentes no relevantes es reducida.

Una buena clasificación es esencial para el diseño de sistemas de recuperación y búsqueda. Aún más, el esquema de clasificación y el sistema de recuperación deben ayudar a discriminar entre varios componentes similares. Una colección puede contener varios componentes similares, difiriendo únicamente en detalles menores de implantación. Un usuario debe seleccionar los componentes que requiere con el menor esfuerzo de adaptacion.

Típicamente, el usuario inspecciona cada elemento de una colección y selecciona el mejor. Pero, si la colección es muy larga, la tarea no es fácil. Se necesita un sistema para evaluar varios componentes similares, para ayudar al usuario a seleccionar los componentes que requieran el menor esfuerzo de conversión.

1.2.1.1 Modelo de reuso

El proceso de reusar software, Prieto y Freeman lo perciben a través de un modelo algorítmico. Este modelo proporciona un ambiente que ayuda a localizar componentes, y con una evaluación estima el esfuerzo de adaptación y conversión. El proceso de

reuso consiste del siguiente algoritmo.

-Dar un conjunto de especificaciones funcionales.
-El usuario busca en una librería de componentes, el candidato que satisfaga todas sus especificaciones.
- Si existe un componente que satisfaga todas sus especificaciones, reusarlo resulta trivial.
- Lo más típico es que, hay varios candidatos, cada uno satisfaciendo algunas especificaciones, a los que se les llama, componentes similares. En este caso, el problema es seleccionar y agrupar los candidatos adecuados, basándose en la medida que más se acerca a los requerimientos, y al esfuerzo requerido para adaptar las especificaciones que no concuerdan.
- Una vez que se tiene la lista ordenada de candidatos, el usuario selecciona el más fácil de reusar y adaptar.

Escrito de una forma más concisa tenemos:

```
BEGIN
  dar especificaciones funcionales
  buscar en la librería
  IF comparación idéntica THEN terminar
  ELSE
    agrupar componenes similares
    FOR EACH componente
       calcular el grado de similitud
    END FOR
    seleccionar el mejor
    modificar componente
  END IF
END
```

1.2.1.2 Esquema de clasificación

Para seleccionar el mejor componente, se necesita una buena clasificación. En el sistema se utiliza el esquema de clasificación por "facetas". Cada faceta representa un atributo relevante de un componente. Después de un arduo estudio sobre una muestra con cientos de programas en libros de texto, y catálogos comerciales de software, se seleccionaron seis facetas que describen a un componente. Las tres primeras describen la funcionalidad (lo que hace), y las restantes tres describen el medio ambiente, del mundo real donde se utiliza el componente.

Funcionalidad. Para representar la funcionalidad de un programa, se escoge el vector < función, objeto, medio >.

Función, el cual es un sinónimo de acción, es el nombre de la operación primitiva desarrollada por el programa (por ejemplo, mover, empezar, o comparar).

Objeto, se refiere a los objetos manipulados por el programa (tales como caracteres, lineas, o variables).

Medio, se refiere a las entidades que sirven como locales donde la acción se ejecuta, tales como estructuras de datos que en muchos casos son las estructuras que soportan a las funciones (como tablas, archivos, o árboles).

Algunos ejemplos son:

< leer, caracter, buffer >
< substituir, tabuladores, archivo >
< buscar, raiz, arbol binario >
< comprimir, lineas, archivo >

Medio ambiente. De la muestra experimental, se seleccionaron tres facetas: (1) las que describen el tipo de

sistema, (2) las que describen el área funcional, y (3) las que describen la localización o lugar de la aplicación. Las facetas determinan el vector < tipo de sistema, área funcional, colocación >.

Ejemplos de tipos de sistemas pueden ser: formateador de reportes, analizador lexicográfico, scheduler, evaluador de expresiones, y un intérprete.

Ejemplos de áreas funcionales pueden ser: control de costos, CAD, cuenta de costos, presupuestos, y manejadores de bases de datos.

La colocación describe donde se ejerce la aplicación, por ejemplo, una oficina de publicidad, un banco, tienda de computadoras, etcétera.

El órden citado de las facetas, se basa en la relevancia a los usuarios. Suponiendo que los usuarios típicos son ingenieros de software que diseñan y construyen nuevos sistemas desde componentes, el órden seleccionado fué función, objeto, medio, tipo de sistema, área funcional, y lugar.

Un componente queda caracterizado por una sexteta que lo describe. Algunos ejemplos son:

< sumar, enteros, arreglo,
  invertir matrices, modelado, fábrica de aviones >,
< comprimir, archivos, disco,
  manejador de archivos, manejador de bases de datos, catálogo de ventas >,
< comparar, descriptores, pila,
  ensamblador, programación, tienda de software >.

Cabe mencionar que no se puede usar ilimitádamente el

lenguaje natural para una descripción. Lo que se tiene es un vocabulario, que controla la descripción de los componentes.

1.2.1.3 Otras características

Una característica muy importante del sistema es la utilización de una gráfica conceptual para medir la similitud entre valores en una faceta, es decir, se puede cuantificar la similitud entre componentes.

Además tiene un mecanismo de evaluación para medir el esfuerzo de reuso. Hay cinco atributos que se han seleccionado como indicadores para medir tal esfuerzo. La tabla 1 presenta los atributos seleccionados y su métrica.

Tabla 1
Métricas para el reuso de atributos

Atributo	Métrica
Tamaño del programa	Líneas de código.
Estructura del programa	Número de módulos , ligas y complejidad ciclomática.
Documentación del programa	Rango subjetivo (1 a 10).
Lenguaje de programación	Relativo a la similaridad.
Experiencia del reusuario	Aptitud en dos áreas: lenguaje de programación y dominio de aplicación.

1.2.2 PARIS: un sistema para reusar esquemas parcialmente interpretados

Un prototipo de un sistema que maneju esquemas parcialmente interpretados es propuesto por Katz, Ritcher y Khe-Sing The [BiPe89a]. El nombre de PARIS se deriva de "partially interpreted schemas". Se creó por el deseo de tener un sistema que pudiera almacenar algoritmos eficientes y correctos, y entonces recuperarlos y utilizarlos en algún proceso. El sistema formaliza y automatiza el reuso de componentes de software.

El sistema mantiene una librería de esquemas parcialmente interpretados, cada uno de los cuales se almacena con un conjunto de aserciones acerca de su aplicabilidad y resultados. Cuando el usuario presenta una sentencia de un problema (es decir, la descripción de los requerimientos de un programa), el sistema busca en la librería un esquema que pueda satisfacer los requerimientos del usuario.

1.2.2.1 Definiciones

Para entender mejor el funcionamiento de este sistema, combiene dar algunas definiciones que se manejan.

Un *esquema de un programa parcialmente* interpretado es sintácticamente un programa o un módulo independiente en algún lenguaje de programación, que contiene entidades abstractas tales como funciones, predicados, símbolos constantes, dominios, o partes abstractas de programas, cada una de las cuales es representada por variables libres.

Cada esquema se acompaña de su *especificación*, la cual

incluye lo siguiente:

1. *Condiciones de aplicabilidad*, las cuales definen restricciones para usar el esquema, tales como posibles tipos de datos, rangos de parámetros y posibles funciones que puede tratar el esquema.
2. *Condiciones de sección*, las cuales son aserciones acerca de secciones de programa que pueden ser substituidas por variables libres.
3. *Condiciones del resultado*, que indican que es lo que el esquema puede lograr. Las aserciones del resultado pueden clasificarse como postaserciones, las cuales definen en términos de entidades abstractas y la entrada, algunas condiciones que se deben mantener después de la ejecución del esquema; aserciones invariantes, las cuales definen condiciones que son verdaderas através de la ejecución del esquema.

La *presentación* de un esquema incluye el texto del esquema, el conjunto de entidades abstractas, una proposición en ''lógica temporal'' de todos los aspectos de la especificación del esquema, y una prueba de correctés de las aserciones del resultado, suponiendo las condiciones de aplicabilidad y sección.

Una *sentencia de programa* consiste de los requerimientos de computación, hechos de la tarea concreta a realizar.

La sentencia de programa se compara con los esquemas existentes, y si se logra un apareamiento satistactorio, se hace una *instanciación*, obteniendo un programa concreto. La instancia obtenida es garantizada correcta.

La librería del sistema consiste de un conjunto de esquemas, en el que cada esquema tiene cinco componentes: lista de entidades, condiciones de aplicabilidad, aserciones de resultado, condiciones de sección, y el cuerpo del esquema. Se supone que el sistema y el usuario están de acuerdo acerca del vocabulario que se utiliza para describir un esquema.

El apareamiento entre la sentencia de un problema y un esquema de especificación se logra iterativamente. Esto significa que el usuario es protegido tanto como es posible de trabajos internos, representaciones, y procedimientos de decisión. Sin embargo, cuando las herramientas automáticas son inadecuadas, se consulta al usuario de la manera más clara posible.

Para evitar mucha consulta al usuario y hacer más eficiente al sistema, se tiene como subsistema al probador de teoremas Boyer-Moore.

1.2.2.2 De especificaciones abstractas a programas concretos

Para empezar una sesión en PARIS, el usuario tiene que dar como entrada una sentencia de problema (ver la figura 1.1). Una sentencia de problema consiste de tres componentes: una lista de entidades (variables, constantes, tipos, dominios, funciones, etc.), condiciones de aplicabilidad, y postcondiciones. Estos componentes son similares, y tienen la misma sintaxis a los componentes de un esquema en la librería del sistema, excepto que estos no tienen entidades abstractas.

Después de que una sentencia de problema ha sido correctamente especificada, PARIS busca en la librería para encontrar un posible candidato. Si el candidato se encuentra, entonces la interpretación de las entidades abstractas se inicia. La interpretación se efectúa en dos etapas: primero las entidades son reemplazadas, y posteriormente las secciones de programa. El reemplazamiento de las entidades puede hacerse en forma manual o automática.

Si el modo manual es seleccionado para interpretar las entidades abstractas, el sistema presentará al usuario todos los

candidatos, hasta que el usuario escoja uno. Cuando se presenta un esquema, el usuario debe decidir si lo toma y sustituye todas las entidades abstractas, o si intenta con otro esquema. Después de que todas las entidades han sido reemplazadas, y antes de que el sistema pida la sustitución de las secciones de programa, el usuario puede pedir al probador de teoremas que verifique si las instanciaciones satisfacen la sentencia del problema.

Si el modo automático se selecciona, el sistema apareará cada entidad abstracta de un esquema con una entidad concreta mencionada en la sentencia de problema, e invocará al probador de teoremas Boyer-Moore para que haga la verificación correspondiente. Si el sistema encuentra un esquema que satisfaga la sentencia de problema, entonces proporcionará un esquema parcialmente instanciado (con todas las entidades abstractas reemplazadas automáticamente), quedando por llenar las secciones de programa. Por otro lado. Si no se encuentra el esquema adecuado, el usuario será informado. El reemplazamiento de las secciones de programa para un esquema, en la versión actual, es efectuado manualmente.

La figura 1.1 muestra los módulos que forman al sistema PARIS, y como el usuario interectua con el sistema en una sesión.

Módulo 1. Especificación del problema. El usuario presenta sus requerimientos de forma que sean entendibles internamente para el sistema.

Módulo 2. Identificación de candidatos. Se reduce el número de candidatos eliminando esquemas irrelevantes. Un esquema se queda en la lista de posibles candidatos, si el conjunto de entidades en la sentencia del problema es un subconjunto del conjunto de entidades de ese esquema.

Módulo 3. Coordinador. Basado en los requerimientos especificados

por el módulo 1, este módulo trata de construir un programa por medio de la interpretación de los candidatos suministrados por el módulo 2. La interpretación de un candidato consiste de dos etapas: aparear las entidades abstractas (usando ya sea el módulo 4 en forma manual o el módulo 5 en forma automática) e insertando las secciones de programa (usando el módulo 6).

Módulo 4. Interpreta las entidades de un esquema, en modo manual(comunicación interactiva con el usuario para interpretar las entidades ebstractas).

Después de que el apareamiento manual se ha hecho, el usuario tiene la opción de checar sus apareamientos por el probador de teoremas.

Módulo 5. Interpreta las entidades de un esquema de forma automática (permutando las entidades abstractas para generar diferentes instancias, e invocando al probador de teoremas para verificar cada instancia).

Módulo 6. Inserta secciones de programa efectuando una comunicación interactiva con el usuario para insertar secciones de programa en candidatos secundarios (los cuales han sido construidos de candidatos preeliminares por medio de la interpretación de las entidades abstractas).

Cuando todas las secciones de programa en un candidato secundario han sido exitosamente interpretadas, el resultado es el programa final.

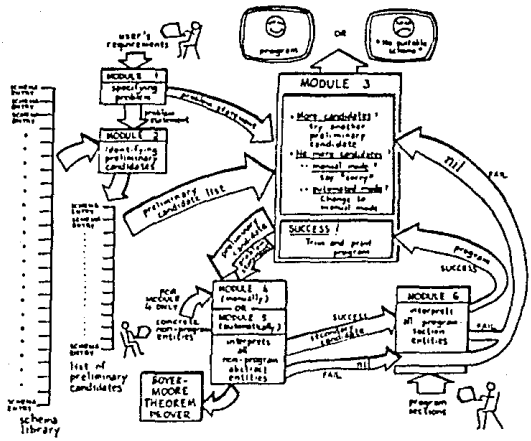


Figura 1.1. Vista del sistema PARIS (tomado de la referencia [BiPe89a], página 263).

1.2.2.3 Un esquema para la inserción en listas ligadas

Para aclarar ideas, se da un ejemplo que ilustra la forma en que se dan especificaciones de un esquema por medio de un programa en C que inserta un nodo en una lista ordenada. Un nodo en una lista contiene dos campos: next y data; next es el apuntador al siguiente nodo de la lista, y data es el valor del nodo. Las funciones leq, equal, y f definen un orden total en los nodos de la lista. El dominio de f es el conjunto de los datos para un nodo, y su rango es el conjunto de valores por medio de los

cuales, la lista es ordenada. Para mantener el orden en la lista, las condiciones de aplicabilidad y postcondiciones se deben establecer.

La lista de entidades para este esquema contiene todas las entidades abstractas, las cuales serán reemplazadas por entidades concretas cuando el esquema sea interpretado. Las condiciones de aplicabilidad imponen algunas condiciones sobre las funciones, y requieren que la lista esté ordenada antes de que se efectúe una inserción. Las postcondiciones indican que después de una inserción, el nuevo nodo debe quedar en el lugar apropiado, y la lista debe quedar ordenada.

La sección de programa S1, asigna el valor del nodo insertado a un espacio de memoria. La operación de asignación se abstrae debido a que ésta se puede efectuar de varias formas, dependiendo de la estructura elegida. El programa principal, el cual es otro módulo independiente, debe satisfacer la invariante de que la lista es ordenada. Esto deja implícito, que después de una inserción, la lista contiene todos los nodos anteriores a la inserción.

El esquema abstracto del módulo de inserción se muestra abajo. Notese que el usuario del esquema no tiene necesariamente que entender el código del cuerpo del esquema, aunque debe saber si sus instanciaciones satisfacen las especificaciones.

```
Entity List
    function f
    function leq
    function equal
    domain D1, D2
    variable data: D1
    variable newdata: D1
    structure node: D1 x pointer(node)
    programsec S1
```

```
Applicability Conditions
    f: -> D2
    leq: D2 x D2 -> boolean
    equal: D1 x D1 -> boolean
    ∀ n : n in  node : n -> next =/ NULL ⇒
      leq(f(n -> data), f((n -> next) ->data))

Result Assertions
    [ ∃ n : n ∈ node : equal(n-> data, newdata)]
    [ ∀ n : n in node : n-> next =/  NULL ⇒
      leq(f(n -> data),f((n -> next)-> data))]

Section Conditions
    S1: precondition: true
        postcondition: equal(new -> data, newdata)

Schema Body
    struct node ( struct node *next; D1 data;)
    insert( listhead, newdata )
    struct node *listhead;
    D1 newdata;
    (
         struct  node *p, *q, *new;
         new = malloc( sizeof( struct node ));

         /* asigna newdata a new -> data */
         S1
         if ( listhead == NULL )
         (    listhead = new;
              listhead -> next = NULL;
         )
         else if ( leq( f(new), f( listhead )) )
              (
                new -> next = listhead;
                listhead = new;
              )
```

```
        else
        {
          p = listhead;
          q = listhead -> next;


          while ( !leq( f(new), f(q)) && q != NULL )
          {
              p = q;
              q = q -> next;
          }
          p -> next = new;
          new -> next = q;
        }
}
```

Este esquema puede fácilmente instanciarse para manejar tipos diferentes de datos D1. Por ejemplo, si D1 es integer, entonces f será la función identidad, leq y equal serán <= y ==, respectivamente; la asignación en S1 será =. Si D1 es una cadena de caracteres, entonces la función f, leq y equal pueden ser funciones de manipulación de strings de C, y la asignación en S1 es simplemente la función de C strcpy.

1.3 *Criticas a estos sistemas*

En esta sección se menionan los inconvenientes apreciados en el sistema de Prieto y Freeman, y en PARIS. Comencemos con el sistema de Prieto y Freeman.

En el sistema de Prieto y Freeman (SPF) los componentes reusables para los que está construido solo proporcionan una

operación, como por ejemplo, un componente que lee lineas de un archivo de texto, o uno que suma matrices. Esta restricción simplifica su mecanismo de clasificación, y evaluación de la similaridad de componentes. Componentes reusables genéricos que se pueden realizar en lenguajes como ADA y Modula-2, pueden proporcionar múltiples operaciones. Es posible crear un componente genérico en ADA que efectue las operaciones de suma resta y mutiplicación de matrices, el cual no se puede clasificar en SPF; se tendría que separar, haciendo un componente para cada operación.

Continuemos ahora con el sistema PARIS. Su sistema de clasificación de esquemas es muy pobre, ya que solo es un conjunto no ordenado de esquemas.

Al igual que en SPF, se debe definir un vocabulario, que en este caso se utiliza para realizar la sentencia de programa, que es la descripción de requerimientos.

La creación de la sentencia de programa es muy laboriosa, debido a que se necesita del conocimiento del lenguaje de espeficación que ellos utilizan para escribir una de las secciones del esquema. Por lo que la interface con el usuario no es muy amigable.

Cuando la sentencia de programa no encuentra un esquema que la satisfaga, el probador de teoremas falla, y es necesario ser un experto en su manejo, para saber porque falla, y hacer las correcciones adeccuadas.

Ni SPF ni PARIS proporcionan mecanismos para describir relaciones de interdependencia entre componentes, para el caso de

SPF, o entre esquemas en el caso de PARIS.

En la seccción de conclusiones de este trabajo, se consideran nuevamente estas criticas al sistema SPF y a PARIS, para una comparación entre ellos y el prototipo de catálogo propuesto.

En el catálogo propuesto en esta tesis, todos los componentes reusables tienen asociada una especificación formal. Sería ideal que todo programa sea especificado, y probado en forma rigurosa. En el siguente capitulo se tratan las ventajas de usar especificaciones en el desarrollo de software.

CAPITULO 2

ESPECIFICACIONES FORMALES

El prototipo que se propone para el manejo de componentes reusables, supone que cada componente tiene asociada una especificación. En este capítulo se dan las ideas y conceptos generales que aportan las especificaciones, en el desarrollo de software.

Una especificación es la descripción de las caracteríticas más importantes de un componente. Una especificación puede ser formal o informal; es formal cuando su significado está definido de una forma precisa, e informal en caso contrario. Por ejemplo, un especificación informal es cualquiera realizada con el lenguaje natural, y una formal, si se utiliza la lógica matemática como lenguaje. Notemos que para hacer una especificación siempre necesitamos un lenguaje para realizarla, conocido como lenguaje de especificación; la definición precisa de un lenguaje de especificación se encuentra en [Wing90]. Las especificaciones formales son las que interesan para maximizar la automatización del desarrollo de software, pero no debemos pensar que son ajenas con las informales, sino complementarias.

2.1 Especificaciones de datos abstractos

Al inicio, los avances en las especificaciones formales fueron relegados debido a la dificultad de construir programas prácticos que las utilizarán. Sin embargo el trabajo desarrollado en la metodología de programación, identificó una unidad de programa, que define datos abstractos, la cual es útil, y práctica para escribir especificaciones formales. Esta unidad consiste de una pareja < T, O >, donde T es un conjunto de datos y O es un

conjunto que tiene como elementos las operaciones realizadas sobre T.

Por ejemplo:

T = { x : x es una cadena de caracteres }

O = { concatenar dos elementos de T,
comparar dos elementos de T,
obtener la longitud de un elemento de T }

Muchas técnicas se han desarrollado para la especificación de datos abstractos [Lisk75], debido a que un dato abstracto se puede tomar como unidad de especificación. Una forma de implantación es a través de módulos que contienen varios procedimientos; en Ada y Modula-2 la implantación se hace de forma natural. Cada procedimiento en el módulo implanta una de las operaciones; el módulo como un todo puede verse como un solo objeto (por ejemplo, un sistema de bases de datos). Es bueno examinar los argumentos a favor de este tipo de módulos como unidades de implantación. Los procedimientos son agrupados porque de alguna manera interactúan entre si: comparten ciertos recursos (por ejemplo, una base de datos), información (por ejemplo, el formato y significado de los datos en la base de datos). Considerando el grupo entero de procedimientos como un módulo, permite ocultar toda la información de las interacciones entre los módulos. El ocultamiento de la información simplifica la interface entre los módulos, y simplifica directamente las especificaciones, porque es precisamente la interface entre los módulos lo que la especificación debe describir.

Como un ejemplo de los problemas que ocurren cuando la

abstracción es ignorada y las operaciones de un grupo se describen dando especificaciones de entrada-salida independientes una de la otra, consideremos la siguiente especificación de la operación apila. Pensando a la operación apila como una función

```
apila: pila X entero --> pila,
```

la especificación de entrada-salida debe definir la información contenida en el valor de salida de apila (el objeto pila regresado por pila) en términos de los valores de entrada de apila (una pila y un entero). Esto puede hacerse definiendo una estructura para los objetos pila, y describiendo el efecto de apila por medio de la estructura. Una pila típica en Pascal puede ser

```
type pila = record
              tope: integer,
              dato: array [ 1.. 100 ] of integer
            end
```

y entonces, el significado de

```
t := apila(p,d)
```

puede establecerse (utilizando la notación de Hoare) como

```
true { t := apila(p,d) }  Vj[ 1 <= j <= p.tope
                            > t.dato[j] = p.dato[j]
                            & t.dato[t.tope] = d
                            & t.tope = p.tope + 1]
```

Una especificación similar puede hacerse para la operación desapila.

Hay varias cosas desagradables con esta especificación. Una de ellas, es que no describe el concepto de pila por medio de su comportamiento, en su lugar delinea una gran cantidad de detalles extraños. Conceptos sobre el comportamiento de la pila, por ejemplo, un teorema estableciendo que desapila regresa el valor más recientemente apilado, pueden ser inferidos únicamente de estos detalles. La inclusión de extraños detalles es indeseable por dos razones. Primero, el inventor del concepto debe estar envuelto en el detalle (la cual es realmente información de implantación) en vez de concentrarse en el concepto directamente. Segundo, la agregación de detalles impide que se tenga la propiedad de minimicidad de una especificación, y es probable que la prueba de correctés sobre una implantación de apila y desapila, en base a representaciones diferentes de pilas se dificulte. Otro problema es que la independencia de apila y desapila es una ilusión; un cambio en la especificación de uno de ellos, nos lleva consecuentemente a un cambio en la especificación del otro. Por ejemplo, nos damos cuenta que las especificaciones de apila y desapila están relacionadas (interpretando la estructura seleccionada para una pila): la decisión de tener un apuntador al tope o al nodo inmediato libre debe respetarse en las especificaciones de apila y desapila, para no tener inconsistencias.

Si una abstracción de datos para una pila se especifica como una unidad, los detalles extraños pueden ser eliminados, y los efectos de las operaciones pueden ser descritos en un nivel más alto. Las especificaciones algebraicas tienen como unidad de especificación a los datos abstractos, y expresan propiedades abstractas de los objetos, en lugar de propiedades específicas, como en el ejemplo anterior. Las operaciones no se describen aisladas, sino que los efectos de una operación se expresan en términos de otra. Por ejemplo, el efecto de desapila

puede definirse en términos de apila por

desapila(apila(p,d)) = p,

que establece que desapila quita el dato que se apiló más recientemente. Ejemplos de especificaciones algebraicas pueden verse desde el artículo de Guttag [Gutt77], y en [Mart86].

2.2 Correctes

Un paso serio en la construcción de software, es verificar si un programa es correcto (correctes), es decir, si hace lo que supusimos que debía hacer. La correctés, no es la única propiedad que nos interesa de un programa terminado, pero es la más importante. Si un programa no es correcto, las demás propiedades (por ejemplo, eficiencia, complejidad), no tienen significado.

Las pruebas para establecer la correctés de programas, al igual que los lenguajes de especificación, pueden clasificarse en formales e informales. La mayoría de las técnicas usadas comúnmente (debugging, probar con grandes cantidades de datos, lectura de programas) son técnicas informales, debido a que dependen en gran parte de la ingenuidad e intuición humana. La continua existencia de errorres en los programas, a los cuales dichas técnicas son aplicadas, atestiguan que son inadecuadas. Los lenguajes formales de especificación son los únicos que nos pueden llevar a la prueba de correctés de software, ya que afortunadamente existen métodos formales, que junto con el lenguaje de especificación, proporcionan la infraestructura necesaria para llevar la especificación de un programa hasta su implantación.

Analizando con más detalle el significado de la correctés, vemos que es un proceso, que inicia con un concepto que solo existe en nuestras mentes, y lo que queremos es saber si un programa que construimos correctamente lo implanta. El concepto puede ser implantado de una infinidad de maneras, pero solo un número finito tienen interés práctico. Esta situación se ilustra en la figura 2.1.

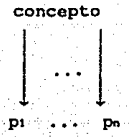


Fig. 2.1. Un concepto y todos los programas que lo implantan correctamente.

En la práctica, el concepto se extrae de la mente de una manera informal, y se hace un programa que lo implanta, aplicandose una técnica de prueba de correctés informal, con consecuencias peligrosas.

Con las técnicas formales, una especificación se interpone entre el concepto y los programas. Su propósito es proporcionar una descripción matemática del concepto, y la correctés de un programa se establece probando que es equivalente a la especificación. La especificación puede ser satisfecha por un conjunto de programas, probablemente infinito, pero solo un número finito tienen interés práctico. Esta situación se

ilustra en la figura 2.2.

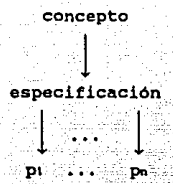


Figura 2.2. Un concepto, su especificación, y todos los programas para los que se puede probar que son equivalentes a la especificación.

2.3 Métodos formales

El desarrollo de un sistema, desde su especificación, implantación, y prueba de corrcetés, es un proceso, que con el tiempo se ha rodeado de una infraestuctura, conocida como método formal, proporcionando una forma sistemática del desarrollo de tal proceso.

Un método es formal, si tiene bases matemáticas, típicamente dadas por un lenguaje de especificación formal. Estas bases proporcionan la manera de definir precisamente las nociones de consistencia y completés, y más aún, especificación, implantación, y correctés. Nos guía, para conocer si una especificación es realizable, probar si un sistema ha sido implantado correctamente, y proporciona propiedades del sistema

sin necesidad de ejecutarlo para determinar su comportamiento.

Un método formal también nos centra en consideraciones fundamentales: quienes usan el método, para que es usado, cuando es usado, y como es usado. Lo más común es que los diseñadores de sistemas usen métodos formales para especificar las propiedades de un sistema deseado. Una gran variedad de métodos formales pueden consultarse en [Wing90].

Sin embargo, cualquiera involucrado en el desarrollo de un sistema, puede hacer uso de los métodos formales. Pueden usarse para captar los requerimientos iniciales del cliente, en el diseño del sistema, implantación, prueba, debugging, mantenimiento, verificación, y evaluación.

Los métodos formales son usados para revelar ambiguedades, e inconsistencias del sistema. Cuando se utilizan en los inicios del sistema, pueden revelar secretos, que únicamente podrían haber sido descubiertos durante pruebas costosas. Cuando son usados al final del desarrollo, pueden determinar la correctés de la implantación de un sistema, y la equivalencia de diferentes implantaciones.

A parte de que un método formal debe tener bases matemáticas, debe indicar al usuario, bajo que circunstancias puede ser aplicado en forma correcta y eficiente. Un producto tangible de un método es una especificación formal, que sirve como contrato, documentación, y un medio de comunicación entre el cliente, el especificador, y el implantador. Debido a sus bases matemáticas, las especificaciones formales son más precisas y concisas que las informales.

Debido a que un método formal no es un programa o un lenguaje de programación, puede o no tener herramientas de soporte. Si la sintaxis de un lenguaje de especificación existe, pueden crearse herramientas para el análisis sintáctico de especificaciones. Si los lenguajes que describen la semántica son suficientemente restringidos, el análisis de la semántica de una especificación puede ser asistido por una máquina. Por lo tanto, las especificaciones formales tienen la ventaja adicional sobre las informales, de que pueden crearse herramientas automáticas que las asistan.

El prototipo que se describe en el capítulo 5, supone que los componentes reusables de software, fueron creados con algún método formal. Un componente tiene asociada una especificación formal, y una o varias implantaciones, en uno o varios lenguajes de programación. Ha sido implantado en Algres, que es un ambiente de programación de bases de datos. El modelo de datos que utiliza es el NF2, que es una de las tendencias de investigación del área de bases de datos. Para entender mejor el origen de Algres, en el siguiente capítulo se revisan los modelos tradicionales de datos, y las perspectivas de investigación de esta rama.

CAPITULO 3

PANORAMA DE LAS BASES DE DATOS

En este capitulo tratamos los caminos a los que se dirige la tecnología de las bases de datos, teniendo como paradigma la actualización de los modelos de bases de datos tradicionales. En particular nos interesa profundizar un poco más en el modelo NF2, que es el modelo utilizado por el manejador de base de datos Algres, herramienta utilizada para implantar el catálogo de componentes reusables.

3.1 Modelos clásicos de bases de datos

La necedidad de organizar datos de una forma clara y rigurosa, ha propiciado durante el tiempo el desarrollo de varios modelos de datos, siendo quizás los más conocidos el relacional, el jerárquico, y el de red. Los dos primeros tienen una orientación "maquinal", la cual proporciona una base eficiente para almacenar y procesar datos, pero no están orientados al usuario final, al cual no le interesa como se almacenan y manipulan los datos. Cada uno de los modelos clásicos está cimentado sobre alguna estructura idealizada de representación de un conjunto de datos, y tiene un conjunto de operaciones asociadas a esta estructura, haciendonos recordar el concepto de datos abstractos, tratado en el capítulo previo.

Debido a la importancia del modelo relacional, se definirá de una forma precisa. Los modelos jerárquico y de red se describirán de una manera informal.

3.1.1 Modelo relacional de datos

El modelo relacional fué propuesto por Codd [Codd70]. Su atracción principal es su claridad matemática, la cual facilita operaciones ("queries") no procedurales de alto nivel, y por lo tanto evita al usuario el conocimiento que concierne a la organización interna de los datos. Entre los tres modelos clásicos de datos, el modelo relacional es considerado el más orientado al usuario.

3.1.1.1 Entidades, atributos y dominios

Entendemos por entidades a los objetos y a sus interrelaciones. Un conjunto E de entidades del mismo tipo se caracteriza por un conjunto de atributos $A_1, A_2, \ldots, A_n$, y se denota como $E(A_1, A_2, \ldots, A_n)$, donde $A_i : E \rightarrow D_i$ es una función cuyo codominio D_i se denomina el dominio del atributo A_i. Si $e \in E$, $A_i(e) \in D_i$ se denomina el valor del atributo A_i de la entidad e.

Ejemplo

Consideremos al tipo de entidad PERSONA, cuyos atributos relevantes pueden ser su rfc, nombre, apellido paterno, apellido materno, y sexo, la cual se puede denotar como

PERSONA(rfc,nombre,ap_paterno,ap_materno,nombre,sexo)

donde los dominios de los atributos son:

rfc:	el conjunto D de cadenas de caracteres de longitud 10
nombre:	el conjunto S de cadenas finitas de caracteres
ap_paterno:	el conjunto S de cadenas finitas de caracteres
ap_materno:	el conjunto S de cadenas finitas de caracteres
sexo:	el conjunto de caracteres {'m','f'}

3.1.1.2 Representación de entidades por medio de tuplas

Dado el tipo de entidad $E(A_1, A_2, \ldots, A_n)$, el conjunto de funciones $A_i: E \rightarrow D_i$ $(i=1,n)$, las cuales definen los atributos de E, determina la función única:

$$(A_1, A_2, \ldots, A_n): E \rightarrow D_1 X D_2 X \ldots X D_n \qquad \text{(i)}$$

donde $D_1 X D_2 X \ldots X D_n$ denota el producto cartesiano de los conjuntos $D_1, D_2, \ldots, D_n$. La función $(A_1, A_2, \ldots, A_n)$ se define como:

$$(A_1, A_2, \ldots, A_n)(e) = (A_1(e), A_2(e), \ldots, A_n(e)) \qquad \text{(ii)}$$

en donde se necesita que para dos entidades diferentes cualesquiera e_1 y e_2, pertenecientes a E, se cumpla que

$$(A_1, A_2, \ldots, A_n)(e_1) \neq (A_1, A_2, \ldots, A_n)(e_2). \qquad \text{(iii)}$$

La condición (iii) establece que $(A_1, A_2, \ldots, A_n)$ es una función inyectiva, y deberá satisfacerse si $\exists j$ $(j=1,n)$ tal que

$$A_j(e_1) \neq A_j(e_2). \qquad \text{(iv)}$$

Que dicho en palabras, significa que entidades diferentes se representan por tuplas diferentes (las tuplas son diferentes si tienen valores diferentes en al menos un atributo).

Ejemplo

Interpretemos la notación

PERSONA(rfc,nombre,ap_paterno,ap_materno,nombre,sexo)

como la función

(rfc,nombre,ap_paterno,ap_materno,nombre,sexo): PERSONA ->
DXSXSXSX{m,f}

donde *D*, y *S* son los conjuntos definidos en el ejemplo anterior. Esta función le asocia a cada persona *p* una tupla de dimensión 5, la cual representa a la persona. Personas diferentes p_1 y p_2 quedarán determinadas por tuplas diferentes.

El conjunto de entidades del tipo $E(A_1,A_2,\ldots,A_n)$ se puede representar por una tabla cuyas columnas corresponderán a los atributos $A_1,A_2,\ldots,A_n$, y cuyos renglones son tuplas particulares que representan a entidades específicas. Una tabla para un conjunto de personas se muestra a continuación:

PERSONA

rfc	nombre	ap_paterno	ap_materno	sexo
SACP620412	Patricia	Sánchez	Conde	f
LORR561025	Roberto	López	Robles	m
MERJ630613	Joaquín	Mendoza	Blanco	m
OLOY541009	Yolanda	Oliva	Ortiz	f
CAMB741006	Beatriz	Castillo	Méndez	f
HERJ560823	Juan	Hernández	Ruiz	m
TOML650712	Luisa	Torres	Marín	f

3.1.1.3 Relaciones

R es una relación sobre los conjuntos $D_1,D_2,\ldots,D_n$ si y solo si $R \subseteq D_1,D_2,\ldots,D_n$. La estructura de la relación *R* se puede describir por la notación $R(A_1,A_2,\ldots,A_n)$ donde cada A_i:R->D_i es un atributo de *R* y *Di* es su dominio.

3.1.1.4 Base de datos relacional y esquema relacional

Una base de datos relacional es un conjunto de relaciones que cambian con respecto al tiempo. Un esquema relacional es una descripción de la estructura de las relaciones en una base de datos relacional.

Un ejemplo de un esquema relacional puede ser el siguiente:

```
DEPARTAMENTO(DEPT#, Nombre, Loc)
EMPLEADO(EMP#, Nombre, Puesto, DEPT#)
PROJECTO(PRO#, Titulo, Fondos)
ASIGNACION(EMP#, PRO#, Tarea)
```

el cual describe cuatro tipos de entidades.

Una base de datos en un instante de tiempo para el esquema anterior puede ser:

DEPARTAMENTO

DEPT#	Nombre	Loc
4	Sistemas	Insurgentes sur 123
2	Contabilidad	Insurgentes Sur 123
6	Mantenimiento	Av. Toluca 412

EMPLEADO

EMP#	Nombre	Puesto	DEPT#
123	Daniel Sánchez	Mensajero	2
34	Luis Rosales	Lider de Proyecto	4
453	Elisa Mendoza	Secretaria	2
45	Raul Mujica	Programador	4
234	Rene Liceo	Electricista	6

PROYECTO

PRO#	Titulo	Fondos
20	Prensa Hidráulica	20000000
12	Lámina Elástica	13000000
21	Sensor Automático	34000000
34	Sierra Universal	32000000

ASIGNACION

EMP#	PRO#	Tarea
34	20	Análisis de fluidos
123	12	Cortes longitudinales
45	21	Dis. de circuitos de cambio
234	12	Verificación de medidas

3.1.1.5 Algebra relacional

El álgebra relacional consiste de un conjunto de operaciones sobre las relaciones.

Proyección

Sea $R(A_1,A_2,\ldots,A_n)$ una relación, $X \subseteq \{A_1,A_2,\ldots,A_n\}$, y $Y = \{A_1,A_2,\ldots,A_n\} - X$. Permutando los atributos de R, se puede representar $R(A_1,A_2,\ldots,A_n)$ como $R(X,Y)$. El resultado de la operación proyección de R sobre los atributos en X se denota por $R[X]$ y se define como

$$R[X] = \{x: \text{existe } y \text{ tal que } (x,y) \in R(X,Y)\}.$$

Selección

Sea $R(A_1,A_2,\ldots,A_n)$ una relación y P una condición lógica definida sobre el conjunto $D_1,D_2,\ldots,D_n$, donde D_i es el dominio del atributo A_i (i=1,n). El resultado de la selección sobre la relación R con respecto a la condición P, se denota como $R[P]$ y se define asi:

$$R[P] = \{x: x \in R \text{ y } P(x) \text{ es veradera}\}.$$

"Join"

Sean $A(A_1,A_2,\ldots,A_n)$ y $B(B_1,B_2,\ldots,B_m)$ relaciones y $X \subseteq A(A_1, A_2,\ldots,A_n)$ y $Y \subseteq B(B_1,B_2,\ldots,B_m)$ conjuntos de atributos. Suponemos que X y Y contienen igual número de atributos, y que los correspondientes tienen el mismo dominio.

Si denotamos por $Z = (A_1,A_2,\ldots,A_n) - X$ y $W = (B_1,B_2,\ldots,B_m) - Y$ entonces, permutando los atributos, las relaciones A y B se pueden representar como $A(Z,X)$ y $B(Y,W)$. El "*join*" natural de las relaciones sobre los atributos X y Y respectivamente se denota por $A[X=Y]B$ y se define como

$$A[X=Y]B = \{(z,x,w): (z,x) \in A, (y,w) \in B, x = y\}.$$

División

Sean $A(X,Y)$ y $B(Z)$ dos relaciones, donde X,Y,Z son conjuntos de atributos. Suponemos que Y y Z contienen el mismo número de atributos y que los dominios de los atributos correspondientes en estos conjuntos son iguales. El resultado de la división de la relación $A(X,Y)$ con la relación $B(Z)$ se denota como $A[Y:Z]B$, y se define como el subconjunto máximo de la proyección $A[X]$ tal que su producto cartesiano con $B(Z)$ está contenido en $A(X,Y)$. De otra forma:

$$A(X,Y) = A[Y:Z]B \times B(Z) \cup R(X,Y)$$

donde $A[Y:Z]B$ es el cociente y $R(X,Y)$ es el residuo de la división. Además dado $C \subseteq A[X]$ tal que $C \times B(Z) \subseteq A(X,Y)$, se debe cumplir que $C \subseteq A[Y:Z]B$.

Operaciones ordinarias sobre conjuntos

Las operaciones en la teoría de conjuntos de unión, intersección, diferencia, y producto cartesiano, en el álgebra relacional se definen de la misma manera, salvo una pequeña restricción. Los operandos de la unión, intersección, y diferencia deben satisfacer la siguiente condición de compatibilidad:

El número de atributos y los dominios de los atributos correspondientes de los operandos de las operaciones de unión, intersección, y diferencia, deben ser los mismos.

3.1.2 *Modelo jerárquico de datos*

Los datos en un modelo jerárquico se organizan en estructuras simples de árbol. Una base de datos jeráquica es una colección de tales árboles --un *bosque*. Esta organización necesariamente requiere que algunos tipos de registros estén subordinados a otros. Una característica crucial del modelo jerárquico es que, algunos registros toman su significado completo cuando son vistos en el contexto de su jerarquía. Desafortunadamente, la estructura jerárquica no es un

modelo natural para muchas aplicaciones, por lo que es necesario usar *registros* lógicos *virtuales* (un registro lógico virtual es un apuntador a otro registro). Estos apuntadores proporcionan una forma que permite a un registro existir en varios árboles, o en dos lugares en el mismo árbol. Sin este concepto, se tendrían que duplicar una gran cantidad de registros, cuando se hiciera una tranformación de un tipo de relación de muchos-a-muchos, en una estructura jerárquica, resultando un alto grado de redundancia en la representación. Aún más, las operaciones en el modelo jerárquico tienen una orientación procedural, y están en este sentido, en un nivel más bajo que las operaciones del modelo relacional, ya que mucha de la organización interna de los datos debe ser visible al usuario.

En la figura 3.1, se tiene la estructura de una base de datos jerárquica para equipos de fooot-ball soccer, la cual

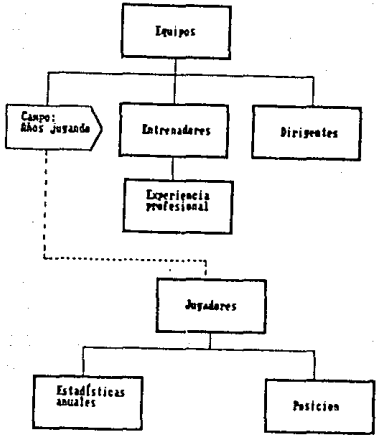


Figura 3.1. Estructura de una base de datos jerárquica para equipos y jugadores de foot-ball, con un registro lógico.

contiene diversos datos. Nótese que apara "ligar" la información referente a equipos y jugadores, se utiliza un registro lógico, el cual se aprovecha para almacenar la antiguedad del equipo.

3.1.3 Modelo de red de datos

El modelo de red de datos fué introducido por el grupo Data Base Task Group (DBTG) en la "Conference on Data Systems Languajes". En este modelo, la información es representada por *registros*, interconectados a través de *ligas* para formar una *gráfica dirigida*. Los registros son agrupados en *conjuntos*, cada uno de los cuales consiste de un registro *propietario* y cero o más registros *miembro*.

El concepto de conjunto es el más importante del modelo del grupo DBTG. El tipo conjunto se define en el esquema, especificando los tipos de los registros del propietario y los miembros. La mayor deficiencia del modelo es que las relaciones de muchos-a-muchos no pueden ser representadas directamente. Para representar esta clase de relaciones, el concepto de *registro de conección* es incluido. Un registro de conección contiene la información comun a los dos tipos de registros ligados por la relación muchos-a-muchos.

El modelo de red tiende a ser procedural, implicando que el usuario tienda a familiarizarse con la representación de bajo nivel de los datos, de tal forma que cuando se está haciendo alguna aplicación, el usuario del modelo, debe tener a un lado la hoja en donde dibujó la gráfica que representa la organización de sus datos, ya que sin ella no puede hacer casi nada. Esto además implica que las operaciones sobre la base de datos sean más complicadas.

En la figura 3.2 se muestra una instancia de una base de datos de red, para una empresa, sus divisiones, departamentos y empleados.

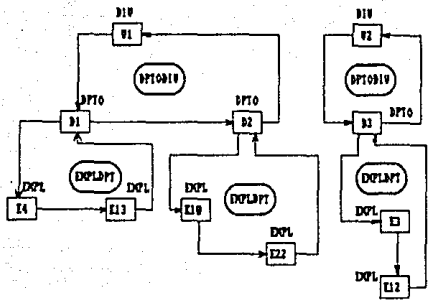


Figura 3.2. Una instancia de una base de datos de red para una empresa, sus divisiones, departamentos y empleados.

3.2 Ventajas y desventajas del modelo relacional

Hemos resumido las características de los modelos tradicionales de datos, para justificar el camino que han tomado las investigaciones sobre los modelos de datos. Como meta de estos esfuerzos se tiene el de crear un modelo ideal general, que pueda aplicarse para representar cualquier aplicación del mundo real. Esa meta es en cierta forma análoga a la que persiguen los interesados en el área de lenguajes de programación, en donde estarían muy contentos si existiera un sólo lenguaje de programacion, y no los miles que existen, capaz de expresar cualquier algoritmo.

El modelo relacional es el foco de las numerosas investigaciones dirigidas a la unificación de los modelos de datos, debido a su claridad, sencillez, y fundamentos matemáticos

[Ullm82], que los modelos jerárquico y de red no poseen, los cuales, complican más sus estructuras, creando conceptos nuevos que les ayuden a salvar sus limitaciones. Por ejemplo, en términos de proporcionar al usuario interfaces amigables, el relacional es el más interesante de los clásicos, debido a su propensión para soportar lenguajes de alto nivel no procedurales. Los lenguajes no procedurales permiten a los usuarios expresar "queries" y "updates" en función de propiedades del resultado, y no por medio del conocimiento concerniente a la organización de la base de datos. Un lenguaje no procedural permite solicitar "queries" en forma simple, en comparación del conjunto de conocimientos y esfuerzo necesario para realizar lo mismo en un lenguaje de programación de propósito general; por lo que un usuario lo puede aprender rápidamente.

El usuario en el modelo relacional maneja estructuras (tablas), que son obviamente más fáciles de manejar que los complicados árboles y redes que se crean en los modelos jerárquico y de red, respectivamente. Sin embargo el modelo relacional no es fácil de adaptar a nuevas aplicaciones, pero tampoco los otros dos. Aunque los modelos jerárquico y de red han sido usados en bastantes casos, se han quedado estancados y no han sufrido mayores modificaciones o extensiones.

3.3 Tendencia del desarrollo de las bases de datos

Una nueva tendencia de la tecnología de las bases de datos es facilitar el desarrollo de aplicaciones complejas no clásicas, y proporcionar capacidad de razonamiento sobre los datos almacenados en la base de datos. Nuevos tipos de aplicaciones tales como sistemas para CAD/CAM, sistemas inteligentes para el manejo de documentos, cartografía, robótica, requieren nuevos tipos de bases de datos, las cuales deben ser más potentes y flexibles que las existentes.

La aparición de estos nuevos dominios de aplicación ha tenido como respuesta, un grande esfuerzo en la comunidad científica para extender, o rediseñar completamente los sistemas de bases de datos. Esta nueva generación de sistemas de bases de datos deberá sostener una gran variedad de aplicaciones, las cuales son mucho más complejas que las aplicaciones tradicionales administrativas.

En particular, los nuevos sistemas de bases de datos necesitan ofrecer nuevos conceptos a nivel de usuario, y más importante es que, necesitan ser más flexibles para soportar los requerimientos tradicionales, y los nuevos.

Una lista de facilidades que debe brindar un nuevo sistema de bases de datos, se puede resumir como sigue:

a) *Tipos de datos y operaciones.* Las aplicaciones no standard necesitan nuevos tipos de datos, porque los existentes actualmente no son adecuados. Información tal como mapas, cartas de navegación, imágenes, piezas de texto, documentos estructurados y dibujos, necesitan ser representados como tipos en un sistema de bases de datos. Los sistemas actuales no contemplan tales nuevos tipos de datos. En algunos casos (por ejemplo, textos), los datos son almacenados como una cadena larga de "strings", la cual no tiene más interpretación por el sistema que ésa. En otros casos, estos nuevos tipos de información son almacenados en sistemas separados, y son extraídos cuando es necesario, e integrados con registros formateados a la base de datos. Esto resulta ser una tarea ardua, y en algunas ocasiones irrealizable. Surgen problemas para asegurar la integridad de los datos, y el buen desarrollo se entorpece. Por lo tanto, las nuevas generaciones de sistemas de bases de datos, deben dar al usuario directamente, nuevos tipos de datos, y más importante, deben permitir la definición de nuevos tipos de datos, sobre los ya construidos. En efecto, cada nueva área de aplicación, tiene en principio, un conjunto de operaciones especializadas para cada tipo de datos específico, que la base de datos debe contener. Se necesi-

sitan operadores especiales para operar textos largos, para comparar imágenes, o para procesar reglas recursivas en aplicaciones basadas en bases de conocimientos.

b) *Representación de conocimiento.* Los nuevos sistemas tienen que explotar el conocimiento que existe sobre la información almacenada en la base de datos. Esto se necesita en varias aplicaciones de bases de conocimientos. Los sistemas de bases de datos deben entonces ser capaces de ofrecer diferentes formalismos para la representación del conocimiento. Como se menciona en [BiGi86], estas necesidades en bases de datos, y otras en inteligencia artificial, han hecho que se unan esfuerzos por parte de los especialistas de ambas ramas, para resolver problemas que en algún momento llegan a ser comunes en las dos disciplinas.

c) *Mecanismos de recobro y control de concurrencia.* Cada nueva área de aplicación, tiene también requerimientos específicos para el control de la concurrencia y recobro de la información, los cuales son diferentes a los tradicionales. Por ejemplo, transacciones largas que manipulan objetos complejos, necesitan mecanismos ad-hoc.

d) *Métodos de acceso y estructuras de almacenamiento.* Las operaciones tradicionales de las bases de datos, como acceso y almacenamiento de información, deben modificarse, debido a que las nuevas áreas de aplicación, requieren de nuevos métodos para efectuar éstas operaciones de manera eficiente, sobre los tipos de datos con características especiales.

Varios enfoques han sido definidos en la literatura con el fin de mejorar la tecnología actual de las bases de datos [Zica88], que incluyen facilidades para desarrollar lo que se mencionó.

Un gran número de organizaciones de investigación, universidades, compañías industriales en Europa, EUA, y Japón,

han desarrollado nuevos sistemas de bases de datos, para abordar nuevas aplicaciones con éxito. Estos esfuerzos se resumen en las siguientes secciones.

3.3.1 Extensiones a las bases de datos existentes.

Este enfoque considera sistemas de bases de datos existentes, básicamente relacionales, y proporciona varias extensiones al modelo teórico básico, con la intención de aumentar la potencia y flexibilidad del sistema, pero manteniendo las ventajas originales del modelo relacional: simplicidad y lenguaje no procedural para la formulación de "queries".

Ejemplos de tales enfoques son los sistemas basados en el modelo de relaciones anidadas, y los sistemas que permiten al usuario la definición de tipos de datos abstractos. En [Kier87] se aborda un sistema que permite extender la noción de dominios relacionales, para incluir tipos de datos abstractos.

3.3.1.1 Relaciones anidadas

Un enfoque para la definición de nuevos modelos de datos, tiende a minimizar las extensiones del modelo relacional, para conservar muchas de sus ventajas. Dentro de este contexto, el relajamiento de la primera forma normal es la principal consideración. Al relajar la primera forma normal se consiente la existencia de relaciones, comúnmente denominadas relaciones anidadas. Una relación anidada permite que los componentes de las tuplas sean a su vez instancias de relaciones, en lugar de valores atómicos. Para las relaciones anidadas, se definen nuevos tipos de operaciones, conservando bastante de la función de las operaciones correspondientes del algebra relacional normal. Entre las nuevas operaciones se enceuntran Nest y Unnest, que han sido investigadas profundamente. Esencialmente, Nest y Unnest permiten pasar de relaciones planas a relaciones anidadas y viceversa.

De esta forma es posible modelar los casos, donde los datos por su definición contienen una estructura de árbol, por ejemplo, formas de oficina, como se muestra en la figura 3.3. Varios es-

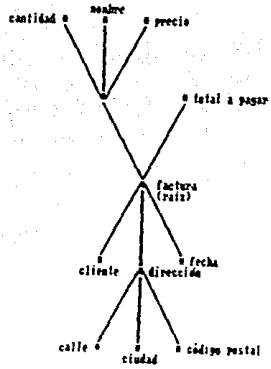


Figura 3.3. Datos de facturas utilizadas en oficinas, con naturaleza arborecente.

tudios teóricos se han hecho sobre el modelo relacional, teniendo como uno de sus productos la factibilidad de extender los resultados del modelo relacional al modelo relacional anidado.

Basado en la idea de relaciones anidadas, el sistema Algres desarrollado en el Politécnico de Milán [Ceri88], ha

sido utilizado como herramienta para desarrollar el prototipo del catálogo de componentes reusables, tratado en esta tesis. Otros esfuerzos de implantación de sistemas de ésta índole son: VERSO [AbBi84] y el implantado en IBM Heidelberg [DaKu86]. El sistema de IBM ha sido experimentado con aplicaciones a la robótica, automatización de fábricas, y manejo de documentos de oficinas. Otros varios prototipos han sido desarrollados basados en el concepto de relaciones anidadas, donde casi la mayoría de ellos, son casos de experimentación, y no están disponibles con fines comerciales.

Otras características que incorporan estos modelos son las de extensión del lenguaje SQL, duplicado de elementos en conjuntos y listas, valores nulos, y la incorporación del operador cerradura ("closure"), que se aplica a expresiones algebraicas, e itera la evaluación de la expresión hasta que el resultado alcanza un punto fijo; esto con el fín de evaluar eficientemente procesos recursivos.

La ventaja básica del modelo relacional anidado (NF2) estriba en su relativa simplicidad, siendo muy apegado al modelo relacional de datos original. Se pueden probar propiedades formales y esto hace al modelo relacional anidado seguro y riguroso [Guch88]. Sin embargo desde el punto de vista práctico, el modelo relacional anidado no es suficiente para cubrir todos los requerimientos de las nuevas aplicaciones, debido a que no todos los datos por naturaleza tienen estructura jerárquica (por ejemplo, gráficas, textos imágenes). Resultados de experimentos con los prototipos existentes darán una mejor respuesta a estas especulaciones. Uno de los objetivos de esta tesis, es experimentar con el prototipo Algres, utilizandolo para crear el catálogo de componentes reusables.

3.3.1.2 Tipos de datos definidos por el usuario

Los sistemas de bases de datos relacionales han sido diseñados, para satisfacer las necesidades tradicionales de

administración de negocios. Los tipos de valores manipulados por estas aplicaciones son simples; la mayoría de las veces están limitados a enteros, reales y cadenas de caracteres. Sin embargo, los límites de los sistemas relacionales se alcanzan cuando tratan de usarse para nuevas aplicaciones. Los tipos de valores manejados por estas nuevas aplicaciones son complejos y específicos a la aplicación. En el modelo relacional [Codd70], nada se dice acerca de los tipos de valores que pueden usarse como dominios relacionales.

Para salvar las deficiencias que surgen con la restricción de los dominios proporcionados por el modelo relacional, se permite en las nuevas extensiones, que el usuario defina sus propios tipos de datos abstractos. Por ejemplo, él puede definir como domino el conjunto de triángulos, con la operación de superficie, y hacer "queries", como "quiero todos los triángulos con superficie 100". Por lo tanto, la potencia del manejador de bases de datos es aumentada, conservando sus propiedades originales. Se puede argumentar que los tipos complejos pueden ser descompuestos en tipos más simples, por el procedimiento de normalización [Codd70]. Sin embargo, al hacer esto , mucha de la semántica de los datos se pierde, y el lenguaje de "queries" no puede expresar tan fácilmente los requerimientos.

3.3.2 Modelos semánticos de datos.

Este enfoque tiende a definir modelos de datos de alto nivel, los cuales son más cercanos a las aplicaciones que el modelo relacional. Estos modelos no son comparables directamente con el modelo relacional o con otros, pero pueden implantarse en particular, sobre el modelo relacional. Aunque en la literatura varios modelos semánticos se han definido, pocos de ellos se han implantado.

La idea que hay detrás de estos modelos es enriquecer el significado de los datos, almacenados en la base de datos, y no tratarlos sólo como cadenas de caracteres. Esto se logra con la creación de nuevos conceptos, como *agregación* y *generalización*.

Agregación es una abstracción en la cual una interrelación entre entidades se considera como otra entidad, pero de un nivel más alto, que se dice ser un objeto agregado.

Ejemplo

Sopongamos que tenemos las entidades:

PROFESOR(rfc, nombre, grado, nombramiento),
MATERIA(clave, nombre, creditos),

un objeto agregado de ellas es el tipo de entidad:

CURSO(PROFESOR, MATERIA, lugar, tiempo_de_duración).

Nótese que CURSO tiene como atributos a las entidades que lo formaron, y además a dos atributos adicionales.

Generalización es una abstracción en la cual un conjunto de tipos de entidades similares se ve como otro tipo de entidad que es más general. Si el tipo de entidad E resulta de la generalización de los tipos de entidades $E_1, E_2, \ldots, E_n$; E se denomina el tipo de entidad genérica de $E_1, E_2, \ldots, E_n$, las cuales se dicen ser subtipos de E. Para definir E de $E_1, E_2, \ldots, E_n$ se ignoran las diferencias entre ellas, y los atributos comunes serán los atributos de E.

Ejemplo

Supongamos que tenemos las entidades

SECRETARIA(rfc, nombre, dirección, velocidad_de_mecanografiado, salario),

INGENIERO(rfc, nombre, dirección, Universidad_de_egreso, salario),

CHOFER(rfc, nombre, dirección, tipo_de_licencia, salario),

su entidad genérica es :

TABAJADOR(rfc, nombre, dirección, salario).

3.3.3 *Sistemas de bases de datos extendibles.*

Este enfoque desarrolla un núcleo para un sistema de bases de datos, el cual es fácilmente extendible, de acuerdo a las demandas de las nuevas aplicaciones. El objetivo de tales sistemas es soportar los prototipos de modelos de datos especializados. Para este propósito, se ofrece un núcleo básico de bajo nivel, el cual puede ser visto como una caja de herramientas para construir los requerimientos específicos.

Antes de concluir este capítulo, no se debe dejar de mencionar que los conceptos de la programación orientada a objetos, actualmente también están siendo utilizados en el área de bases de datos con mucho éxito [Kimw90], [AtBa90], [CaCe90].

En el siguiente capítulo se tratan com más detalle las relaciones anidadas, como parte del modelo NF2. Además, se describe Algres, que es una implantación real del modelo, y ha sido utilizado como herramienta de implantación del catálogo de componentes reusables.

CAPITULO IV

EL MODELO NF2 DE DATOS y ALGRES

En este capitulo daremos una definición precisa del modelo de datos NF2 [ShPi82], [PiTr85], [Guch88]. Describiremos las características del manejador de bases de datos Algres, que es una implantación de este modelo de datos, y ha sido desarrollado en el Politécnico de Milán, y actualmente se encuentra en una fase de experimentación. Uno de los objetivos de esta tesis fué también experimentar con Algres, utilizandolo para representar el catálogo de componentes reusables de software, que consta básicamente de implantaciones y especificaciones.

4.1 El modelo NF2 de datos

El modelo relacional original de bases de datos presentado por Codd [Codd70] permite que estructuras complejas de datos, sean representadas por relaciones, en donde los atributos de estas relaciones deben ser valores atómicos. Cuando los datos han sido representados de esta forma, se dice que estan en la *primera forma normal* (*1NF*). Pero no todos los datos se representan de manera natural por relaciones cuyos atributos, deben ser valores atómicos. El modelo NF2 relaja la primera forma normal, permitiendo que los atributos puedan ser a su vez relaciones, dando como consecuencia lo que se conoce como *relación anidada*. Teniendo ahora relaciones anidadas, el algebra relacional tradicional se puede extender, como lo muestran Gucht y Ficher [Guch88].

Para formalizar la noción de relacion anidada, se necesita el concepto auxiliar de esquema. Un esquema especifica el formato de una relación anidada.

4.1.1 *Notaciones*

Sea Ω el universo de atributos. Un *esquema* S es un árbol para el cual los nodos hoja son elementos distintos de Ω.

El nodo raíz se denota por R(S).

El conjunto de nodos que no son hojas (*interior*) se denota por int(S).

El conjunto de hojas (*atributos*) de S se denota por att(S).

El conjunto de todos los nodos interiores que definen *subesquemas* propios de S, es denotado por sub(S). De otra forma, sub(S) = int(S) - { R(S) }.

El conjunto de todos los nodos interiores que tienen únicamente nodos hoja como hijos, es llamado el conjunto *frontera* de S, y se denota por frn(S).

Si frn(S) = { R(S) }, S es llamado un *esquema plano*.

Si las nociones de att, int, sub, y frn, se aplican a un nodo M que pertenece al int(S), esas nociones podrán aplicarse al subárbol T(M), que tiene a M como raíz. Por lo tanto, att(M) se escribirá en lugar de att(T(M)), etc.

Para un nodo M en S, el conjunto de hijos de M, denotado por C(M), es particionado en los atributos A(M) = C(M) ∩ att(S) y H(M) = C(M) ∩ int(S), ver la figura 4.1.

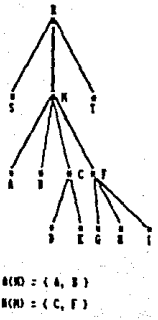


Figura 4.1. A(M) y H(M).

Si las funciones C, A, y H son aplicadas al esquema S, entendemos que en realidad se aplican a la raiz R(S). Por lo tanto, C(S) significa C(R(S)), etc.

Si se etiqueta un nodo con Y*, donde Y denota a un conjunto de nodos de S, se entiende que C(Y*) = Y, ver fig. 4.2.

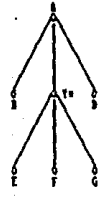


Figura 4.2. Y*, con Y = { E, F, G }.

Finalmente, decimos que dos nodos M,N que están en int(S) son *incomparables* en S, si uno no es descendiente del otro.

Ahora podemos definir una relación anidada. Una relación anidada consiste de un esquema S y una instancia de la raíz R(S), donde una instancia de un nodo se define recursivamente como:

1. Una instancia de un nodo hoja A de S es un valor atómico simple que pertenece al conjunto dom(A), el conjunto de todos los valores posibles del atributo A.

2. Una instancia de un nodo interior M de S es un conjunto finito de tuplas, donde cada tupla t es un mapeo con dominio C(M)tal que, para cada N ∈ C(M), t(N) es una instancia de N.

Si S es un esquema plano, una estructura sobre S es llamada una *relación plana*, o simplemente una *relación*. Una *base de datos* es un conjunto finito de relaciones anidadas.

Como ejemplo de una relación anidada consideremos el esquema de la figura 4.3, con raíz r,

Figura 4.3. Esquema de una raíz anidada.

si denotamos con I(N) la instancia de un nodo N, tenemos que una tupla de I(r) tendrá la forma:

(I(a), I(e), I(b)),

a y b son atributos, por lo que sus instancias son valores atómicos. Una instancia del atributo e consistirá de un conjunto de tuplas, que de acuerdo a la definición cada tupla tendrá la forma

(I(c), I(d)),

donde c y d son atributos, por lo que sus instancias son valores atómicos.

Si el dominio de los atributos a y b es el conjunto de los naturales, y el dominio de c y d es el conjunto de caracteres, una instancia de r puede ser

```
I(r) = {
        ( 1, { ( a, b ), ( a, x ) }, 4 ),
        ( 23, { ( s, e ), ( q, w ), ( t, d ), ( b, t ) }, 34 ),
        ( 13, { ( f, k ), ( o, d ) }, 78 ),
        ( 89, { ( s, g ) }, 6 )
       }.
```

Como otro ejemplo consideremos el esquema ARTICULO de la fig. 4.4. (CLAVES es el nombre del atributo que describe las palabras claves).

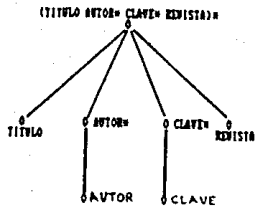


Figura 4.4. El esquema ARTICULO.

Por comodidad, en lugar de escribir (AUTOR)*, escribimos AUTOR*. Por lo que según nuestras definiciones tenemos que

R(ARTICULO) = (TITULO, AUTOR*, CLAVE*, REVISTA)*

int(ARTICULO) = (R(ARTICULO), AUTOR*, CLAVE*)

att(ARTICULO) = (TITULO, AUTOR, CLAVE, REVISTA)

sub(ARTICULO) = (AUTOR*, CLAVE*)

frn(ARTICULO) = (AUTOR*, CLAVE*).

En este caso, sub(ARTICULO) = frn(ARTICULO), pero en general no sucede así (tómese un esquema con profunfidad mayor de 3). Cláramente ARTICULO no es un esquema plano. Finalmente,

A(ARTICULO) = (TITULO, JOURNAL)

H(ARTICULO) = (AUTOR*, CLAVE*).

En la fig. 4.5 mostramos la instancia A sobre el esquema ARTICULO.

(TITULO	AUTOR*	CLAVE*	REVISTA)*
(<T1,	(<A1>, <A2>, <A3>),	(<D1>, <D2>, <D3>)	R1>;
<T2,	(<A1>, <A4>)	(<D1>, <D2>)	R1>;
<T3	(<A5>)	(<D4>, <D5>)	R2>)

Figura 4.5. Instancia A del esquema ARTICULO.

En la figura 4.5 se muestran convenios de notación utilizados en algunos manejadores de bases de datos, que usan el modelo relacional extendido. Las comas separan componentes de tuplas, y el ';' separa miembros de conjuntos.

Para proporcionar un aspecto más entético y de fácil lectura, se acostumbra utilizar una notación tabular para la instancia de una relación anidada. Los renglones denotan diferentes tuplas, y las columnas denotan componentes diferentes. Tuplas con un solo componente no se rodean de los paréntesis "<" y ">". En la figura 4.6 se representa la instancia A en forma tabular.

(TITULO	AUTOR*	CLAVE*	REVISTA)*
T1	(A1, A2, A3)	(D1, D2, D3)	R1
T2	(A1, A4)	(D1, D2)	R1
T3	(A5)	(D4, D5)	R2

Figura 4.6. Instancia de A, en forma tabular.

4.1.2 *El algebra relacional extendida*

Para explotar la información almacenada en la base de datos se requiere de operadores que la manipulen. Los operadores tradicionales en el algebra relacional: union, diferencia, producto cartesiano, selección, y proyección [Ullm82], se pueden extender al modelo relacional extendido [Caca89], conservando básicamente su significado. Dos nuevos operadores NEST y UNNEST, son creados de manera natural, para manipular la estructura de las relaciones anidadas.

Sea t una tupla sobre el esquema S, y sea $Y \subset C(S)$. La *porción Y de t*, denotada por $t[Y]$, es la restricción de la tupla t a Y (una tupla es definida como un mapeo, lo cual permite el uso del concepto de restricción de un mapeo. Para un objeto simple 0, $t[0]$, es una tupla con un solo componente y $t(0)$ es el valor de la entrada en la posición 0 de t).

Sea s una relación anidada sobre el esquema S, y sea $Y \subset C(S)$. Se define $NEST_Y(s)$ como una relación anidada s_Y sobre el esquema S_Y , ver fig. 4.7, donde S_Y es obtenida insertando un nuevo hijo Y^* de R(S) dentro de S, y haciendo todos los miembros de Y hijos de Y^* en S_Y . Además se debe tener que

$C(S_Y) = (C(S) - Y) \cup (Y^*)$, y $C(Y^*) = Y$.

Una tupla $t_Y \in s_Y$, si y solo si existe una tupla $t \in s$ tal que

1. $t_Y [C(S) - Y] = t[C(S) - Y]$, y
2. $t_Y (Y^*) = \prod_Y ((x \in s \mid x[C(S) - Y] = t[C(S) - Y]))$.

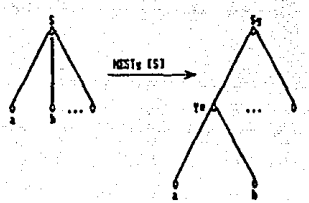


Figura 4.7. La operación NEST, con Y = {a, b}.

Sea s una relación anidada con esquema S, y sea $M \in H(S)$. Entonces UNNEST$_M$ (s) es una relación anidada s_m sobre el esquema S_m, donde S_m es obtenido borrando M de S, y haciendo los hijos de M hijos de R(S). Es decir, $C(S_m) = (C(S) - (M)) \cup C(M)$. Ver figura 4.8. Una tupla $t_m \in S_m$ si y solo si existe una tupla $t \in s$ tal que:

1. $t_m [C(S) - (M)] = t[C(S) - (M)]$
2. $t_m [C(S)] \in t(M)$.

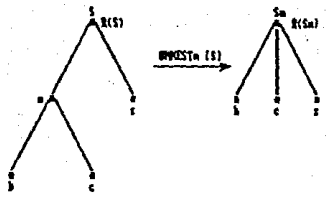


Fig. 4.8. La operación UNNEST.

Por ejemplo, consideremos la relación plana r, definida sobre el esquema (PADRE, HIJO)*, que se muestra en la fig. 4.9. Una tupla (p,c) pertenece a r si p es un padre de c. En la figura 4.10 mostramos las relaciones anidadas

s = NEST$_{\text{PADRE}}$ (r),

u = NEST$_{\text{HIJO}}$ (r),

v = NEST$_{\text{HIJO}}$ (NEST$_{\text{PADRE}}$ (r)),

w = NEST$_{\text{PADRE}}$ (NEST$_{\text{HIJO}}$ (r)).

PADRE	HIJO
p1	c1
p1	c2
p2	c1
p3	c2
p1	c3
p3	c3
p4	c4
p5	c4

Fig. 4.9. La relación r.

PADRE*	HIJO
{p1,p2}	c1
{p1,p2}	c2
{p1,p3}	c3
{p4,p5}	c4

Relación s

PADRE	HIJO*
p1	{c1,c2,c3}
p2	{c1,c2}
p3	{c3}
p4	{c4}
p5	{c4}

Relación u

PADRE*	HIJO*
{p1,p2}	{c1,c2}
{p1,p3}	{c3}
{p4,p5}	{c4}

Relación v

PADRE*	HIJO*
{p1}	{c1,c2,c3}
{p2}	{c1,c2}
{p3}	{c3}
{P4,P5}	{c4}

Relación w

Fig. 4.10. Las relaciones anidadas *s*, *u*, *v*, y *w*.

Este ejemplo ilustra que dos operaciones NEST no son necesariamente conmutativas. Nótese que la relación semántica entre PADRE e HIJO es expresada claramente con las relaciones anidadas, en lugar del uso de una relación plana. Obsérvese además que

r = UNNEST$_{PADRE}$* (s)
= UNNEST$_{HIJO}$* (u)
= UNNEST$_{PADRE}$* (UNNEST$_{HIJO}$* (v))
= UNNEST$_{HIJO}$* (UNNEST$_{PADRE}$* (v))

En la siguiente sección describiremos el manejador de base de datos Algres, que es una implantación del modelo relacional anidado.

4.2 Algres

Algres es un ambiente de programación de bases de datos, que utiliza el modelo relacional anidado de datos. Es un producto, del proyecto del mismo nombre, del Politécnico de Milán, encontrándose actualmente en una etapa de experimentación y desarrollo [Ceri88]. Actualmente una versión de Algres se encuentra funcionando en la Microvax del I.I.M.A.S. de la U.N.A.M., y es la herramienta de programación que se utilizó para implantar el catálogo de componentes reusables.

Además de las propiedades que hereda Algres del modelo NF2, también cuenta con un operador que permite la implantación de "queries" recursivas. El algebra relacional tradicional no puede utilizarse para estos fines, lo que la hace incompleta. Una forma

de salvar esta situación, en los manejadores relacionales de bases de datos, es proporcionando una interface con algún lenguaje imperativo como C o Pascal.

El sistema Algres, proporciona varios elementos para el desarrollo de bases de datos, que a continuación se tratan.

4.2.1 *Modelo de datos*

Un modelo de datos (basado en el modelo de relaciones anidadas o NF2) que incorpora tipos elementales: character, string, integer, real, boolean. Tipos estructurados como RECORD, SET, MULTISET, y SEQUENCE.

RECORD
Los campos de un RECORD son encerrados en (,).
Un RECORD (registro) es una simple tupla, pero sus campos pueden tener una estructura compleja.

```
( 2364023, "Cuernavaca", < ( "Pablo" ) ( "Juan" )
  ("Julian") > )
```

Este es un RECORD con tres campos, de tipos integer, string, y (lista) SEQUENCE.

SET
Los elementos de un SET (conjunto) son encerrados en (,).
Los elementos de un SET (conjunto) deben ser homogeneos, pero pueden ser de un tipo complejo.

```
( ("40") ("12") ("3c") )
( (40) (12) (3) )
```

Estas son dos instancias de dos diferentes conjuntos. En el primero, sus elementos son del tipo string, y en el otro del tipo integer. Las repeticiones de un elemento se consideran como una sola aparición.

MULTISET

Los elementos den un MULTISET (conjunto con repetición) son encerrados en [,].
Deben ser homogeneos sus elementos, pero pueden ser de un tipo complejo. Pueden repetirse elementos, a difenecia de un conjunto.

```
[ ("Bertha", 17), ("Adriana", 22), ("Nancy", 20),
  ("Ivonne", 19), ("Bertha", 17) ]
```

Esta es una instancia de un MULTISET, con una estructura compleja. Los elementos son del tipo RECORD con un campo del tipo string, y uno del tipo integer.

SEQUENCE

Loa elementos de una SEQUENCE (conjunto ordenado, o lista) son encerrados en <,>.
Los elementos de una lista también deben ser homogeneos, o pueden haber repeticiones del mismo elemento.

```
< ("Cancun", 807), ("Manzanillo", 645), ("Playa Azul", 423)
  ("Acapulco", 412) >
```

En este ejemplo los elementos de la lista son del tipo RECORD, con dos campos, un string, y un integer.

Como una combinación de varios tipos estructurados consideremos la relación r, con tipo MULTISET:

r[a, s<b,c, t{d}, e>, f]

en la cual todos los atributos elementales: a, b, c, d, e, f, son enteros, y los atributos *s*, *t*, son una lista y un conjunto, respectivamente. Una instancia estructuralmente compatible con r puede ser

```
[ (6, < (0,1, ( (5) (6) ), 0)  (0,2, ( (6) (7) (5) ), 400) >,
   6578)
  (11, < (0,0, ( (3) (7) (9) ), 8)  (1,2, ( (2) (3) ), 500) >,
   3256)
  (6, < (0,1, ( (5) (6) ), 0)  (0,2, ( (6) (7) (5) ), 400) >,
   6578)
 ]
```

En el modelo relacional anidado (NF2) como se mencionó en el capítulo anterior, una relación anidada consiste de un esquema y una instancia de su raiz. En Algres se usa la palabra objeto para designar a una relación anidada. Un objeto tiene un esquema que consiste de una estructura jerárquica de profundidad arbitraria. Para la relación r del ejemplo anterior, se muestra su esquema en la figura 4.1. Nótese que los nodos que corresponden a atributos no elementales, son etiquetados con el símbolo correspondiente del tipo estructurado.

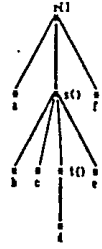


Figura 4.11. Esquema de la relación *r*.

4.2.2 *Algres-prefix*

Algres-prefix es un lenguaje algebraico, que proporciona:

i) Operadores clásicos relacionales con las extensiones necesarias para relaciones anidadas tales como selección, proyección, producto cartesiano, join, unión y diferencia.

ii) Operadores de reestructuración CS (Construct Structure) y DS (Delete Structure), que corresponden a Nest y Unnest del modelo NF2, los cuales modifican la estructura de un objeto complejo.

Consideremos el objeto escuela, definido en Algres:

```
DEF escuela:
(
        clase: integer,
        CLASE:
        (
             ALUMNO: ( nombre: string,
                       PUNTOS: [ puntos: integer ]
                     )
        )
);
```

Con la operación:

```
                DS [ CLASE ] escuela
```

obtenemos el nuevo esquema:

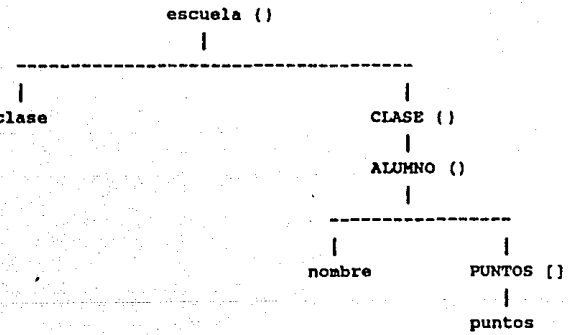
```
escuela: (  clase: integer,
            ALUMNO: ( nombre: string,
                      PUNTOS: [ puntos: integer ]
                    )
         )
```

Comparando las instancias antes y después de DS tenemos:

escuela, antes de DS

```
escuela <- ( (1, ( ( Alejandro, [ 6, 7 ]  )
                   ( Sandra, [ 8, 7] )
                   ( Andres, [ 7, 7 ] )
                 )
               )
              ( 2, ( ( Pablo, [ 8, 7 ] )
                     ( Eric, [ 7, 7 ] )
                     ( Hilda, [ 7, 6 ] )
                   )
             )
           )
```

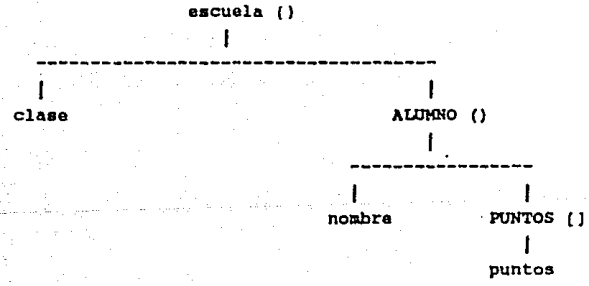
cuyo esquema también se puede dibujar como el árbol:



escuela, después de DS

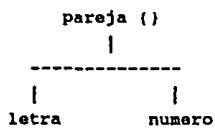
```
escuela <- ( (1,   ( Alejandro, [ 6, 7 ]  )  )
             (1,   ( Sandra, [ 8, 7] )  )
             (1,   ( Andres, [ 7, 7 ] )  )
             (2,   ( Pablo, [ 8, 7 ] )  )
             (2,   ( Eric, [ 7, 7 ] )  )
             (2,   ( Hilda, [ 7, 6 ] )  )
                   )
            )
          )
```

cuyo esquema también se puede dibujar como el árbol:



La operación CS requiere como parámetros el nombre de la estructura a ser creada, y la lista de atributos definidos en esta. Por ejemplo consideremos al objeto pareja (de una letra y un número), con esquema e instancia como sigue:

pareja: (letra: char, numero: integer)

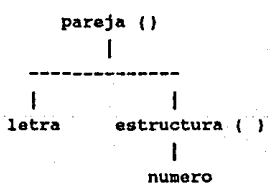


((a, 1) (a, 2) (b, 3))

CS aplicada a pareja sobre el atributo numero, construyendolo como conjunto, multiset, y lista arroja los resultados

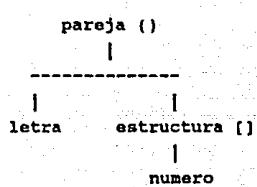
CS [estructura: (numero)] pareja =
((a, (1, 2)) (b, (3)))

cuyo esquema es



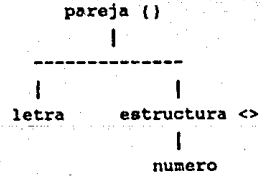
```
CS [ estructura: [ numero ] ] pareja =
                    ( ( a, [ 1, 2 ] )  ( b, [ 3 ] ) )
```

cuyo esquema es




```
CS [ estructura: [ numero ] ] pareja =
                    ( ( a, < 1, 2 > )  ( b, < 3 > ) )
```

cuyo esquema es



Refiriéndonos al ejemplo del objeto escuela, si se quiere tener a los estudiantes en orden alfabético, supongamos que partimos de escuela definido como sigue

```
escuela: (  clase: integer,
            ALUMNO: ( nombre: string,
                        PUNTOS: [ puntos: integer ]
                      )
         )
```

con instancia

```
escuela <- ( (1,  ( Alejandro, [ 6, 7 ]  )  )
             (1,  ( Sandra, [ 8, 7] )  )
             (1,  ( Andres, [ 7, 7 ] )  )
             (2,  ( Pablo, [ 8, 7 ] )  )
             (2,  ( Eric, [ 7, 7 ] )  )
             (2,  ( Hilda, [ 7, 6 ] )  )
                  )
           )
         )
```

Las operaciones

```
SQ [ ASC nombre: CLASE ]
CS [ CLASE: < ALUMNO > ] escuela
```

(La operación SQ ordena alfabéticamente con respecto al nombre del alumno).

crean el objeto

```
DEF escuela:
(
        clase: integer,
        CLASE:
        <
             ALUMNO: ( nombre: string,
                       PUNTOS: [ puntos: integer ]
                     )
        >
)
```

con instancia

```
escuela <- ( (1, <
                    ( Alejandro, [ 6, 7 ]  )
                    ( Andres, [ 7, 7 ] )
                    ( Sandra, [ 8, 7] )
                  >
             )
             ( 2, <
                    ( Eric, [ 7, 7 ] )
                    ( Hilda, [ 7, 6 ] )
                    ( Pablo, [ 8, 7 ] )
                  >
             )
           )
```

iii) Facilidades para efectuar "queries" utilizando "tuple functions" (TF). Se tienen cuatro TF: "Tuple update" (TU), "Tuple extension" (TE), "Selective update" (SU), y "Selective extension" (SE).

TU actualiza la instancia de un objeto, sin cambiar su esquema, alterando en todas las tuplas los atributos indicados.

TE adiciona un atributo a todas las tuplas de su operando, inicializandolo con un valor especificado por una función.

SU actualiza solo ciertas tuplas de un objeto dependiendo del valor de un predicado.

SE agrega un atributo a todas las tuplas, incializándolos con un valor seleccionado con un if-then-else.

Por ejemplo para el objeto escuela ya definido con anterioridad

```
DEF escuela:
(
        clase: integer,
        CLASE:
        (
             ALUMNO: ( nombre: string,
                       PUNTOS: [ puntos: integer ]
                     )
        )
)
```

Si todos los estudiantes salieron exitosos en sus clases al final del año, podemos actualizar sus nuevas clases con la instrucción

```
TU [ CLASE.clase := CLASE.clase + 1 ] escuela.
```

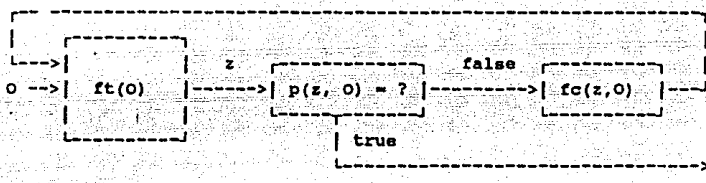
También cuenta con funciones para evaluar la cardinalidad de un objeto, su número de atributos, transformación de tipos estructurados, y otras más que facilitan las "queries".

iv) Un operador de punto fijo FP("Fix Point"), que es la única operación en Algres que controla secuencia de instrucciones. Con FP es posible implantar un ciclo, y "queries" recursivas.

El operador de punto fijo se aplica repetidamente a una

función ft (transformador) que a su vez se aplica sobre un objeto, combinando su resultado con una nueva función fc (combinador), el cual calcula el nuevo objeto que será aplicado en el siguiente ciclo. El ciclo se repite hasta que el predicado de salida p es true.

Un diagrama de FP es el siguiente:



Los tres bloques funcionales corresponden respectivamente al transformador, el predicado de éxito, y al combinador.

El significado es el siguiente: O es el nombre del operando de FP.

O se toma como el argumento inicial de ft, posiblemente con otros argumentos O1, O2, ..., On, con lo que se calcula el valor z.

Antes de ir a la función de combinación fc, el predicado de salida p(z,O) se calcula; este predicado puede contener más argumentos ademas de z,O. Si p es true, el ciclo es interrumpido y el resultado de la operación es O.

Si p es falso, la función de combinación fc(z,O) se calcula; posiblemente con más argumentos. La función fc calcula el nuevo valor de O (que obviamente se reemplazará en todos los lugares donde aparezca O).

En código Algres FP se escibe como sigue:

```
R <- FP [ z := ft(O);
          p(z,O);
          fc(z,O)
        ] O
```

que expresado en Pascal significa:

```
fin := false;
R := O;
repeat
    z := ft(R);
    if p(z,R)
       then
          fin := true
       else
          R := fc(z,R)
until fin
```

El resultado de FP es del mismo tipo que O.

Ejemplo. Dada la relación hijo-padre,

```
DEF Padres:
{
     hijo; string,
     padre: string
}

{
   ( Jorge, David )
   ( Marcos, Alejandro )
   ( David, Marcos )
   ( Roberto, Jorge )
}
```

calculemos la relación decendiente-ancestro:

```
DEF Temp_ancestros
( ( decendiente: string,  ancestro: string ) )

DEF Ancestros ( ( decendiente: string, ancestro: string) )



Temp_ancestros <- Padres

Ancestros <- FP [
                  z := PJ [ hijo, ancestro ]
                       JN [ padre = decendiente ]
                       Padres Temp_ancestros;
                  CONTEQ ( z; Temp_ancestros );
                  UN Temp_ancestros z
               ]  Temp_ancestros
```

Los simbolos que aparecen en este cálculo significan:

PJ : Proyección
JN : Join
CONTEQ : contenido o igual

El órden de aplicación de las operaciones, es análogo al que se usa en la composición de funciones en el contexto matemático: en g(f(x)), primero se aplica f, y después g. Así, en el cálculo de z, primero JN se aplica a los argumentos Padres, Temp_ancestros, y al resultado se le aplica la proyección PJ, dejando el resultado en z.

La instancia resultante para ancestros es

```
{
  ( Roberto, Jorge )
  ( Roberto, David )
  ( Roberto, Marcos )
  ( Roberto, Alejandro )
  ( Jorge, David )
  ( Jorge, Marcos )
  ( Jorge, Alejandro )
  ( David, Marcos )
  ( David, Alejandro )
  ( Marcos, Alejandro )
}
```

Algres está integrado con un sistema de bases de datos relacional (Informix), pero no está implantado arriba de él. Se usa por razones de eficiencia. Existe una máquina abstracta llamada RA ("Relational Algebra") que soporta el modelo de datos extendido, y el lenguaje algebraico extendido (algres-prefix). Una parte muy importante del ambiente Algres (Fig. 5.9) consiste del traductor de Algres al lenguaje del interprete RA. Algres ha sido diseñado para usarse en máquinas que utilizan el sistema operativo UNIX.

Las aplicaciones en Algres se pueden efectuar directamente en Algres-prefix. Sin embargo al programador se le proporcionan también los lenguajes ALGRESQL, DATALOG, y ALICE. ALGRESQL es una extensión del SQL tradicional, es decir, un lenguaje parecido al inglés para efectuar "queries", a diferencia de Algres-prefix que es un lenguaje netamente algebraico. Sin embargo existe una correspondencia biunívoca entre Algres-prefix y ALGRESQL. Programas en ALGRESQL son traducidos a Algres-prefix antes de su ejecución. DATALOG es un lenguaje de programación lógica, sintácticamente similar a Prolog, para utilizar la capacidad deductiva del lenguaje sobre las relaciones. ALICE ("Algres In C Embeded") es una combinación de Algres-prefix y el lenguaje C [Lamp89], resultando ser un lenguaje híbrido, donde cualquiera

de los dos métodos de programación puede ser utilizado, o si se prefiere se pueden mezclar. Las operaciones relacionales pueden en este contexto parametrizarse con resultados computados en C. Se pueden además, convertir objetos de Algres en estructuras de C, y viceversa. Cuando se escribe un programa en Alice, primero se traduce a un programa en C puro, que se compila posteriormente. De estos lenguajes solo se tienen implantados actualmente, Algres-prefix y Alice.

El catálogo de componentes reusables se implantó primero en Algres-prefix, para experimentar con él, y posteriormente se utilizó Alice para mejorarlo, quedando en Alice la última versión. Este catálogo se describe en el siguiente capítulo.

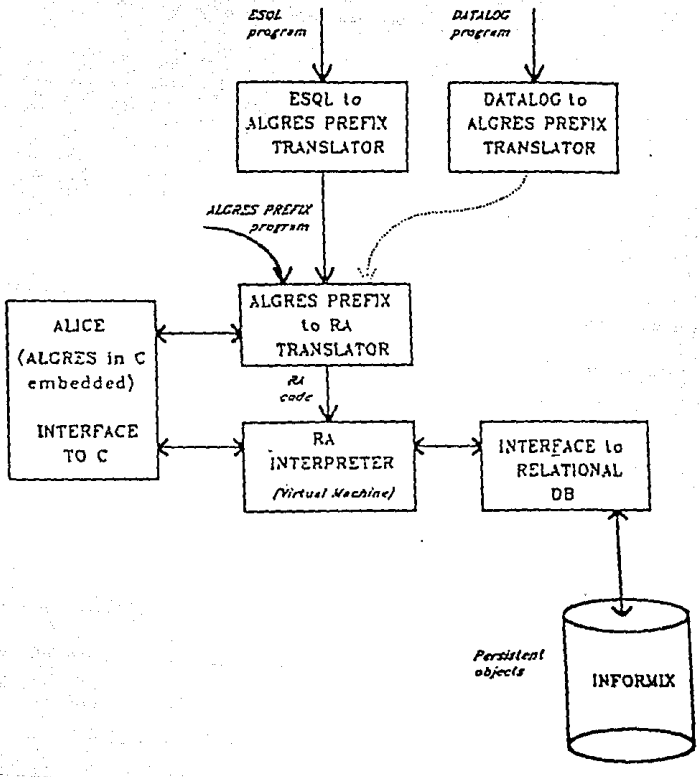


Figura 4.12. Ambiente Algres (Tomada de [Ceri88]).

CAPITULO 5

CATALOGO DE COMPONENTES REUSABLES

5.1 Objetivos

La creación del catátologo de componentes reusables de software tiene como objetivos:

1. Ayudar a los diseñadores y programadores de software, la creación de nuevos componentes complejos, utilizando los componentes reusables existentes.
2. Proporcionar la información necesaria acerca de las características de la implantación de un componente, a través de su especificación, e información de reuso.
3. Experimentar con el ambiente de programación Algres, como herramienta de implantación del catálogo.

5.2 Preliminares

Un componente reusable tiene asociada una especificación, y una o varias implantaciones, como se muestra en la figura 5.1.

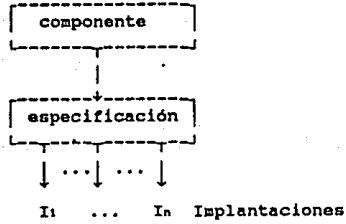


Fig. 5.1. Un componente, su especificación, y sus implantaciones.

El catálogo contiene información de los componentes, clasificados por tópicos, proporcionada por una especificación algebraica p a ra cada uno, e información de reuso de sus implantaciones. Esto permite ver a un componente como una caja negra, cuyo funcionamiento está garantizado, es decir, cada implantación satisface su especificación.

Como lenguaje de especificación se ha usado el lenguaje algebraico LESPAL [Okta89], el cual ha sido desarrollado en el I.I.M.A.S, de la Universidad Nacional Autónoma de México. LESPAL permite la definición modularizada, jerárquica, y parametrizada de componentes de software.

Otros supuestos considerados en el diseño son:

Cada especificación almacenada en el catálogo, tiene asociada al menos una implantación en un lenguaje de programación.

Las implantaciones son correctas, es decir, satisfacen su especificación.

En el conjunto de especificaciones, se definen las relaciones de herenia *(H)*, e importación *(I)*:

Definición 1. *x H y*, si la especificación *x* es heredera de las propiedades de la especificación *y*.

Esta relación, ocasiona que se creen jerarquías hereditarias, donde tenemos ancestros y descendientes, bajo el contexto anteriormente descrito. En la figura 5.2 tenemos árboles que ilustran un conjunto de especificacines, y la estructura que origina *H*. Notemos que de manera natural la relación *H*, se puede modelar con el modelo de datos que proporciona el ambiente de programación Algres.

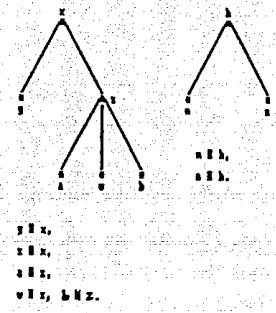


Figura 5.2. Estructura originada por *H* en un conjunto de especificaciones.

Definición 2. *x I y*, si la especificación *x* importa a la especificación *y*.

La relación I también origina una estructura jerárquica en el conjunto de especificaciones, como se muestra en la figura 5.3.

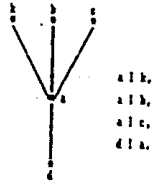


Fig. 5.3 Estructura de la relación I sobre un conjunto de especificaciones.

Es importante no confundir el significado de las relaciones de importación y de herencia. Cuando escribimos φ H λ, es decir, φ es heredera de la especificación λ, se debe entender que todas las propiedades que tenga λ, las tendrá φ, pero φ tendrá además sus caracteristicas muy particulares. Por ejemplo, si φ es la especificación de árbol binario, y λ es la especificación de árbol en general, entonces φ será una gráfica dirigida, tendrá una raiz, tendrá nodos, que pueden tener desde cero, hasta un número finito de hijos, tendrá nodos hojas, etcétera, propiedades que hereda de λ. Pero, φ tendrá la propiedad especifica, de que cualquier nodo solo puede tener cero o dos hijos.

El hecho de que la especificación α, importe a la especificación β, simplemente significa que α, solo utilizará funciones, constantes, tipos, etcétera, que ya han sido especificados en β, sin que haya transmisión de propiedades entre α y β.

El catálogo contiene cuatro componentes:

1. Diccionario general de tópicos.
2. Colección de especificaciones.
3. Colección de implantaciones.
4. Colección de dependencias entre especificaciones.

Estos componentes tienen asociados uno o más objetos complejos en Algres, cuyos esquemas describimos a continuación.

5.3 Diccionario de tópicos

El diccionario de tópicos contiene la información general, acerca de los componentes de software del catálogo. Esta organizado como un conjunto de temas. Cada tema contiene un conjunto, cuyo nombre es Modulos. El conjunto Modulos,

proporciona el nombre de los componentes pertenecientes a cada tema, el nombre del lenguaje de especificación, y los nombres de los lenguajes en que se encuentra implantado cada componente.

El esquema Algres del diccionario de tópicos es el siguiente:

```
DEF Dicc_Topicos:
{
     Tema: string,
     Modulos:{
               nom_comp: string,
               lengs_esp: string,
               lengs_impl: < nom: string >
             }
}
```

Una instancia de Dicc_Topicos puede ser:

```
Dicc_Topicos <- { ( Tipos basicos { (Boolean, LESPAL, <C> )
                                    (Integer, LESPAL, <C,
                                                Modula-2>)
                                    (Natural, LESPAL, <C, ADA>)
                                  }
                  )
                  ( Ordenamientos { ( Burbuja, LESPAL, <C>)
                                    ( quick-sort, LESPAL,
                                      < ADA, Modula-2, C > )
                                    }
                  )
                }
```

5.4 Conjunto de especificaciones

Se tiene un objeto complejo en Algres, que mantiene el conjunto de especificaciones, que refleja el formato seguido en la escritura de una especificación en LESPAL. Se requieren en principio dos objetos auxiliares:

```
DEF sorts: < sort: string >;

DEF operaciones: < nom_op: string,
                   dominios: < dominio: string >,
                   codominio: string
                 >;
```

Recordar que <> representa a una lista. Ahora podemos definir la relación adecuada para las especificaciones:

```
DEF especificaciones:
( nom_esp: string,
  enriched_by: string,
  parametros: <
                nom_param:string,
                sorts: VS[],
                operaciones: VS[]
               >,
  exports: < sorts: VS[],
             operaciones: VS[] >,
  imports: ( n_esp: string ),
  sorts: VS[],
  operaciones: VS[],
  variables: < nombre: string, tipo: string >,
  axiomas: < linea: string >
)
```

Una instancia del objeto "especificaciones", puede ser:

```
especificaciones <-
( (Stacks,
    < (Elements, <Elem>, <> ) >,
    <( <Stack,Elem,>,
       < (empty, <>,Stack ),
         (push, < Elem, Stack >, Stack ),
         (pop, < Stack >, Stack ),
         (top, <Stack>, Elem),
         (is-empty, <Stack>, Boolean )
       >
     )
    >,
   ( Boolean ),
   <Stack,Elem>,
   <      (empty, <>,Stack ),
          (push, < Elem, Stack >, Stack ),
          (pop, < Stack >, Stack ),
          (top, <Stack>, Elem),
          (is-empty, <Stack>, Boolean )
    >,
   < (e,Elem), (s,Stack) >,
   < ([S1] pop(empty) = error) ,
     ([S2] pop(push(e,s)) =s ),
     ([S3] is-empty(empty) = true),
     ([S4] is-empty(push(e,s)) = false),
     ([S5] top(empty) = error ),
     ([S6] top(push(e,s)) = e )
   >
)

   (B-Stacks,
    < (Tam-maximo, <Tam-max>, < (Max, <>, Nat) > ) >,
    < >,
   ( Natural ),
   <>,
   < (size,<Stack>,Nat) (is-full,<Stack>,Bool) >,
```

```
    <>,
    < ([S7] size(empty)=0),
      ([S8] size(push(e,s)) = succ(size(s))),
      ([S9] is-full(s) = true WHEN eq(size(s),Tam-Max) = true),
      ([S10] is-full(s) = false WHEN eq(size(s),Tam-Max) = true),
      ([S11] push(e,s) = error WHEN eq(size(s),Tam-Max) = true) >
  )
)
```

Recalcamos que el atributo "imports" describe la interface entre la especificación, y las especificaciones que ayudan a crearla. El atributo "exports", define la interface entre la especificación, y las especificaciones que la utilizan. El atributo "enriched", expresa que la especificación es heredera de otra especificación.

5.5 Conjunto de implantaciones

La información acerca de las implantaciones en el catálogo, se representa por el objeto "implementación". Para cada especificación se tienen un conjunto de implantaciones, que pueden estar realizadas en diversos lenguajes de programación. Cada implantación tine un nombre de referencia (nombre del archivo, que se encuentra en el atributo "codigo"), e información adicional, como autor (marca), descripción de complejidad, e información de reusabilidad.

El esquema Algres de "implementación" es:

```
DEF implementacion:
(
  nom_esp: string,
  implts:(
           lenguaje: string,
           variables: (
                        codigo: string,
                        marca: string,
                        complejidad: string,
                        inf_reus:  string
                      )
         )
)
```

Una instancia del objeto "implementacion", puede ser:

```
implementacion <-
(
 ( Stacks, ( (Modula-2, ( (Stacks.MOD,
                           Armando Hdz. S.,
                           Comp.Stacks.Mod.txt,
                           Reus.Stacks.Mod.txt)

                          (Stacks1.MOD
                           Armando Hdz. S.,
                           Comp.Stacks1.Mod.txt,
                           Reus.Stacks1.Mod.txt)
                          ) )

             (ADA, ( (Stacks.ADA,
                           Armando Hdz. S.,
                           Comp.Stacks.ADA.txt,
                           Reus.Stacks.ADA.txt)))

           )
 )
```

```
( Bounded-Stacks, ( (Modula-2,
                        ( (BStacks.MOD,
                           Armando Hdz. S.,
                           Comp.BStacks.Mod.txt,
                           Reus.BStacks.Mod.txt)

                          (BStacks1.MOD
                           Armando Hdz. S.,
                           Comp.BStacks1.Mod.txt,
                           Reus.BStacks1.Mod.txt)
                          ) )

                    (ADA, ( (BStacks.ADA,
                           Armando Hdz. S.,
                           Comp.BStacks.ADA.txt,
                           Reus.BStacks.ADA.txt)))

          )
 )
)
)
```

5.6 Dependencias entre especificaciones

Las relaciones *H* e *I* sobre el conjunto de especificaciones, pueden utilizarse para construir objetos Algres que nos proporcionan información importante acerca de las dependencias entre las especificaciones. Esta dependencia se expresa en una especificación en LESPAL por medio de las clausulas "imports", y "enriched_by". Que como ya se ha mencionado, con "imports" se indica, que especificacines se importan, y con "enriched", se expresa quien es el padre de la especificación tratada.

Se pueden derivar tres objetos a partir del objeto especificaciones: *ancestros, descendientes, e importaciones.*

El objeto *ancestros*, es un objeto con esquema

```
DEF ancestros:
( espec: string,
  conj_ances: ( ances: string )
)
```

que es un conjunto que nos da para cada especificación, el conjunto de nombres de sus ancestros.

El objeto *ancestros* se puede construir con el operador de punto fijo FP, a partir del objeto *especificaciones*, como se muestra en el apéndice A.

Supongamos que la relación *H* en el objeto especificaciones, nos induce la estructura jerárquica mostrada en la figura 5.4,

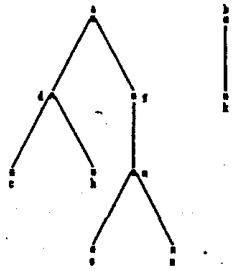


Figura 5.4. Una jerarquía inducida por la relación *H*.

donde por ejemplo *o H m*, *n H m*, *m H g*, etcétera. El objeto "ancestros", para este caso es:

```
{ (o,{m,g,a}),
  (n,{m,g,a}),
  (m,{g,a}),
  (g,{a}),
  (c,{d,a}),
  (h,{d,a}),
  (d,{a}),
  (k,{b})
}.
```

El objeto *descendientes* se puede construir a partir de los objetos *especificaciones*, y *ancestros* como se muestra en el apéndice B. La estructura de *descendientes* se define así:

```
DEF descendientes:
{
  nom_esp: string,
  conj_desc: { hijo: string }
} ,
```

dando para cada especificación, el conjuto de nombres de sus descendientes. Para el conjunto de especificaciones implícito en la figura 5.1 tenemos que el objeto *descendientes* tendrá como instancia

```
{ (a,{d,c,h,g,m,o,n} ),
  (d,{c,h}),
  (g,{m,o,n}),
  (m,{o,n}),
  (b,{k}) }.
```

El objeto *importaciones* se puede construir a partir del

objeto *especificaciones*, como se muestra en el apéndice C. La estructura de importaciones se define así:

```
DEF importaciones:
(
  nom_esp: string,
  conj_imps: ( cimps: string )
),
```

proporcionando para cada especificación, el conjunto de nombres de especificaciones que la importan.

Consideremos las siguientes especificaciones relacionadas por *I*:

a I b,
a I c,
b I k,
d I a,

donde las especificaciones *c* y *k* no "importan" a ninguna especificación.

Para esta relación *I* particular, la instancia del objeto *importaciones* será:

((a,d), (b,a), (c,a) (k,b)).

5.7 Interacción con el sistema

El catálogo de componentes reusables, es un prototipo que interactua con el usuario por medio de menús. Se puede ilustrar

la trayectoria de interacción a través de los menús, por el árbol de la figura 5.5. Para tener una sesión con el prototipo simplemente se ejecuta el comando *ccreusa*, que significa catálogo de componentes reusables.

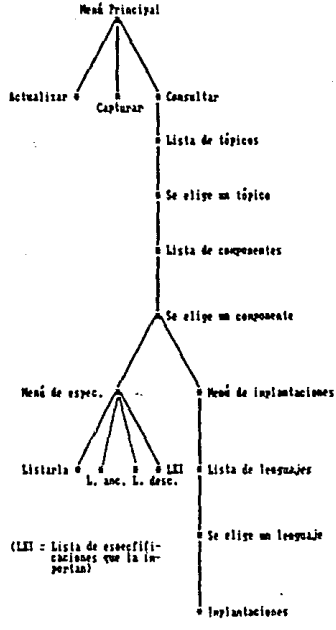


Figura 5.5. Arbol de menús.

En la raíz del árbol se encuentra el menú principal, que presenta las opciones de *captura*, *actualización*, y *consulta*.

La opción de captura permite capturar el *diccionario de tópicos*, las *especificaciones*, y las *implantaciones*.

La opción de actualización permite actualizar los objetos que contiene la información concerniente a los conjuntos de ancestros, descendientes, e importaciones; y si se desea, se pueden observar en ese momento.

La opción de consulta presenta una lista de *tópicos*, para los cuales existen componentes reusables. Se le pide al usuario que seleccione el tópico que desee, y para ese tópico se le muestran los nombres de los componentes reusables que existen. Se le pide que escoja un componente. Una vez elegido el *componente* se le presentan dos opciones: consultar información acerca de su *especificación* o de sus implantaciones.

Si decide consultar la información acerca de su especificación, puede preguntar cual es el conjunto de sus *ancestros, hijos (descendientes), y* especificaciones que la *importan*. Además puede listar la especificación del componente.

Si decide consultar la información referente a sus *implantaciones*, se muestra en que lenguajes está implantado el componente, pidiendo que elija un lenguaje de implantación. Una vez elegido el lenguaje, el sistema mostrará al usuario el nombre del archivo que contiene la versión del componente, su autor, información referente a su complejidad, e información de reuso.

Este prototipo es un esfuerzo con el fin de promover la creación, y utilización de componentes reusables de software;

esperando que se hagan en el futuro más sistemas con este fin. Esto debe ayudar a incrementar la rapidez de diseño, e implantación de sistemas de software.

Para mayor información sobre la ubicación del prototipo en la Micro-vax del I.I.M.A.S, ver el apéndice D, que además indica como recompilar el ambiente de programación Algres en otra máquina similar con sistema operativo Unix.

CONCLUSIONES

Son pocos los sistemas que existen análogos al catálogo de componentes reusables que se propone. En la literatura solo se encontraron dos: el de Prieto y Freeman [PrFr87], y el sistema PARIS, de Katz, Richter, y Khe-Sing The [BiPe89a].

El sistema de Prieto y Freeman (SPF) tiene características muy interesantes, pero los componentes reusables para los que está construido solo proporcionan una operación, como por ejemplo, un componente que lee lineas de un archivo de texto, o uno que suma matrices. Esta restricción simplifica su mecanismo de clasificación, y evaluación de la similaridad de componentes. Componentes reusables genéricos que se pueden realizar con lenguajes como ADA y Modula-2, pueden proporcionar múltiples operaciones. Se puede hacer un componente genérico en ADA que efectue las operaciones de suma resta y mutiplicación de matrices, el cual no se puede clasificar en SPF; se tendría que separar, haciendo un componente para cada operación. En nuestro catálogo se puede clasificar ese componente, en el tópico de operaciones sobre matrices, sin necesidad de particionarlo.

En nuestro prototipo de catálogo se puede reconsiderar la clasificación de los componentes, punto que realzan Prieto y Freeman. Este aspecto puede mejorarse para el sistema final, utilizando los conocimietos de clasificación de productos literarios.

En PARIS si es posible manejar componentes reusables que proporcionan multiples operaciones (se utiliza el término "esquema parcialmente interpretado", en lugar de "componente reusable").

El sistema de clasificación de esquemas es muy pobre, ya que solo es un conjunto no ordenado de esquemas.

Al igual que en SPF, se debe definir un vocabulario, que en este caso se utiliza para realizar la sentencia de programa, que no es más que la descripción de requerimientos.

La elaboración de la sentencia de programa es muy laboriosa, ya que se necesita del conocimiento del lenguaje de especificación que ellos utilizan para escribir una de las secciones del esquema. Por lo tanto la interface con el usuario no es muy amigable.

Cuando la sentencia de programa no encuentra un esquema que la satisfaga, el probador de teoremas falla, y se necesita ser un experto en su manejo, para saber porque falla, y hacer las correcciones pertinentes.

Tanto SPF como PARIS incluyen ayuda al usuario para localizar un componente reusable, conociendo sus requerimientos. El prototipo propuesto se puede mejorar en este sentido, utilizando un sistema experto que interactue con él, y que contenga como base de conocimientos, además de la que proporciona la base de datos del catálogo, información adicional, que le permita al sistema experto, por ejemplo, deducir si el usuario requiere un componente de l tópico de estructuras de datos, componentes básicos, sorts, etcétera. En este estado, de los componentes existentes en el tópico, hay que indagar cual es el que necesita el usuario. El sistema experto, debe proporcionar mínimamente el tópico y el nombre del componente, que existen en el catálogo, y pueda utilizar la información que le puede proporcionar.

Nuestro prototipo no puede realizar una instancia de un componente en forma automática. Esta tarea se le deja al usuario, aunque se le indica como relizarlo. Se puede anexar esta característica en un trabajo posterior.

Ni SPF ni PARIS proporcionan las relaciones de interde-

dependencia entre especificaciones formales (ancestros, descendientes, e importaciones), que se han implantado en el prototipo.

El prototipo tiene como supuesto que los componentes reusables han sido creados a partir de algún método formal. Actualmente los métodos formales se usan para:

-Identificar muchas deficiencias, aunque no todas, en un conjunto de requerimientos informalmente establecidos, detectar discrepancias entre una especificación y su implantación, asi como para encontrar errores en programas y sistemas.

-Especificar problemas de tamaño medio no triviales, especialmente el comportamiento funcional de programas secuenciales, tipos de datos abstractos y hardware.

-Entender profundamente el comportamiento de sistemas largos, y complejos.

Una de las tendencias de investigación de la comunidad de bases de datos, es la de extender el modelo relacional. El ambiente de programación Algres, el cual se encuentra dentro de esta linea de investigación, es producto del proyecto con el mismo nombre, y ha sido desarrollado en el Politécnico de Milán. Algres no se encuentra completamente terminado, pero se tiene ya una versión. Se escogió como herramienta de implantación del prototipo, para experimentar con él, analizando su utilidad para nuestra aplicación.

La naturaleza de las especificaciones manejadas por el catálogo, vistas como datos, se puede representar fácilmente en Algres, debido a que las relaciones H e I inducen jerarquías, que se representan directamente en el modelo de datos, sobre el conjunto de especificaciones. La implantación

de los objetos *ancestros, descendeintes, e implantaciones*, se puede hacer a partir del conjunto de especificaciones con el operador de punto fijo. Por lo tanto, Algres resultó ser adecuado para el prototipo.

La actualización que se realiza en el prototipo, consiste solamente en crear nuevamente las estructuras de *ancestros, descendientes, e importaciones*, cuando se captura un nuevo componente. Una actualización para el caso en que se actualiza una tupla del conjunto de especificaciones, se puede realizar así:

Sea $\Psi = (c_1, c_2, \ldots, c_n)$ el conjunto de especificaciones. Si se modifica la tupla c_i, dando lugar a la nueva tupla η_i, entonces Ψ, debe actulizarse por la asignación

$$\Psi \leftarrow (\Psi - (c_i)) \cup (\eta_i)$$

Para actualizar los conjuntos de *ancestros, descendientes, e importaciones*, solo hay que activar las funciones que los calculan con el nuevo valor de Ψ.

Esta actualización sería conveniente incluirla cuando el prototipo, pase a ser un sistema liberado.

Apéndice A

Construcción del objeto ancestros

El esquema del objeto ancestros es el siguiente:

```
DEF ancestros:
(
   espec: string,
   conj_ances: ( ances: string )
)
```

Para cada especificación se tiene el conjunto de sus ancestros.

Se necesitan además los objetos auxiliares:

```
DEF T: VS [ ancestros ]
DEF P: VS [ ancestros ]
DEF ancestros_red: VS [ ancestros ]
```

Se calcula el objeto T:

```
T <- CS [ conj_anc: ( enriched ) ]
     PJ [ nom_esp, enriched ]
     SL [ NOT(enriched = " ")  ] especificaciones
```

Haciendo una copia en P:

```
P <- T
```

T contiene la información referente a las especificaciones con atributo "enriched" no nulo.

El siguiente calculo genera el conjunto de ancestros, pero con redundancia:

```
ancestros_red <- FP[ Par := PJ[ P.espec , T.conj_ances ]
                            TU [ T.conj_ances :=
                                 UN P.conj_ances T.conj_ances ]
                            JN [ MEMBER( T.espec; P.conj_ances) ]
                               ] P T;
                      CONTEQ(Par; T);
                      UN T Par
                    ] T
```

Finalmente, la redundancia se elimina fácilmente mediante:

```
ancestros <- CS [ anc: [ ances ] ]
              DS [ conj_ances ] ancestros_red
```

La redundancia que se ha referenciado se puede ilustrar con un ejemplo:

Supongamos que en el conjunto de especificaciones, las especificaciones *a*, b, c, d, e, están relacionadas así: *a H* b, b *H* c, c *H* d, d *H* e. Para la especificación *a*, en el objeto "ancestros_red", tendremos las tupas

```
{ ..., (a, { b }),(a, { b, c }), (a, { b,c,d }),
                                 (a, { b,c,d,e }), ...)
```

que se pueden reducir a la tupla

```
{ ...,(a, { b,c,d,e }), ...}
```

que dice que la especificación *a*, tiene como encestros a las especificaciones b,c,d,e.

Apéndice B

Construcción del objeto descendientes

El esquema del objeto descendientes es el siguiente:

```
DEF descendientes:
(
   nom_esp: string,
   conj_desc: ( hijo: string )
)
```

Para cada especificación se tiene el conjunto de sus descendientes.

Se necesitan además el objeto auxiliar:

```
DEF Temp_desc: VS [ descendientes ]
```

Se calcula Temp_desc

```
Temp_desc <- UN ancestros
             TE [ ances_vacio := ( ) ]
             PJ [ nom_esp ]
             SL [ enriched = " "  ] especificaciones
```

el cual contiene a la estructura ancestros, más las especificaciones que son las raices de los árboles en la jerarquía, es decir, su conjunto de ancestros es vacio. El cálculo comienza con la selección de las especificaciones raices, que tienen el atributo "enriched" nulo, de las cuales solo se proyecta su nombre. A estas especificaciones se les agrega el conjunto de ancestros vacio, con la operación TE. Finalmente se hace la union entre el conjunto de ancestros, y las especificaciones raices.

El objeto descendientes se obtiene mediante la sucesión de operaciones JN, PJ, y CS:

```
descendientes <- CS [ tconj_desc: ( ancestros.nom_esp ) ]
                 PJ [ Temp_desc.nom_esp, ancestros.nom_esp ]
                 SL [ MEMBER( Temp_desc.nom_esp; conj_ances ) ]
                    Temp_desc ancestros
```

Se efectúa un join entre "Temp_desc" y "ancestros", para obtener cada uno de los hijos de cada especificación, con la condición de que el nombre de la especificación sea miembro del conjunto conj_ances (en el epéndice A se encuentra la estructura de "ancestros"). Ahora se proyectan los nombres de las especificaciones en "Temp_desc" y "ancestros", obteniendo el resultado final con un "nest" sobre el atributo ancestros.nom_esp, reuniendo a todos los hijos de cada especificación.

Apéndice C
Construcción del objeto importaciones

El esquema del objeto importaciones es el siguiente:

```
DEF importaciones:
(
   espec: string,
   conj_imps: ( cimps: string )
)
```

Para cada especificación se tiene el conjunto de especificaciones que la importan.

Se necesitan además los objetos auxiliares:

```
DEF coimp: ( t: string )
DEF temp_imp: VS [ importaciones ]
```

Se calcula el objeto "coimp", que contiene únicamente los nombres de las especificaciones:

```
coimp <- PJ [ nom_esp ] especificaciones,
```

Se calcula ahora el objeto "temp_imp":

```
temp_imp <- PJ [nom_esp, imports ] especificaciones,
```

que contiene la especificación, y el conjunto de especificaciones importadas.

Finalmente se obtiene el objeto importaciones:

```
importaciones <- CS [ conj_imps: ( temp_imp.espec ) ]
                 PJ [ t, temp_imp.espec ]
                 JN [ MEMBER( t; temp_imp.conj_imps ) ]
                                    coimp temp_imp,
```

donde el join localiza en que lugar es importada cada especificación. La proyección da las parejas (c, Γ_1), (c, Γ_2), ..., (c, Γ_n), donde cada especificacción Γ_i importa a la especificación c. Por último se hace un "nest" sobre el atributo "temp_imp.espec" para obtener la tupla (c, (Γ_1, Γ_2, Γ_3, ..., Γ_n)), para cada especificación c.

Apéndice D

Uso del catálogo de componentes reusables, e instalación del ambiente Algres para una máquina con sistema operativo Unix.

El Catálogo de Componentes Reusables (*ccreusa*) se encuentra en el ambiente de programación Algres. El subdirectorio exacto de la localización de *ccreusa* es /Algres/doc/examples.

Inicialicación del catálogo de componentes reusables

Antes de tener la primera sesión con *ccreusa* se deben inicializar algunos objetos, a través del intérprete de Algres llamando a *ali*. La inicialización se efectua de la siguente manera:

Entrar al subdirectorio /Algres/doc/examples, lo cual se logra mediante el comando

```
% cd /Algres/doc/examples
```

(Ver la observación al final de la siguiente sección)

LLamar al intérprete de Algres

```
% ali
```

el cual responderá con el "prompt"

```
ALGRES>
```

Correr el programa en Algres-prefix *inidt* (inicialización del diccionario de tópicos)

```
ALGRES>run inidt
```

creará el archivo *dt.dat*, para lo cual nos pedira ciertos datos, debiéndose contestar a todo con "enter", excepto a la última pregunta que solo tiene las respuestas: yes o no (y/n), a la cual hay que contestar con y (yes). Se regresará nuevamente al intérprete el cual contestará con el "prompt"

```
ALGRES>
```

Este proceso se debe repetir en forma análoga para los programas *iniple, iniesp, iniesp1, inicon2, inicon3, inicon4*, es decir se deberá seguir la secuencia de comandos

```
ALGRES>run iniple
( contestar con "enter", y con "y" a la última pregunta )

ALGRES>run iniesp
( contestar con "enter", y con "y" a la última pregunta )

ALGRES>run iniesp1
( contestar con "enter", y con "y" a la última pregunta )

ALGRES>run inicon2
( contestar con "enter", y con "y" a la última pregunta )

ALGRES>run inicon3
( contestar con "enter", y con "y" a la última pregunta )

ALGRES>run inicon4
( contestar con "enter", y con "y" a la última pregunta )
```

El objeto que crean en el archivo correspondiente se lista en la tabla siguiente:

Programa	Objeto	Archivo
inidt	diccinario de tópicos	*dt.dat*
iniple	conjunto de implantaciones	*imple.dat*
iniesp	parte del conjunto de especificaciones.	*espe.dat*
iniesp1	parte del conjunto de especificaciones.	*esp1.dat*
iniesp2	parte del conjunto de especificaciones.	*esp2.dat*
iniesp3	parte del conjunto de especificaciones.	*esp3.dat*
iniesp4	parte del conjunto de especificaciones.	*esp4.dat*

La partición del objeto especificaciones se debe a la ineficiencia de una de las interfaces del ambiente Algres, de la versión utilizada, específicamente la interface de captura.

Una vez ejecutados estos programas de inicialización, se debe salir del intérprete mediante el comando "exit" o "quit"

```
ALGRES>exit
```

para regresar a unix, en el subdirectorio /Algres/doc/examples.

Sesiones posteriores a la inicialización

Una vez realizada la inicialización del catálogo, para cualquier sesión de trabajo con el catálogo se deberá única-

mente entrar al subdirectorio /Algres/doc/examples, y ejecutar el comando

% ccreusa

Observación: El proceso de inicialización y el llamado a *ccreusa* se deben relizar activando el modo gráfico de la terminal de trabajo.

Instalación del ambiente Algres para una máquina con sistema Unix

Actualmente se tiene una versión de Algres en la Microvax del I..I.M.A.S. de la U.N.A.M.. Si se quiere llevar a otra máquina con UNIX se debe respetar la organización de subdirectorios establecida. En el subdirectorio /Algres/commands se encuentra el archivo *alcompile*, el cual es un programa en UNIX que recompila el sistema (escrito en C). Por lo que solo hay que ejecutar el comando

% sh alcompile

o también

% chmod +x alcompile

% alcompile

con los posibles errores de una nueva compilación, que deben corregirse si los hay para la versión correspondiente de UNIX.

Los programas:

altra (traductor de algres-prefix al lenguaje de la máquina virtual)

ali (intérprete de Algres-prefix)

alice (traductor de Algres-prefix + C a C)

al terminar la instalación, deben poder ser llamados desde el subdirectorio /Algres/doc/examples.

REFERENCIAS

[AbBi84] Abiteboul, S. y Bidoit, N., "Non First Normal For Relations to Represent Hierarchically Organized Data", Proc. of the 3rd ACM PODS, Waterloo, Ontario, Canada, 1984.

[AtBa90] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S., "The Object-Oriented Database System Manifiesto", Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, de mayo 23 a mayo 25, SIGMOD RECORD, Vol.,19, ISSUE 2, junio, 1990.

[BiGi86] Bic, Lubomir y Jonhatan, Gilbert, "Learning from AI: New Trends in Database Technology", Computer, Vol.19, No. 3, Marzo, 1986.

[BiPe89a] Biggerstaff, Ted y Perlis Alan, *Software Reusability volume I Concepts and Models*, ACM PRESS, Addison-Wesley Publishing Company, New York, 1989.

[BiPe89b] Biggerstaff, Ted y Perlis Alan, *Software Reusability volume I I Concepts and Models*, ACM PRESS, Addison-Wesley Publishing Company, New York, 1989.

[Caca89] Cacae, F., Lamperti, G., *Algres User Manual*, EEC Esprit Project 432 METEOR, MPI, Enero, 1989.

[CaCe90] Cacace, F., Ceri, S., Crespi-Reghizzi, S., Tanca, L., y R. Zicari, "Interacting Object-Oriented Data Modelling with a Rule-Based Programming Paradigm", Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, de mayo 23 a mayo 25, SIGMOD RECORD, Vol.,19, ISSUE 2, junio, 1990.

[Ceri88] Ceri, S., Crespi-Reghizi, S., Gottlob, G., Lamperti, F., Lavazza, L., Tanca, L., Zicari, R., "The ALGRES Project", in Advances in Database Technology -EDBT 88, Lectures Notes in Computer Science, No. 303, 1988.

[Codd70] Codd, E., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, volúmen 13, número 6, junio de 1970.

[DaKu86] Dadam, P., Kuespert, K., Andersen, F., Blanquen, H., Erbe, R., Guenauer, J., Lum, V., Pistor, P., Walch, G., "A DBMS Prototype to Support NF2 Relations: An Integrated View on Flat Tables and Hierarchies", ACM-SIGMOD Conference, Washington, D.C., mayo de 1986.

[Dijk88] Dijkstra, Edsger W. y W.H. Feijen, *A Method of Programming*, Addisson-Wesley, 1988.

[Grie81] Gries, D., *The Science of Programming*, Springer-Verlag, 1989.

[Guch88] Gucht, Dirk Van, Ficher, Patrick C., "Multilevel Nested Relational Structures", Journal of Computer and System Sciences, No. 36, 1988.

[Gutt77] Guttag, John, "Abstract Data Types and the Development of Data Structures", Communications of the ACM, Vol. 20, No. 6, junio de 1977.

[Harp86] Harper, R., D. McQeen y R. Milner, "Standard ML LFCS Report", Series Department of Computer Science, University of Edinburgh, ECS-LFCS-86-2, 1986.

[Hoar89] Hoare, C.A.R. y C.B. Jones (editor), *Essays in Computing*, Prentice Hall International, 1989.

[Kier87] Kiernan, G. y I. Morize, "Le support de Domaines Complexes dans SABRINA: Une Approche par Integration d'un Intérorateur LISP", BDJ, 1987.

[Kimw90] Kim, Won, "Research Directions in Object-Oriented Database Systems", ACM Transactions on Data Base Systems, Vol. 15, No. 4, diciembre, 1990.

[Lampe 89] Lamperti, G., Lavazza, L., Crespi, S., Milani, D., Riva, D., "Interfacing C with a data-base management system for nested relations", Departamento de electrónica, Politécnico de Milán, Italia, 1989.

[Lins89] Lins, C., *The Modula-2 Software Component Library*, Springer Compass International, 1989.

[Lisk75] Liskov, Barbara y Stephen Zilles, "Specification Techniques for Data Abstraccions", IEEE Transactions on software enginnering, vol. SE-1, No.1, marzo 1975.

[Lyon84] Lyon, Michael, "Salvagin your Software Assest(Tools-Based Maintenance)", Proc. Nat'l Computer Conf., AFIPS Press, Reston, Va., 1981.

[Mart86] Martin, Johannes, *Data Types and Data Structures*, Prentice Hall International, 1986.

[McIl76] McIlroy, M., "Mass-Produced Software Components", Software Engineering Concepts and Tecniques, Petrocelli/ Charter, Bruselas, Bélgica, 1976.

[Okta89] Oktaba, Hanna, *Tipos de datos abstractos y estructuras de datos*, comunicaciones técnicas, IIMAS-UNAM, México, 1989.

[OkCe90] Oktaba, H., Pérez de Celis, C., Zicari, R., "Algres Prototype of Reusable Software Modules Based on Algebraic Specifications", International Conference on Database and Expert Systems Applications (DEXA 90), Technical University of Viena, Gußhausstraße 25-27, Vienna, Austria, Agosto 29-31, 1990.

[PrFr87] Prieto, Rubón y Peter Freeman, "Classifying Software for Reusability", IEEE Software, enero 1987.

[ShPi82] Schek, H., Pistor, P., "Data Structures for an Integrated Data Base Management and Information Retrieval System", Proc. VLDB Conf. México, septiembre de 1982.

[PiTr85] Pistor, P. y Traunmüller, R., "A DATA BASE LANGUAJE FOR SETS, LISTS, AND TABLES", Reporte del Heidelberg Scientific Center, Tiergartenstr. 15, D - 69000 Heidelberg, teléfono (0 62 21) 4 04 - 0, Copyright IBM Deutschland GmbH 1985.

[TaMa84] Tajima, Denji y Matsubara Tomoo, "Inside The Japanese Software Industry", Computer, vol 17, No. 3, Marzo 1984.

[Ullm82] Ullman , Jeffrey, *Principles of data base systems*, Computer, Science Press, U.S.A, 1982.

[Wing90] Wing, Jeannette, "A Specifier's Introduccion to Formal Methods", Computer, Vol. 23, No. 9, septiembre, 1990.

[Zica88] Zicari, Roberto, "Extending the current data-base technology: an overview", Reporte del Politécnico de Milán, Departamento de Electrónica, Italia, 1988.