Interfaces Gráficas como Tablas Asociativas

Gabriel Bustos Sölch y Álvaro E. Campos
Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Casilla 306, Santiago 22, Chile
Fax: +56 2 686-4444
[gbustos,acampos]@ing.puc.cl

Resumen

Las herramientas para la generación de interfaces gráficas se dividen en herramientas de bajo y alto nivel. Entre las herramientas de alto nivel se encuentran aquellas basadas en lenguajes, es decir, aquellas en las que una interfaz se especifica como un programa. El objetivo del presente trabajo es establecer un modelo para la especificación de interfaces vía un grafo dirigido acíclico de tablas. Dicho grafo representa la estructura de una interfaz gráfica si se establece la analogía entre una ventana y una tabla.

Sin embargo, una estructura de tablas es sólo una estructura de datos. Para que la interfaz definida por la estructura pueda ser vista y el usuario pueda interactuar con ella, es necesario que se active un monitor sobre las tablas, el cual vigila que la correspondencia tabla-ventana se mantenga mientras se encuentra activa la interfaz.

Palabras Claves: Interfaces humano computador; lenguajes de programación

Introducción

La principal motivación de este artículo es proveer de una herramienta de interfaces de usuario de alto nivel, basada en un lenguaje declarativo, que sea flexible en cuanto a la funcionalidad de la interfaz gráfica que represente. Dicho lenguaje declarativo debe explotar las potencialidades de un lenguaje de programación que acepte:

- Tablas asociativas heterogéneas como tipo de dato incorporado al lenguaje
- Llamadas a funciones
- Funciones como tipos de datos

En la discusión que sigue se presenta definiciones de términos usados en este artículo, para entonces mostrar el esquema conceptual de fondo desarrollado para la representación de interfaces gráficas como un conjunto de tablas asociativas heterogéneas. Luego se presenta las funciones y argumentos que el programador debería dominar, para poder programar con este ambiente.

A continuación se presenta la adaptación que sufre el modelo al ser llevado a un lenguaje concreto como EZ [FRASER1, FRASER2].

Definiciones

La interfaz de usuario (en inglés *UI*, para *User Interface*) de un programa es la parte que maneja la salida a pantalla y las entradas de datos de la persona que ocupa el programa. El resto del programa se denomina aplicación, o semántica de la aplicación.

Las herramientas para la construcción de interfaces de usuario son herramientas (aplicaciones, conjunto de funciones) diseñadas para que el programador no tenga que crear en forma manual cada uno de los elementos que la componen, sino que éstas puedan ser creadas en forma automática a partir de una especificación sencilla. Estas herramientas han recibido distintos nombres en los últimos años, siendo el más popular el de *User Interface Management Systems* (*UIMS*). En el transcurso de este trabajo, sin embargo, se adopta la nomenclatura de Myers [MYERS1, MYERS2], y se las denomina *herramientas de interfaces de usuario (HIU)*.

Esquema Conceptual

Una *tabla asociativa heterogénea* es una tabla con dos campos y una cantidad ilimitada de filas, en las que se almacena todo tipo de datos. En estas tablas el

índice es el primer campo y el dato asociado es el segundo. En el caso del modelo que se describe a continuación, se requiere que el tipo de dato `tabla` pueda almacenar en cualquiera de sus campos datos de tipo `tabla`, números, procedimientos (que se usarán como manejadores de eventos) y strings.

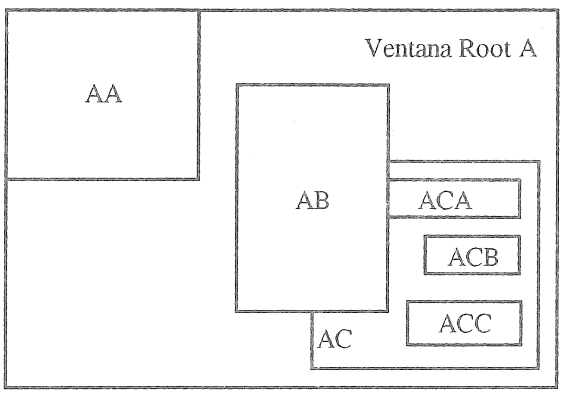

Figura 1: Apariencia de una jerarquía de ventanas.

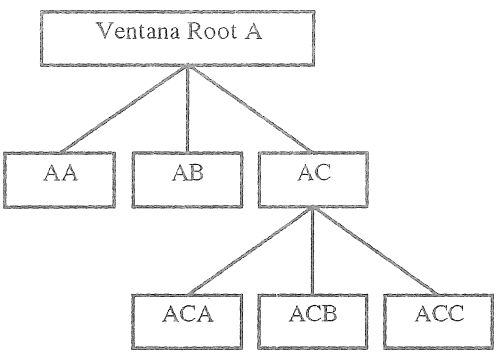

Figura 2: Ventanas vistas como una jerarquía de objetos.

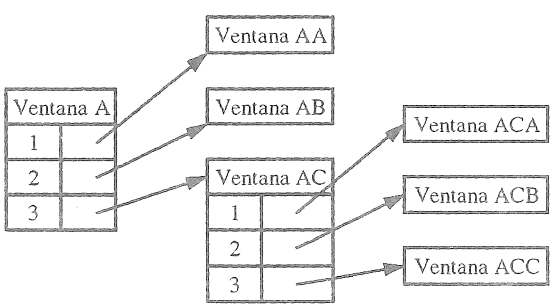

Figura 3: Representación de la jerarquía de ventanas como tablas asociativas.

Por otro lado, una interfaz gráfica se compone de muchas ventanas. La ventana principal alberga un conjunto de ventanas *hijas* que pueden traslaparse y que a su vez pueden tener dentro de sí a otras ventanas. Las ventanas hijas no despliegan fuera de los bordes de la madre y todo lo que se dibuje pixel a pixel dentro de una ventana no puede exceder de los márgenes preestablecidos por ella. Este esquema se presenta mediante un ejemplo en la Figura 1.

Surge entonces en forma natural la idea de representar una interfaz gráfica compuesta por una jerarquía de ventanas como una jerarquía de tablas, si se establece la analogía entre ventana y tabla. Esto se muestra en la Figura 2 y en la Figura 3. La *jerarquía* es formalmente un *grafo dirigido acíclico de tablas*, ya que puede suceder que una tabla que especifica un conjunto de ventanas y dibujos muy usado (Por ejemplo el botón OK) sea hija de varias otras tablas que simbolizan ventanas.

Más aún, una jerarquía completa de ventanas (como la de un botón) también puede ser caracterizada por una sola gran tabla, si esa jerarquía ha de repetirse muchas veces en el desarrollo de una aplicación. Se denomina *widgets* a estas jerarquías prefabricadas [MYERS1, MYERS2] y corresponden a los botones, etiquetas, texto, etc.; como los que se presentan en la Figura 4.

Una jerarquía de tablas asociativas no es más que una estructura de datos, o mejor: un programa declarativo. Para hacer visible la interfaz, es necesario *activarla*. Al activarla, un ente que se denomina *monitor*, verifica que la consistencia entre la estructura de tablas y la de ventanas se mantenga en todo instante. Entonces, todo cambio que cualquier programador realice sobre tablas activadas será reflejado automáticamente en la interfaz gráfica. Si un usuario activa una tabla que representa un botón que tiene como texto "OK", aparecerá dicho botón en pantalla. Pero si luego, mientras está activada la tabla del botón, decide cambiarle el campo texto a la tabla del botón por uno que contenga "Sí", el usuario debería ver dicho cambio, de manera que la tabla siga siendo consistente con lo que se presenta en pantalla.

Para que el programador de la aplicación pueda especificar el diseño y funcionalidad de la interfaz, se provee de un pequeño conjunto de funciones, las cuales se presentan a continuación:

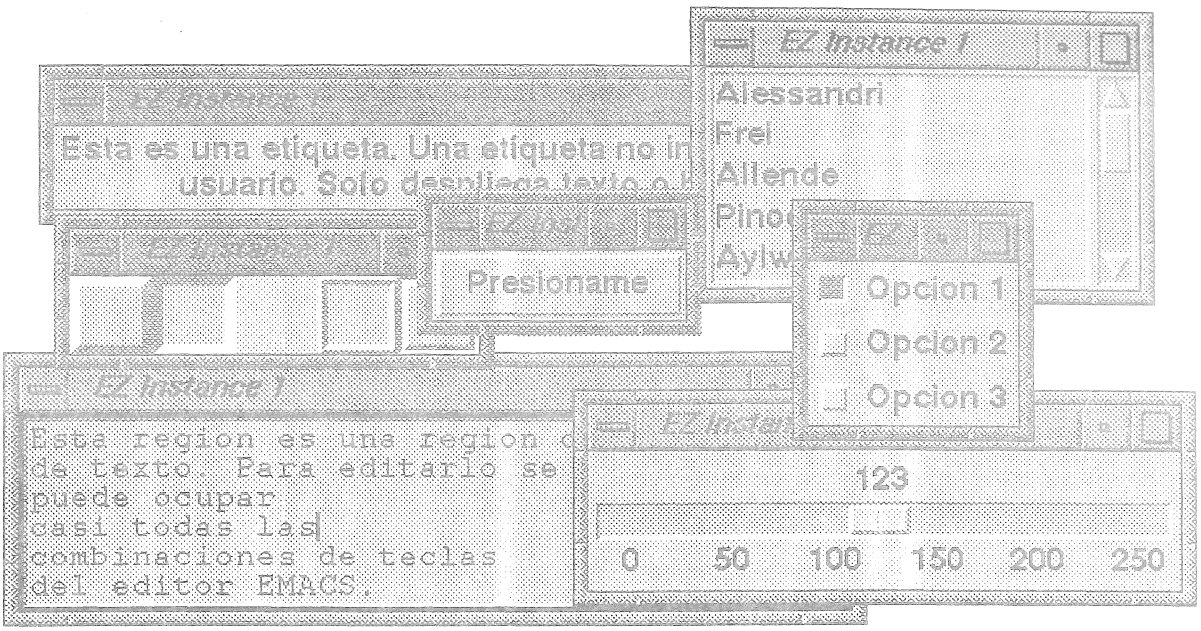

Figura 4: Ejemplos de distintos *widgets* prefabricados. Se presenta una etiqueta, una lista de ítemes, un conjunto de marcos, un botón, un grupo de cajas de check, un editor simple y una escala.

- *tabla*=gui_widget(*nombre del widget*, $nombre_1$, $valor_1$, ..., $nombre_n$, $valor_n$)
- *tabla* = gui_widget(*nombre del widget*, $tabla_1$, ..., $tabla_n$)
- *id* = gui_activate(*tabla*)
- gui_deactivate(*id*)

`gui_widget` genera una tabla asociativa que representa el widget especificado mediante el *nombre del widget*. Si el *nombre del widget* es *vbox* o *hbox*, entonces los argumentos que le siguen serán todos tablas, las cuales representan elementos de interfaz que han de agruparse en forma vertical u horizontal respectivamente. Es importante notar que estas funciones simplemente generan tablas en las que se copian los argumentos entregados en los campos de la tabla. La función `gui_widget` está pensada sólo para facilitar la generación de tablas. Por ejemplo:

```
t=gui_widget("button", "text",
"Ejecutar P", "command", P)
```

es idéntico a hacer:

```
t.widget="button"
t.text="Ejecutar P"
t.command=P
```

en que `P` es un procedimiento.

Análogamente, es posible hacer lo mismo para el caso en que el *nombre del widget* sea *vbox* o *hbox*. Por ejemplo:

```
t=gui_widget("button", "text",
"Ejecutar P", "command", P)
s=gui_widget("button", "text",
"Ejecutar Q", "command", Q)
r=gui_widget("vbox", t, s)
```

es idéntico a hacer:

```
t.widget="button"
t.text="Ejecutar P"
t.command=P

s.widget="button"
s.text="Ejecutar Q"
s.command=Q

r.group="vbox"
r.1=t
r.2=s
```

Nótese como en el caso de ser un widget de agrupamiento, el nombre del campo que indica a su vez el nombre del elemento no es `widget`, sino `group`.

En general se proveen 3 tipos de elementos de interfaz o *widgets*:

1. Elementos estándares, como botones, listas, etiquetas, etc.
2. Elementos de agrupamiento, como `hbox` y `vbox`.
3. Elementos de nivel de abstracción intrínseco.

En el último tipo se tienen los *widgets* que permiten definir nuevos *widgets* a partir de ventanas, texto, dibujos y el manejo de eventos. El modelo que se plantea considera uno de estos widgets y se le denomina *TextSquare*.

Para *activar* la interfaz (ejecutar la interfaz) se introduce la tercera función `gui_activate`. Ella le indica al monitor de interfaces que comience con el monitoreo de la tabla que se entrega como argumento, esperándose que ésta corresponda a una tabla con la estructura típica de las tablas que representan *widgets*[1]. Al activarse una tabla, se activan también sus hijas en la jerarquía de tablas, las cuales representan en sí un *widget* o ventana. También se despliegan las ventanas especificadas en pantalla.

Finalmente, la función `gui_deactivate` desactiva el monitoreo realizado sobre una tabla, eliminando las ventanas que estaban asociadas a ella en el sistema de ventanas.

El Modelo y EZ

EZ es uno de los lenguajes de programación que soporta tablas asociativas heterogéneas y llamadas a funciones. Se escogió principalmente porque además de proveer estas características, también provee un espacio de direccionamiento persistente y distribuido, y un manejo transparente de procesos livianos. Otra razón importante, fue el que su código fuente estuviera públicamente disponible.

EZ, en su versión distribuida [CAMPOS], cumple con cada una de las propiedades de transparencia que se deben tener en cuenta al diseñar un sistema distribuido [TANEN]. Esto es, transparencia en cuanto a:

- ubicación: Los usuarios no requieren saber dónde se almacenan los datos,
- migración: Los recursos pueden ser cambiados de ubicación sin cambiar los nombres,
- replicación: Los usuarios no pueden saber cuantas copias existen,
- concurrencia: Múltiples usuarios pueden compartir recursos en forma automática, y
- paralelismo: Las actividades pueden ocurrir en paralelo sin que los usuarios lo sepan.

En particular, un usuario puede definir una variable en un programa, y en otro servidor puede haber algún otro proceso que lea esa misma variable.

Llevado al ámbito de las interfaces gráficas, esto implica que toda definición de interfaz debe ser compartida por todos los usuarios del ambiente sin importar su ubicación. Surge entonces la necesidad de especificar las interfaces teniendo en cuenta que la especificación es común a todos los usuarios del ambiente. Por ejemplo, uno de los botones más usados en una interfaz gráfica es el de "OK". En un ambiente distribuido como el que se plantea, se especifica ese botón con una tabla una única vez, pudiendo activarla distintos procesos en distintos servidores en la red, sin importar en qué servidor se almacene físicamente.

Por otro lado, existe también una analogía entre la activación de una interfaz gráfica y la ejecución de un proceso EZ [HANSON]: Ambos sólo existen mientras la aplicación se ejecute. Sin embargo, las tablas que representan la interfaz pueden ser ejecutadas por varios usuarios en forma simultánea, así como una activación de un procedimiento puede ser ejecutada por varios usuarios a la vez.

Para mantener la consistencia entre los datos contenidos en la estructura de tablas y la información desplegada, se plantea que cada vez que una tabla esté activa (y, por lo tanto, se muestre su interfaz en pantalla), el monitor debe revisar las direcciones de memoria en la que se encuentran ubicados los datos de dicha tabla, de manera de reflejar cambios en la interfaz gráfica. El monitor además debería también entregar mensajes en la dirección inversa, es decir, debe comunicar a EZ los cambios en la interfaz (como por ejemplo, cambios en el texto de un procesador de texto) para así reflejarlos en la tabla correspondiente.

Implementación

La implementación del sistema que se presenta se realizó en UNIX [THOMPSON] usando Xlib (biblioteca del sistema de ventanas X Windows)

[1] Si no es así, entonces se prevee que exista algún mecanismo de manejo de errores dependiente tanto del lenguaje en el que el modelo está inserto como de su implementación.

[SCHEIFL, NYE1, NYE2] para implementar el widget TextSquare, Tcl/Tk [OUSTER] en los widgets estándares y C [KERNI] como lenguaje de programación. Ésta consistió principalmente en adaptar un ambiente de programación como EZ de manera que soportara interfaces gráficas.

EZ es un sistema distribuido que calza en el esquema del paradigma de cliente-servidor. Se tienen varios computadores conectados en red, en los que se ejecuta una o más instancias del servidor EZ. Este servidor EZ atiende requerimientos de uno o más clientes que se conectan remotamente (es decir desde otro computador) o localmente (es decir, como un proceso más en un sistema operativo multitareas y multiprocesos).

Un cliente es un programa que simplemente transmite el código EZ a ejecutar, a un servidor EZ. Una vez realizado esto, genera un proceso, llamado *Writer*, el cual se encarga de desplegar en la ventana en la que se ejecutó el cliente, todas las salidas a `stdout` que genere la ejecución del programa EZ transmitido. Un esquema simplificado de esto se presenta en la Figura 5.

En un servidor EZ se puede distinguir 5 estructuras básicas que se ejecutan concurrentemente mediante un sistema de *threads* o procesos livianos que comparten el área de direccionamiento del servidor:

- El Compilador que realiza la conversión del código fuente a código intermedio para el intérprete.
- El Listener que está listo para atender nuevos clientes y que recibe de ellos el código fuente a ser ejecutado y las entradas de teclado solicitadas en el programa (rutina `read`).
- El Interprete de un programa que un cliente manda por la red a un determinado servidor.
- El Distributed Virtual Memory Manager (DVMM), que es el encargado de administrar la memoria virtual distribuida.
- El Recolector de Basura, que es el encargado de detectar objetos que no están siendo referenciados por ninguna variable en el ambiente, de manera de liberar esos espacios de memoria.

A estas 5 se agrega una sexta: El monitor de interfaces.

En la Figura 6 se muestra cómo fluye la información a través de los módulos mencionados anteriormente.

El código EZ a ser ejecutado es enviado por el cliente para ser recibido por el Listener dentro del servidor. El Listener luego pasa el código fuente a un compilador, quien a su vez tiene la tarea de generar un código en un lenguaje intermedio que corresponda al programa recibido por el Listener. Este código es guardado en estructuras almacenadas en el espacio de direccionamiento virtual manejado por el DVMM.

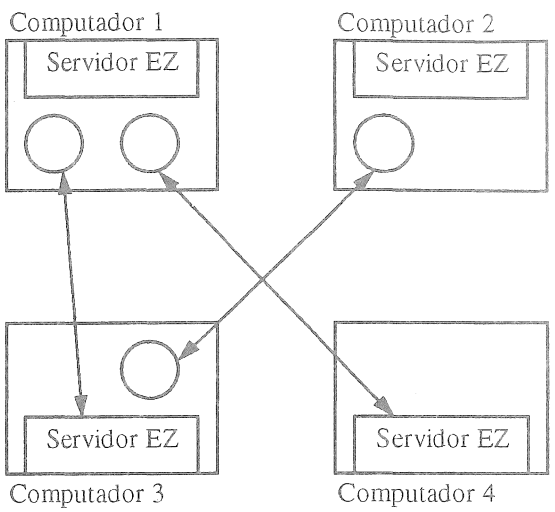

Figura 5: Interconexión entre clientes y servidores en el sistema EZ distribuido.

El código intermedio así generado es entonces interpretado por el Intérprete, el cual lee tanto el código mismo como los datos requeridos por él desde el espacio de direccionamiento de EZ. Claramente, no es posible obviar un fuerte interacción entre el intérprete y el DVMM. También es el intérprete el encargado de solicitar al Listener que espere por entradas de datos que se le pidan al usuario del programa durante su ejecución.

Las salidas a `stdout` generadas por la ejecución del código generado deben ser vistas por el usuario sentado frente al terminal que ejecutó el cliente EZ. Es por eso que el intérprete se conecta cada vez que se ejecuta un `write` (rutina predefinida en EZ) al proceso *Writer*, que el mismo cliente EZ había generado. Este *Writer* escribirá entonces todas las salidas del intérprete en el terminal del usuario.

Por otro lado, el sexto elemento, el monitor de interfaces, es el ente encargado de mantener la coherencia entre el estado de la interfaz gráfica y el estado de las variables EZ ligadas a ella.

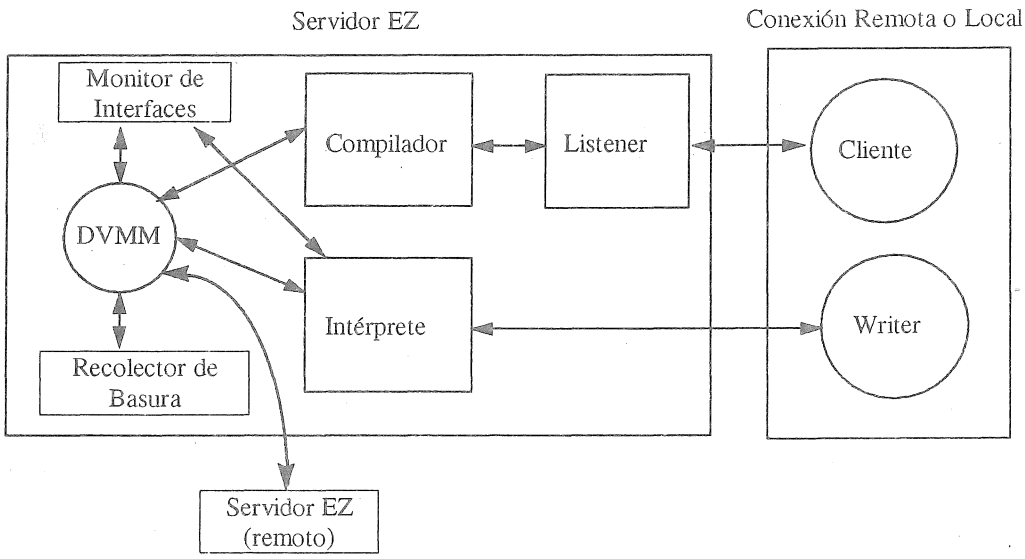

Figura 6 : Estructura interna del servidor EZ con el monitor incorporado.

Para realizar esta labor, el monitor es inicializado con una estructura de tablas conteniendo la descripción de la interfaz dada por el programador de EZ. Cada vez que se active dicha estructura (es posible que una estructura sea activada más de una vez) el monitor de interfaces realiza las siguientes tareas:

- Generar y desplegar la interfaz gráfica correspondiente a las tablas recibidas como parámetro.
- Guardar internamente información vital para la mantención de la coherencia.
- Guardar en la memoria virtual información de las interfaces necesaria para asegurar la persistencia en la activación.
- Revisar periódicamente y en ocasiones estratégicas, posibles incoherencias.
- Actualizar la interfaz y el ambiente EZ para que diferencias entre los dos sean eliminadas.
- Eliminar los registros internos y la interfaz desplegada al momento de desactivar una estructura de tablas o al cerrar la ventana madre asociada.
- Sincronizarse con los otros monitores en la red.

Cada vez que una estructura de tablas es activada, se genera un identificador único (distinto a todos los ya generados y aún activos) para dicha instancia de la tabla. Es responsabilidad del programador el guardar este identificador para así poder posteriormente desactivar la interfaz.

Junto con el identificador, también se genera una *ventana madre* en la pantalla del usuario del tipo *Toplevel*. Ella contiene todos los *widgets* asociados a la interfaz ubicados jerárquicamente bajo la raíz del árbol de tablas. Una ventana *Toplevel* tiene las siguientes características:

- Es una ventana manejada por el manejador de ventanas (Window Manager) y puede ser, por lo tanto, reubicada y redimensionada usando los controles que el manejador provee para ello.
- Es la ventana madre, lo que significa que corresponde a la raíz de la estructura de tablas activada y es dentro de ella donde se ubican todas sus ventanas hijas. La madre de una ventana *Toplevel* es la pantalla misma, es decir, la ventana *root*.

Dentro del contexto de EZ, y tal como se observa en la Figura 6, el monitor interactúa con el DVMM para la obtención de las páginas de memoria que corresponden a las tablas involucradas y para la actualización de los campos de éstas ante cambios en la interfaz. También interactúa con el intérprete, revisando cada una de las direcciones de memoria virtual escritas por él, para detectar cambios en los registros de las tablas justo antes que se produzcan.

Es por lo tanto necesario que el monitor de interfaces guarde internamente una tabla con todas las direcciones de las páginas involucradas con una cierta instancia de interfaz. Esta tabla debe ser de rápido acceso e indexada por direcciones de páginas,

conteniendo para cada una de ellas las instancias que están monitoreando sobre ella.

Para reflejar cambios de la interfaz en el ambiente EZ, el monitor debe interceptar al intérprete justo antes de comenzar la ejecución de una secuencia de operaciones atómicas, de manera de actualizar los datos que éstas pudieran ocupar.

Esta última técnica de interceptar al intérprete tiene sentido, pues al manejar el servidor varios procesos EZ, éstos están continuamente ejecutándose en pequeños bloques de instrucciones (instrucciones generadas por el compilador), para luego dar lugar al siguiente proceso EZ, el cual también sólo es ejecutado en una pequeña porción de su código, etc.

Con esto, es imperioso que el monitor guarde un *checksum* del contenido variable de la interfaz gráfica, de manera de poder comparar de manera eficaz la nueva situación de la interfaz con la antigua. En efecto, si se ha de mantener la consistencia entre la información almacenada en el ambiente EZ y aquella desplegada en pantalla, es necesario que periódicamente se esté revisando que la información desplegada no haya cambiado con respecto a la última oportunidad en la que se revisaron inconsistencias entre ambos ambientes. Dicha revisión debe ser lo más eficiente posible de manera que la frecuencia con que se revisa no deteriore la eficiencia del sistema. Por lo tanto, se guarda el *checksum* para poder comparar rápidamente el estado actual de la interfaz con el anterior.

Otro método para mantener esta consistencia está actualmente en investigación y consiste en que el monitor registre los cambios sobre la interfaz para luego actualizar sólo las tablas pertinentes justo antes de ejecutar las instrucciones atómicas. Éste método eliminaría la posibilidad de que un cambio no fuera detectado por el método del checksum.

También es necesario ejecutar procedimientos EZ cada vez que se active un *widget* que tenga asociado un comando (Ejemplo: Un botón con un procedimiento EZ asociado). La salida de este procedimiento debe ser la misma que la del cliente que envió el código fuente EZ, sin importar si en el intertanto el cliente ha dejado de existir o si aún persiste. Consecuentemente, el monitor también guarda las direcciones de *sockets* [STEVENS] de cada uno de los procesos Writer de manera que la salida pueda llegar efectivamente a su destino. Dichos datos son inherentes a la activación de la interfaz que actualmente se está ejecutando. Es decir, si una tabla se activó siendo el terminal A la salida del proceso EZ, entonces será el terminal A, el indicado para recibir todas las salidas de los manejadores de eventos que se ejecuten por cuenta de la interfaz.

Persistencia y Distributividad

Un desafío importante fue el incorporar la capacidad de que la interfaz fuera tan persistente como los datos y la ejecución de los programas EZ.

Buscando siempre minimizar la carga sobre el intérprete EZ, se buscó la mínima cantidad de variables que era necesario guardar en el ambiente EZ, de manera de poder garantizar la persistencia de la interfaz ante detenciones en la ejecución del servidor. Las variables escogidas:

- El número de la instancia actualmente en ejecución,
- La jerarquía de tablas asociadas a la interfaz, y
- La dirección de los *sockets* que corresponden al Writer que está recibiendo la salida de la ejecución del programa.

Con estos datos almacenados en el espacio de direccionamiento virtual de EZ, es posible que cada monitor, al activar una nueva interfaz, genere un identificador único a partir de los identificadores presentes en la variable `interfaces`. El acceso a ella debe ser, sin embargo, controlado vía un semáforo binario al momento de activar.

El ambiente distribuido, en tanto, acarrea consigo el siguiente problema: Cada vez que se activa una interfaz, el monitor debe guardar en memoria una tabla de acceso rápido con las direcciones de las páginas correspondientes a las tablas activadas. Sin embargo, si existen varios monitores en una red, entonces puede ocurrir que una celda monitoreada por el monitor A, pero no así por el monitor B, sea modificada por B, generando una inconsistencia con respecto a la interfaz desplegada por el monitor A. Para evitar esto, existen dos soluciones posibles:

- Guardar la tabla de direcciones en la memoria virtual de EZ. Esto puede ser muy ineficiente.
- Guardar en todas las tablas de direcciones, presentes en la RAM de cada uno de los monitores, la misma información, aún si una

dirección sólo está siendo monitoreada en un servidor remoto.

Para implementar la segunda alternativa, se hace necesario que un monitor entregue la lista de direcciones correspondientes a la interfaz a todos los servidores EZ activos en la red (broadcast mediante two-phase-commit) *antes* de mostrarla en pantalla. Esta lista debe ir acompañada de la dirección del monitor que está activando y del identificador de interfaz generado para ella.

Así, un monitor remoto B que escriba sobre una dirección correspondiente a una interfaz activa en otro monitor A, le avisa a éste sobre las modificaciones hechas, permitiéndole actualizar su interfaz gráfica con los nuevos contenidos de las tablas.

Un ejemplo

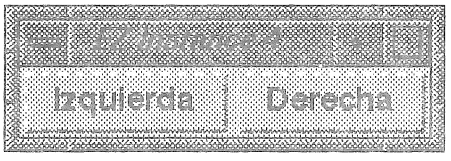

Figura 7 : Dos botones uno al lado del otro

El código para el ejemplo presentado en la Figura 7 es el siguiente:

```
a = gui_widget ("button","text",
"Izquierda")
b.widget = "button"
b.text = "Derecha"
c = gui_widget ("hbox", a, b)
id = gui_activate (c)
```

Si el programador desea que se ejecute un procedimiento al presionar el botón izquierdo, entonces debe reemplazar la primera línea por la siguiente:

```
a=gui_widget ("button", "text",
"Izquierda", "command", P)
```

Conclusiones

El presente artículo muestra una forma de especificar interfaces gráficas como una composición de tablas asociativas heterogéneas, soportadas como un tipo de dato por el lenguaje de programación en el que se desea especificar la interfaz.

Dicha especificación es lo suficientemente poderosa, proveyendo al programador tanto de elementos de interfaz prefabricados (*widgets*) como de la posibilidad de generar sus propios elementos.

La idea central para representar interfaces como una jerarquía de tablas es la de establecer las siguientes analogías:

Una ventana puede ser representada como una tabla. Por lo tanto, una jerarquía o *grafo dirigido acíclico de tablas* equivale a una jerarquía de ventanas, y una jeraquía de ventanas es idéntico a decir: interfaz gráfica.

Una jerarquía de tablas especificando la estructura de ventanas es simplemente un tipo de dato. Si ha de generarse una interfaz a partir de ella, es necesario que se ejecute o active la estructura de tablas, tal como se hace con el código de un programa. Es decir, un conjunto de tablas corresponde a un programa mientras que una activación corresponde a su ejecución.

Entonces, dado que una interfaz es una jerarquía de tablas, se concluye que toda interfaz gráfica basada en ventanas, como las descritas, puede representarse de esta manera.

Por otro lado, el modelo está pensado para adaptarse a lenguajes que soporten (1) tablas, strings, números y procedimientos, y (2) el llamado a funciones. El modelo presentado es entonces una extensión natural a todo lenguaje que cumpla esas características.

Los objetivos primarios fueron satisfechos en cuanto al modelo se refiere. Pero, ¿es este modelo implementable? Una implementación en un lenguaje que reune las características exigidas para permitir la implementación del modelo así lo demuestra.

El prototipo implementado para EZ permite la generación de interfaces basadas tanto en elementos prefabricados (botones, etiquetas, etc.) como en elementos originales (ventanas, texto, imágenes y eventos). Dicho prototipo, además de cumplir con los objetivos primarios ya enunciados, satisface los objetivos secundarios que salen al paso al considerar las características del lenguaje EZ (persistencia, manejo de procesos livianos), y las inherentes a la implementación (memoria distribuida).

El hecho que mucha de la información relativa a la activación de la interfaz se almacene en el espacio de direccionamiento de EZ permite que todos los monitores en la red tengan acceso a ella y permite que se regenere el estado de las interfaces una vez que el sistema es detenido por un comando EZ como `stop`.

La HIU presentada corresponde, en consecuencia, y de acuerdo a Myers, a una herramienta de alto nivel construida sobre herramientas de bajo nivel intrínsecas (Xlib) y de elementos de interfaz (Tcl/Tk), en la que se *especifica* las características de la interfaz usando una estrategia basada en un lenguaje de programación.

Si bien un lenguaje de programación que soporta llamadas a procedimientos no es un lenguaje declarativo (sino que imperativo), se puede decir que la especificación de la interfaz, dentro del ámbito de las cuatro funciones para ello definidas, corresponde a un esquema de lenguaje declarativo. Con esto, se tiene como gran ventaja el que el dieñador de la interfaz no tiene que preocuparse de los tiempos en los que ocurren los eventos y puede así concentrarse en la información que debe ser desplegada y almacenada en las tablas.

Sin embargo, al usar un esquema de lenguaje declarativo, existen algunos tópicos de funcionalidad que se atan al monitor, no permitiéndole al diseñador que intervenga directamente en ellos. Pero este esquema ha probado ser muy existoso para describir la distribución de *widgets*, y también en permitir a programadores concentrarse en las grandes líneas que componen su interfaz, ya que la funcionalidad de aspectos que la mayoría de las veces se implementan de la misma manera (manejo del evento de entrada del puntero del ratón a una ventana por ejemplo) se manejan en forma automática. De hecho, en el presente modelo basta con que se conozca las 4 funciones para que se comience a programar sus interfaces.

Pero eso no es todo: La especificación de interfaces como tablas asociativas puede ser usada como lenguaje intermedio para un generador de interfaces gráficas basado en una especificación gráfico-interactiva. En ese caso basta con que el generador gráfico-interactivo construya la jerarquía de tablas.

Para mostrar la interfaz generada, el generador llamaría finalmente a `gui_activate`. Todo lo referente a la interacción, el manejo de eventos y la mantención de la coherencia funcionaría en forma automática. En resumen: La implementación de un generador gráfico-interactivo es un paso pequeño una vez que se provee de un mecanismo de interpretación de tablas asociativas como interfaces de usuario.

A futuro es necesario continuar el trabajo investigando en cómo incrementar la eficiencia del sistema, sin por ello reducir su flexibilidad, tanto en un sistema distribuido como en un sistema residente en un computador. Esto puede ser llevado a cabo buscando formas de pre-compilar la especificación de una interfaz gráfica, de manera de manipular esa interfaz como un ente cerrado.

Existen otros sistemas, como InterViews [LINTON], que consideran un esquema similar para el caso de clases en C++. Interesante es entonces el analizar otras estructuras de datos que permitan establecer analogías similares a las propuestas para tablas y objetos.

Agradecimientos

Los autores desean agradecer el apoyo de Fondecyt, a través del proyecto 196-0381.

Referencias

[CAMPOS] Alvaro E. Campos, *Distributed, Garbage-Collected, Persistent, Virtual Address Spaces*, Ph.D. thesis, Princeton University, Princeton, NJ, 1993.

[FRASER1] Christopher W. Fraser and David R. Hanson, *A High-Level Programming and Command Language*, in Proceedings of the SIGPLAN'83 Symposium on Programming Language Issues in Software Systems, 18(6), pp. 212-219, 1983.

[FRASER2] Christopher W. Fraser and David R. Hanson, *High-Level Language Facilities for Low-Level Services*, Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages, New Orleans, LA, pp. 217-224, 1985.

[HANSON] David R. Hanson and Makoto Kobayashi, *EZ Processes*, in Proceedings of the International Conference on Computer Languages, New Orleans, LA, pp. 90-96, 1990.

[KERNI] Brian Kernighan and Dennis Ritchie, *El Lenguaje de Programación C*, Prentice Hall, Englewood Cliffs, NJ, segunda edición, 1991.

[LINTON] Mark A. Linton and Paul R. Calder, *The Design and Implementation of InterViews*, The Evolution of C++, The MIT Press, pp. 75-86, 1993.

[MYERS1] Brad A. Myers, *User Interface Software Tools*, ACM Transactions on Human-Computer Interaction, 2(1), pp. 64-103, 1995.

[MYERS2] Brad A. Myers and Mary Beth Rosson, *Survey on User Interface Programming*, in Proceedings SIGCHI'92, Monterrey, CA, pp. 192-202, 1992.

[NYE1] Adrian Nye, *Xlib Programming Manual*, Vol. 1 of *The Definitive Guides to the X Window System*, third edition, O'Reilly & Associates, Sebastopol, CA, 1995. Versión 11 releases 4 y 5.

[NYE2] Adrian Nye, *Xlib Reference Manual*, Vol. 2 of *The Definitive Guides to the X Window System*, third edition, O'Reilly & Associates, Sebastopol, CA, 1995. Versión 11 releases 4 y 5.

[OUSTER] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, New York, NY, 1994.

[SCHEIFL] James Gettys and Robert W. Scheifler, *Xlib - C Language X Interface*, X Consortium Web site http://www.x.org/, 1994.

[STEVENS] W. Richard Stevens, *UNIX Network Programming*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[TANEN] Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, Upper Saddler River, NJ, 1995.

[THOMPSON] Thompson, Ken, *UNIX Implementation*, Bell System Technical Journal 57(6), pp. 1931-1946, 1978