Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXV, pp. 961-975
Alberto Cardona, Norberto Nigro, Victorio Sonzogni, Mario Storti. (Eds.)
Santa Fe, Argentina, Noviembre 2006

MANEJO DINÁMICO DE MEMORIA EN UN PROGRAMA DE ELEMENTOS FINITOS ORIENTADO A OBJETOS

Roberto J. Godoy [a], Martín A. Santa María [b] y Alberto Cardona [c]

[a] *Facultad de Ingeniería y Ciencias Hídricas, Universidad Nacional del Litoral
Ciudad Universitaria, Paraje El Pozo. S3001XAI, Santa Fe, Argentina,
rjgodoy@yahoo.com, http://fich.unl.edu.ar*

[b] *Facultad de Ingeniería y Ciencias Hídricas, Universidad Nacional del Litoral
Ciudad Universitaria, Paraje El Pozo. S3001XAI, Santa Fe, Argentina,
martin_santa_maria@yahoo.com.ar, http://fich.unl.edu.ar*

[c] *Centro Internacional de Métodos Computacionales en Ingeniería.
Güemes 3450 – PTLC, S3000GLN Santa Fe, Argentina,
acardona@intec.unl.edu.ar, http://www.cimec.org.ar*

Palabras Clave: Elementos finitos, manejo de memoria, conteo de referencias, punteros inteligentes, programación orientada a objetos.

Resumen. El manejo de memoria, en programas orientados a objetos con gran cantidad de clases interrelacionadas y un flujo de ejecución complejo, es un problema que no puede abordarse en forma manual. En el presente trabajo se desarrolla una adaptación de la técnica de conteo de referencias para un programa de elementos finitos teniendo en cuenta requerimientos de tiempo de ejecución, así como también la conservación de la legibilidad y mantenibilidad del código fuente preexistente.

Se introduce el problema de manejo de memoria dinámico, mencionando diferentes metodologías existentes y detallando las particularidades de la implementación propuesta. El enfoque combina ventajas de técnicas tradicionales logrando coexistencia con punteros comunes, soporte de tipos incompletos y destrucción explícita.

Para verificar la acción del mecanismo y el costo computacional adicional que implica, se presenta un conjunto de pruebas que muestran que el método funciona, bajo las condiciones para las que fue diseñado, sin agregar costo computacional significativo.

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

1 INTRODUCCIÓN

La implementación se realizó en Oofelie (*Object Oriented Finite Element Led by Interpreter Executor*), un software de elementos finitos orientado a objetos desarrollado en lenguaje C++.

El programa posee un intérprete de comandos desde el cual el usuario puede crear y destruir en forma dinámica objetos de diversos tipos (e.g.: elementos, matrices, dominios). El manejo de memoria tradicional no permite realizar estas operaciones en forma simple, por lo que fue necesario diseñar un mecanismo eficiente para ello. Con este propósito, se realizó una implementación de un conteo de referencias basado en punteros inteligentes, con adaptaciones al problema en particular.

Existen diferentes esquemas para la administración de memoria, cada uno de ellos con distintos grados de complejidad computacional y, por lo tanto, adecuados para distintas situaciones. Considerando que el tiempo de ejecución es un factor crítico en los programas relacionados con el cálculo numérico (A. Cardona, I. Klapka, y P. Devloo, 2001), es necesario elegir un mecanismo que minimice el costo computacional inherente al manejo dinámico de memoria.

Por otra parte, el tamaño del grupo de desarrollo (alrededor de 40 personas), su variabilidad, y la cantidad de código preexistente (aproximadamente 800 unidades de compilación) hacen imprescindible reducir el impacto ocasionado en el código.

2 DESCRIPCIÓN DEL PROGRAMA

En Oofelie existen principalmente dos familias de clases: matemáticas y físicas (Klapka, Cardona, y Géradin, 1998). Las primeras se utilizan para el tratamiento de problemas algebraicos (escalares, vectores, matrices) mientras que las clases físicas (grados de libertad, dominio, análisis, particiones, desplazamientos) están relacionadas con la descripción del problema físico y su relación con las clases matemáticas.

Como se observa en la Figura 1, los objetos físicos poseen propiedades que se representan mediante instancias de la familia de clases físicas, creándose así una estructura de árbol. La clase base para esta familia es `PhySet`, que posee una lista de referencias a otros objetos `PhySet` (Cardona, Klapka y Géradin, 1994). En la Figura 2 se puede observar un esquema parcial de la jerarquía de esta clase.

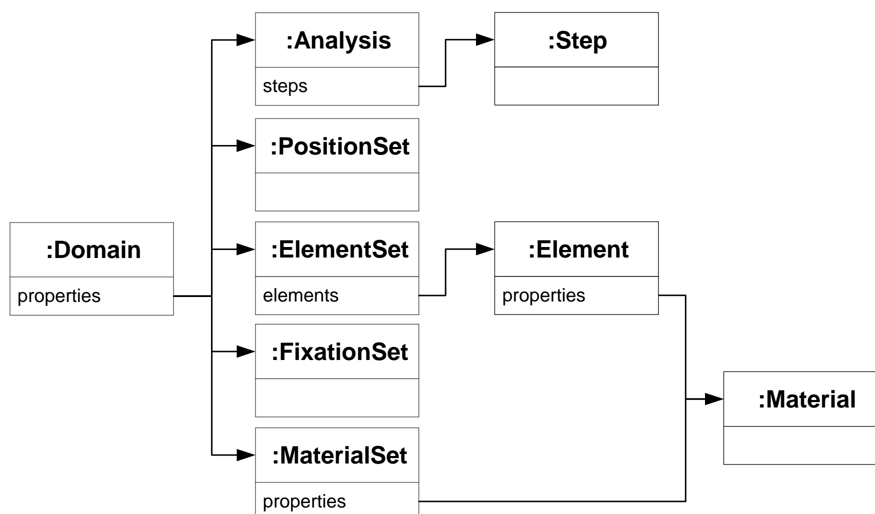

Figura 1: Ejemplo de una estructura de árbol de elementos físicos.

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

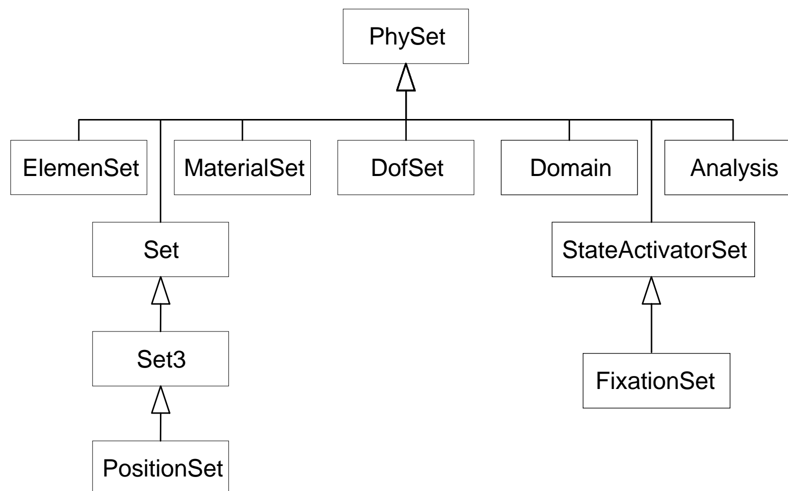

Figura 2: Parte de la jerarquía `PhySet`.

El programa también cuenta con un intérprete de comandos con un lenguaje de alto nivel, que traduce las instrucciones del usuario a operaciones de bajo nivel que son ejecutadas por un módulo administrador de ejecución. Este diseño permite un uso flexible para propósitos generales, pero complica la estructura de referencias a memoria al no ser trivial la determinación del momento en que un objeto deja de ser utilizado y puede reclamarse el espacio de memoria ocupado.

3 ENFOQUES TRADICIONALES DE MANEJO DE MEMORIA

El manejo de memoria consiste en la asignación y liberación de bloques de memoria a medida que un programa lo solicite. Particularmente, la liberación de memoria puede realizarse en forma manual o mediante técnicas automatizadas, también conocidas como *técnicas de recolección de basura*, pues consisten en identificar y liberar los bloques asignados que ya no serán accedidos desde la aplicación (Simsek, 2005). En el lenguaje C++ no existe un mecanismo intrínseco para tal fin, por lo que este debe ser introducido a nivel de aplicación.

Estos métodos automáticos pueden clasificarse en dos grupos: *basados en rastreo* y *basados en conteo de referencias*. Los primeros mantienen un grafo dirigido cuyos nodos son los objetos y sus arcos las referencias entre ellos. Este grafo es periódicamente recorrido para identificar los objetos que no son alcanzables y, por lo tanto, pueden ser eliminados. Por otra parte, como su nombre lo indica, los métodos de conteo de referencia llevan un contador de la cantidad de referencias al objeto y cuando este contador cae a cero el objeto es destruido.

El rastreo de referencias es muy costoso pero resulta aplicable en cualquier situación, mientras que el conteo de referencias es un esquema mucho más simple pero falla cuando existen referencias circulares, porque ningún contador cae a cero. Para solucionar este inconveniente existen técnicas mixtas, de costo intermedio, que utilizan contadores pero mantienen un grafo para detectar referencias circulares.

El conteo de referencias en C++ puede implementarse mediante clases template que imitan, mediante sobrecarga de operadores, el comportamiento de los punteros comunes (e.g.: derreferenciación, asignación) y adicionan lógica de manejo de memoria. Estas clases se conocen como *punteros inteligentes*.

Según la ubicación del contador de referencias, el enfoque de punteros inteligentes puede ser *intrusivo* o *no intrusivo* (Alexandrescu, 2001). En el primer caso el contador es un

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

miembro del objeto, mientras que en el segundo, cada puntero inteligente posee un puntero a un contador común, independiente del objeto (ver Figura 3). Debe notarse que con el enfoque intrusivo es necesario modificar la declaración de las clases apuntadas.

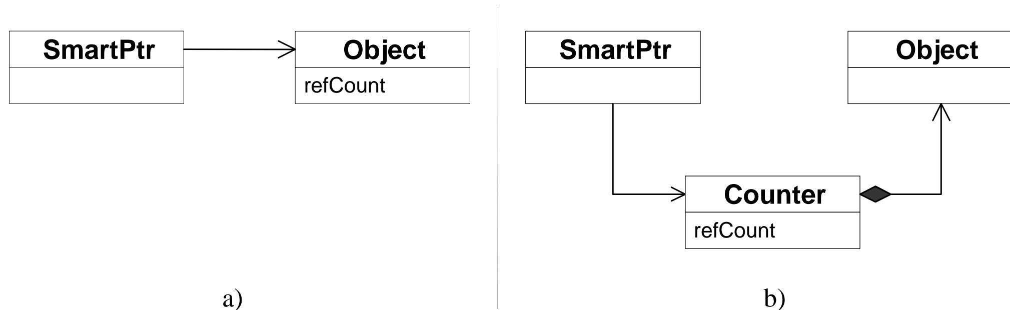

Figura 3: Punteros inteligentes intrusivos (a) y no intrusivos (b).

4 IMPLEMENTACIÓN PROPUESTA

Es deseable minimizar el impacto en la legibilidad y usabilidad del código preexistente, conservando la semántica de las construcciones del lenguaje. Se adopta entonces un enfoque híbrido entre ambas modalidades: el contador de referencias se ubicará fuera del objeto (no intrusivo) pero se modifica la clase apuntada agregándole un puntero al contador (ver Figura 4). Como se explica en lo que resta del trabajo, este esquema presenta ventajas respecto a los enfoques tradicionales, en términos de los requerimientos mencionados.

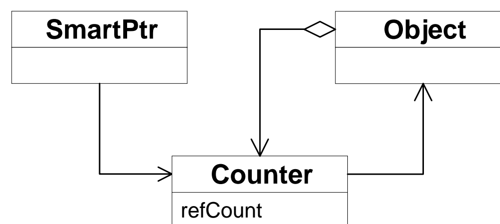

Figura 4: Enfoque propuesto.

Los punteros inteligentes serán implementados por la clase template `SmartPtr`. Puesto que muchas de las clases existentes y la mayor parte de los objetos instanciados forman parte de la jerarquía `PhySet`, inicialmente se introducirá el mecanismo sólo para esta clase y sus descendientes.

Como se puede observar en el apéndice (8.1), la estructura del contador está formada por un puntero común al objeto y un campo entero donde se mantiene el conteo de las referencias. El objeto incorpora un puntero al contador y una firma cuya finalidad se explica en la sección 4.4. El ciclo de vida del contador está manejado conjuntamente por el objeto y los punteros inteligentes; los detalles de esta implementación se explicitan en el apéndice (8.2 y 8.3).

Un puntero inteligente encapsula en forma transparente un puntero común a una dirección de memoria mediante la sobrecarga de los operadores (asignación, derreferenciación y flecha). De este modo, la mayoría de las construcciones que habitualmente se utilizan con punteros comunes también funcionan con punteros inteligentes.

4.1 Costo computacional

Cualquier enfoque de manejo de memoria implica la realización de ciertas operaciones que

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

producen un costo computacional adicional. En particular, en el caso de los punteros inteligentes, este costo se debe a la sobrecarga de los operadores mencionados, que ya no realizan una simple operación, sino que involucran un conjunto de instrucciones con una lógica compleja.

En principio, para que el mecanismo maneje eficazmente la destrucción de objetos sería necesario que todos los punteros a `PhySet` sean de tipo inteligente. Con el objetivo de minimizar el costo computacional incurrido por el mecanismo se considera que el alcance de los punteros puede ser local (intra-procedimiento) o global (inter-procedimiento – por ejemplo, variables miembro). Los del primer tipo generalmente están asociados a operaciones iterativas de lectura y cálculo sobre las que el manejo por punteros inteligentes introduciría operaciones innecesarias.

En base a esto, se propone ampliar el enfoque tradicional permitiendo el uso simultáneo de punteros comunes con ámbito local y punteros inteligentes con ámbito global. Aunque es poco frecuente, existen algunos casos especiales en los que al menos un puntero local debe ser convertido al tipo inteligente para evitar la destrucción del objeto cuando no está siendo referenciado por ningún puntero global.

4.2 Destrucción explícita e implícita

En la programación convencional en C++ las sentencias `delete` son necesarias y su omisión es considerada una mala práctica. Si se utiliza un mecanismo tradicional de punteros inteligentes, la destrucción explícita de un objeto manejado por el mecanismo genera errores en tiempo de ejecución, puesto que el puntero inteligente queda referenciando basura. En tal caso se requiere la omisión total de las sentencias de destrucción, produciendo un cambio drástico en la costumbre de los programadores.

Además, si se cuenta con código preexistente, es necesaria su revisión completa para asegurar que no existan llamadas a la sentencia `delete` para ningún puntero a un objeto manejado por el mecanismo. Determinar automáticamente cuales de los punteros contienen realmente referencias a tales objetos es una tarea dependiente del contexto y de difícil automatización. Se requeriría identificar a los punteros en el ámbito en que fueron declarados, y analizar el flujo de ejecución para asegurar que realmente apuntan a un objeto manejado por un puntero inteligente cuando son destruidos.

Puesto que las destrucciones de punteros existentes no están incorrectamente ubicadas, se adopta un enfoque más sencillo: soportar directamente la destrucción del objeto, haciendo que los punteros inteligentes se auto-anulen cuando su objeto es destruido, esto es, cuando se elimina explícitamente un `PhySet`, su contador será modificado para que apunte a `NULL` en vez de al objeto que está siendo destruido (ver Figura 5). Luego, a medida que los punteros inteligentes que apuntaban al objeto abandonan su ámbito de ejecución, se irán destruyendo y decrementando el contador de referencias, que será destruido cuando el conteo llegue a cero.

Así un objeto puede ser destruido por el programador mediante un `delete` explícito, o automáticamente cuando todos los punteros inteligentes dejen de referenciarlo o salgan de su ámbito.

Al almacenar el contador fuera del objeto, se pueden dar 3 situaciones:

- El conteo de referencias cae a cero: la responsabilidad de eliminar al objeto y al contador recae en el puntero inteligente.
- El conteo de referencias es distinto de cero y se hace un `delete` explícito: el contador no se elimina, pero es anulado en el destructor del objeto.
- Nunca se creó un puntero inteligente y se hace un `delete` explícito del objeto: se elimina el contador en el destructor del objeto.

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

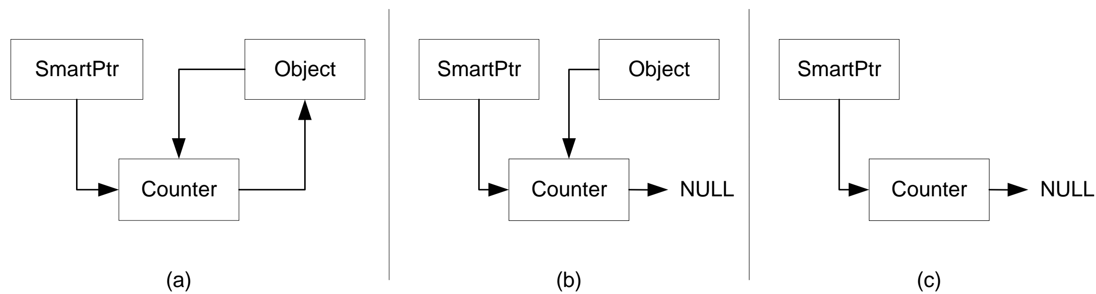

Figura 5: secuencia de auto-anulación por destrucción explícita: (a) estado inicial, (b) anulación del puntero, (c) destrucción del objeto.

4.3 Tipos incompletos

Al momento de declarar un puntero inteligente, es necesario que el tipo apuntado sea completo. Esto no ocurre en todos los casos porque pueden existir declaraciones incompletas de tipos. Para salvar este inconveniente se sumó un parámetro al template, permitiendo especificar una clase completa al momento de instanciar la plantilla. Si este parámetro está presente, debe ser una superclase de la clase apuntada dentro de la jerarquía de `PhySet` (por defecto se utiliza la misma clase apuntada). Esto permite obtener el contador y ejecutar el destructor correcto cuando la clase no es completa.

4.4 Espacio de pila

En la implementación de los punteros inteligentes debe tenerse en cuenta que un puntero a un objeto creado en espacio de pila (i.e.: espacio de memoria estático) no debe destruirse mediante `delete`, sino que el destructor se ejecuta automáticamente al abandonar el ámbito (Stroustrup, 1997). No existe ninguna función estándar en C++ que permita saber si un objeto fue creado en pila o en espacio de memoria dinámico y, por lo tanto, no se pueden determinar con exactitud los casos en que el `delete` es seguro.

Un objeto es creado en memoria dinámica por medio del operador `new`, que se ejecuta inmediatamente antes que el constructor de la clase. Por ello, se adopta la estrategia de sobrecargar este operador para asignar una firma en el objeto (http://www.tarma.com/index.htm#/articles/1997jan.htm). La firma es un campo entero ubicado en el objeto y cuyo valor no es modificado por el constructor. El valor del campo firma es una constante conocida si el objeto fue creado mediante `new`, pero será basura en caso contrario, sabiendo así si el objeto se encuentra en memoria dinámica o no.

4.5 Coexistencia de punteros

En el enfoque no intrusivo tradicional, no pueden utilizarse punteros comunes (Alexandrescu, 2001) porque al asignar uno de ellos a un puntero inteligente se crearía un nuevo contador. Por otro lado, un enfoque intrusivo sí lo permite pero no se puede implementar la auto-anulación cuando se realiza una destrucción explícita.

El enfoque híbrido, con las consideraciones mencionadas, no solo permite el uso simultáneo de punteros comunes e inteligentes, sino también la asignación entre ellos. Por este motivo, no es necesario modificar la signatura de las funciones para recibir parámetros de tipo puntero inteligente, sino que se puede recibir un puntero común y, de ser necesario, asignarlo nuevamente a un puntero inteligente.

Además, para simular el comportamiento de punteros comunes, también se soportan

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

conversiones implícitas dentro de la jerarquía de clases. Esto permite mejorar la legibilidad del código y asignar un puntero inteligente de `Clase1` a un puntero inteligente de `Clase2` si `Clase1` es asignable a `Clase2`. El mecanismo de conversión descansa en la compatibilidad de asignación entre punteros comunes, por lo que las conversiones implícitas entre punteros inteligentes se verifican en tiempo de compilación, evitando una posible conversión incorrecta de tipo cuyas consecuencias se manifestarían durante la ejecución.

5 MODIFICACIONES EN EL CÓDIGO FUENTE

Debido al soporte adecuado de coexistencia de punteros no se requieren grandes cambios en la implementación de los métodos. Las principales modificaciones ocurren en la declaración de variables miembro y contenedores de la clase apuntada (`PhySet` y sus descendientes).

La elevada cantidad de clases que potencialmente habría que modificar, ocasiona que la realización manual de esta tarea sea difícil y propensa a errores. La incorporación de un esquema de manejo de memoria sobre un programa extenso ya escrito se vuelve intratable si no se cuenta con un enfoque automatizable que garantice la correcta aplicación de los cambios en todos los casos.

Además, Oofelie se amplía constantemente agregando nuevas funcionalidades y clases, con el objetivo de satisfacer necesidades concretas. El diseño orientado a objetos y la independencia entre distintos módulos hacen posible que frecuentemente se liberen versiones. Si bien la diferencia entre sucesivas versiones es mínima, al automatizar la aplicación del esquema resulta más sencillo aplicar las modificaciones en versiones recientes, mientras se trabaja en él.

Para automatizar las tareas más repetitivas y tediosas se desarrolló un analizador de sintaxis (*parser*) que examina la estructura del código fuente mediante expresiones regulares y máquinas de estado finito. Los casos con pocas ocurrencias, para los cuales no se justificaba desarrollar un enfoque automatizado, fueron resueltos en forma manual.

Las tareas automatizadas consisten en reconocer la jerarquía de clases de `PhySet`, encontrar los punteros miembros a objetos de estas clases y modificar su declaración. También se automatizó la búsqueda de referencias a la jerarquía en argumentos de template.

Los contenedores templatizados de objetos de la jerarquía de `PhySet` fueron modificados manualmente, reemplazando punteros comunes por punteros inteligentes, logrando así su persistencia hasta la destrucción del contenedor. También se realizó en forma manual la adición del segundo parámetro del template en los casos necesarios.

Surgieron algunas pocas situaciones particulares, entre las cuales puede mencionarse el caso de `dynamic_cast`, en el que no funciona la conversión implícita y es necesario prefijar al puntero inteligente con `&*` para obtener la dirección de memoria del objeto apuntado.

6 PRUEBAS

Se ejecutó la batería de pruebas usada para validar los desarrollos de Oofelie. La mayoría de las pruebas, en lo que concierne a herencia simple, tuvieron éxito ocurriendo pocos errores que no fueron consecuencia del método sino de algunos supuestos implícitos en el diseño del sistema, que tendrán que ser revisados.

Por otro lado, se analizaron en detalle algunas pruebas de la batería para determinar los efectos del enfoque en el programa. Debido a que en la versión actual de Oofelie no están implementados los destructores de todas las variables de interfaz por el problema del manejo de memoria presentado, la mayoría de los objetos quedan referenciados hasta la terminación

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

del programa y en consecuencia la acción del mecanismo no puede apreciarse. Por esta razón, el objetivo de las pruebas realizadas sobre la batería está orientado únicamente a determinar la sobrecarga del mecanismo en el tiempo de ejecución.

A continuación, en 6.1, se muestran los resultados obtenidos de una prueba en C++, desarrollada fuera de Oofelie, para verificar que el mecanismo libera efectivamente la memoria; y luego, en 6.2 y 6.3, se muestran los resultados más representativos de algunos de los casos de prueba de la batería mencionada.

6.1 Prueba de memoria

Se desarrolló una prueba que intenta simular el entorno de ejecución del programa evitando las destrucciones de los objetos apuntados, pero, a diferencia de Oofelie, se deja actuar el mecanismo de manejo de memoria permitiendo que las referencias a los objetos creados se anulen antes de su terminación.

Esta prueba consiste en la ejecución de una función que crea dinámicamente un conjunto de objetos `PhySet` y `FatSet`, e inicializa con ellos un arreglo de punteros. A continuación, se realizan diferentes tipos de asignaciones, elegidas aleatoriamente, sobre posiciones `i`, `j` también aleatorias:

- `ptr[i]=ptr[j]`
- `ptr[i]=new PhySet()`
- `ptr[i]=new FatSet()`
- `ptr[i]=NULL`.

En este caso, el tipo `PhySet` posee una estructura muy parecida, pero simplificada, del tipo introducido en la sección 2. El tipo `FatSet` es un descendiente de este pseudo `PhySet` que incorpora un arreglo de datos de 4KB, y es utilizado para mostrar que el método funciona en casos de polimorfismo.

La función se invoca un número finito de veces en forma recursiva, durante algunas iteraciones determinadas en forma aleatoria, de este modo existirán arreglos en distintos ámbitos. Con el propósito de obtener resultados comparables entre distintas pruebas, se inicializa al generador de números aleatorios con la misma semilla.

Para lograr una resolución aceptable debieron introducirse demoras antes de cada asignación. Esto hace que los tiempos de ejecución de ambas versiones sean prácticamente iguales y no tenga sentido compararlos.

Los resultados obtenidos se muestran en la Figura 6, donde se grafica la evolución de la memoria utilizada por el programa durante la ejecución del algoritmo de prueba. La línea punteada muestra los resultados para un arreglo de punteros comunes, mientras que la línea continua muestra los resultados para un arreglo de tipo puntero inteligente.

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

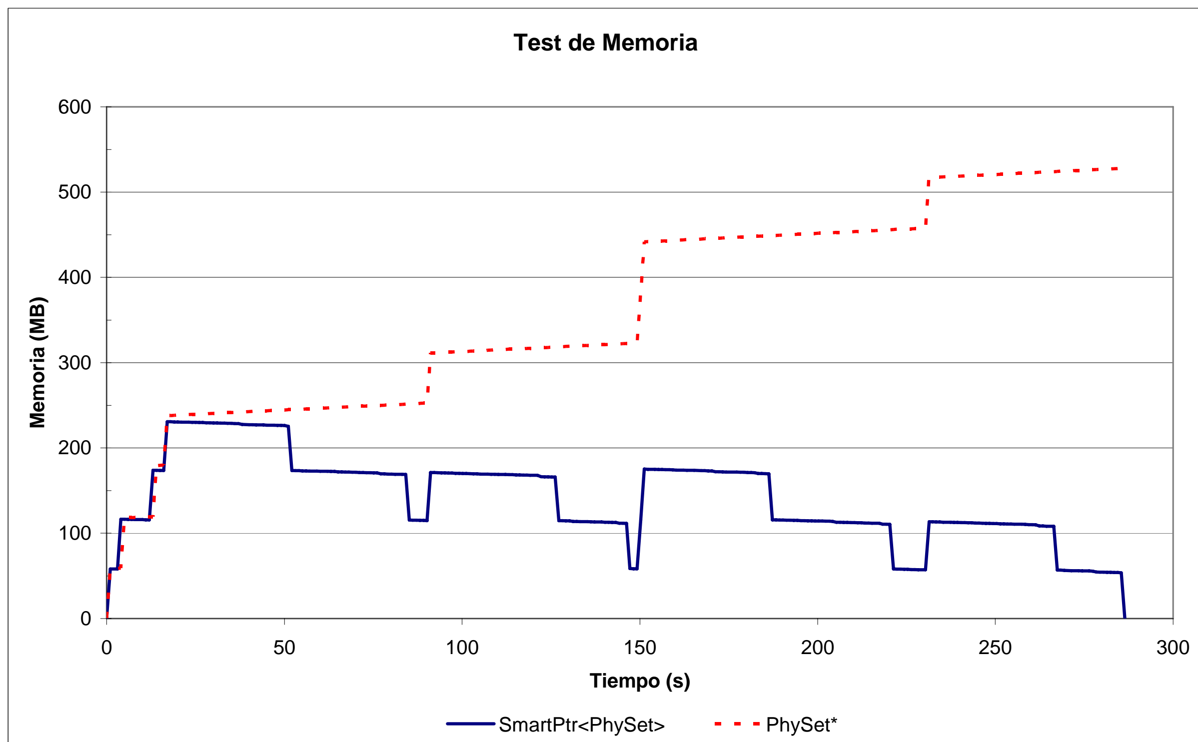

Figura 6: Resultados de la prueba de memoria.

Los escalones ascendentes pueden observarse en ambos casos, y corresponden a la creación de objetos durante la inicialización del arreglo en una llamada a la función. Los escalones descendentes, que solo se verifican para el caso de punteros inteligentes, corresponden a la destrucción de objetos por el mecanismo al abandonar el ámbito de la función.

La pendiente positiva de la línea punteada se debe a la creación de nuevos objetos que no son destruidos luego de su utilización, mientras que la pendiente negativa de la línea continua se debe a que algunas asignaciones de punteros inteligentes ocasionan que un contador caiga a cero y el objeto correspondiente sea destruido. Puede observarse que al terminar la ejecución de la función de prueba, el mecanismo liberó la totalidad de la memoria anteriormente asignada.

6.2 Caso de prueba "StressedBeamPiezoModal"

Este problema consiste en un análisis de vibraciones de una viga con material piezoeléctrico teniendo en cuenta el acoplamiento entre la deformación mecánica y el potencial eléctrico que se genera en el material. El problema cuenta con una malla de 3330 elementos hexaédricos y cuadráticos ejecutándose tres veces, cada una con diferentes parámetros. Los resultados obtenidos se muestran en la Figura 7.

En este caso se puede observar que la versión original realiza destrucciones, aunque no completas, de los objetos creados luego de cada ejecución y, por lo tanto, la versión que implementa el mecanismo de conteo de referencias también lo hace. Aún así, se esperaría que esta última realice liberaciones de memoria en otros instantes de la ejecución. Existen destrucciones por anulación del conteo de referencia, aunque son mínimas, como se muestra en la Figura 8.

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

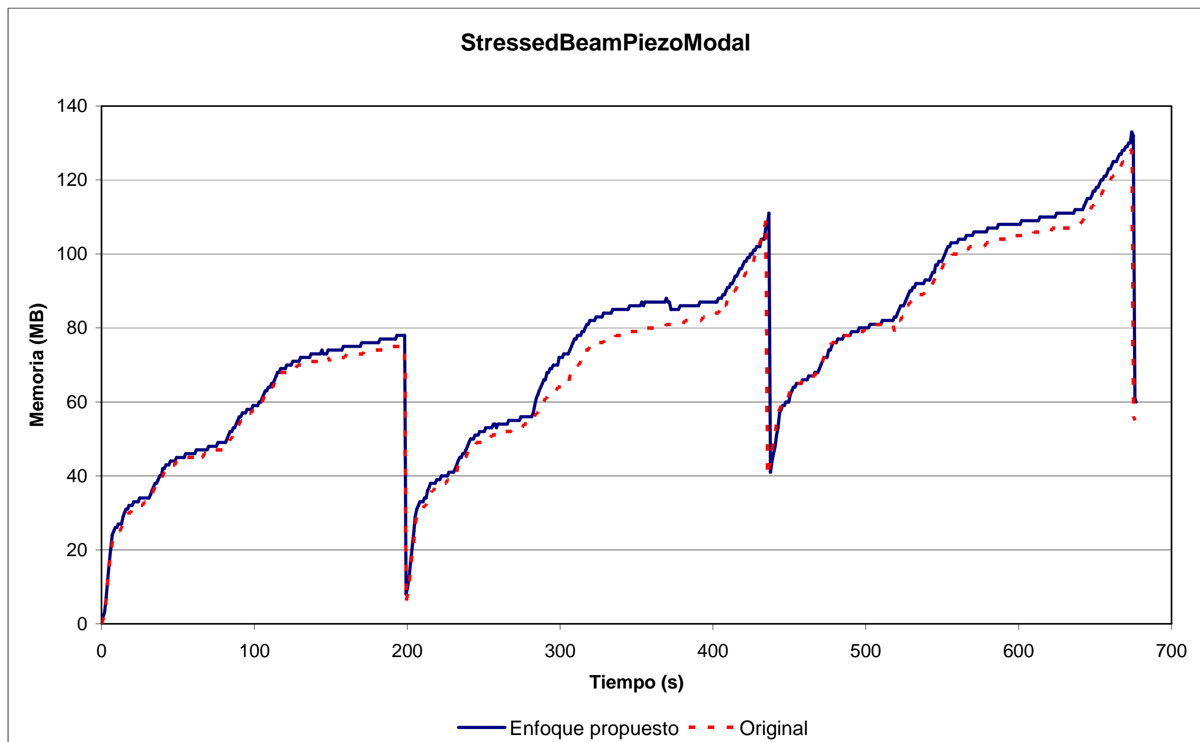

Figura 7: Resultados de la prueba `StressedBeamPiezoModal`.

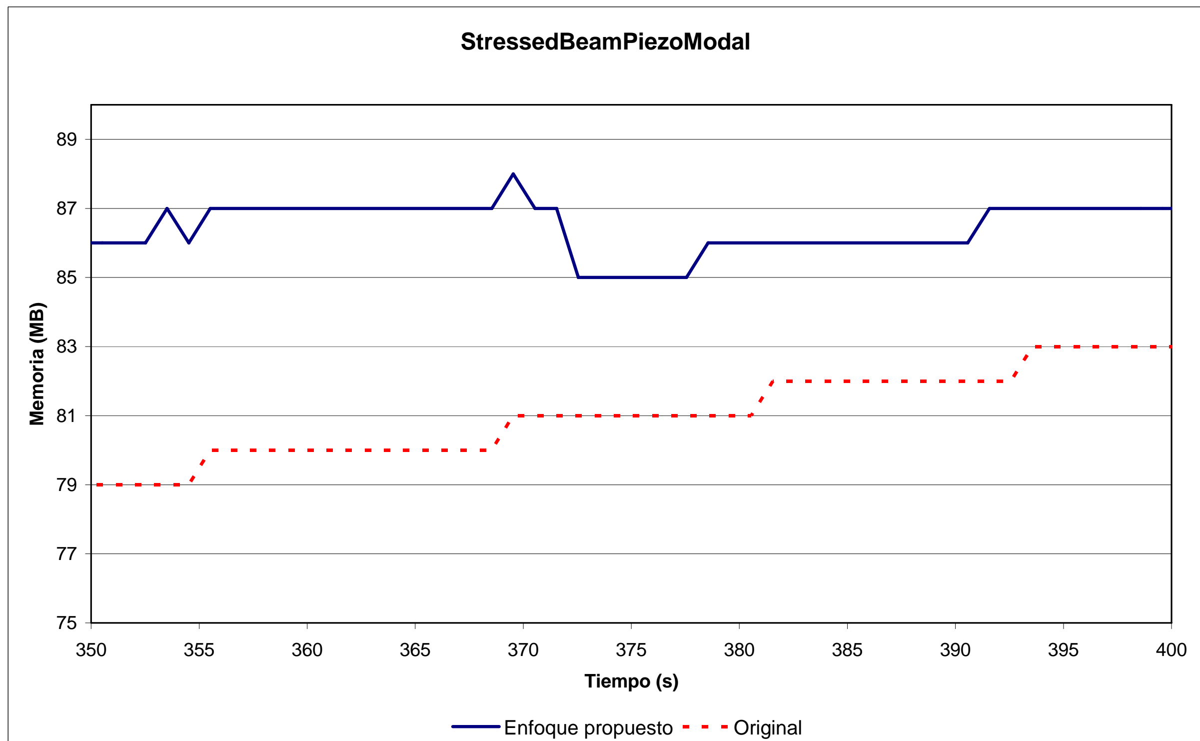

Figura 8: Detalle de la prueba `stressedBeamPiezoModal`, entre 350 y 400 segundos.

Este comportamiento se debe a las características del algoritmo de prueba que crea un conjunto de objetos necesarios para realizar los cálculos y luego realiza las destrucciones observadas. Esta claro que el mecanismo de memoria no tiene posibilidad de eliminar objetos

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

puesto que todos ellos necesitan existir hasta la finalización del algoritmo de prueba. Por otro lado, al realizarse tres cálculos independientes, se debería liberar toda la memoria adquirida al finalizar cada uno. Esto no sucede por a las referencias mencionadas que no son destruidas.

Por otro lado, la sobrecarga producida por el mecanismo en el tiempo de ejecución es despreciable (0.831 segundos – 0.12%), lo cual muestra la potencia del correcto uso de los punteros inteligentes combinados con punteros comunes.

6.3 Caso de prueba "Mptp"

Consiste en un análisis de vibraciones en un motor piezoeléctrico. La discretización consiste en una malla de elementos hexaédricos. Las mediciones de tiempo de ejecución y carga de memoria se muestran en la Figura 9.

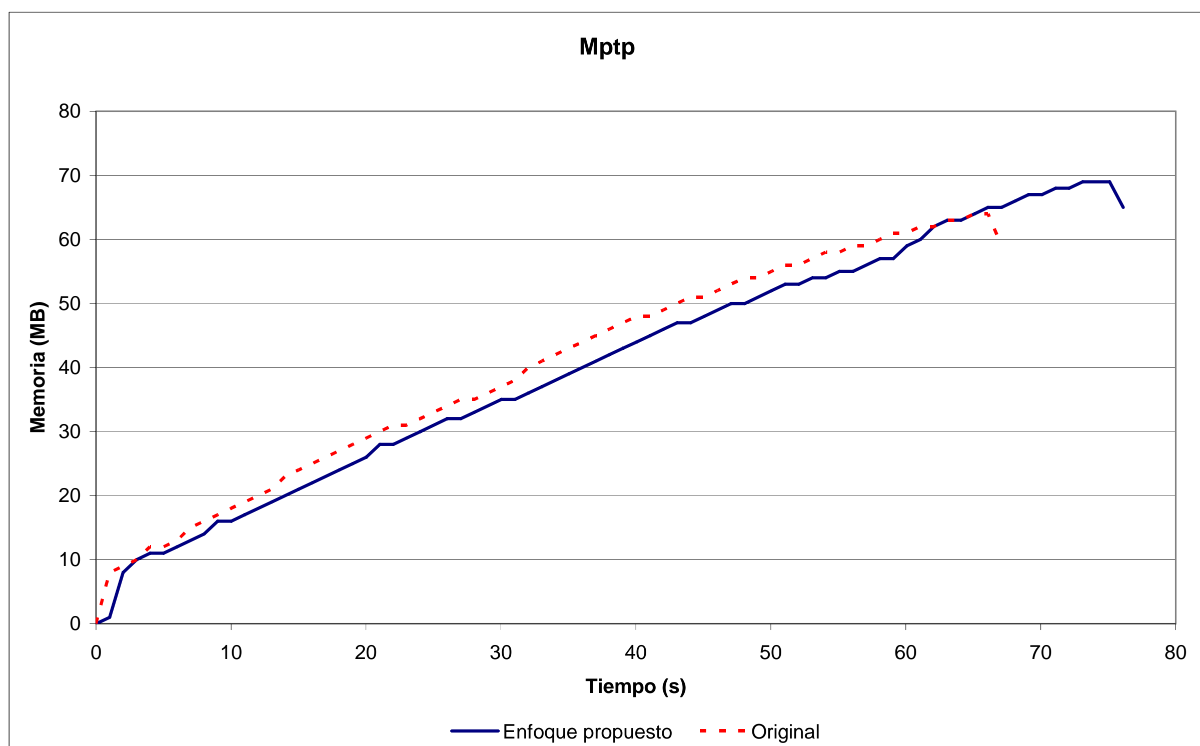

Figura 9: Resultados de la prueba `mptp`.

Se puede observar la sobrecarga del mecanismo tanto en el tiempo de ejecución como en uso de memoria. Esto último se debe, principalmente, a que prevalecen referencias a los objetos desde la interfaz. Estas referencias mantienen el conteo de referencias distinto de cero evitando sus destrucciones.

A diferencia de la prueba anterior, se observa una sobrecarga considerable en el tiempo de ejecución (9.012 segundos – 13.43%). Esto se debe a que la utilización directa de variables miembro en un bucle no representaba una sobrecarga significativa en la versión original, pero con la introducción de los punteros inteligentes se produce un costo computacional adicional importante para las operaciones de asignación (el costo se incrementa en un orden de magnitud) y menor, aunque no despreciable, en las derreferenciaciones (una o dos instrucciones de microprocesador adicionales).

Aunque no se encuentra implementada, la solución a este inconveniente consiste en asignar el puntero inteligente a un puntero común, y utilizar este último en las operaciones locales, especialmente en aquellas que sean iterativas.

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

7 CONCLUSIÓN

Se presentó un esquema de manejo de memoria híbrido basado en punteros inteligentes para un programa de elementos finitos orientado a objetos. El enfoque propuesto demostró ser adecuado para los requerimientos del programa, reuniendo ventajas de diferentes enfoques y nuevas funcionalidades para proporcionar una baja sobrecarga computacional (inherente al mecanismo de manejo de memoria), coexistencia de punteros inteligentes y comunes, eliminación de posibles referencias a objetos destruidos (“*dangling pointers*”) y soporte de tipos incompletos. También se permite la liberación inmediata de memoria, mediante destrucciones explícitas, cuando el programador lo crea necesario.

Si se realiza un manejo tradicional, únicamente basado en `delete`, no siempre es sencillo identificar en que puntos del código deben liberarse los bloques asignados. El mecanismo de punteros inteligentes exime al programador de esta tarea, evitando posibles pérdidas de memoria (por la no destrucción de objetos) y errores en tiempo de ejecución (debidos a destrucciones anticipadas).

Sin embargo, el enfoque falla cuando existen referencias circulares, por lo que, cuando estas referencias sean importantes, deben utilizarse otros mecanismos de manejo de memoria. Para el programa analizado, en particular, tales referencias se conocen a priori y se están efectuando modificaciones que extienden la validez del método en estos casos.

Las funcionalidades agregadas respecto a los enfoques tradicionales hacen que la implementación propuesta requiera mínimas modificaciones en el código fuente y que estas modificaciones resulten automatizables en su mayor parte.

Una buena política de uso de los punteros inteligentes permite que la sobrecarga introducida por el mecanismo sea despreciable. No obstante, en algunos casos los reemplazos automáticos realizados no resultaron en un código fuente óptimo; estas situaciones deberán revisarse manualmente.

Actualmente Oofelie no implementa las destrucciones de objetos desde la interfaz, porque no es posible determinar el momento exacto en que deben eliminarse. El enfoque propuesto permitirá que se incorporen estas destrucciones haciendo un uso más eficiente de la memoria. De este modo será posible ejecutar problemas de mayor tamaño y complejidad.

8 APÉNDICE

8.1 Estructura del contador

```
template <class X> struct Counter {
 public:
   int       refCounter;
   X*        ptr;
   Counter(X* x) : ptr(x), refCounter(0) {}
   Counter(X* x,int ref) : ptr(x), refCounter(ref) {}
};
```

8.2 Template SmartPtr

```
template <class X,class Gnomon=X> class SmartPtr{
 public:
   static const int signature=0x57647652L;
 private:
   Counter<X>* counter;
   static Counter<X>* nil;
```

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
  void acquire(X* obj) {
      acquire((Counter<X>*) ((Gnomon*)obj)->get_counter());
  }

  void acquire(Counter<X>* c) throw(){ // increments the counter
      counter = c;
      (c->refCounter)++;
  }

  void release(){
      if (--counter->refCounter == 0) dealloc();
      counter = nil;
  }

  void dealloc(){
      if (counter->ptr){
          if (((Gnomon*)counter->ptr)->getSignField()==signature)
             delete (Gnomon*) counter->ptr;
      } else {
          if (counter!=nil) delete counter;
      }
  }

 public:

  SmartPtr(X* p = NULL) {
      if(p){  acquire(p);}
      else {  counter = nil;}
  };

  SmartPtr(const SmartPtr& r) {acquire(r.counter);}

  template <class Y> operator SmartPtr<Y>() {
      return SmartPtr<Y>(counter->ptr);
  }

  ~SmartPtr() {release();}

  SmartPtr<X,Gnomon>& operator=(X* obj){
      if (obj==NULL) release();
      else if (this->counter->ptr != obj) {
          release();
          acquire(obj);
      }
      return *this;
  }

  SmartPtr<X,Gnomon>& operator=(const SmartPtr<X>& r)      {
      if (r.counter==nil) release();
      else if (this->counter != r.counter){
          release();
          acquire(r.counter);
      }
      return *this;
  }

  operator X*()   const throw() {return   counter->ptr;}
  X& operator*()  const throw() {return *(counter->ptr);}
  X* operator->() const throw() {return   counter->ptr;}

  void print(std::ostream & outp);
};
```

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
template <class X, class Gnomon>
Counter<X>* SmartPtr<X,Gnomon>::nil= new Counter<X>(NULL,0x80000000);

template <class X, class Gnomon>
void SmartPtr<X,Gnomon>::print(std::ostream & outp) {
  if (counter && counter->ptr)
      outp << (counter->ptr) << ':' << (counter->refCounter);
  else
      std::cout<<"NULL";
}

template <class X, class Gnomon>
std::ostream & operator << (std::ostream & outp,
                            SmartPtr<X,Gnomon> & sp){
  sp.print(outp);
  return outp;
}
```

8.3 Clase PhySet

```
class PhySet : public VirtualSet {
 private:
  int              signfield;
  Counter<PhySet>* counter;

 public:
  inline void * operator new(size_t sz);

  PhySet (PhySet * _pere = NULL);
  PhySet (const PhySet &object);
  virtual ~PhySet();

  Counter<PhySet>* get_counter(){return counter;}
  int getSignField()            {return signfield;}
  ...
}

inline void *PhySet::operator new(size_t sz) {
  void *block    = ::operator new(sz);
  PhySet *ptr    = static_cast<PhySet *>(block);
  ptr->signfield = SmartPtr<PhySet>::signature;
  return block;
}

PhySet::PhySet (PhySet * _pere) {
  counter=new Counter<PhySet>(this);
  ...
}

PhySet::PhySet(const PhySet &object) : VirtualSet(object) {
  counter=new Counter<PhySet>(this);
  ...
}

PhySet::~PhySet() {
  ...
  if (counter->refCounter==0)
      delete counter;
  else
      counter->ptr = NULL;
}
```

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

REFERENCIAS

A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied,* Addison-Wesley, 2001

A. Cardona, I. Klapka, y P. Devloo. Object Oriented Programming in FE Analysis, *Encyclopedia of Vibration,* Ed. S.G. Braun, D.J. Ewins and S.S. Rao, Academic Press Ltd., 2001.

A. Cardona, I. Klapka y M. Géradin. Design of a new finite element programming environment. *Engineering computations*, vol. 11, 365-381, 1994.

I. Klapka, A. Cardona y M. Géradin. An Object-Oriented Implementation of the Finite Element Method for Coupled Problems. *Revue Européene des Eléments Finis*, vol. 7, 469-504, 1998

B. Simsek. *Memory Management for System Programmers*, 2005. Disponible en http://www.enderunix.org/simsek/articles/memory.pdf

B. Stroustrup. *The C+ + Programming Language,* Addison-Wesley, 1997.

Copyright © 2006 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar