XI Jornadas de Ingeniería del Software y Bases de Datos
JISBD 2006
José Riquelme - Pere Botella (Eds)

DISEÑO DE PRIMITIVAS DE REFLEXIÓN ESTRUCTURAL EFICIENTES INTEGRADAS EN SSCLI[i]

José Manuel Redondo[1], Francisco Ortin[2], Juan Manuel Cueva[2]

1: Laboratorio de Tecnologías Orientadas a Objetos
Escuela Universitaria de Ingeniería Informática de Oviedo (EUITIO)
Universidad de Oviedo
Facultad de Ciencias. C/Calvo Sotelo S/N 33007 (Oviedo) (Asturias) e-mail:
redondojose@uniovi.es, web: http://www.ootlab.uniovi.es/

2: Laboratorio de Tecnologías Orientadas a Objetos
Escuela Universitaria de Ingeniería Informática de Oviedo (EUITIO)
Universidad de Oviedo
Facultad de Ciencias. C/Calvo Sotelo S/N 33007 (Oviedo) (Asturias) e-mail:
{ortin, cueva}@lsi.uniovi.es, web: www.di.uniovi.es/{~ortin, ~cueva}

Keywords: Reflexión estructural, máquina virtual, rendimiento, leng. Dinámicos, *JIT*

Abstract. *Los lenguajes dinámicos permiten a los programas una mayor flexibilidad y adaptabilidad en tiempo de ejecución, siendo cada vez más utilizados en el desarrollo de aplicaciones Web entre otros escenarios. No obstante su aplicación a otros contextos suele verse relegada principalmente por sus carencias de rendimiento. La compilación bajo demanda sobre máquinas virtuales es una técnica ampliamente utilizada actualmente, pero no en entornos de procesamiento de lenguajes dinámicos. Hemos usado esta técnica junto a la plataforma .Net para poder ejecutar lenguajes dinámicos de forma nativa, obteniendo buenos resultados de rendimiento que abren una vía para poder aplicar estos lenguajes en nuevos escenarios de ingeniería del software.*

1. INTRODUCCIÓN

Actualmente se ha incrementado el número de aplicaciones comerciales que emplean lenguajes dinámicos en problemas que necesitan un alto nivel de flexibilidad y/o adaptabilidad. Podemos citar aplicaciones concretas de este tipo de lenguajes, como

[i] Este proyecto ha sido parcialmente financiado por *Microsoft Research* (*Extending Rotor with Structural Reflection to support Reflective Languages*, (04-05)), el Ministerio de Ciencia y Tecn. (Programa Nacional para la Investigación, Desarrollo e Innovación, TIN2004-03453) y por la U. de Oviedo (UNOV-06-MB-534).

desarrollo *Web* o soporte para programación orientada a aspectos dinámica (*DAOP*). Esto se debe a que en muchas ocasiones lenguajes "estáticos" como *C++* no son capaces de proporcionar la flexibilidad y capacidad de adaptación necesarias, al carecer de características que la proporcionarían, como reflexión. No obstante, su falta de rendimiento en tiempo de ejecución puede ser un factor negativo a la hora de seleccionarlos como medio de desarrollo, por lo que este trabajo trata de solventar este problema explorando una vía cuyo objetivo es mejorarlo de forma significativa.

Dado que en la actualidad las máquinas virtuales han experimentado un constante avance en términos de rendimiento, gracias a técnicas como compilación bajo demanda (*JIT* [1]) con optimización adaptativa (*HotSpot*) [2], este trabajo muestra el resultado incorporar soporte nativo para primitivas de reflexión de lenguajes dinámicos a estas máquinas, comprobando que es posible lograr una mejora significativa en sus tiempos de ejecución y comparando los resultados obtenidos con los que un lenguaje dinámico proporciona ante las mismas pruebas.

Este artículo describirá aspectos de los lenguajes dinámicos importantes para este trabajo y enunciará las líneas generales del diseño de las modificaciones planteadas, para luego describir como se ha implementado dicho diseño y presentar los resultados y conclusiones obtenidos.

2. LENGUAJES DINÁMICOS

Los lenguajes dinámicos permiten examinar y efectuar modificaciones sobre su estructura, comportamiento y entorno, permitiendo su automodificación y la generación dinámica de código. Su objetivo principal es modelar la dinamicidad que puede necesitarse cuando se diseñan aplicaciones que tienen una gran dependencia de su contexto, debida la movilidad de la propia aplicación (nuevos requisitos, reglas de negocio,…) o de sus usuarios [3]. De esta forma, estos lenguajes permiten el uso de metaprogramación, reflexión, reconfiguración y distribución dinámica y movilidad de una forma más sencilla. No obstante, el empleo de sistemas de tipos dinámicos (que posponen ciertas comprobaciones de tipo a tiempo de ejecución, y pueden hacer que los errores derivados de su uso incorrecto no se detecten al compilar el programa) y su bajo rendimiento (debido al coste adicional impuesto por las operaciones que proporcionan su flexibilidad y una ejecución basada en la interpretación de código) influyen negativamente a la hora de seleccionarlos para desarrollar ciertos proyectos, como aquellos en los que el rendimiento de la aplicación final es más importante que la productividad del programador.

Además, estos lenguajes dinámicos se implementan frecuentemente sobre máquinas virtuales (lo que aporta ventajas como la movilidad, distribución y portabilidad de su código [4]), por lo que, para tratar de mejorar su rendimiento, en este trabajo se empleará una máquina virtual ya existente para añadirle soporte nativo para las primitivas de reflexión poseídas por estos lenguajes. Esta máquina poseerá compilación bajo demanda (*JIT* [1]), lo que le permitirá optimizar el rendimiento en tiempo de ejecución de todos los programas que ejecute, y que por

tanto pueda mejorar el rendimiento de las primitivas implementadas por encima del ofrecido por otros sistemas similares actualmente.

3. DISEÑO DEL MODELO COMPUTACIONAL DE OBJETOS

El sistema modificado permitirá, en una primera fase de desarrollo, añadir, modificar y eliminar miembros (atributos y/o métodos) a cualquier clase u objeto (reflexión estructural), creándolos dinámicamente si fuese necesario. Estas nuevas operaciones deben integrarse perfectamente tanto en los servicios ofrecidos por la máquina como en su modelo computacional, aspecto este último que plantea ciertos problemas. La mayoría de las máquinas utilizables para nuestro proyecto poseen un modelo de orientación a objetos basado en clases, lo que hace que la estructura de todos los objetos deba ser un reflejo fiel de la que posea su clase asociada. Por tanto, si se modifica la estructura de una clase, todas sus instancias deberán ser actualizadas convenientemente para reflejar estos cambios. Este proceso de actualización se conoce en el campo de las *BBDD* como evolución de esquemas, y puede hacerse cuando la clase es modificada o bien cuando se va a usar dicha modificación [5], ofreciendo así una solución a este primer problema. En cambio, si lo que se modifica es sólo una instancia particular, entonces su clase ya no reflejará su estructura, algo que no puede ocurrir al romper también las reglas del modelo. Este mismo problema se trató de solucionar en el desarrollo de la plataforma reflectiva *Metaxa* [6], empleando una técnica también proveniente del campo de las *BBDD* (versionado de esquemas [7]), consistente en generar una clase "sombra" a partir de la original por cada instancia modificada, que sea su nuevo tipo. Esto no es operativo dada la elevada complejidad y pérdida de eficiencia que introduce.

Para solucionar este último problema hemos incorporado al modelo computacional de la máquina virtual original conceptos del modelo de objetos orientado a prototipos empleado por algunos lenguajes dinámicos [8] [9], y que se adapta mejor a las operaciones reflectivas. En este modelo no existe el concepto de clase, por lo que toda modificación que se realice sobre cualquier entidad no violará ninguna norma, al no existir una vinculación estructural entre dos entidades como en el otro. Además, la relación de herencia entre objetos es una asociación dotada de una semántica adicional de delegación jerárquica de mensajes, en contraposición al mecanismo de concatenación usado por lenguajes estáticos. Esto provoca que a la hora de buscar miembros añadidos dinámicamente, el sistema deba hacer una búsqueda menos eficiente que recorra toda la jerarquía del objeto o clase de partida para encontrar la información, pero permite una mayor flexibilidad. Por otra parte, todo el comportamiento compartido por una serie de objetos (sus métodos) puede alojarse en un objeto especial relacionado con ellos, llamado *trait object*, mientras que su estructura común (atributos) estaría contenida en otro, denominado *prototype object*, a partir del cual se generarían instancias por clonación. Por tanto, la modificación de un método en un *trait object* afectaría automáticamente a todos los objetos relacionados con él, produciéndose una evolución de esquemas correctamente.

Dado que se debe mantener el modelo de clases original para mantener la compatibilidad con el código heredado, este se ha ampliado (no sustituido) con los conceptos vistos, de manera que cuando se empleen capacidades reflectivas las clases serán *trait objects* (usando un esquema de actualización perezosa) mientras que los objetos serán tratados como prototipos, permitiéndose pues que agrupen tanto atributos como métodos propios a cada uno de ellos sin necesidad de modificar su clase.

4. IMPLEMENTACIÓN

Para la implementación del sistema se ha elegido la plataforma *.NET* dado su soporte de interoperabilidad de lenguajes, su rendimiento y la existencia de distribuciones de la misma que no establecen limitaciones para su modificación interna. Precisamente una de esas distribuciones *SSCLI* [10] (*Shared Source Common Language Infrastructure*, también llamado *Rotor*), fue la seleccionada principalmente por su similitud con el sistema comercial *CLR* entre otras razones. La implementación de las modificaciones aparece representada en la figura 1, describiéndose sus elementos a continuación.

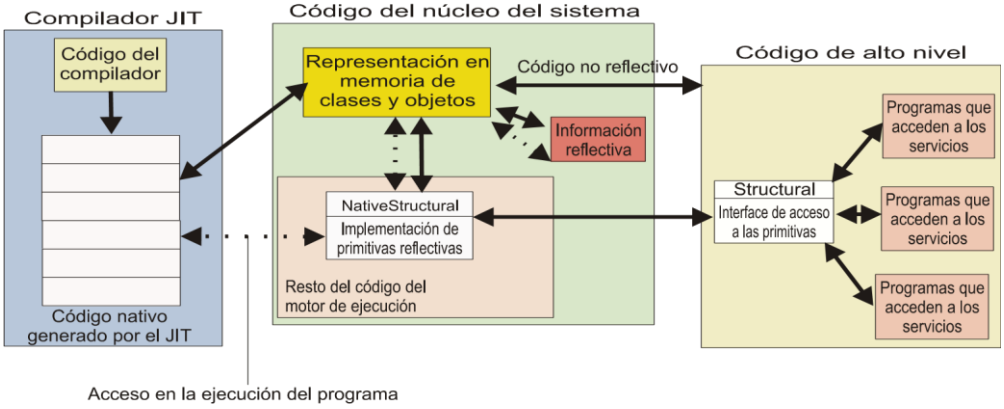

Figura 1. Estructura básica del diseño planteado, con los elementos más importantes

- Todas las primitivas reflectivas y servicios ofertados están implementados dentro del motor de la máquina (en *C*) en una única clase (*NativeStructural*), integrada con el resto del mismo. Esta clase ocultará a los potenciales usuarios sus detalles de implementación y las interacciones con otros objetos internos, utilizando el patrón *Facade*. Mediante un interfaz desarrollado a alto nivel (*Structural*), ubicado en la biblioteca de clases estándar (*BCL*), cualquier lenguaje de la máquina accederá a los servicios anteriores sin necesidad de ser modificado.
- Se ha modificado la semántica de un conjunto determinado de instrucciones del lenguaje intermedio *IL* (acceso a atributos y llamadas a métodos), de manera que el código nativo generado por el compilador *JIT* pueda acceder en tiempo de ejecución (línea de puntos) a los servicios de la clase central, empleando adecuadamente la posible información reflectiva añadida de forma transparente al lenguaje. Las primitivas de la clase central ya están preparadas para operar de acuerdo al modelo computacional modificado descrito, y de esta forma se empleará siempre el mismo conjunto de primitivas para soportar las características reflectivas independientemente del punto de acceso a las mismas.
- La información reflectiva se mantendrá de forma privada a cada clase y objeto, de manera que se pueda acceder a ella solo a través de la clase a la que pertenece, mediante la ampliación de un objeto asociado a toda clase e instancia llamado *SyncBlock*. La clase central podrá así acceder tanto a la información reflectiva como a la no reflectiva a partir de una referencia cualquiera a su propietario.

5. RESULTADOS OBTENIDOS

Para hacer los estudios comparativos de rendimiento se ha comparado nuestro sistema (denominado *RRotor*) con el lenguaje dinámico *Python* [11], principalmente debido a su amplia utilización, robustez y rendimiento. Hemos realizado tres estudios: Rendimiento de las primitivas reflectivas implementadas usando *benchmarks* que usen reflexión, penalización de rendimiento en la que se incurre al incorporar las modificaciones al sistema original y rendimiento del sistema con código que no use reflexión. Se han empleado las últimas versiones disponibles de varias implementaciones de *Python*, como *CPython* y *ActivePython*, ambas desarrolladas en *C*, así como *Jython* e *IronPython*, que generan código para una máquina virtual (*Java* y *.NET* respectivamente) que emula las características dinámicas, aunque se mostrarán solo resultados de las dos primeras, mucho más eficientes. Para el estudio de las primitivas reflectivas se han medido usando los diferentes escenarios que aparecen en la figura 2, usando 10.000 iteraciones en cada caso:

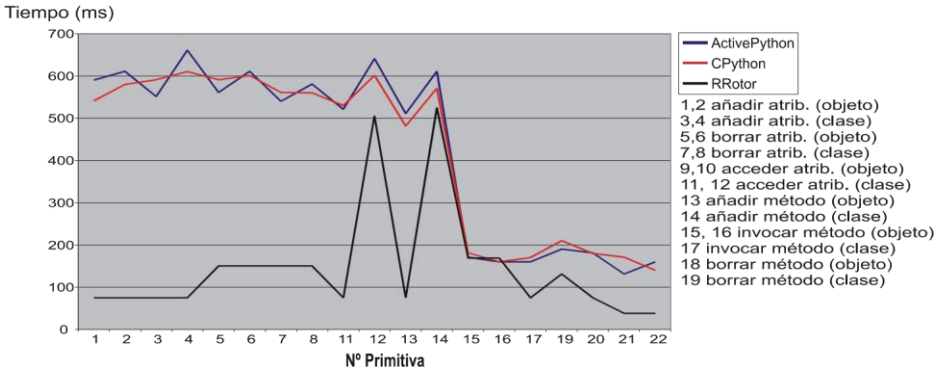

Figura 2. Rendimiento comparado de las primitivas reflectivas en diferentes escenarios

Nuestra modificación de SSCLI es aproximadamente 3,1 veces más rápida que *CPython* y *ActivePython*, aunque debe tenerse en cuenta que aún no se soporta el modelo computacional completo de este lenguaje, al no permitir que un objeto pueda cambiar de tipo dinámicamente. Por otra parte, el coste de incorporar flexibilidad al sistema original se ha medido empleando el *benchmark Tommti* [12], que no emplea operaciones reflectivas. Sólo se ha detectado una pérdida de rendimiento significativa en los accesos a miembros, caso en el cual nuestro sistema es unas 0,6 veces más lento, tanto en el acceso a atributos (instrucción *ldfld*) como en la invocación de métodos (instrucción *call*) de las instancias. Por otra parte, en la ejecución de programas reales en *C#*, se ha obtenido que nuestro sistema es 0,81 veces y 2,14 veces más lento que el original ante programas de *test* como *LCSCBench* (*parser LR*) y *AHCBench* (algoritmo de compresión) respectivamente. Finalmente, para comprobar cuál ha sido la ganancia de rendimiento obtenida por nuestro sistema respecto a lenguajes dinámicos con código no reflectivo, se ha usado el mismo *benchmark Tommti* con *RRotor* y *CPython* (traduciendo su código a este lenguaje), obteniendo que nuestro sistema es 7,41 veces más rápido de media.

7. CONCLUSIONES

Diversas máquinas virtuales, tales como *Java* o *.Net*, utilizan en su implementación técnicas de generación de código bajo demanda y optimización adaptativa, pero no ha habido resultados positivos de su utilización en la ejecución de lenguajes dinámicos. En nuestro trabajo hemos demostrado cómo la compilación *JIT* constituye una técnica adecuada para optimizar las primitivas reflectivas de lenguajes dinámicos, obteniendo un rendimiento unas 7 veces superior cuando se ejecuta código no reflectivo y 3,1 veces cuando el código es reflectivo. El principal cambio que ha producido estos resultados es el haber ampliado la semántica de la máquina abstracta con el modelo computacional orientado a objetos basado en prototipos, dando un soporte adecuado a la reflexión estructural sin perder compatibilidad con el código existente, que también podrá hacer uso de las nuevas funcionalidades.

REFERENCES

[1] J. Aycock. *A Brief History of Just-In-Time*. ACM Computing Surveys, Vol. 35, No. 2, June 2003, pp. 97–113.

[2] Sun Developer Network. *The Java HotSpot Virtual Machine*.

[3] *2nd Workshop on Object-Oriented Lang. Engineering for the Post-Java Era: Back to Dynamicity*. ECOOP 2004 Workshop Reader, Lecture Notes in Computer Science, Vol. 3344 (2004).

[4] F. Ortin. *Sistema Computacional de Programación Flexible Diseñado Sobre Una Máquina Abstracta Reflectiva No Restrictiva*. T. Doctoral. D. de Informática. U. Oviedo (2001)

[5] L. Tan, T. Katayama. Meta operations for type management in object-oriented databases - a lazy mechanism for schema evolution. Proceedings of First International Conference on Deductive and Object-Oriented Databases, DOOD, (1989).

[6] M. Golm, J. Kleinöder. MetaJava - A Platform for Adaptable Operating- System Mechanisms. Lecture Notes in Computer Science 1357 (Springer-Verlag, 1997).

[7] J. Roddick. A Survey of Schema Versioning Issues for Database Systems. Information and Software Technology, 37 (1995).

[8] A.H. Borning. Classes versus Prototypes in Object-Oriented Languages. Proceedings of the ACM/IEEE Fall Joint Computer Conference 36-40 (1986).

[9] D. Ungar, G. Chambers, B.W. Chang and U. Hölzl. Organizing Programs without Classes. Lisp and Symbolic Computation. Kluwer Academic Publishers (1991).

[10] D. Stutz, T. Neward, G. Shilling. SSCLI Essentials. O'Reilly & Associates (2003)

[11] G. Van Rossum, Fred L., Jr. Drake. The Python Language Reference Manual. Network Theory (2003).

[12] T. Bruckschlegel. Microbenchmarking C++, C#, and Java. Dr. Dobb's Portal (2005).