Depurador Declarativo para Plataforma .NET

Autores:
Laura Alemany de Aguinaga,
Víctor Manuel Arias Torrijos,
Alberto Sánchez Carrillo

Profesor director (departamento SIC):
Rafael Caballero Roldán

Curso académico: 2007/2008
Asignatura: Sistemas Informáticos,
Facultad de Informática,
Universidad Complutense de Madrid

Índice

Resumen

Resumen

El proyecto consiste en la construcción de un Depurador Declarativo para la Plataforma .NET de Microsoft.

El Depurador construido es multilenguaje, es decir, acepta aplicaciones implementadas con los lenguajes pertenecientes a la Plataforma .NET, como C#, Visual Basic, C++, Fortran, Haskell, Java, Pascal, Perl, Phyton o Smalltalk.

Realizando una copia de la estructura de la aplicación por medio de la Reflexión, se obtiene el código IL (bytes generados por el compilador de .NET Just-in-time), que es traducido a código MSIL (Microsoft Intemediate Language). Al nuevo Ensamblado se le añade código instrumentado, permitiendo así que al realizar una nueva ejecución se pueda extraer la traza relativa a lo sucedido durante la misma. Con estos datos se genera un árbol de cómputo en el que cada nodo representa la información en tiempo de ejecución de una función del programa.

Además, se ha implementado un sistema de localización de errores, que interactuando con el usuario clasifica los nodos como válidos o inválidos y consigue determinar cuál es la función crítica, mostrando al usuario dónde está el error.

Summary

The project consists of the construction of a Declarative Debugger for the Microsoft´s .NET Platform.

The Debugger built is multilingual, that is to say, accepts applications implemented with the languages belonging to the Platform .NET, like C#, Visual Basic, C++, Fortran, Haskell, Java, Pascal, Perl, Phyton or Smalltalk.

Making a copy of the application´s structure by means of Reflection, you get the IL code (generated bytes from the .NET´s compiler, Just-in-Time) wich is translated into MSIL code (Microsoft Intermediate Language). Orchestrated code is added to the new Assembly, allowing like this that when making a new execution the relative trace to the happened during the same one can be drawn. With this data a computation tree is generated in which each node represents information in execution time about one of the program´s function.

Moreover, an error location system has been implemented, that interacting with the user classifies the nodes as valid or invalid and manages to determine which is the critical function, showing to the user where the mistake is.

1.INTRODUCCIÓN Y OBJETIVOS

1.a) Importancia de la depuración

En los tiempos actuales, un depurador tiene una vital importancia para cualquier persona encargada de implementar software. Cuando un programador o analista se dispone a realizar una implementación, comete errores y para localizar y poder solucionar esos errores se necesita una herramienta potente y especializada.

Cuando realizamos un determinado programa y éste falla ¿A qué se debe este error? La respuesta es que el programador crea un defecto en el código. Cuando el código es ejecutado, el defecto causa una infección en el estado del programa, que más tarde se hace visible como un error.

Pero si hemos realizado un programa de miles y miles de líneas, ¿cómo podemos localizar un error de ejecución? Para poder localizar cualquier tipo de error se han desarrollado entornos de programación que proporcionan cada vez mejores instrumentos y técnicas de localizar fallos: puntos de ruptura, posibilidad de ejecutar paso a paso el programa, inspectores de objeto, etc. Sin embargo, mientras los lenguajes de programación y la ingeniería del software han ido avanzando en los últimos años en el asunto de los depuradores, seguimos usando técnicas que ya se empleaban en los años 70.

El desarrollo de programas software está asociado desde sus orígenes con el problema de la detección de errores. Para este problema se ha intentado definir técnicas y herramientas que automatizaran esta tarea. Las técnicas de depuración de síntomas de errores observados en tiempo de ejecución se basan en la inspección de algún tipo de traza del cómputo. Existen diversos paradigmas de programación (específicamente los paradigmas que incluyen resolución de restricciones y/o evaluación perezosa) que poseen un mecanismo de cómputo tan complejo que prácticamente no resulta viable reproducir los detalles operacionales en la traza empleada para depurar.

Un camino para poder resolver esta dificultad y sobre el cuál se va a orientar nuestro proyecto es la depuración declarativa, donde la traza del cómputo se estructura como un árbol de cómputo que explica el resultado finalmente observado como síntoma de error mostrando dependencias entre los resultados intermedios. La idea clave es que los árboles de cómputo pueden ser analizados sin descender al nivel operacional, realizando consultas a un oráculo que dispone de información sobre el comportamiento esperado del programa.

1.b) Depuración declarativa

Para poder explicar la metodología de nuestro proyecto, hace falta realizar una breve introducción y descripción de la depuración declarativa.

La idea de la depuración declarativa surgió en el contexto de los lenguajes declarativos, donde la falta de herramientas auxiliares como depuradores ha supuesto una limitación para el éxito de estos lenguajes.

La idea básica de la programación declarativa es desarrollar lenguajes de programación en los que los cómputos se correspondan con demostraciones en una cierta lógica asociada. La ventaja de la programación declarativa es que el programador sólo debe describir cuáles son las características del problema, sin tener que preocuparse en detalle de cómo los cómputos son llevados a la práctica. Además, el alto nivel de abstracción de estos lenguajes permite probar formalmente la corrección de los programas y razonar acerca de sus propiedades de forma sencilla. La suma de estas ideas han dado lugar a dos corrientes distintas: la programación lógica y la programación funcional.

La programación lógica representa sus programas como fórmulas lógicas, en particular como *cláusulas*, y emplea mecanismos propios de la lógica matemática para la resolución de objetivos. El lenguaje más conocido de la programación lógica es Prolog.

La programación funcional, en cambio, utiliza una lógica ecuacional y sus programas consisten en un conjunto de funciones. Los objetivos se ven en este paradigma como expresiones a evaluar mediante un mecanismo conocido como reescritura. Los lenguajes funcionales más conocidos son Haskell y ML.

Para dar lugar a lenguajes que tratan de aunar las ventajas de la programación declarativa y de la programación funcional, surgió en los años 80 la programación lógico-funcional. Uno de los primeros lenguajes en esta línea es LOGLISP de J.A. Robinson y E.E. Sibert. Otros lenguajes de este mismo tipo son BABEL, *TOY* y Curry.

El origen de los lenguajes declarativos y su naturaleza provoca que herramientas más tradicionales como la ejecución del programa paso a paso (en los lenguajes imperativos), no puedan ser aplicados de manera sencilla al paradigma de la programación declarativa. Esto es debido al alto nivel de abstracción de los lenguajes declarativos, que produce un gran alejamiento entre el significado del programa y cómo es éste ejecutado en la práctica. Para poder afrontar esta dificultad surgió la depuración declarativa.

La depuración declarativa, introducida por E. Shapiro [Shapiro 82], parte de un comportamiento no esperado del programa, llamado síntoma inicial, y construye un árbol de prueba para el cómputo asociado a dicho síntoma inicial. En este árbol aparece reflejado el comportamiento del programa durante el cómputo y no se tiene en cuenta el orden de ejecución. Después, dicho árbol, es recorrido por el depurador, que va preguntando al usuario por la validez de los resultados de los subcómputos realizados hasta localizar el origen del error. Una correcta construcción del árbol de cómputo nos debe garantizar que cada nodo tenga asociado:

- El resultado de algún subcómputo realizado durante el cómputo principal.

- El fragmento de código del programa depurado responsable de dicho subcómputo.

Por tanto, los nodos hijos se corresponderían con los cómputos auxiliares que han sido necesarios para llegar al resultado almacenado en el nodo padre, teniendo cada uno de ellos a su vez su resultado y fragmento de código asociados. Si se localiza un nodo con un resultado incorrecto, de manera que los resultados de sus hijos sí lo son, tendremos que el fragmento de código asociado a dicho nodo ha producido un resultado incorrecto a partir de resultados auxiliares correctos. Por lo tanto, podremos afirmar que se trata de un fragmento de código erróneo y sería señalado por el depurador como la causa del error.

1.c) Aplicaciones a lenguajes orientados a objetos

Depurador declarativo para JAVA

En este apartado, para ir realizando una pequeña introducción hacia nuestro proyecto, se procede a explicar un proyecto basado en un depurador declarativo para un lenguaje imperativo, en concreto para el lenguaje orientado a objetos JAVA.

Para los creadores de este proyecto, R. Caballero, C. Hermanns y H. Kuchen, los depuradores de lenguajes imperativos como JAVA, tienen la posibilidad de realizar trazas muy sofisticadas pero no llegan a aprovechar todas sus ventajas. Por ello decidieron realizar un depurador declarativo para un lenguaje imperativo como JAVA.

La idea de usar depuradores declarativos en lenguajes imperativos no es nueva. Shahmehri y P. Fritzson presentaron un depurador declarativo para el lenguaje imperativo Pascal. La principal desventaja de este proyecto fue que el árbol se obtenía usando una transformación del programa lo cuál limitó la eficiencia del depurador.

Pero las diferencias respecto a este depurador declarativo para Pascal que han introducido los creadores de esta aplicación son:

Java es un lenguaje mucho más complejo y este depurador incluye la depuración de objetos. Además para la implementación de este depurador declarativo en Java se han basado en Java Platform Debugging Architecture (JPDA). JPDA tiene una arquitectura basada en eventos y usa la entrada y salida de métodos para producir el árbol de cómputo. Este depurador se concentra sólo en la lógica de llamadas de método, almacenándolos de un modo estructurado (en el árbol de cómputo) que permite al depurador deducir el método incorrecto de las respuestas del usuario.

Además, para mejorar la eficacia del depurador declarativo proponen usar un generador de casos de pruebas. Este instrumento divide los valores de entrada posibles de cada método en clases de equivalencia. El comportamiento correcto o incorrecto de una llamada a un método para cualquier representante de una clase de equivalencia implicará la validez o no validez del método para otros miembros de la clase. Por lo

tanto el depurador puede usar esta información para deducir el estado de varios nodos de una sola respuesta de usuario. Esta mejora puede reducir radicalmente el número de nodos considerados durante una sesión de depuración.

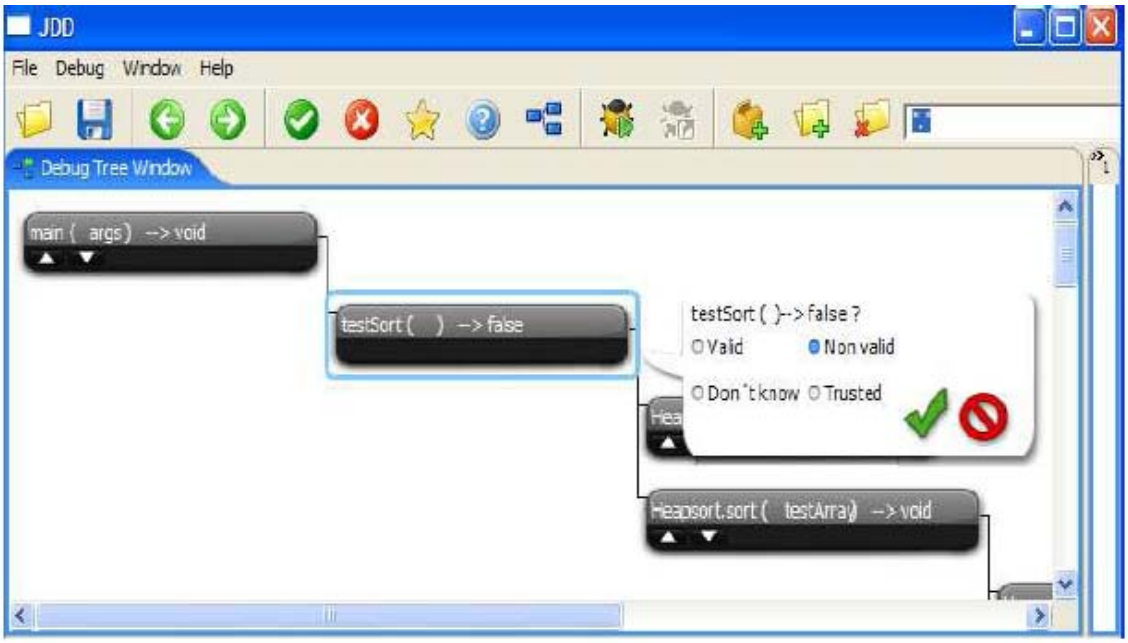
Ya que el objetivo del depurador es descubrir métodos incorrectos en programas Java, cada nodo del árbol contendrá la información sobre alguna llamada a un método que ocurrió durante el cómputo.

Sea N un nodo en el árbol de cómputo que contiene la información de un método f. Entonces los nodos hijos de N corresponderán a las llamadas a métodos ocurridas en la definición de f que ha sido ejecutada durante el cómputo del resultado guardado en N. Esta estructura del árbol de cómputo garantiza que un nodo no válido con hijos válidos, corresponderá a un método incorrecto y que por lo tanto la técnica de eliminación de fallos propuesta es correcta. Sin embargo la comprobación de la validez de un nodo en un árbol de cómputo Java es mucho más complejo que por ejemplo en un lenguaje funcional. Esto es debido a que aparte de devolver resultados, los métodos en los lenguajes orientados a objetos pueden cambiar tanto el objeto llamador como los estados de los parámetros. Toda esta información debe estar disponible al usuario en el momento de la depuración para descubrir la validez de los nodos. Por ello, los creadores de este depurador consideraron que la información almacenada en cada nodo de los árboles de cómputo sea:

- El nombre del método correspondiente a la llamada.
- Los valores de entrada de los parámetros y, en caso de objetos y arrays, los valores de salida si han sido modificados por el método.
- El valor de retorno del método.
- El estado (por ejemplo los atributos) del objeto llamador que contiene el método. Como en el caso de los parámetros tanto la entrada como los estados de salida son necesarios para comprobar la validez del nodo.

Para simplificar la eliminación de fallos, el depurador marca en un color diferente los nombres de los parámetros y atributos cambiados durante la ejecución del método. El inspector de objeto del depurador permite al usuario comprobar los cambios de estado detalladamente.

En la siguiente imagen se muestra el aspecto de la interfaz gráfica que presenta este depurador declarativo. Como se puede observar, a parte de las diferentes opciones que presenta la interfaz, está la opción de seleccionar un método para que nos muestre la diferente información que dicho método almacena.



Una vez que se selecciona un método y se observa el valor de su información, se pregunta al usuario si dicho nodo o método es o no válido.

Como ya hemos comentado en un apartado anterior, en contraste con un depurador convencional, cualquier depurador declarativo tiene la ventaja de que el árbol de cómputo registra la ejecución entera del programa y esta ejecución puede ser investigada el tiempo que el usuario estime oportuno.

Una de las limitaciones que tiene este depurador es que no soporta los hilos (threads) debido a su complejidad y otro problema es la depuración de programas muy grandes o programas que trabajan con estructuras de datos grandes, que requerirán mucha memoria para guardar el árbol de cómputo entero. Aunque dichos autores planifican en el futuro afrontar esta dificultad almacenando el árbol de cómputo en una base de datos.

JAVADD

Otra herramienta de depuración declarativa es JAVADD. Para la creación de este proyecto, sus creadores, Hani Z. Girgis y Bharat Jayaraman se han basado en la investigación anterior realizada sobre el sistema JIVE para JAVA. Dicho sistema soporta los pasos de programación hacia delante y hacia detrás, y además realiza una visualización de estados de ejecución mediante el uso de diagramas de objeto y diagramas de secuencia.

Para ilustrar la depuración declarativa, un aspecto crucial de entendimiento de un programa es conocer como las variables toman valores diferentes durante la ejecución. La manera de imprimir dichos valores es el modo estándar de obtener esta información.

En JAVADD usan el término " depuración declarativa " para referirse a un juego flexible de preguntas sobre estados de ejecución individuales y también sobre la historia

de ejecución de un programa (o parte de la historia) .A diferencia de la herramienta de JAVA explicada en el apartado anterior, estas preguntas que realiza el depurador declarativo son registradas en una base de datos.

Los ejemplos utilizados incluyen preguntas para encontrar todos los valores asignados a una variable sobre su vida (historia de ejecución en el programa); que variable tiene un cierto valor; que resultado devuelve la invocación de un procedimiento; si una cierta declaración fue ejecutada; etc.

Una característica importante es que estas preguntas pueden ser planteadas interactivamente (en cualquier paso de ejecución) mediante una interfaz gráfica de usuario.

En este proyecto proponen dos amplias categorías de preguntas para realizar una depuración declarativa completa:

1) Preguntas sobre estados de ejecución individuales, donde cada estado consiste en el juego de objetos activo e invocaciones de método. Los ejemplos de este tipo incluyen preguntas sobre un objeto o una variable y su valor.

2) Preguntas sobre la historia entera de ejecución de un programa, o un subconjunto de la historia. Los ejemplos de este tipo incluyen preguntas sobre todos los valores asignados a una variable sobre su historia, si una declaración particular alguna vez fue ejecutada o no; etc.

1.d) ¿Qué se pretende hacer en este proyecto?

Tradicionalmente la depuración es un proceso en el cual el programador tiene que continuar la ejecución de un programa paso a paso y objeto-por-objeto para encontrar la causa de un error. Cómo hemos introducido en apartados anteriores, los depuradores de lenguajes imperativos como JAVA, tienen la posibilidad de realizar trazas muy sofisticadas pero no llegan a aprovechar todas sus ventajas. Para facilitar este proceso, en las tres últimas décadas se han desarrollado entornos de programación (Eclipse, NetBeans, Visual Studio) que proporcionan cada vez mejores instrumentos y técnicas: puntos de ruptura, posibilidad de volver algún paso atrás en la ejecución, inspectores de objeto, etc.

La herramienta que pretendemos realizar en nuestro proyecto es un depurador declarativo para un lenguaje orientado a objetos. Este depurador tiene la ventaja de que el árbol de cómputo registra la ejecución entera del programa y esta ejecución puede ser investigada el tiempo que el usuario estime oportuno. El principal objetivo de nuestro depurador es indicarnos la validez de un determinado método comprobando si los hijos de dicho método son o no válidos.

Para ello nuestro proyecto va a consistir en la creación de un árbol de cómputo que refleje el comportamiento de un programa escrito en un determinado lenguaje fuente. Dicho árbol lo representaremos en una interfaz gráfica y estará constituido por

nodos, los cuáles serán métodos que aparte de su nombre tendrán el valor de sus parámetros de entrada (en el caso de tenerlos) y su valor de salida.

Una vez construido el árbol, el usuario se podrá manejar por la interfaz interactuando con los nodos. Siempre que seleccione un determinado nodo del árbol, el depurador le formulará una pregunta indicándole si es válido o no el nodo. El usuario observando los valores que tiene dicho nodo dará validez o no al nodo. Si el nodo es válido se pondrá verde; en caso contrario se pondrá rojo. En el momento en que exista un nodo rojo (no válido) que tenga todos sus hijos verdes (válidos), el depurador indicará que dicho nodo es el causante del error.

La principal ventaja de este depurador es que si queremos averiguar el error de un método no necesitamos depurar paso a paso el programa, ya que la ejecución total del programa queda reflejada en nuestro árbol de cómputo.

La herramienta elegida para la realización del depurador declarativo ha sido la plataforma .NET. Se ha optado por elegir dicha plataforma porque posee numerosas ventajas que en el siguiente apartado vamos a proceder a explicar.

1.e) Elección de la plataforma .NET: código MSIL

En este apartado de la introducción, se va a explicar los motivos que nos llevaron a tomar la decisión de elegir la plataforma .NET. En primer lugar se realizará una completa descripción de los elementos que componen la plataforma .NET y de todas las ventajas que nos proporcionan para nuestro entorno de programación. Posteriormente, se explicará todos los beneficios que nos proporciona en relación a nuestro proyecto.

Microsoft .NET © es una plataforma sencilla de desarrollo para distribuir el software en forma de servicios que puedan ser referenciados remotamente, y puedan comunicarse a su vez con otros de manera totalmente independiente de la plataforma usada por el usuario, así como del lenguaje de programación usado y del modelo de componentes con el que hayan sido desarrollados.

El Common Language Runtime (CLR) es el corazón de la plataforma .NET. Es el encargado de gestionar la ejecución de las aplicaciones que han sido desarrolladas para él y las que ofrece numerosos servicios que simplifican el desarrollo y favorecen la fiabilidad y seguridad. Alguna de sus características más notables son las siguientes:

Modelo de programación consistente: A todos los servicios ofrecidos por el CLR se accede por un modelo de programación orientado a objetos.

Modelo de programación sencillo: Con el CLR desaparecen elementos complejos de manejar incluidos en los sistemas operativos actuales, como por ejemplo el registro de Windows.

Eliminación de las DLL: Las DLL's, es un problema que radica al sustituirse bibliotecas antiguas por nuevas. En la plataforma .NET, las versiones nuevas de las

bibliotecas DLL pueden coexistir con las versiones antiguas lo cuál facilita mucho la tarea de instalación y desinstalación de software.

Ejecución multiplataforma: El CLR actúa como una máquina virtual, encargándose de ejecutar las aplicaciones diseñadas para la plataforma .NET en lenguaje MSIL. Es decir, cualquier plataforma para la que exista una versión del CLR, podrá ejecutar una aplicación diseñada en la plataforma .NET.

Integración de lenguajes: Es posible programar en cualquier lenguaje para el que exista un compilador que genere código para la plataforma .NET, como si estuviéramos programando en el lenguaje nativo de la plataforma .NET. Microsoft, ha desarrollado el lenguaje de última generación C# que genera código de este tipo, así como versiones de sus compiladores para Visual Basic (Visual Basic .NET), C++ o JScript (JScript .NET).

Recoleccción de basura: El CLR incluye un recolector de basura, que evita que el programador tenga que tener en cuenta cuándo es necesario liberar la memoria de los objetos que ya no se utilizan.

Seguridad de tipos: El CLR facilita la detección de errores de programación difíciles de detectar, comprobando que toda conversión o casting de tipos que se realice durante la ejecución de una aplicación .NET se haga de modo que ambos tipos sean compatibles.

Aislamiento de procesos: El CLR asegura que desde cualquier proceso no se pueda acceder a código o datos pertenecientes a otro, lo que garantiza la seguridad en las aplicaciones y evita errores comunes de acceso a datos de otro proceso.

Soporte multihilo: El CLR es capaz de trabajar con aplicaciones divididas en múltiples hilos de ejecución, que pueden ir evolucionando por separado, en paralelo o intercalándose, según el número de procesadores de la máquina donde se ejecute. Las aplicaciones pueden lanzar nuevos hilos, destruirlos, suspenderlos por un tiempo, y demás operaciones útiles para sincronizar los diferentes hilos.

Una vez explicado las ventajas que nos proporciona el CLR, vamos a explicar el código que generan los compiladores para la plataforma .NET, que es el código MSIL.

Los compiladores que generan código para la plataforma .NET no crean código máquina para las diferentes plataformas hardware. Generan código escrito en un lenguaje intermedio denominado Microsoft Intermediate Language. Cuando decimos que un compilador crea código para la plataforma .NET, estamos diciendo que crea código MSIL, que es el único lenguaje que es capaz de interpretar el CLR.

MSIL se trata de un lenguaje de un nivel de abstracción más elevado que los lenguajes de código máquina de las CPUs existentes, ya que permite por ejemplo trabajar directamente con objetos (crearlos, destruirlos, inicializarlos, etc) así como tablas y excepciones (lanzarlas, capturarlas y manejarlas).

El compilador de C#, desarrollado por Microsoft, genera código fuente MSIL, así como que Microsoft ha desarrollado extensiones de sus compiladores de Visual Basic y C++ que generan también código MSIL. Pero no son éstos los únicos lenguajes

que generan código MSIL, sino que son bastantes los lenguajes que han optado por desarrollar una extensión que genere código MSIL para la plataforma .NET, como por ejemplo Fortran, Haskell, java, Pascal, Perl, Python, o Smalltalk. Como se puede observar el número de lenguajes que se puede utilizar es bastante amplio, ya que debido a la arquitectura de la plataforma .NET, podemos programar nuestra aplicación en cualquiera de estos lenguajes, accediendo a todos los servicios ofrecidos por el CLR.

Las principales ventajas del código MSIL es que facilita la ejecución multiplataforma y la integración entre lenguajes al ser independiente de la CPU. Sin embargo, dado que las CPU's no pueden ejecutar directamente código MSIL, es necesario convertirlo antes al código nativo de la CPU en la que vayamos a ejecutar nuestra aplicación. De esto se encarga un componente del CLR, denominado Compilador JIT (Just in Time). Este compilador convierte dinámicamente el código MSIL a código nativo según sea necesario

Una vez explicado de forma completa la arquitectura .NET, ya sabemos que las aplicaciones primero se convierten al lenguaje MSIL para posteriormente convertirlas al lenguaje nativo de la arquitectura hardware donde se ejecuten. Podemos pensar que la actuación del compilador JIT puede disminuir el rendimiento de la aplicación compilada, debido a que debe invertirse un cuantum de tiempo adicional para volver a compilar la aplicación dependiendo de la arquitectura del procesador. Esto es cierto, pero comparándolo con arquitecturas similares, como Java, resulta ser más eficiente ya que cada código no es interpretado y compilado al lenguaje nativo cada vez que se ejecuta, sino que es traducido una única vez; cuando se llama al método al que pertenece. Además, el hecho de que el compilador JIT tenga acceso en tiempo de ejecución a la máquina donde se ejecuta la aplicación hace que posea mucha más información de ella, por lo que puede realizar más optimizaciones que las que podría realizar cualquier compilador tradicional.

La primera condición para desarrollar nuestro depurador declarativo era realizarlo para un lenguaje orientado a objetos. Sólo nos quedaba elegir nuestro entorno de programación y elegimos .NET por todas las ventajas explicadas en este apartado como la ejecución multiplataforma, la eliminación de las DLL, la recolección de basura, la seguridad de tipos , etc.

Pero la principal característica que nos motivó a elegir esta plataforma es que nuestro depurador declarativo es capaz de desarrollar su misión independientemente del lenguaje fuente utilizado (siempre y cuando sea un lenguaje admitido por la plataforma .NET, como C#, Fortran, Haskell, Java,…). Esto es debido a que hemos trabajado directamente con el código MSIL que genera la plataforma al compilar cualquiera de los lenguajes anteriormente citados. Aunque hay que destacar que el lenguaje utilizado para la realización de nuestro depurador ha sido C#.

2.Análisis y diseño

2.a.Casos de uso

Los siguientes casos de uso muestran todas las posibles acciones que pueden suceder a lo largo de una ejecución normal del programa. Ello constituye el esquema básico de funcionamiento de la aplicación.

01.- Mostrar interfaz de usuario

Al iniciarse la aplicación, aparece una ventana que muestra la GUI de usuario donde observamos cinco posibles opciones. Podremos decidir por lo tanto entre:

- Abrir y Generar
- Ayuda Ildasm
- Ejecutar
- Abrir Árbol
- Salir

Para la realización de este objetivo será necesario la utilización del entorno gráfico de C#, incluyendo aquí tanto menús para abrir ficheros, paneles para mostrar árboles, posibles barras de scroll, etc.

02.- Seleccionar opciones en el menú

Mediante una lectura de los periféricos (ratón y/o teclado) se deberá poder seleccionar, en los botones habilitados para ello, “Abrir y Generar”, “Ayuda Ildasm”, ”Ejecutar” y “Abrir Árbol”. También se podrá seleccionar el aspa para “Salir” de la aplicación. Estas acciones se realizarán mediante el uso del ratón, haciendo clic sobre dichos botones, o con el teclado, haciendo uso de la tecla TAB para colocarnos sobre el botón deseado, y después con el uso de tecla INTRO seleccionar la opción deseada.

03.- Salir de la aplicación

Hacer clic sobre el aspa que se encuentra en la parte superior derecha de la ventana de la aplicación implica la terminación del programa.

04.- Abrir y Generar

Al pulsar este botón utilizando el ratón o el teclado, nos aparecerá un explorador de ficheros, donde gracias a una nueva lectura de los periféricos (ratón y/o teclado) se deberá poder seleccionar la ruta y el fichero que deseamos abrir. Dicho fichero será el ejecutable del programa del cuál queremos realizar la depuración declarativa. Al seleccionar el fichero mediante un doble click del ratón o mediante la tecla INTRO, nos aparecerá en un panel de la parte derecha el análisis del código MSIL del programa a depurar. Hay que destacar, que para seleccionar el fichero, éste debe ser

obligatoriamente un fichero en código MSIL, en caso de ser un archivo no correcto, se producirá una excepción inesperada en el programa y este finalizará su ejecución, o bien no hará nada.

05.- Ejecutar

Al seleccionar este botón mediante el teclado o el ratón, procederá a ejecutarse el programa que queremos depurar y el cuál hemos elegido en el caso de uso 04 – Abrir y Generar. Además se creará un fichero .txt que será la representación del árbol de cómputo del programa.

06.- Abrir Árbol de Cómputo

Al seleccionar este botón mediante el teclado o el ratón, se mostrará en el panel central el árbol de ejecución del programa a depurar mediante el cuál realizaremos la depuración declarativa. Dicho árbol se representará a partir del fichero .txt creado en el caso de uso 05- Ejecutar.

07.- Ayuda Ildasm

Al seleccionar este botón mediante el teclado o el ratón, se nos abrirá dos paneles para poder visualizar en uno el código MSIL del programa a depurar y en otro el código MSIL del ejecutable creado por nosotros para poder realizar un fichero .txt que represente el árbol de cómputo.

08.- Expandir nodo

Mediante un simple clic en el icono +, que aparece a la izquierda de cada uno de los nodos que se pueden desplegar, se podrá expandir dicho nodo. A su vez se puede realizar la misma acción interactuando con las teclas de dirección del teclado. Pulsar la tecla DERECHA cuando el nodo contraído deseado esté seleccionado, implica expandir dicho nodo.

09.- Contraer nodo

Mediante un simple clic en el icono -, que aparece a la izquierda de cada uno de los nodos que se encuentren desplegados, se podrá contraer dicho nodo. A su vez se puede realizar la misma acción interactuando con las teclas de dirección del teclado. Pulsar la tecla IZQUIERDA cuando el nodo expandido deseado esté seleccionado, implica contraer dicho nodo.

10.- Asignar validez de un nodo

Mediante un doble clic sobre un nodo, se mostrará un cuadro de diálogo en el que se deberá indicar si es válido o no dicho nodo, o bien cancelar. Mientras se está haciendo la pregunta el nodo tendrá un color distinto al normal cuando simplemente se selecciona. Distintas respuestas derivan en distintos casos:

-En caso de afirmar que el nodo es válido, éste aparecerá en color verde.

-En caso de negar que el nodo es válido, éste aparecerá en color rojo.

-En caso de cancelar, no se le asignará ni estado correcto ni estado erróneo, lo dejará sin asignar, dando igual si antes estaba seleccionado como nodo correcto o incorrecto.

Se seleccione una respuesta u otra, el nodo sobre el que se hacía la pregunta aparecerá después deseleccionado.

2.b.Gestión de Riesgos del proyecto

Definimos riesgo como la probabilidad de que se produzca alguna circunstancia adversa en la elaboración de un proyecto.

En este apartado de la memoria, vamos a describir la gestión de riesgos llevada a cabo durante la realización de nuestro proyecto y detallaremos las siguientes fases seguidas para cada clase de riesgo encontrado:

Identificación de riesgos
– En esta fase, describimos la identificación de cada uno de los riesgos de nuestro proyecto, además de especificar que tipo de riesgos son.

Análisis de riesgos
– En esta etapa de análisis, evaluamos la probabilidad y las consecuencias de cada tipo de riesgo, además de atribuir prioridades a dichos riesgos.

Planificar los riesgos
– La planificación consiste en realizar planes para evitar o minimizar los riesgos que se presenten en el proyecto, es decir cómo vamos a afrontar la aparición de un determinado riesgo. Como se puede observar en el apartado siguiente, para cada aparición de un posible riesgo en nuestro proyecto, hemos definido una estrategia a seguir para poder solucionarlo. En el caso de que la estrategia no sea válida para eliminar el riesgo, hemos definido un plan de contingencia capaz de afrontar dicho riesgo.

Seguimiento de los riesgos
– Una vez completada las tres primeras fases para cada riesgo identificado,hemos realizado un seguimeinto exhaustivo de cada uno de ellos para que no se vuelva a presentar.

Para cada tipo de riesgo, vamos a explicar de una forma esquemática las fases comentadas anteriormente. En nuestro proyecto hemos tenido un gran número de riesgos de todo tipo, pero como es obvio nos centraremos en los riesgos más

importantes y que más tiempo nos ha llevado solucionar. Los riesgos encontrados en nuestro desarrollo de la herramienta han sido: riesgos tecnológicos, riesgos de personal, riesgos de organización y riesgos de estimación.

Riesgos Tecnológicos

Este tipo siempre están relacionados y aparecen cuando se realiza un nuevo proyecto o una ampliación del mismo y se desconoce una determinada herramienta. En nuestro caso han sido:

Información Riesgo Tecnológico 1			
Fecha	Octubre	Estado Inicial	Presente
Probabilidad	Alta	Consecuencias	Serias
Tipo	Predecible	Estado Actual	Eliminado por estrategia
Descripción	Desconocimiento del funcionamiento de un depurador declarativo y de sus fundamentos teóricos		
Estrategia	Buscar todos los miembros del grupo todo tipo de información que nos aporte los conocimientos necesarios para la realización del proyecto		

Información Riesgo Tecnológico 2			
Fecha	Noviembre	Estado Inicial	Presente
Probabilidad	Alta	Consecuencias	Serias
Tipo	Predecible	Estado Actual	Eliminado por estrategia
Descripción	Desconocimiento del funcionamiento de la herramienta utilizada para la realización de nuestro depurador, el Visual Estudio.NET 2005		
Estrategia	Lectura y seguimiento de un tutorial para familiarizarnos con el entorno de la herramienta		
Plan de contingencia	Buscar otra herramienta de programación para realizar nuestro proyecto		

Información Riesgo Tecnológico 3			
Fecha	Noviembre- Diciembre	Estado Inicial	Presente
Probabilidad	Alta	Consecuencias	Serias
Tipo	Predecible	Estado Actual	Eliminado por estrategia
Descripción	Desconocimiento del funcionamiento del lenguaje intermedio de microsof MSIL y de los comandos utilizados		
Estrategia	Como en los riesgos tecnológicos anteriores, la estrategia a seguir ha sido utilizar varios tutoriales (algunos online) acerca de este tipo de lenguaje		

Información Riesgo Tecnológico 4			
Fecha	Enero	Estado Inicial	Presente
Probabilidad	Alta	Consecuencias	Serias
Tipo	Predecible	Estado Actual	Eliminado por estrategia
Descripción	Desconocimiento del funcionamiento del programa Ildasm y reflector, utilizados para mostrar el código MSIL que genera cualquier ejecutable		
Estrategia	Como en los riesgos tecnológicos anteriores, la estrategia a seguir ha sido utilizar toda la información disponible para poder utilizar dichos programas		
Plan de contingencia	Buscar otro tipo de herramienta para poder obtener el código MSIL de un ejecutable		

Como hemos dicho anteriormente, hemos definido los riesgos tecnológicos más importantes y que más tiempo nos ha llevado eliminar.

Riesgos de personal

Este tipo de riesgos están relacionados con las personas que componen el grupo de proyecto. El principal riesgo que hemos tenido en la realización del depurador ha sido la incompatibilidad horaria de los tres miembros del grupo junto con la del profesor para poder realizar las reuniones semanales.

Información Riesgo De Personal 1			
Fecha	Octubre-Noviembre	Estado Inicial	Presente
Probabilidad	Alta	Consecuencias	Serias
Tipo	Predecible	Estado Actual	Eliminado por estrategia
Descripción	Incompatibilidad horaria de los componentes del grupo de proyecto en el horario del primer cuatrimestre.		
Estrategia	Búsqueda de un horario y día libre común para los tres miembros del grupo y para el profesor		
Plan de contingencia	Trabajo en subgrupos o trabajo online		

Información Riesgo De Personal 2			
Fecha	Febrero-Marzo	Estado Inicial	Presente
Probabilidad	Alta	Consecuencias	Serias
Tipo	Predecible	Estado Actual	Eliminado por estrategia
Descripción	Incompatibilidad horaria de los componentes del grupo de proyecto en el horario del segundo cuatrimestre.		
Estrategia	Búsqueda de un horario y día libre común para los tres miembros del grupo y para el profesor		
Plan de contingencia	Trabajo en subgrupos o trabajo online		

Riesgos de Organización

Este tipo de riesgos se presentan principalmente cuando está construida la arquitectura de la aplicación y se empieza a implementar dicha arquitectura mediante código de programación. Por lo tanto este riesgo nos ha aparecido a partir de Diciembre que es cuando hemos empezado a implementar el código.

Información Riesgo De Organización 1			
Fecha	Diciembre	Estado Inicial	Presente
Probabilidad	Alta	Consecuencias	Serias
Tipo	Predecible	Estado Actual	Eliminado por estrategia
Descripción	Organización del grupo del proyecto para agrupar el código generado por cada miembro del proyecto		
Estrategia	Debido a que generalmente cada miembro del grupo ha trabajado en un módulo diferente del proyecto, para la organización hemos utilizado un foro y un grupo de trabajo para explicar y actualizar cada una de las versiones modificadas		
Plan de contingencia	En el supuesto caso de que nuestra estrategia no hubiera erradicado el riesgo, el profesor del proyecto nos habría proporcionado e instalado un controlador de versiones para una mejor organización		

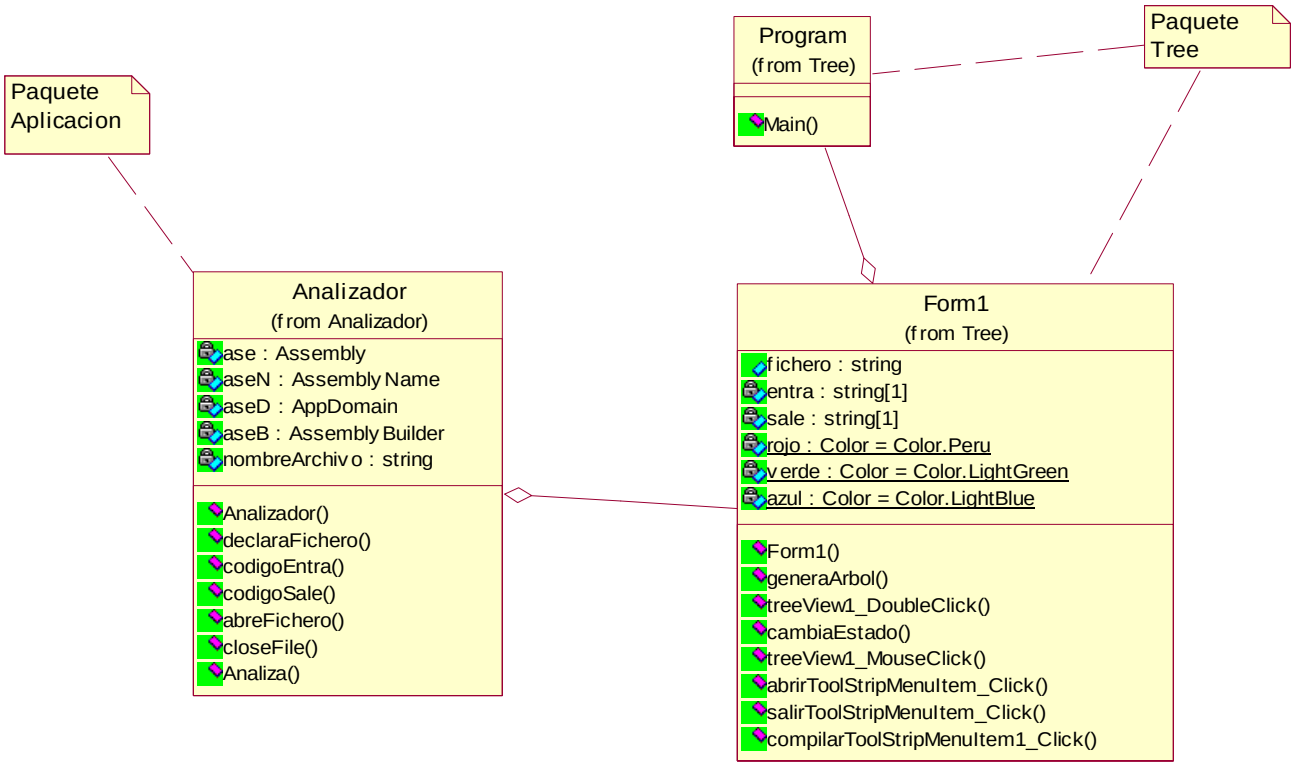
Riesgos de Estimación

Este tipo de riesgos aparecen cuando inicialmente, al empezar el proyecto, el grupo se fija una serie de objetivos a realizar. En un principio, nuestro principal objetivo era realizar un depurador declarativo para un lenguaje orientado a objetos. Pero debido al tiempo necesitado para la investigación de los fundamentos teóricos de la aplicación y de las herramientas sólo hemos conseguido realizar un depurador declarativo utilizable para tipos básicos.

Información Riesgo Estimación 1			
Fecha	Octubre-Noviembre	Estado Inicial	Presente
Probabilidad	Alta	Consecuencias	Tolerables
Tipo	Predecible	Estado Actual	Eliminado por estrategia
Descripción	El tiempo calculado para desarrollar la totalidad de los requisitos sea insuficiente		
Estrategia	No especificar más requisitos de los que posteriormente se van a poder hacer		
Plan de contingencia	Disminuir el número de requisitos especificado.		

2.c.Diagrama de clases UML

En este apartado, procedemos a explicar los diagramas de clases de nuestro proyecto. En un primer diagrama UML representamos una idea general de nuestro proyecto. Hay dos paquetes principales, Aplicación y Tree. El programa principal, que se encuentra en el paquete Tree, crea un formulario (interfaz gráfica de la aplicación). Este formulario se comunica con el paquete de la aplicación mediante una instancia de la clase Analizador. Esta clase es la encargada de generar el fichero de texto que representa el árbol de cómputo para su posterior vista en la interfaz gráfica.



Los dos siguientes diagramas UML representan las clases que intervienen en cada uno de los paquetes de nuestro proyecto.

PAQUETE TREE

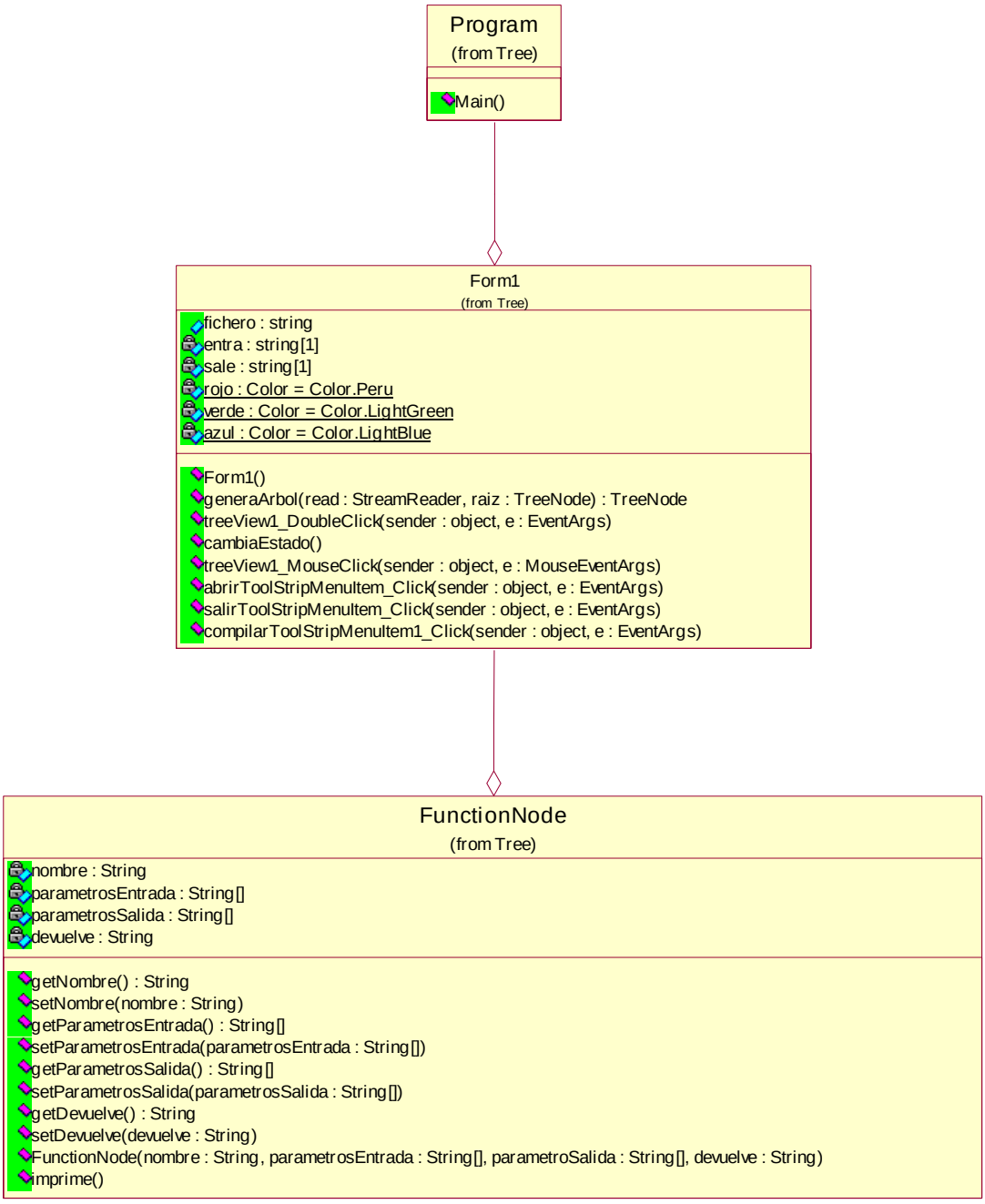
El paquete Tree estará compuesto por las clases que se encargan de implementar la interfaz gráfica que nos muestra el árbol de cómputo de nuestro depurador declarativo.

Este paquete constará por tanto de las tres siguientes clases:

- Program: ejecuta el programa principal creando el formulario o interfaz gráfica del depurador declarativo.

- Form1: es la clase encargada de mostrarnos la interfaz gráfica de la herramienta, como las opciones que tenemos, el árbol de cómputo, etc. Dicho árbol se realizará partiendo de un fichero de texto (el cuál se realiza en el paquete aplicación explicado anteriormente) que estará compuesto por el nombre de cada método del programa, el valor de entrada de los parámetros del método y el valor devuelto por dicho método.

- FunctionNode: es la clase encargada de proporcionar las características de los nodos del árbol de cómputo. Como hemos explicado anteriormente, los nodos se corresponderán con las llamadas a los métodos, por tanto esta clase se encargará de proporcionarnos la información mostrada de dichos métodos.


Program
(from Tree)
Main()
Form1
(from Tree)
fichero : string
entra : string[1]
sale : string[1]
rojo : Color = Color.Peru
verde : Color = Color.LightGreen
azul : Color = Color.LightBlue
Form1()
generaArbol(read : StreamReader, raiz : TreeNode) : TreeNode
treeView1_DoubleClick(sender : object, e : EventArgs)
cambiaEstado()
treeView1_MouseClick(sender : object, e : MouseEventArgs)
abrirToolStripMenuItem_Click(sender : object, e : EventArgs)
salirToolStripMenuItem_Click(sender : object, e : EventArgs)
compilarToolStripMenuItem1_Click(sender : object, e : EventArgs)
FunctionNode
(from Tree)
nombre : String
parametrosEntrada : String[]
parametrosSalida : String[]
devuelve : String
getNombre() : String
setNombre(nombre : String)
getParametrosEntrada() : String[]
setParametrosEntrada(parametrosEntrada : String[])
getParametrosSalida() : String[]
setParametrosSalida(parametrosSalida : String[])
getDevuelve() : String
setDevuelve(devuelve : String)
FunctionNode(nombre : String, parametrosEntrada : String[], parametroSalida : String[], devuelve : String)
imprime()

PAQUETE APLICACIÓN

El diagrama UML mostrado en la siguiente página, se corresponde con el paquete Aplicación. Consta de las clases encargadas de crear un fichero de texto con los nombres de todos los métodos del programa que se pretende depurar, con sus respectivos valores de parámetros y valores devueltos. Está compuesto por las siguientes clases:

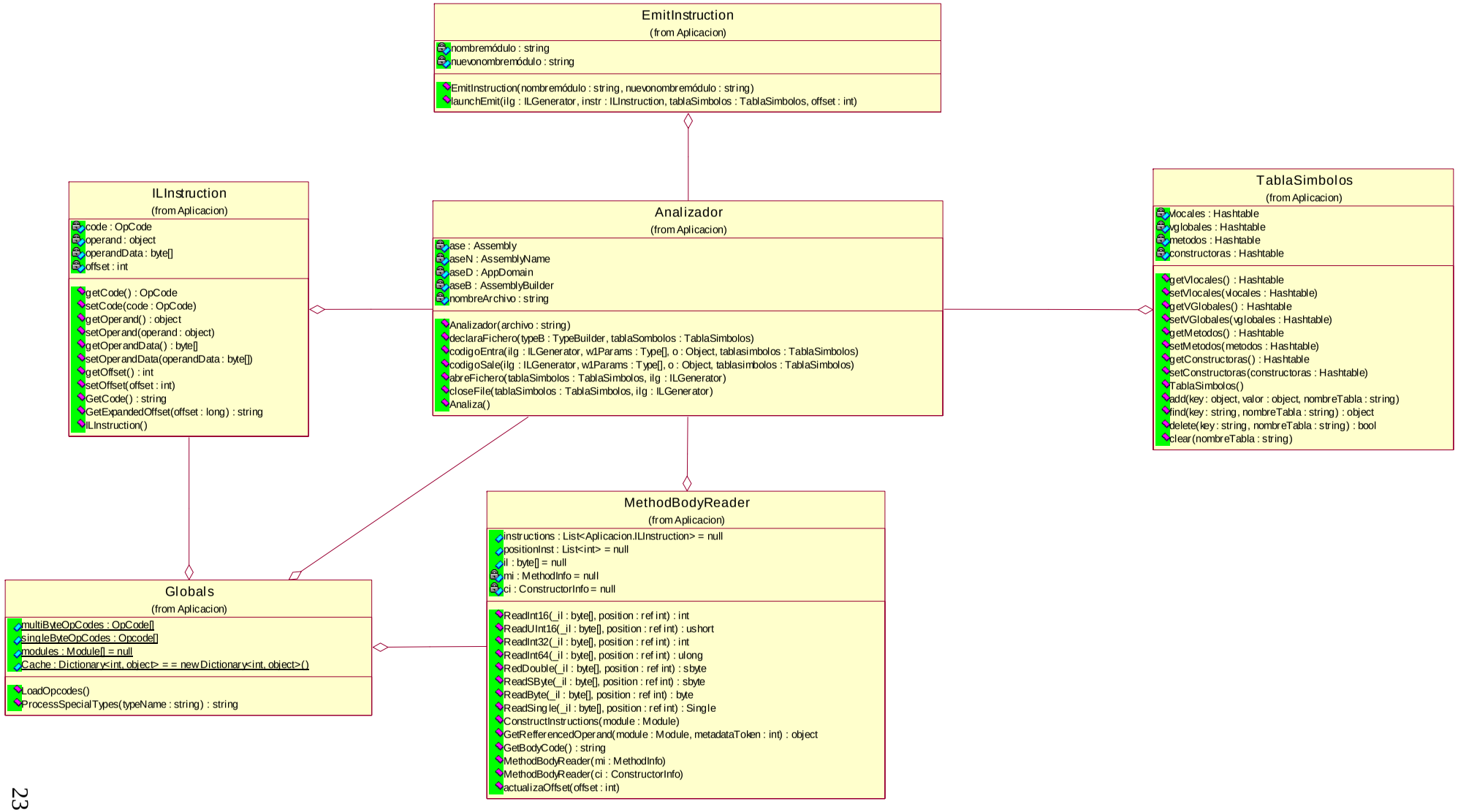
- Analizador: Es la clase más importante del paquete. Esta clase tiene una instancia de cada una de las clases del paquete. En términos generales, se encarga de coger el ejecutable del programa que queremos depurar. A partir de ese ejecutable, construimos otro ejecutable añadiéndole instrucciones nuevas del lenguaje MSIL utilizando la clase EmitInstruction. Este ejecutable será el encargado de crear un fichero de texto con la información de cada método, para que posteriormente se nos muestre en nuestro árbol de cómputo.

- MethodBodyReader: Mediante esta clase, obtenemos el lenguaje MSIL del cuerpo de los métodos, para así poder tratar instrucción por instrucción cada método.

- TablaSímbolos: En esta clase almacenamos los métodos del programa a depurar, sus constructoras, sus variables locales y sus variables globales.

- EmitInstruction: Mediante esta clase emitimos nuevas instrucciones MSIL que se añaden en el nuevo ejecutable que generará el fichero de texto anteriormente citado con la información de los métodos.

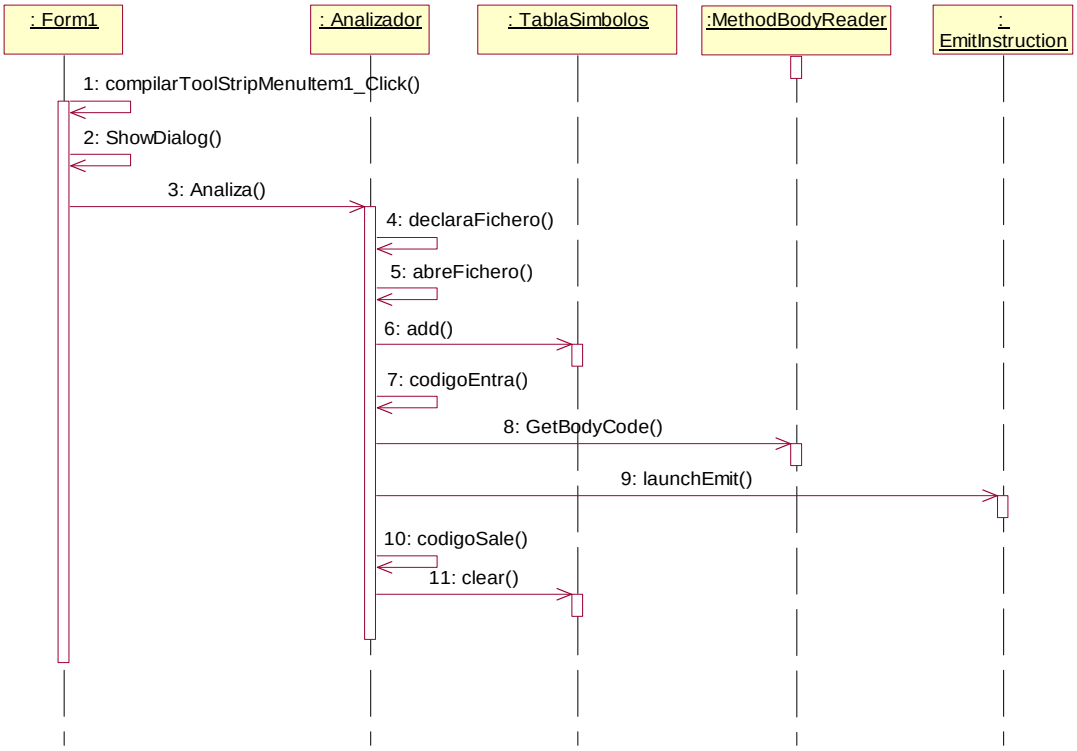
- ILInstruction y Globals: Son clases que se encargan de trabajar con cada una de las instrucciones MSIL que obtenemos del programa a depurar.


EmitInstruction
(from Aplicacion)
nombremódulo : string
nuevonombremódulo : string
EmitInstruction(nombremódulo : string, nuevonombremódulo : string)
launchEmit(ilg : ILGenerator, instr : ILInstruction, tablaSimbolos : TablaSimbolos, offset : int)
ILInstruction
(from Aplicacion)
code : OpCode
operand : object
operandData : byte[]
offset : int
getCode() : OpCode
setCode(code : OpCode)
getOperand() : object
setOperand(operand : object)
getOperandData() : byte[]
setOperandData(operandData : byte[])
getOffset() : int
setOffset(offset : int)
GetCode() : string
GetExpandedOffset(offset : long) : string
ILInstruction()
Analizador
(from Aplicacion)
ase : Assembly
aseN : AssemblyName
aseD : AppDomain
aseB : AssemblyBuilder
nombreArchivo : string
Analizador(archivo : string)
declaraFichero(typeB : TypeBuilder, tablaSombolos : TablaSimbolos)
codigoEntra(ilg : ILGenerator, w1Params : Type[], o : Object, tablasimbolos : TablaSimbolos)
codigoSale(ilg : ILGenerator, w1Params : Type[], o : Object, tablasimbolos : TablaSimbolos)
abreFichero(tablaSimbolos : TablaSimbolos, ilg : ILGenerator)
closeFile(tablaSimbolos : TablaSimbolos, ilg : ILGenerator)
Analiza()
TablaSimbolos
(from Aplicacion)
vlocales : Hashtable
vglobales : Hashtable
metodos : Hashtable
constructoras : Hashtable
getVlocales() : Hashtable
setVlocales(vlocales : Hashtable)
getVGlobales() : Hashtable
setVGlobales(vglobales : Hashtable)
getMetodos() : Hashtable
setMetodos(metodos : Hashtable)
getConstructoras() : Hashtable
setConstructoras(constructoras : Hashtable)
TablaSimbolos()
add(key : object, valor : object, nombreTabla : string)
find(key : string, nombreTabla : string) : object
delete(key : string, nombreTabla : string) : bool
clear(nombreTabla : string)
Globals
(from Aplicacion)
multiByteOpCodes : OpCode[]
singleByteOpCodes : Opcode[]
modules : Module[] = null
Cache : Dictionary<int, object> = = new Dictionary<int, object>()
LoadOpcodes()
ProcessSpecialTypes(typeName : string) : string
MethodBodyReader
(from Aplicacion)
instructions : List<Aplicacion.ILInstruction> = null
positionInst : List<int> = null
il : byte[] = null
mi : MethodInfo = null
ci : ConstructorInfo = null
ReadInt16(_il : byte[], position : ref int) : int
ReadUInt16(_il : byte[], position : ref int) : ushort
ReadInt32(_il : byte[], position : ref int) : int
ReadInt64(_il : byte[], position : ref int) : ulong
RedDouble(_il : byte[], position : ref int) : sbyte
ReadSByte(_il : byte[], position : ref int) : sbyte
ReadByte(_il : byte[], position : ref int) : byte
ReadSingle(_il : byte[], position : ref int) : Single
ConstructInstructions(module : Module)
GetRefferencedOperand(module : Module, metadataToken : int) : object
GetBodyCode() : string
MethodBodyReader(mi : MethodInfo)
MethodBodyReader(ci : ConstructorInfo)
actualizaOffset(offset : int)

2.d.Diagramas de secuencia

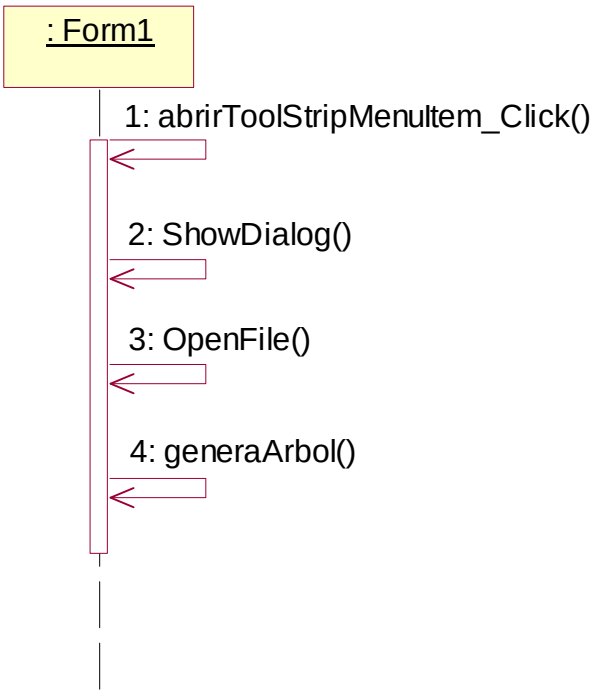
En este apartado adjuntamos dos diagramas de secuencia que representan dos funcionalidades de nuestro proyecto. El primer diagrama de secuencia corresponde con la funcionalidad de generar un árbol de cómputo. Dado un ejecutable de un programa a depurar, se selecciona y se crea un fichero de texto que representa nuestro árbol de cómputo.

Diagrama que representa generar un árbol de cómputo



Nuestro segundo diagrama de secuencia, representa la funcionalidad de mostrar árbol de cómputo. Mediante la opción de abrir fichero de nuestra interfaz gráfica, abrimos un fichero de texto (generado anteriormente) que representa el árbol de cómputo. Una vez abierto, aparece en nuestra interfaz de usuario el árbol de cómputo visualmente y podremos empezar a realizar la depuración.

Mostrar árbol de cómputo



3.Tecnología .NET

3.a.Proceso de compilación/ejecución en .NET

Microsoft.NET es un conjunto de tecnologías en las que Microsoft ha invertido mucho tiempo de investigación y desarrollo estos últimos años con un único fin: conseguir desarrollar una plataforma potente y sencilla para distribuir el software en forma de servicios que puedan ser suministrados remotamente, y además puedan comunicarse y combinarse unos con otros de manera totalmente independiente de la plataforma, lenguaje de programación y modelo de componentes con los que hayan sido creados. A todo esto se le conoce como la *plataforma .NET*, y a todos estos servicios de los que hemos hablado se les llama *servicios Web*.

Para crear aplicaciones para esta plataforma, tanto aplicaciones tradicionales (aplicaciones de consola, aplicaciones de ventana, etc) como servicios Web, Microsoft ha publicado el llamado "kit de desarrollo de software" conocido como *.NET Framework SDK*, que incluye herramientas necesarias para su desarrollo, distribución y ejecución, y Visual Studio .NET, que ofrece al desarrollador hacer todo lo anterior con una interfaz visual basada en ventanas. Estas dos herramientas las podemos descargar de forma gratuita desde la dirección http://www.msdn.microsoft.com/net, aunque la última sólo está disponible para subscriptores MSDN Universal (los no subscriptores pueden pedirlo desde la misma dirección y recibirlo por correo normal, también de manera gratuita).

Una manera rápida de resumir cómo se compila y ejecuta en .NET se puede deducir del gráfico que se muestra a continuación. Se observa que el código a compilar se escribe en un lenguaje concreto de alto nivel, y se compila con su correspondiente compilador. Ninguno de estos compiladores generará código máquina para CPUs x86 ni para cualquier otro tipo de CPU concreta, sino que habrán generado código MSIL. A partir de aquí, el CLR dará a las aplicaciones la sensación de que se están ejecutando sobre una máquina virtual, y precisamente MSIL es el código máquina de esa máquina virtual.

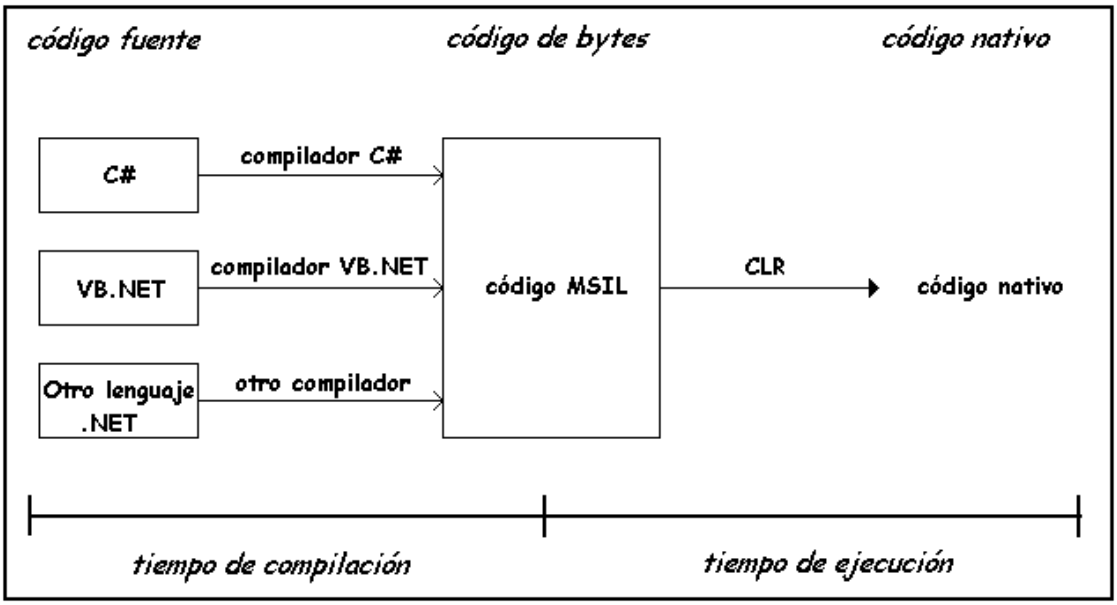


Figura1: Diagrama CLR

Las distintas fases que se muestran en la figura se explicarán en apartados posteriores.

3.b.MSIL

Los compiladores que generan código para la plataforma .NET no crean código máquina para las diferentes plataformas hardware, ya que generan un código escrito en un lenguaje intermedio llamado *Microsoft Intermediate Language* (MSIL). Cuando se dice que un compilador crea código para la plataforma .NET, en realidad estamos diciendo que crea código MSIL, que es el único lenguaje que es capaz de interpretar el CLR. Esto favorece el desarrollo de nuevos compiladores y herramientas, ya que no se necesita generar código para cada lenguaje ensamblador, sólo para el lenguaje intermedio MSIL.

Antes de continuar explicaremos brevemente qué es esto del CLR. Es el *Common Language Runtime*, el componente de máquina virtual del .NET Framework de Microsoft. Es la implementación del estándar *Common Language Infrastructure* (CLI) que define un ambiente de ejecución para los códigos de los programas. Es por eso que los programadores que usan CLR suelen escribir su código en lenguajes como C# o VB.Net.

En tiempo de compilación, un compilador .NET convierte el código MSIL. En tiempo de ejecución, el *compilador en tiempo de ejecución* (Just-in-time compiler) del CLR convierte el código MSIL en código nativo para que pueda ser interpretado por el sistema operativo. Podemos observarlo más claramente en la figura 1. Este compilador convierte dinámicamente el código MSIL a código nativo según sea necesario. Se distribuye en tres versiones principales: compilador JIT normal, económico y prejitter.

- El compilador JIT normal es el que se suele usar por defecto. Éste sólo compila el código MSIL a código nativo a medida que va siendo necesario, ya que así se ahorra tiempo y memoria, al evitarse tener que compilar innecesariamente el código que nunca se va a ejecutar.
- El compilador JIT económico funciona de forma similar al compilador normal, solo que no realiza ninguna optimización de código al compilar, sino que traduce cada instrucción
MSIL por su equivalente en el código máquina sobre el que se ejecute la aplicación. Está pensado para ser usado en CPUs con poca potencia o memoria, ya que necesita menos tiempo para compilar y menos memoria.

Además de las dos versiones anteriores del compilador JIT, Microsoft distribuye una tercera versión llamada prejitter, que se distribuye como una aplicación en línea de comandos mediante la cual es posible compilar completamente cualquier ejecutable o biblioteca que contenta código gestionado y convertirlo a código nativo.

Volviendo al CLR, éste solamente se ejecuta en sistemas operativos de Microsoft Windows, a pesar de que algunas implementaciones del CLI se ejecuten en sistemas operativos no Windows. La forma en que el CLR se relaciona con la máquina virtual permite a los programadores pasar por alto muchos detalles específicos de la

CPU que esté ejecutando el programa. Además, el CLR permite otros servicios como la administración de memoria, de hilos, el manejo de excepciones, la seguridad y la recolección de basura.

Sigamos con MSIL. Se trata de un lenguaje de un nivel de abstracción mayor que el de los lenguajes de código máquina de las CPU´s existentes, ya que por ejemplo permite trabajar directamente con objetos, ya sea para crearlos, inicializarlos, destruirlos, etc., o con excepciones (lanzarlas, capturarlas, etc.). Podemos ver la forma que tiene este código MSIL en la Figura 2.El compilador de C# (que hemos usado para nuestro proyecto) genera código fuente MSIL como hemos comentado antes. Además, Microsoft ha desarrollado extensiones de sus compiladores de VB y C++ que también generan MSIL. Aunque estos no son los únicos que lo generan, ya que otros lenguajes, como pueden ser Fortran, Haskell, Java, Pascal, Perl, Phyton o Smalltalk han optado por desarrollar una extensión que genere código MSIL para .NET. Por tanto estamos ante una oferta bastante amplia, ya que debido a la arquitectura de la plataforma .NET, podríamos programar nuestra aplicación en cualquiera de estos lenguajes, accediendo además a todos los servicios que proporciona el CLR, anteriormente comentados.

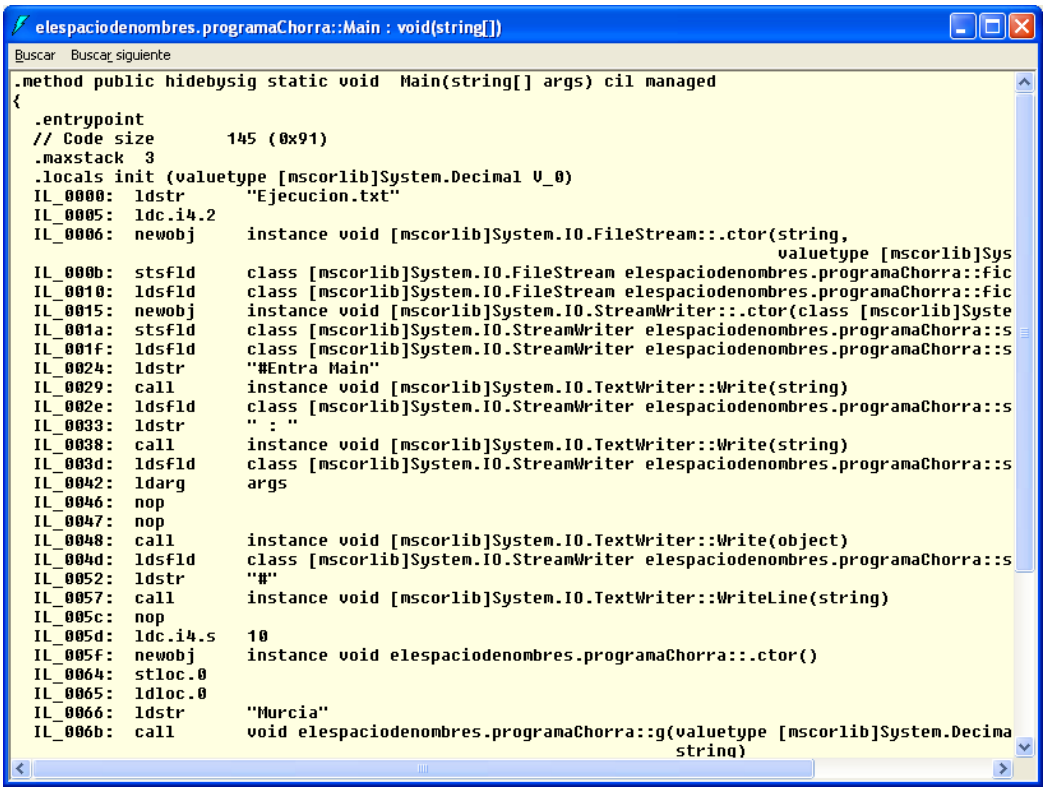


Figura 2: Código MSIL

Las principales ventajas del código MSIL es que facilita la ejecución multiplataforma y la integración entre lenguajes al ser independiente de la CPU. No obstante, dado que las CPU´s no pueden ejecutar directamente código MSIL, se necesita convertirlo antes al código nativo propio de la CPU en la que vamos a ejecutar nuestra aplicación. Y por eso hemos comentado más arriba la gran utilidad del compilador JIT. Podríamos pensar que la actuación de este compilador JIT puede disminuir el rendimiento de la aplicación compilada, debido a que debe invertirse un *quantum* de tiempo adicional para volver a compilar la aplicación dependiendo de la arquitectura del procesador. Y esto es cierto, pero si lo comparamos con arquitecturas de características

similares (Java, por ejemplo), resulta que es más eficiente, ya que el código no es interpretado y compilado a lenguaje nativo cada vez que se ejecuta, sino que se traduce una sola vez: cuando se llama al método al que pertenece. Además, que el compilador JIT tenga acceso en tiempo de ejecución a la máquina donde se ejecuta la aplicación, hace que posea mucha más información que ella, por lo que puede hacer más optimizaciones que las que podría hacer cualquier compilador tradicional.

Cuando el compilador produce código MSIL también genera metadatos. Los metadatos describen tipos que aparecen en el código, incluidas las definiciones de los tipos, las firmas de los miembros de tipos, los miembros a los que se hace referencia en el código y otros datos que el motor en tiempo de ejecución utiliza. El lenguaje intermedio de Microsoft y los metadatos se incluyen en un archivo ejecutable portable (PE), que se basa en el formato *Common Object File Format* (COFF) utilizado para contenidos ejecutables. Este formato de archivo utilizado, el cual contiene código MSIL o código nativo y metadatos, permite al sistema operativo reconocer imágenes de CLR. La presencia de metadatos junto con el código MSIL permite además crear códigos autodescriptivos, con lo cual las bibliotecas de tipos y el *Lenguaje de Definición de Interfaces* (IDL) son innecesarios. El motor en tiempo de ejecución localiza y extrae los metadatos del archivo cuando son necesarios durante la ejecución.

MSIL incluye instrucciones para carga, almacenamiento, inicialización y llamadas a métodos en los objetos, así como instrucciones para operaciones lógicas y aritméticas, flujo de control, acceso directo a memoria, control de excepciones y otras operaciones. Todas las operaciones en MSIL son ejecutadas en la pila. Cuando se llama a una función, sus variables locales y parámetros son colocados en la pila. El código de función que comienza en este estado de pila puede introducir algunos valores más en la pila, hacer operaciones con estos valores, y desapilar valores de la pila. La ejecución de funciones y comandos MSIL se hace en tres pasos:

1- Apilar parámetros de función u operandos en la pila.
2- Ejecutar los comandos MSIL o llamar a la función. El comando o la función desapila sus operandos o parámetros de la pila e introduce en ella los valores devueltos.
3- Se leen los resultados de la pila.

No obstante, el primer y tercer paso son opcionales, pensemos por ejemplo en una función de tipo *void* que no tenga parámetros de entrada. Los comandos MSIL usados para apilar valores son los load (ld), y para desapilar store (st).

Algunas directivas importantes a resaltar en MSIL, y que hemos tenido que hacer uso de ellas, son las siguientes:

- .entrypoint: define por dónde va a empezar la aplicación, es decir, a qué función va a llamar el .NET Runtime cuando comience el programa. Cualquier programador está acostumbrado a que suela ser el método Main, pero con esta directiva podremos elegir el método que queramos, aunque debe cumplir la condición de que sea un método estático. Lógicamente, esta directiva sólo puede aparecer en uno de los métodos de la aplicación.

- .maxstack: define la máxima profundidad de la pila usada por el código de función. En nuestro caso no tuvimos que preocuparnos por ella, ya que el compilador de C# ajusta siempre el valor exacto necesario para cada función.
- .locals init: sirve para definir las variables locales que se van a usar en el método donde están definidas. init especifica que las variables deben estar inicializadas a los valores por defecto de sus respectivos tipos.

Algunas instrucciones del lenguaje MSIL que también tuvimos que usar o conocer son los siguientes:

- **ldstr** cadena: carga la cadena en la pila.
- **call** función (parámetros): llama a la función estática. Los parámetros de la función se deben haber cargado en la pila previamente.
- **pop**: desapila un valor de la pila.
- **ret**: vuelve de una función.
- **ldc.i4.**n: carga una constante de 32 bits (n de 0 a 8) en la pila.
- **stloc.**n: guarda un valor de la pila en un número de variable local n, entre 0 y 3.
- **add:** suma dos valores, los cuales deben estar guardados en la pila previamente. La función desapila los dos valores, los suma, y vuelve a guardar el resultado en la pila.
- **sub:** resta dos valores, con el mismo procedimiento que add.
- **mul:** multiplica dos valor, con el mismo procedimiento que add.
- **bge.s** etiq: va a la etiqueta etiq si el valor1 es mayor o igual que el valor2. Dichos valores deben estar cargados en la pila previamente.
- **blt.s** etiq: salta a la etiqueta etiq si valor1 es más pequeño que valor2. Al igual que antes, ambos valores deben estar cargados en la pila previamente.
- **br.s** etiq: salta a la etiqueta etiq.
- **dup**: duplica el valor que se encuentra en la cima de la pila.
- **ldarg.n**: carga el argumento n en la pila. En una función que no sea estática, el argumento 0 está oculto y apunta a esta instancia.

Para terminar, conviene hablar un poco acerca de los ofuscadores. Es una técnica que nos permite cambiar el nombre de los símbolos de los ensamblados, así como otros "trucos" para frustrar la acción de los descompiladotes. Cuando se aplica de forma correcta, la ofuscación aumenta la protección contra la descompilación, dejando la aplicación intacta. Los ofuscadores primitivos cambian el nombre de los identificadores que se encuentran en dicho código por algo ilegible mediante técnicas de hash o desplazamiento aritmético, por ejemplo. Es bastante efectivo, el problema es que lógicamente es una técnica reversible, y por tanto poco protectora. Existen aplicaciones (pe. Dotfuscator) que además de cambiar los nombres introduce nuevas formas de crear confusión que hacen muy difícil realizar el trabajo inverso.

La ofuscación es un proceso que se aplica a código MSIL compilado, no a código fuente, ya que este nunca se modifica de ninguna manera. El código MSIL ofuscado es funcionalmente equivalente al código MSIL tradicional, y se ejecuta en el CLR con el mismo resultado (aunque esto no funciona a la inversa). En la figura 3 podemos observar esta equivalencia.

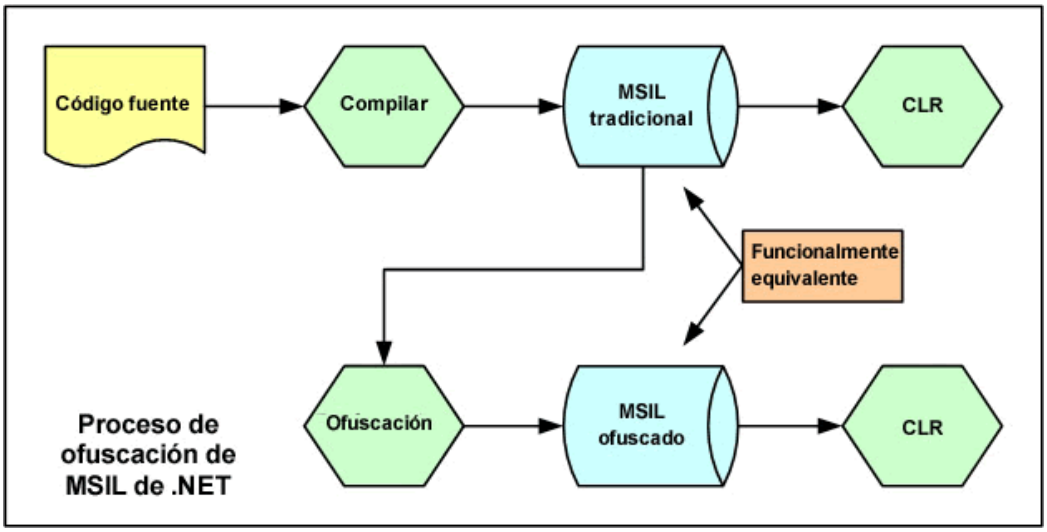


Figura 3: Ofuscación

Ninguna técnica de ofuscación es segura al cien por cien

3.c.Iildasm & Ilasm

El programa ILDasm (Desensamblador de Lenguaje Intermedio, Intermediate Language Disassembler) viene incorporado con el .NET Framework SDK (FrameworkSDK\Bin\ildasm.exe), y nos permite ver el código MSIL en un formato legible. Gracias a esta propiedad, podíamos abrir cualquier tipo de archivo ejecutable .NET (por lo general un archivo *.exe*, aunque en otras ocasiones puede ser un fichero *.dll*), y así poder ver su código MSIL para poder entenderlo mejor.

A su vez existe el programa contrario, el ILAsm (Ensamblador de Lenguaje Intermedio, Intermediate Language Assembler), el cual genera archivos ejecutables a partir de un fichero escrito en código MSIL (podemos encontrarlo en la ruta WINNT\Microsoft.NET\Framework\vn.nn.nn). El archivo de texto creado por ILDasm se puede utilizar como entrada del ensamblador del ILAsm, ya que es una técnica muy útil cuando se compila código en un lenguaje de programación que no admite todos los atributos de metadatos en tiempo de ejecución, por ejemplo. Una vez compilado el código y ejecutados los resultados mediante ILDasm, el archivo de texto de MSIL resultante se puede editar manualmente para agregar los atributos que faltan. A continuación se podría ejecutar este archivo de texto mediante el ensamblador de MSIL para producir un archivo ejecutable final. Si bien es cierto que actualmente no se puede utilizar esta técnica con archivos PE que contienen código nativo incrustado, como podría darse el caso en los producidos por Visual C++.

Aunque es igual de importante que el anterior, la realidad es que para la realización de nuestro proyecto no hemos tenido que hacer uso de esta herramienta, y sin embargo sí del ILDasm. Es por ello que nos centraremos en cómo hemos utilizado el ILDasm, y dejamos aparte cualquier explicación con respecto al ILAsm.

Cualquier programador que comienza a desarrollar en .NET debe estar interesado en saber qué sucede a bajo nivel en el .NET Framework. Aprender MSIL

aporta al usuario el reto de entender algunas cosas que están ocultas para el programador común de VB o C#. Es por ello que saber MSIL da más poder a un programador de .NET. Eso no quiere decir que vayamos a escribir directamente programas en código MSIL, pero en algunos casos es de gran utilidad abrir código MSIL con el programa ILDasm y ver qué cosas se han hecho, tal y como tuvimos que realizar nosotros en nuestro proyecto. Y es que en algunos casos es bastante difícil de comprender simplemente mirándolo.

Hemos usado la interfaz gráfica para el usuario predeterminada en el desensamblador de MSIL para ver los metadatos y el código desensamblado de los archivos que generábamos en una vista jerárquica. Para ello se podría escribir "ildasm" en la línea de comandos, sin proporcionar el argumento *PEfilename* ni ningún tipo de opción, aunque nosotros simplemente íbamos al directorio donde se encontraba el archivo ejecutable y hacíamos doble clic sobre él. Pero veamos primero en qué consiste dicho programa:

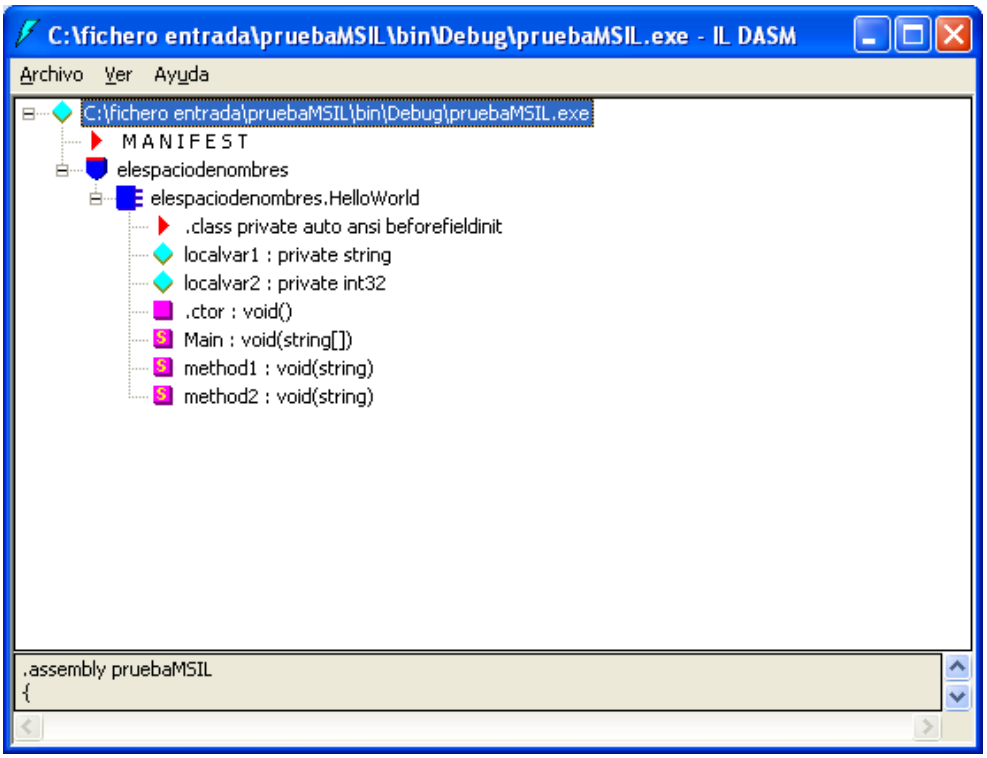


Figura 4: GUI principal ILDasm

Si arrancamos ILDasm, de algunas de las maneras sugeridas anteriormente, la GUI nos mostrará algo del estilo a la Figura 4, en la cual podemos ver el árbol del programa escrito en código MSIL. Hay toda clase de símbolos y distintos tipos mostrados en el árbol (como podemos ver en la Figura 5), pero si nos fijamos bien se pueden apreciar claramente las variables globales, métodos, clases, y en general todo tipo de información que contenía el programa.

Símbolo	Significado
	Más información
	Espacio de nombres
	Clase
	Interfaz
	Clase de valor
	Enumeración
	Método
	Método estático
	Campo
	Campo estático
	Evento
	Propiedad
	Manifiesto o un elemento de información de clase

Figura 5: Leyenda ILDasm

Si hacemos doble clic en un nodo hoja, como por ejemplo en uno de los métodos, se nos muestra toda la información que contiene dicho método. En caso de que no sea un nodo hoja, el nodo se expandirá para mostrar todos sus correspondientes descendientes. También podemos ver la metainformación que contiene pulsando ctrl+M. Se nos mostraría algo del estilo a lo que se puede ver en la Figura 6. Por tanto, para trabajar en este entorno sólo necesitamos conocer la combinación de teclas anteriormente citada, y hacer doble clic sobre cualquier aspecto que nos interese para obtener información.

Y, ¿por qué este programa nos ha sido de tanta ayuda? Conforme íbamos añadiendo nuestras propias sentencias de código MSIL en el código original, cometíamos errores que simplemente ejecutando el archivo no podíamos distinguir. Sin embargo, utilizando esta aplicación de Microsoft, podíamos ver el código IL traducido a código MSIL y ver cómo interpretaba el compilador las instrucciones en código MSIL que estábamos emitiendo. ¿Cómo aprendíamos a emitir correctamente instrucciones en código MSIL? Abríamos dos aplicaciones de ILDasm, y en una de ellas cargábamos nuestro archivo modificado, y en el otro el original, e íbamos comparando línea a línea donde teníamos el error. Un trabajo tedioso, pero eficaz. Para arrancar las dos aplicaciones de ILDasm con los correspondientes programas, ejecutábamos al finalizar nuestro código las siguientes instrucciones (duplicadas, una para el código modificado y otro para el original):

Página Web que explica cómo analizar el código IL de un método. Aquí encontramos el código del programa que hizo Sorin Serban y que utilizamos en nuestro proyecto.

```
System.Diagnostics.Process proc1 = new
System.Diagnostics.Process();
proc1.EnableRaisingEvents = false;
proc1.StartInfo.FileName = “C:\\...\\ildasm.exe”;
proc1.StartInfo.Arguments = “archivo1.exe”;
proc1.Start();
```

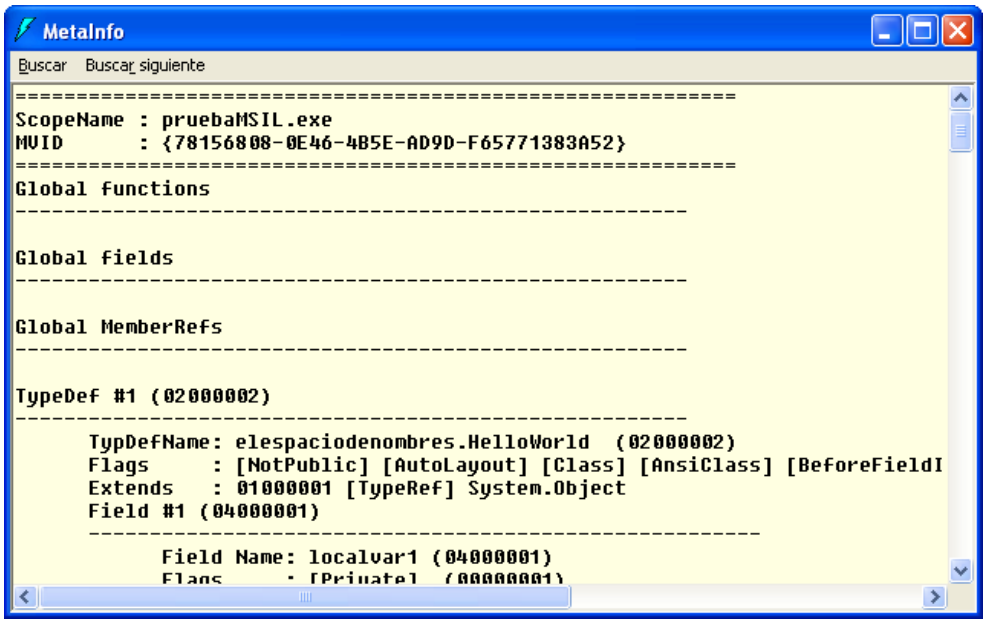


Figura 6: Metainformación ILDasm

Destacar además que ILDasm sólo muestra descripciones de metadatos para archivos de entrada .obj y .lib. El código MSIL de estos tipos de archivos no se desambla. Pero en nuestro caso sólo usamos archivos .exe, con los cuales sí podíamos ver si estaban administrados o no. Si el archivo no lo estaba, la herramienta en cuestión mostraba un mensaje que indicaba que el archivo no tenía encabezado válido de Common Language Runtime y no se podía desensamblar. Sin embargo, si el archivo estaba administrado, podíamos ejecutar la herramienta de manera correcta.

3.d.Reflector

Existe otra opción distinta al ILDasm para poder “descompilar” de código máquina a código MSIL, y así poder leerlo y entenderlo con mayor facilidad. Se trata del Reflector, un programa creado por Lutz Roeder, empleado de Microsoft. En la figura 7 podemos observar la versión que hemos usado nosotros, la 5.1.2.0.

(Figura en la siguiente página)

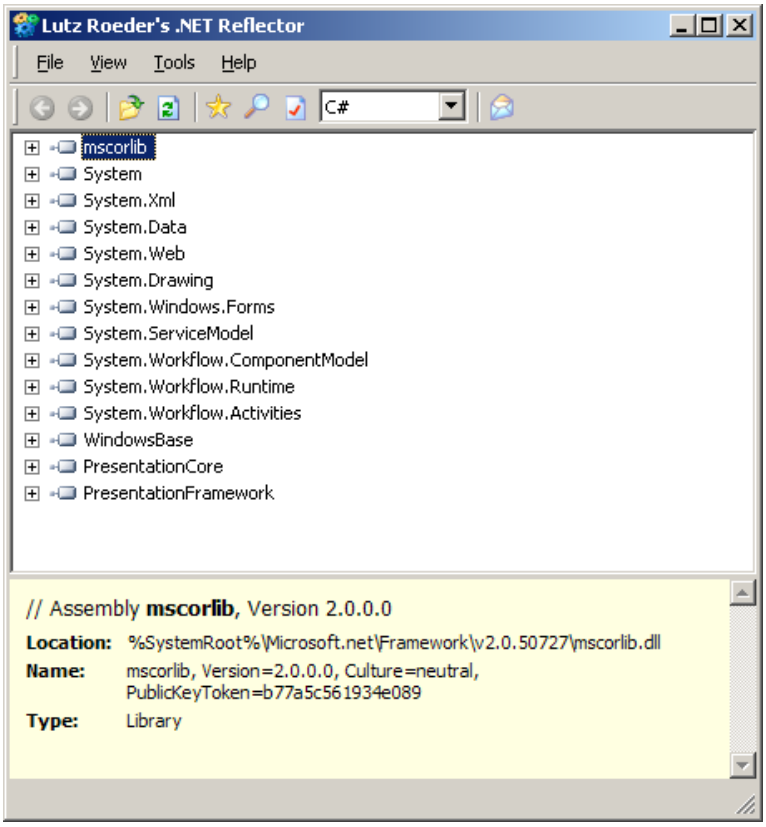


Figura 7: Reflector.exe

Para empezar a usarlo, primero debemos descargar el ZIP de la página de Lutz, http://www.aisto.com/roeder/dotnet. Éste contiene tres ficheros: Reflector.exe, el archivo ReadMe, y el Reflector.exe.config. Debemos descomprimir el ZIP en una ruta específica de nuestro ordenador, y hacer doble clic en Reflector.exe. Esto lanzará Reflector, el cual carga automáticamente algunos ensamblados de la librería de clases del .NET Framework:

- mscorlib
- System
- System.Xml
- System.Data
- System.Web
- System.Drawing
- System.Windows.Forms
- etc.

Podemos hacer clic en + para expandir cualquiera de los ensamblados anteriores, o bien abrir un ensamblado nuestro, expandirlo, y así mostrar las clases que existen en nuestro propio ensamblado. De nuevo haremos clic en una clase para ver sus métodos y propiedades. Hasta ahora, parece que Reflector sólo es un visualizador de clases (Visual Studio .NET incluye un visualizador de clases llamado *ObjectBrowser*). Pero su verdadero poder radica en sus capacidades descompiladoras. En la barra de herramientas superior podemos seleccionar el lenguaje a descompilar, ya sea C#, IL, Visual Basic, Delphi, MC++ o Chrome.

Si queremos ver la sintaxis en alto nivel para un método en concreto, seleccionamos en nuestro caso C#, que es el lenguaje en el que esta escrita nuestra

aplicación, y hacemos un simple clic sobre el método deseado, obteniendo nuestro código legible en lenguaje de alto nivel, como se muestra en la figura 8. La razón de que nos fuese útil esta aplicación fue que, debido a que nosotros modificamos directamente el código MSIL, en ocasiones introducíamos instrucciones erróneas, y con la aplicación ILDasm no podíamos apreciarlo. Sin embargo, en Reflector, cuando hay instrucciones que él entiende que son erróneas, nos avisa además que el código MSIL puede estar ofuscado, y por ello no puede mostrar dicha instrucción. Así nos dábamos cuenta de cuándo estábamos escribiendo líneas incorrectas a bajo nivel.

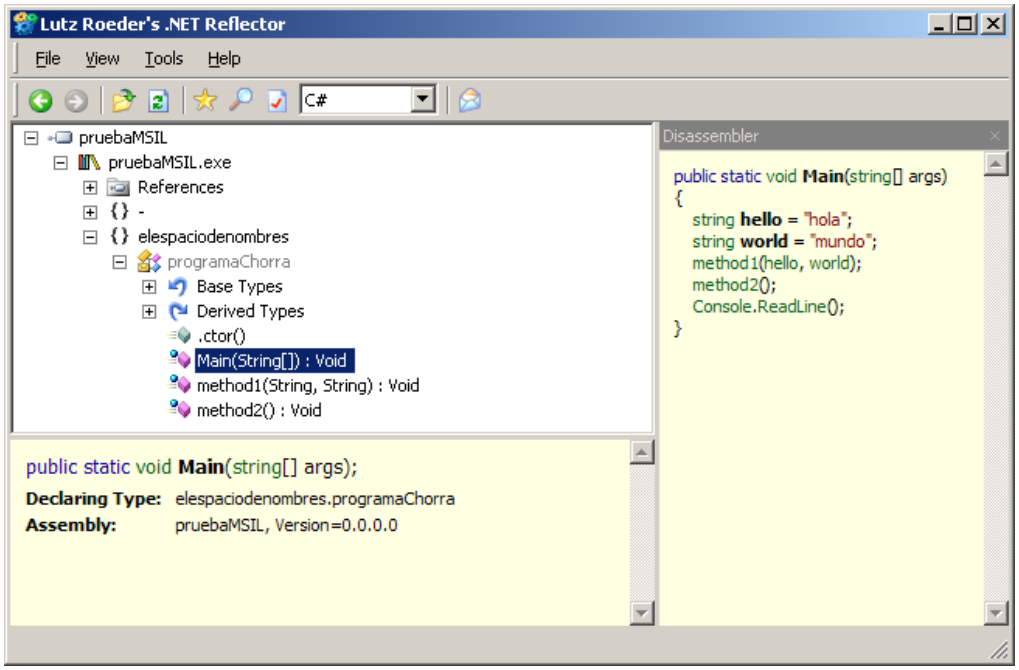


Figura 8: Desensamblando

Otro punto a favor de esta aplicación es su extensibilidad. Reflector permite a cualquier desarrollador crear Add-Ins, que se pueden cargar en el Reflector dinámicamente, y así obtener nuevas funcionalidades, como dibujar un grafo del ensamblado con todas sus dependencias, dibujar el diagrama de secuencia, o mostrar las diferencias que existen entre dos versiones distintas del mismo ensamblado (todas estas actualizaciones se pueden encontrar en http://www.codeplex.com/reflectoraddins).

3.e.Reflexión

Cuando el código fuente de un programa se compila, normalmente se pierde la información sobre la estructura del programa conforme se genera el código de bajo nivel (que normalmente suele ser lenguaje ensamblador). Si un sistema permite reflexión, se preserva la estructura como metadatos en el código generado. Por tanto, la reflexión es la capacidad que tienen algunos lenguajes para acceder a su estructura mediante su metadata (o metainformación, los datos de los datos).

Los lenguajes de .NET Framework 2.0 (y superiores) proveen esta funcionalidad mediante el espacio de nombres *System.Reflection*. Éste contiene clases (e interfaces) que proporcionan una panorámica administrada de los campos, métodos y tipos

cargados, además de tener la posibilidad de crear e invocar tipos dinámicamente; es decir, tenemos información acerca de los ensamblados cargados y los tipos definidos en ellos, tales como clases, interfaces y tipos de valor. La reflexión también la podemos usar para crear instancias de tipo en tiempo de ejecución, además de poder invocarlas y tener acceso a ellas.

¿Y por qué necesitamos esto? Porque nuestra técnica de depuración necesita generar una "copia modificada" del código MSIL correspondiente al programa erróneo. Esta copia contiene las mismas instrucciones y tiene el mismo funcionamiento que la original, salvo porque genera una traza que permite representar un cierto árbol de cómputo. Por tanto, para crear el programa modificado debemos ser capaces de leer y alterar un programa MSIL, y esto requiere del uso de la reflexión.

A continuación mostramos los pasos mínimos que hay que dar para generar un código MSIL usando las propiedades de reflexión:

Primer paso:

Consiste en obtener un ensamblado, que son las unidades de creación de las aplicaciones .NET Framework. Éstos conforman la unidad fundamental de implementación, control de versiones, ámbitos de activación, reutilización y permisos de seguridad. Un ensamblado es una colección de tipos y recursos creados para funcionar todos ellos en conjunto, y formar una unidad lógica de funcionalidad. Los ensamblados proporcionan a *Common Language Runtime* la información necesaria para conocer las implementaciones de tipos. Para el motor en tiempo de ejecución, un tipo no existe si no es en el contexto de un ensamblado. Por tanto, un ensamblado realiza todas estas funciones:

- Contiene el código que ejecuta *Common Language Runtime*. El código del lenguaje intermedio de Microsoft (MSIL) de un archivo ejecutable no se ejecuta si no tiene asociado un ensamblado. Tener en cuenta que cada ensamblado solamente puede tener un punto de entrada (en nuestro caso será el método Main del programa a analizar).

- Crea un límite de seguridad. El ensamblado es a quien se solicita y concede los permisos, sin él no podríamos, por ejemplo, acceder a métodos privados.

- Crea un límite de tipos. La identidad de todos los tipos incluyen el nombre del ensamblado en el que se encuentran. Así pues, no es el mismo tipo "MiTipo", cargado en el ensamblado "Ensamblado1", que el tipo "MiTipo" cargado en el ensamblado "Ensamblado2". Lo podemos ver mejor en la Figura 9:

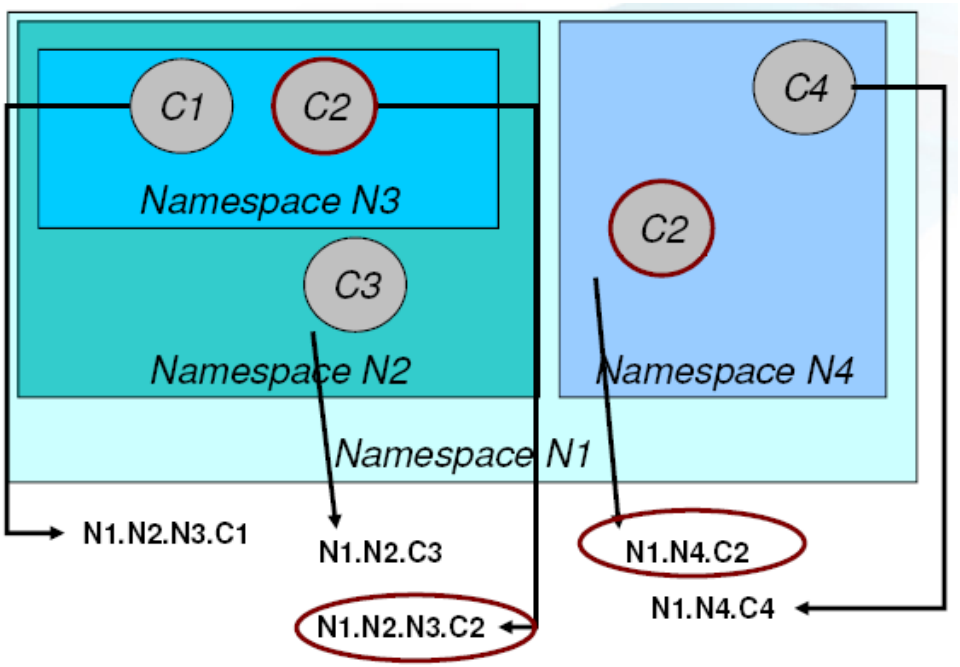


Figura 9: Espacio de nombres

- Crea un límite de ámbito de referencia. El manifiesto del ensamblado contiene metadatos del ensamblado que son usados para satisfacer las solicitudes de recursos y resolver los tipos. Especifica los tipos y recursos que se exponen fuera del ensamblado, además de enumerar el resto de ensamblados de los que depende.

- Forma un límite de versión. El ensamblado es la unidad "versionable" más pequeña de *Common Language Runtime*; todos los tipos y recursos del mismo ensamblado pertenecen a esa misma versión. El manifiesto del ensamblado contiene las dependencias de versión que se especifiquen para los ensamblados que son dependientes de éste.

- Crea una unidad de implementación. Cuando se inicia la aplicación, sólo deben estar los ensamblados a los que llama la aplicación al inicio. El resto de ensamblados, como por ejemplo los que contienen clases de utilidad, se pueden recuperar más tarde a petición. De esta manera, se mantiene la simplicidad y transparencia de las aplicaciones la primera vez que se descargan.

- Permite la ejecución simultánea.

Los ensamblados pueden ser estáticos o dinámicos. Los estáticos pueden incluir tipos de .NET Framework (interfaces y clases), así como recursos para el ensamblado (mapas de bits, archivos JPEG, archivos de recursos, etc.). Los ensamblados estáticos se almacenan en el disco, en archivos ejecutables portables PE. También se puede utilizar .NET Framework para crear ensamblados dinámicos, que se ejecutan directamente desde la memoria y no se guardan en el disco antes de su ejecución. Los ensamblados dinámicos se pueden guardar en el disco una vez que se hayan ejecutado

Para crear nuestro ensamblado basta declararlo y cargarlo desde el archivo que deseemos analizar:

```
Assembly ase = Assembly.LoadFrom(archivo);
```

En nuestro caso no queremos crear un ensamblado partiendo de cero, sino hacer una copia modificada de uno ya existente (concretamente del programa a depurar), y por eso nos limitamos inicialmente a cargar el archivo en el que se encuentra este programa.

Segundo paso:

Un ensamblado está compuesto por uno o varios módulos (paquetes) implementados como una unidad, por tanto nuestro siguiente paso es obtener todos estos módulos declarados en el ensamblado. Un módulo consta de una o varias clases e interfaces (llamadas tipos). Puede que haya varios espacios de nombres contenidos en un solo módulo, e incluso un espacio de nombres puede abarcar a varios módulos. En nuestro caso obtenemos el array de módulos contenido en el ensamblado que estamos copiando y modificando de la siguiente manera:

```
Module[] m = ase.GetModules();
foreach (Module mod in m)
        { ... }
```

Tercer paso:

Obtenemos todos los tipos que hayan sido declarados en el modulo en el que nos encontramos. Los tipos representan declaraciones de tipo, es decir, tipos de clase, tipos de interfaz, tipos de matriz, etc.

Estos tipos son la raíz de la funcionalidad de la reflexión, y constituye el modo principal de obtener acceso a los metadatos. Para poder obtener información acerca de los métodos, constructores, campos, propiedades, etc., es necesario utilizar dichos tipos.

Un objeto de esta clase que representa un tipo es único. Con esto queremos decir que dos referencias al objeto *Type* hacen referencia al mismo objeto solamente si representan el mismo tipo. Por tanto nos permite comparar estos objetos con la igualdad de referencia.

Sin la clase *ReflectionPermission*, el código sólo puede obtener acceso a los miembros públicos de los ensamblados cargados (como por ejemplo el método *Object.GetType*, que es uno de los cuales debíamos ignorar cuando lo detectábamos, ya que no íbamos a realizar ninguna acción sobre él), y desagraciadamente no hemos podido implementarlo. Por tanto en nuestra aplicación no tendremos acceso a métodos privados.

Type es una clase base abstracta que nos permite diversas implementaciones. El sistema siempre proporcionará el *RuntimeType* de la clase derivada. En la reflexión, todas las clases que comienzan por la palabra *Runtime* se crean sólo una vez por cada objeto del sistema y admiten operaciones de comparación.

Se puede obtener la referencia al objeto *Type* asociado a un tipo de muchas maneras, pero en nuestro caso lo realizamos apoyándonos en el módulo que ya tenemos:

```
Type[] t = mod.GetTypes();
foreach (Type tip in t)
{ … }
```

Cuarto paso:

Una vez tenemos el tipo podemos hacer muchas cosas con ello, desde la obtención de los atributos de un objeto, o la ejecución de métodos que en principio no están accesibles, por ejemplo. A continuación uno de los ejemplos más simples que podríamos mostrar, como puede ser la manera de conseguir los nombres de todos los métodos que se encuentran en la clase:

```
MethodInfo[] metI = tip.GetMethods();
foreach (MethodInfo met in metI)
{
        string nombreMétodo = met.Name;
 }
```

Bien es cierto que esta característica da al .NET Framework y a sus familiares cercanos una potencia nada desdeñable y a la que los programadores no estamos acostumbrados, y por ello se requiere un gran esfuerzo para poder trabajar con la reflexión.

Quinto paso:

Por último, podemos mencionar la manera en la que obtenemos el código asociado a cada método. Esto se consigue a través de un array de bytes, del que iremos sacando una a una todas las instrucciones para ir emitiéndolas y generar nuestro nuevo archivo modificado, en el que estaremos además incluyendo nuestro propio código para generar la traza.

```
MethodBodyReader reader = new
MethodBodyReader(met);
for (int j = 0; j < reader.instructions.Count; j++)
{ … }
```

3.f.Visual Studio 2005

Para poder empezar a desarrollar nuestro proyecto, necesitábamos un entorno adecuado para realizarlo. Debíamos adquirir y configurar un entorno de desarrollo que fuera adecuado para nuestro proyecto, y que nos permitiese una mayor facilidad de manejo y entendimiento que otros. Por ello seleccionamos Visual Studio 2005 SP1, que es el IDE (Integrated Development Environment) de Microsoft.

Visual Studio es una herramienta de desarrollo que fue diseñada por la compañía Microsoft. Su primera aparición fue en 1997 con el nombre de Visual Studio 97, y existían dos versiones distintas: Professional y Enterprise. Sucesivamente han ido apareciendo nuevas versiones hasta llegar a la actual, Visual Studio 2008, diseñado para aprovechar las ventajas que ofrece el nuevo sistema operativo Windows Vista. Sin embargo, como hemos comentado antes, la versión que hemos elegido para nuestro trabajo ha sido Microsoft Visual Studio 2005 Professional Edition, con la ampliación del Service Pack 1, y con la versión del .NET Framework 2.0.50727, ya que por medio de nuestra facultad se nos facilitaba de manera gratuita. Aunque a partir de dicha versión Microsoft empezó a distribuir versiones que usaban menos características de forma gratuita, con las sucesivas también se podría haber hecho el trabajo que hemos realizado, ya que entre otras razones, sólo hemos hecho uso de C#, el cual es uno de los muchos lenguajes de programación que ofrece y con los que se puede trabajar en esta plataforma.

La elección de este entorno de desarrollo fue, principalmente, porque ya lo conocíamos de un año anterior, en el que habíamos realizado otro tipo de proyecto en un grupo de 17 personas para la asignatura de Ingeniería del Software. Además, habíamos comprobado también que todas las clases que íbamos a usar a priori estaban incorporadas en el IDE. Por tanto, estas son las razones por las que elegimos esta opción que nos facilitaría en gran medida su montaje.

Detenernos en la explicación de cómo se realizó la instalación y la configuración del programa es algo que no vamos a explicar aquí, ya que si quisiésemos entrar a tal explicación, se podría rellenar un libro entero, y ese no es nuestro objetivo. Lo que sí comentaremos es que un proyecto de Visual Studio tiene configuraciones independientes para las versiones de lanzamiento y depuración del programa. Como se puede desprender de los nombres, la versión de depuración se genera para la depuración y la versión de lanzamiento para la distribución final de lanzamiento. Esto podemos verlo en la figura 10.

Visual crea automáticamente estas configuraciones, y establece las opciones predeterminadas apropiadas y otros valores. Con la versión de depuración del programa (que es la que hemos usado nosotros) se compila sin optimizar y con toda la información de depuración simbólica. La optimización complica la depuración, ya que la relación entre el código fuente y las instrucciones generadas es bastante más compleja.

Por otro lado existe la configuración Release, que no contiene información de depuración simbólica y está totalmente optimizada, por lo que hubiéramos tenido más dificultades para encontrar errores.

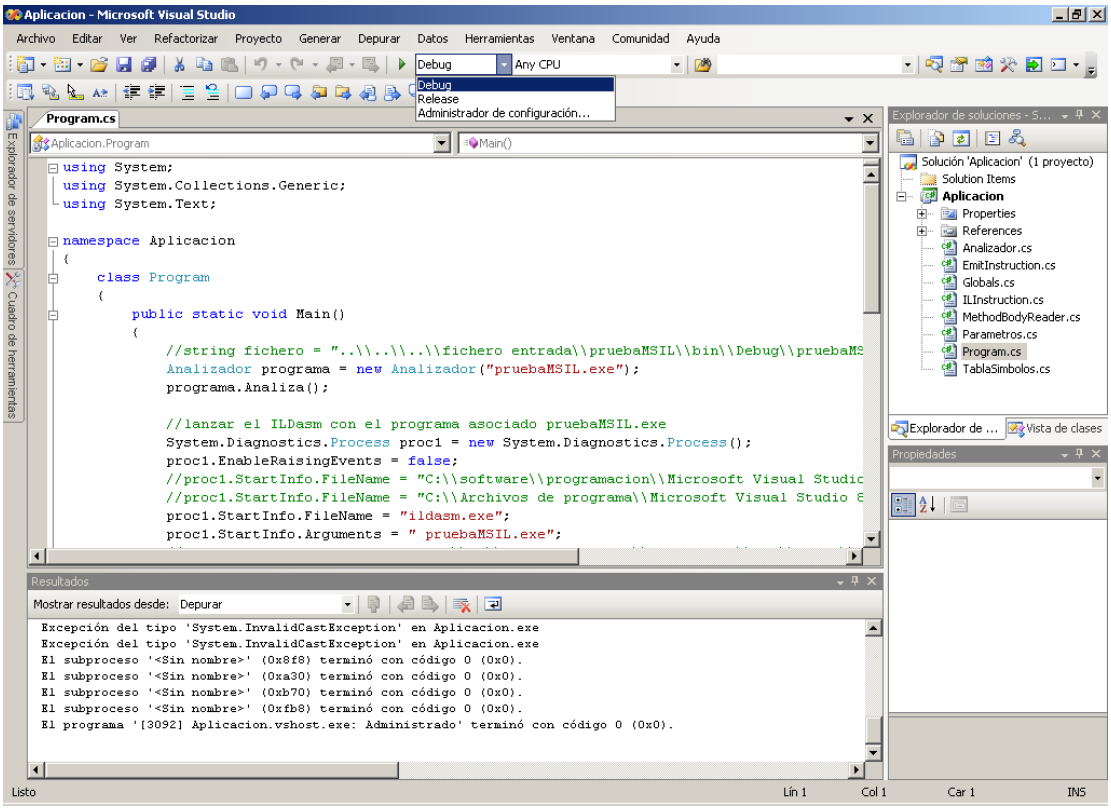


Figura 10: Visual Studio 2005

No obstante, el uso del depurador no ha sido muy grande, debido a la particularidad de nuestro proyecto. Con esto queremos decir que, lo normal en cualquier desarrollo de una aplicación suele ser que los errores se encuentren bien al compilar (errores en la sintaxis), o bien al ejecutar (errores en la semántica). Pero en nuestro caso, en varias ocasiones ocurría que habiendo compilado y ejecutado el programa generado, se detectaba el error, y esto complicaba mucho su detección. Además, ninguna de las herramientas que ofrece Visual Studio podía ayudarnos en dicha tarea, por lo que sólo podíamos usar nuestro ingenio para ir solucionando errores de este tipo. Es por esta razón que no hemos hecho mucho uso del depurador de Visual Studio.

4.Desarrollo

Este apartado explica cómo fue el proceso de desarrollo de nuestro proyecto, empezando por los primeros planteamientos en los que nos preguntábamos por la viabilidad de la idea inicial, hasta que logramos programar una aplicación real que poco a poco se ajustaba más a nuestros requisitos.

La primera fase del proyecto consistió en pensar sobre las alternativas de desarrollo, es decir, una vez concretado qué queríamos conseguir, empezamos a plantear diferentes formas de implementarlo, para asegurarnos así de optar por el mejor camino. De cada una de las alternativas que propuestas sopesamos las ventajas y desventajas (Más información sobre esta fase en el apartado de *Alternativas de desarrollo*). De entre todas las alternativas propuestas la que nos pareció más atractiva por sus posibilidades y eficiencia, fue la de utilizar el código intermedio de Microsoft, también llamado código MSIL.

La idea base consiste manipular el código MSIL para incluir código oculto en la compilación de la aplicación que se quiere analizar, es decir, utilizar código instrumentado. Este código instrumentado nos sirve para que después al realizar una ejecución del programa ofrecido por el usuario, podamos extraer información sobre la ejecución y a partir de ella generar un árbol con todos los datos sobre la las operaciones acaecidas durante la ejecución.

Este proyecto fue principalmente de investigación, hasta ahora no tenemos constancia de que nadie haya realizado un trabajo similar sobre la misma tecnología. Es por ello que los primeros meses tuvimos que dedicar mucho tiempo a documentarnos. A continuación explicamos cómo fuimos llevando a cabo esta investigación:

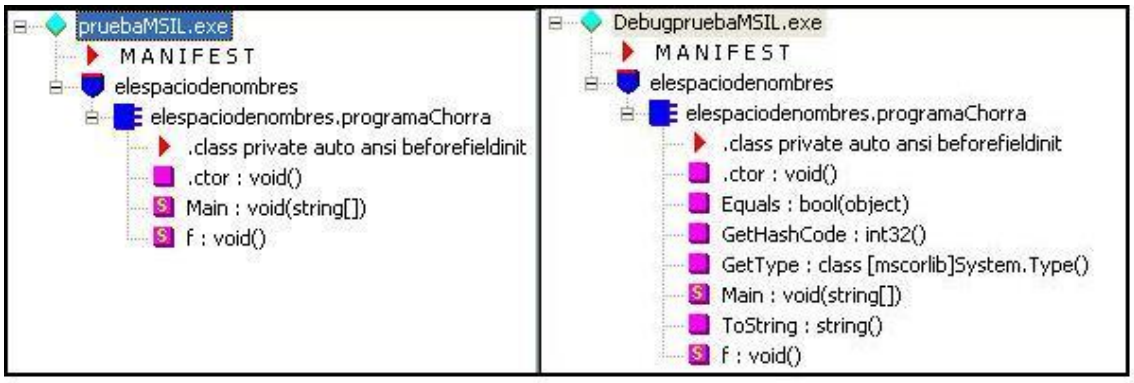
El primer paso en la investigación estaba dirigido a determinar si la implementación elegida era realmente viable. Para ello comenzamos a estudiar sobre un concepto nuevo para nosotros: la Reflexión. Como ya comentamos anteriormente, en .NET contamos con una técnica denominada Reflection (o Reflexión) .Esta técnica permite trabajar de manera dinámica con los tipos de datos en tiempo de ejecución. Entre otras cosas, permite inspeccionar tipos de datos para examinar su contenido y también permite generarlos.

Una vez entendidas las posibilidades de la reflexión, nos propusimos generar una copia exacta de la estructura de una aplicación proporcionada por le usuario. Este objetivo nos llevó bastante tiempo, de hecho, mientras lo intentábamos intentamos buscar ayuda diversas fuentes, y comprobamos que hasta ahora nadie se había planteado utilizar la reflexión para conseguir una copia exacta de una aplicación ya existente, lo que estábamos intentando era una idea nueva. Encontramos programas para generar el esqueleto sencillo de una nueva aplicación, pero no encontramos nada que consiguiera copiar exactamente todas las características que definen cada elemento de la estructura. Después de algún tiempo y mucho esfuerzo, conseguimos completar esta primera meta, era el primer programa que existía capaz de crear una nueva aplicación con las mismas características de otra ya compilada proporcionada por parámetro. O dicho de otra

manera, habíamos conseguido crear un nuevo Ensamblado con una copia exacta de módulos, paquetes, clases y métodos de la aplicación original.

En paralelo a la implementación de la primera versión del proyecto, comenzamos a utilizar un programa desensamblador para poder interpretar el código compilado que estábamos generando. Nosotros podíamos ejecutar el nuevo Ensamblado que habíamos creado, pero no podíamos ver la estructura interna de este nuevo Ensamblado, para ello utilizamos este programa, Ildasm de Microsoft. Dedicamos a este programa un apartado entero de la documentación, pues aunque no forme parte del proyecto, nos sirvió para depurar y puede ser muy útil para facilitar la comprensión de cada fase del proyecto con muestras visuales.

A continuación ofrecemos una muestra visual de lo que habíamos conseguido tras esta fase (imagen a partir del Programa de Microsoft Ildasm): A la izquierda está la estructura del programa original que queremos copiar, a la derecha aparece la estructura copia que habíamos conseguido hasta ahora (hay que destacar que en la copia, no sólo se habían incluido las constructoras y métodos propios de la aplicación original, sino también se habían incluido otros métodos que debieran ser externos pero que el compilador había agregado al crear el código compilado. Estos métodos los suprimimos de la copia más adelante, así no generaríamos más información de la estrictamente necesaria).



Sin embargo hay algo fundamental que todavía no habíamos conseguido: Al hacer una inspección minuciosa, esperábamos encontrar las variables utilizadas, así como los ciclos y las decisiones adoptadas dentro de un método cuerpo. Microsoft ha descuidado esta necesidad, pero todavía nos proporcionó algo: el código IL.

Esto no es suficiente, pues se trata de un array de bytes que primero debemos aprender a descifrar. Lo que necesitábamos era una serie de objetos que representaran la realidad de las instrucciones de código IL. Para conseguir esta meta nos basamos en las investigaciones de un informático de Rumanía, Sorin Serban. Sorin que recientemente creó una aplicación que capaz de descifrar el este array de bytes y desglosarlo en intrucciones de código MSIL. Sus investigaciones fueron fundamentales para acelerar nuestro proyecto y por ello le estamos muy agradecidos.

Para entender bien en qué consiste esta transformación de código IL a código MSIL (Microsoft Intermediate Language), vamos a ver un ejemplo con el programa clásico “Hola Mundo”. En este caso partimos del programa en código C #:

```
public void SayHello()
{
        Console.Out.WriteLine( " Hello world" );
}
```

Al llegar el cuerpo del método `SayHello` utilizando la reflexión, conseguimos el código IL, es decir, tenemos una serie de bytes, tales como:

```
0,40,52,0,0,10,114,85,1,0,112,111,53,0,0,10,0,42
```

Esto no es muy legible, por eso queremos transformarlo, para poder procesarlo. Este mismo programa, una vez transformado a código MSIL tendrá el siguiente aspecto:

```
0000 : nop
0001 : call System.IO.TextWriter System.Console::get_Out()
0006 : ldstr " Hello world"
0011 : callvirt instance System. Void
System.IO.TextWriter::WriteLine()
0016 : nop 0017 : ret
```

Una instrucción de código IL es una ristra de bytes que en realidad representa el par*: <opcode, operand>*.

- *Opcode* es un valor que nos informa sobre el tipo de la intrucción actual,
- *Operand* es la dirección de la información sobre los metadatos de la entidad.

Por ejemplo, opcode puede decirnos que estamos tratando una instrucción de carga en pila, y el atributo operand nos da información más detallada como: el tipo de lo que vamos a cargar en la pila, el valor de ese elemento, la dirección dónde buscarlo...

Así con el fin de llegar a un código más legible, se transforman estos bytes en un nuevo tipo formado por estos dos atributos <operation_code, operand>. La forma de conseguir esto a partir del código IL del cada método conseguido aplicando técnicas de Reflexión, se basa en:

- Obtener el siguiente byte y ver qué operador que estamos tratando.

- Dependiendo del operador, los metadatos token se define en los próximos 1, 2, 3 o 4 bytes. Obtener los metadatos muestra del operand.
- Almacenar el par <opcode, operand>.
- Repertir el proceso si no estamos al final de la cadena IL.

En resumen, gracias a las investigaciones de Sorin Seban, pudimos trabajar con un nuevo tipo que define la instrucción en vez de tener que trabajar directamente con bytes.

Ahora ya teníamos acceso al código de las instrucciones, el siguiente paso era aprender a emitirlas una a una. Al emitir una instrucción es como se asocia realmente la instrucción al Ensamblado, como nosotros queríamos intercalar instrucciones extra, necesitábamos entender cómo se emitían instrucciones de una en una. Descubrimos que cada tipo de instrucción es diferente y se emite de manera diferente, en concreto existen 240 tipos. Sin duda se trata de demasiados tipos para un proyecto con un periodo de duración de un curso lectivo, así que nos centramos en los más comunes y nos quedamos con 17:

- El operando es un entero de 8 bits.
- El operando es un símbolo (token) de cadena de metadatos de 32 bits.
- El operando es el argumento entero de 32 bits de una instrucción máquina de conmutación.
- El operando es un entero de 16 bits que contiene el ordinal de una variable local o un argumento.
- El operando es un destino de bifurcación entero de 8 bits.
- El operando es un símbolo (token) de metadatos de 32 bits.
- El operando es un destino de bifurcación entero de 32 bits.
- El operando es un símbolo (token) de firma de metadatos de 32 bits.
- El operando es un símbolo (token) de FieldRef, MethodRef o TypeRef.
- El operando es un símbolo (token) de metadatos de 32 bits.
- El operando es un entero de 32 bits.
- El operando es un entero de 64 bits.
- El operando es un número de punto flotante IEEE de 32 bits.
- El operando es un número de punto flotante IEEE de 64 bits.
- No hay operando.
- El operando es un entero de 8 bits que contiene el ordinal de una variable local o un argumento.
- El operando es un símbolo (token) de metadatos de 32 bits.

Implementar cada uno de estos 17 tipos fue muy complicado y llevó mucho tiempo, primero empezamos por los más visuales o sencillas, como :

- *Nop*, que pertenece al grupo no hay operando.

- *Readline ()* que entraría en el último grupo, símbolo (token) de metadatos de 32 bits;

- *Writeline(Hola)* que necesita carga previa en la pila del dato *Hola* (símbolo (token) de cadena de metadatos de 32 bits), y la posterior llamada al método, símbolo (token) de metadatos de 32 bits.

Hay que destacar que en estos dos casos se trata de llamadas a una librería externa, en ambos casos fue fácil hacer la llamada, sin embargo, las llamadas internas fueron mucho más difíciles de implementar, esto lo resolvimos más adelante.

Lo siguiente que conseguimos fue crear una emisión diferente de la instrucción por cada tipo posible para una variable:

- *String* (Pertenece al grupo: operando es un símbolo (token) de cadena de metadatos de 32 bits),
- *Entero de 8 bits* (Pertenece al grupo: operando es un entero de 8 bits),
- *Entero de 32 bits* ((Pertenece al grupo: operando es un entero de 32 bits),
- *Entero de 64 bits* (Pertenece al grupo: operando es un entero de 64 bits),
- *Número en punto flotante de 32 bits* (Pertenece al grupo: operando es un número de punto flotante IEEE de 32 bits.
- *Número en punto flotante de 64 bist* (Pertenece al grupo: operando es un número de punto flotante IEEE de 64 bits.

Recapitulando todo lo que habíamos hecho, ahora podíamos copiar un sencillo programa con el siguiente aspecto:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace elespaciodenombres
{
    class programa
    {
        //Programa principal
        public static void Main(string[] args)
        {
            nop;
            Console.WriteLine("Hola Mundo");
            Console.WriteLine(123);        }
    }
}
```

Una vez llegamos a este punto, intentamos copiar las variables locales y las variables globales. A primera vista parecían copiarse bien y la emisión de la instrucción parecía correcta, sin embargo, en al ejecutar el nuevo programa descubríamos que no se podía acceder a su valor. Paralelamente a estos intentos, probamos a hacer instrucciones de llamada a funciones internas al módulo, esto tampoco funcionaba. Nos costó mucho depurar esta parte, pero finalmente encontramos el problema que curiosamente era el mismo en los dos casos: Las instrucciones que lanzábamos eran copias exactas de las originales, y por ello, al ejecutarse, se intentaba acceder a la función interna del código original (en el caso de llamadas a funciones) o a a la variable del código original (en el caso de las variables locales o globales) en vez de acceder a las de nuestro propio código. Teníamos que conseguir cambiar estas referencias al antiguo Ensamblado por otras nuevas a nuestro nuevo Ensamblado. La solución a primera vista parecía difícil y

estuvo a punto de hacernos creer que lo que intentábamos no era posible. Después de mucho tiempo intentando encontrar una solución, lo conseguimos:

Cada vez que encontrásemos una variable o un método debíamos guardarlos en una estructura que fuera accesible durante las emisiones de código y así redireccionar las llamadas a las creadas en nuestro nuevo Ensamblado. Es decir, necesitábamos una Tabla de Símbolos. Esta es la estructura de nuestra Tabla de Símbolos:

Variables locales	Nombre (Key)	Valor
Variables globales	Nombre (Key)	Valor
Métodos	Nombre (Key)	Valor
Constructoras	Nombre (Key)	Valor

En este punto, nuestra aplicación podía copiar programas algo más complejos como este:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace elespaciodenombres
{
    class programa
    {
        public static string factor;

        //Constructora
        public programa()
        {

        }

        //Función interna doble
        public static void doble()
        {
            Console.WriteLine("Empieza doble");
            Console.ReadLine();
            int a = 5;
            a = a * 2;
            Console.WriteLine(a);
            Console.ReadLine();
        }

        //Programa principal
        public static void Main(string[] args)
        {
            doble();
            factor="Fin";
            Console.WriteLine(factor);
            Console.ReadLine();
        }
    }
}
```

Como podemos observar, todavía quedaban por realizar copias de elementos importantes de la estructura, sin embargo, a partir de aquí nos situamos en un punto importante de la investigación, pues ya habíamos demostrado que nuestro proyecto era posible, pues partiendo del código ya compilado de una aplicación podíamos manipular la estructura y el código a nuestro antojo, y habíamos hecho una aplicación que lo ratificaba.

Nuestro siguiente objetivo fue ampliar los elementos copiados para poder manipular programas más complejos.

El primer paso fue lograr tratar las instrucciones de salto. Como ya habíamos comprobado, los saltos podían estar direccionados al Ensamblado original y si simplemente los copiábamos del Ensamblado original, no señalarían a nuestro programa. Sin embargo no fue así, realizando varias pruebas descubrimos que el compilador los trata como saltos relativos, por lo que también eran válidos desde nuestro Ensamblado. Así que implementamos los emit para las instrucciones de: *If-the-else y While*.

El siguiente paso fue conseguir el paso de parámetros a una función. Para eso primero necesitábamos volcar el valor de cada uno de los parámetros en la pila, para que posteriormente fueran recogidos por la llamada a la función. Eso fue fácil, pues ya habíamos implementado las instrucciones para tratar variables y tipos. Lo que llevó más tiempo fue copiar las características de los parámetros asociados a la función, es decir, cuántos son y de qué tipo.

Y por último tratamos el valor devuelto por una función. El valor se vuelca en la pila y para que pueda ser recogido al restablecer el contexto.

Lo único que queda pendiente fue la copia de las constructoras, sí se empezó, pero la falta de tiempo hizo que decidiésemos pasar a la siguiente fase y dejarlo en un segundo plano.

Veamos un ejemplo del tipo de programa que habíamos conseguido hasta el momento:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace elespaciodenombres
{
    class programa
    {

        public static string gv;

        //Constructora
        public programa()
        {
        }

        public static int f(int a)
        {
            Console.WriteLine(6);
            Console.ReadLine();
            int b = 0;
            int c = 0;
            a = a + 1;
            b = b + 2;
            c = c + 3;
            while (a <= 2){
                Console.WriteLine(a);
                Console.WriteLine(b);
                Console.WriteLine(c);
                Console.ReadLine();
                a++;
            }
            return a;
        }

        public static void p()
        {
            Console.WriteLine("p");
            Console.ReadLine();
        }

        public static void l()
        {
            Console.WriteLine("l");
            Console.ReadLine();
            p();
        }

        public static void n()
        {
            Console.WriteLine("n");
            Console.ReadLine();
            l();
        }
```

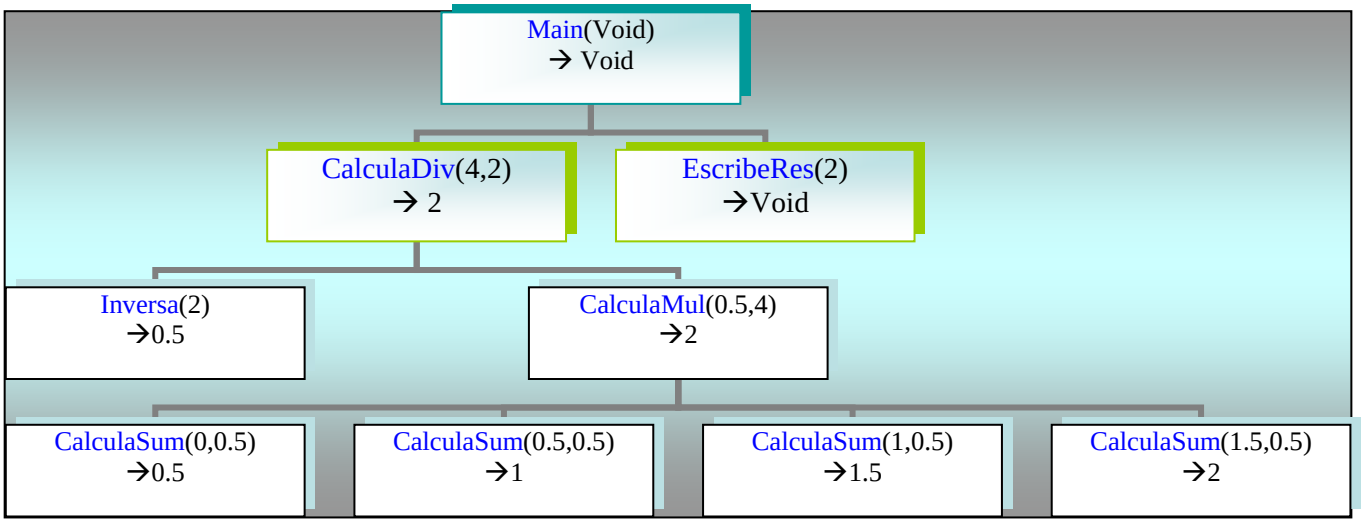
```
        public static void m()
        {
            Console.WriteLine("m");
            Console.ReadLine();
            n();
        }

      public static void k()
        {
            Console.WriteLine("k");
            Console.ReadLine();
        }

        public static void h()
        {
            Console.WriteLine("h");
            Console.ReadLine();
        }

        public static void g(int a)
        {
            if (a>0)
                Console.WriteLine("g");
            Console.ReadLine();
            h();
            k();
        }


  public static void Main(string[] args)
  {

  Console.WriteLine("Inicio programa");
  Console.ReadLine();

  gv = "variable global";
  Console.WriteLine(gv);
  Console.ReadLine();

  int j = f(0);
  int i = f(2);

  g(6);
  m();

  Console.WriteLine("fin");

  }

  }

}
```

Con esto dimos por finalizada esta la fase de copia del Ensamblado. La siguiente fase consistió en incluir código extra entre las instrucciones del programa original. Estas nuevas instrucciones generarían un fichero de texto en el que se al ejecutar el nuevo Ensamblado se copiaran los valores de los parámetros de entrada y de salida a cada función. Como ya llevábamos tiempo estudiando las instrucciones de código MSIL y cómo emitirlas, supimos resolverlo bastante rápido. Todo lo que hubo que hacer fue:

- Incluir instrucciones para crear un fichero de texto, "Ejecucion.txt", que incluimos en el programa como variable global y, por tanto, también en la Tabla de Símbolos.
- Añadir instrucciones al principio de cada función que graben en el fichero el valor de los parámetros a la entrada
- Añadir instrucciones justo antes de salir de la función (inmediatamente antes de la instrucción *ret* de código MSIL) que graben en el fichero el valor devuelto por la función.

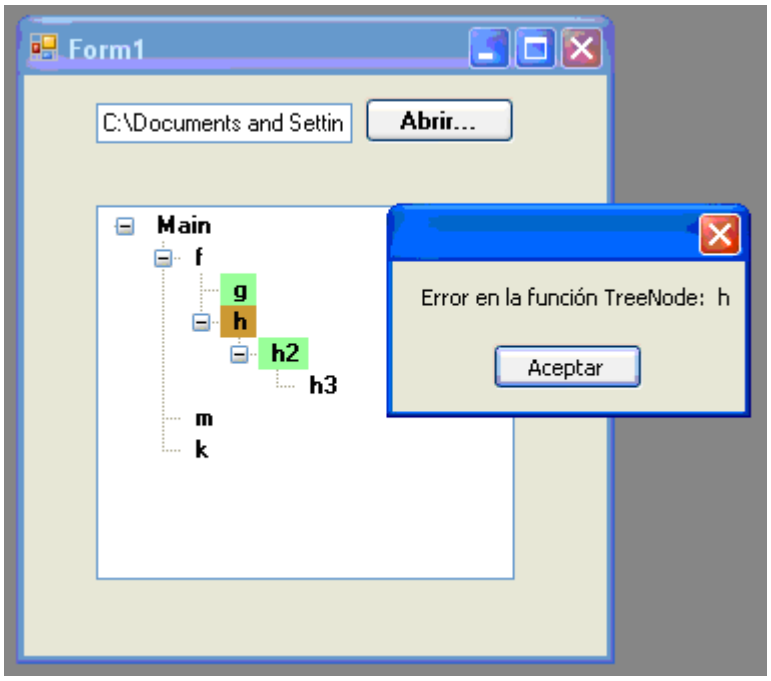
A partir de este fichero, ya podíamos crear un árbol con los resultados de toda la ejecución del programa del usuario.

El árbol creado posee como nodos las funciones y la información de los valores de entrada y salida para una ejecución concreta. Este árbol es una herramienta de depuración rápida y eficaz. El usuario puede observar en todo momento en qué orden se han ido realizando las llamadas, con qué parámetros y qué valores se han devuelto. El siguiente dibujo representa la información que contiene le árbol tras una ejecución de un programa sencillo al que se le pasan dos parámetros de entrada “4” y “2” en este caso, para calcular su división y mostrar el resultado por pantalla:



Además de mostrar la información del árbol, el siguiente paso fue utilizar esta información para implementar un sistema de localización de errores. Este sistema consiste en realizar una búsqueda a través del árbol guiada por preguntas hechas al usuario.

El usuario sólo tiene que seleccionar un nodo y el programa le preguntará por la validez de dicho nodo. Si el usuario responde que el nodo no es válido, no significa que este sea el nodo problemático, podría ser que simplemente esté arrastrando el error generado por uno de sus nodos hijos. La forma de conseguir detectar la función dónde está le fallo es relativamente sencilla. Para saber que una función es la causante del error, tan sólo hay que comprobar si todos sus hijos estén marcados como válidos, lo que significa que el error no existía hasta llegar a la función actual y que es en esta función dónde se ha generado. Si se detecta la función errónea automáticamente el sistema informa al usuario de qué función es la causante del error.



Y por último, la fase final del proyecto consistió en testear la aplicación para comprobar su correcto funcionamiento y en mejorar la interfaz gráfica para hacerla más sencilla e intuitiva.

Por fin, después de mucho esfuerzo y trabajo hemos conseguido crear un Depurador Declarativo para la Plataforma .NET capaz de ayudar en el desarrollo de aplicaciones en diversos lenguajes, y con el que no es necesario ejecutar todas las trazas para encontrar un error en una aplicación, pues el usuario tiene acceso a los datos de toda la ejecución al mismo tiempo y además cuenta un sistema capaz de detectar en qué función está error buscado.

5.Evaluación final del proceso de desarrollo

En este apartado se detalla cómo fue la planificación inicial que se pensó para llevar a cabo nuestro proyecto siguiendo el proceso unificado de desarrollo.

- **Inicio:**
  · Comienzo → Octubre de 2007
  · Final → Noviembre de 2007

  En este periodo decidimos las especificación del proyecto (qué queríamos conseguir), las tecnologías que vamos a usar para su implementación (Microsoft Visual Studio 2005 para desarrollar una aplicación que pudiera manipular código MSIL y así implementar un depurador multilenguaje) y nos decidimos por un lenguaje de programación que se adaptara a nuestras necesidades (C#).

- **Elaboración:**
  · Comienzo → Noviembre de 2007
  · Final → Diciembre de 2007

  Durante este periodo empezamos a diseñar los requisitos del proyecto, a especificar todos los detalles, y a crear la arquitectura base para el Depurador. Paralelamente todos los miembros del grupo continúan el estudio sobre las tecnologías que se van a usar y a investigar sobre el código MSIL y la Reflexión, realizando pequeños programas para su estudio.

- **Construcción:**
  · Comienzo → Diciembre de 2007
  · Final → Mayo 2008

  Se comienza a programar el Depurador. Se dedica mucho tiempo en cada paso porque se trata de un proyecto de investigación, es decir, hasta el momento no existe ningún depurador con estas características y cada avance implica antes mucho tiempo dedicado a la investigación.

- **Transición:**
  · Comienzo → Mayo 2008
  · Final → Junio 2008

  Durante este periodo se realizan pruebas y se crea una interfaz gráfica que facilite la utilización del Depurador al usuario.

6.Alternativas de desarrollo:

Al iniciar el proyecto teníamos claro que queríamos conseguir y cuál iba a ser el entorno de desarrollo, sin embargo no estaba tan claro cuál iba a ser el método en que nos íbamos a basar. Por ello, los primeros días los dedicamos a plantear diversas alternativas de desarrollo para lograr un Depurador Declarativo para la Plataforma .NET. Este apartado está dedicado a explicar por qué ciertas alternativas fueron descartadas y por qué nos decidimos por internarnos en el estudio del código intermedio de Microsoft, MSIL.

La alternativa más intuitiva era la de realizar una traducción fuente a fuente. Es decir, dado un programa desensamblar su código y generar otro código en el que además de incluir las instrucciones del programa original, se añadiera un nuevo parámetro asociado a cada método. Este nuevo parámetro representaría el árbol de depuración para el método. Este método es válido para cualquier lenguaje de la plataforma .NET, porque partimos del código ensamblado. Sin embargo, a pesar de esta ventaja, esta alternativa la descartamos la primera, debido a su que se trataba de un método muy poco eficiente. Para llevar a cabo la ejecución entera del Depurador habría que partir del código ensamblado, desensamblarlo, modificarlo, y más tarde volverlo a ensamblar.

La siguiente alternativa que nos planteamos estaba relacionada con manipular directamente el código MSIL. Podíamos fabricar el Depurador como un metaintérprete que ejecutase el programa y creara el árbol simultáneamente. El problema de esta opción es que el depurador necesitaría una gran cantidad de memoria para poder almacenar toda la información requerida al mismo tiempo. Por tanto esta alternativa también fue descartada al dar con otra mejor.

Si estudiábamos a fondo el código intermedio de Microsoft, no sólo podríamos interpretarlo y obtener información en tiempo de ejecución, podríamos hacer un sistema que utilizara código MSIL instrumentado para más tarde poder obtener información sobre lo sucedido durante la ejecución del programa a analizar.

Utilizando esta técnica podíamos implementar un Depurador con las siguientes ventajas:

El Depurador sería válido para cualquier lenguaje de la plataforma :NET.

Tendríamos acceso a la información necesaria sobre la ejecución incluso después de realizarse la misma.

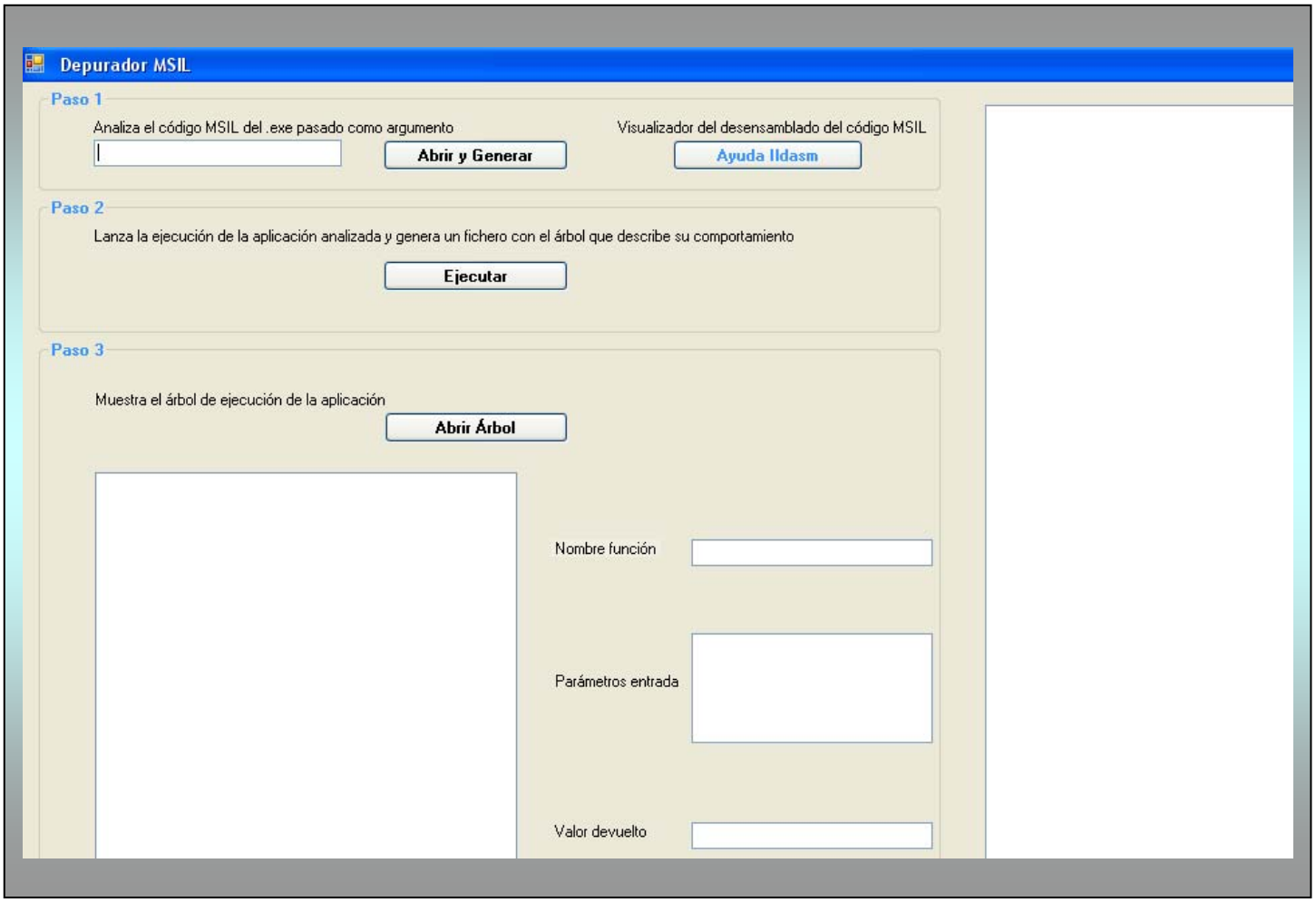
No consumiríamos grandes cantidades de memoria simultáneamente,

No necesitaríamos realizar ningún desensamblado.

Para realizar todo esto sólo teníamos que estudiar bien la forma de manipular el código MSIL (para poder emitir las instrucciones una a una según las necesitásemos) y la forma de funcionamiento interno del compilador de la plataforma .NET (para poder añadir código instrumentado entre las instrucciones sin afectar a la funcionalidad del código original).

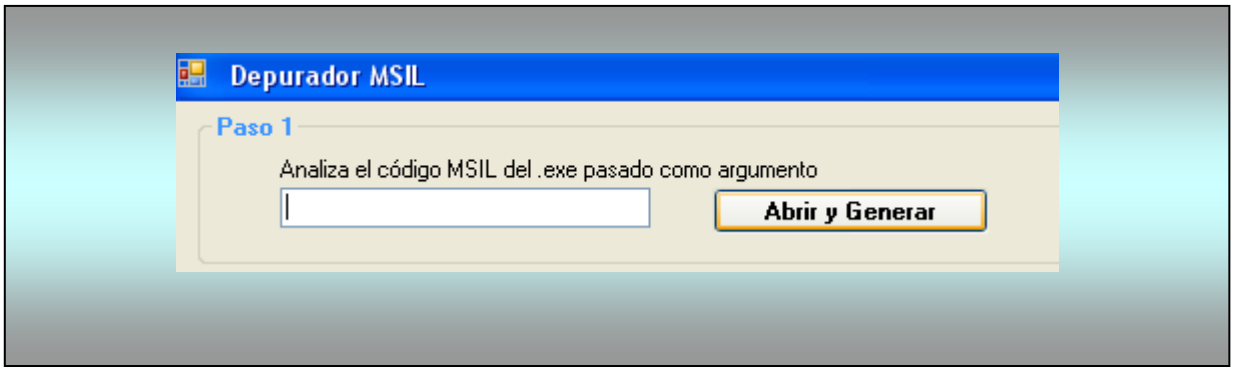
7.Manual de usuario:

En este apartado, explicaremos los pasos que debe seguir un usuario para poder realizar una depuración declarativa de su programa. En primer lugar, en la siguiente figura, mostramos el aspecto general que presenta la interfaz gráfica de nuestro proyecto.

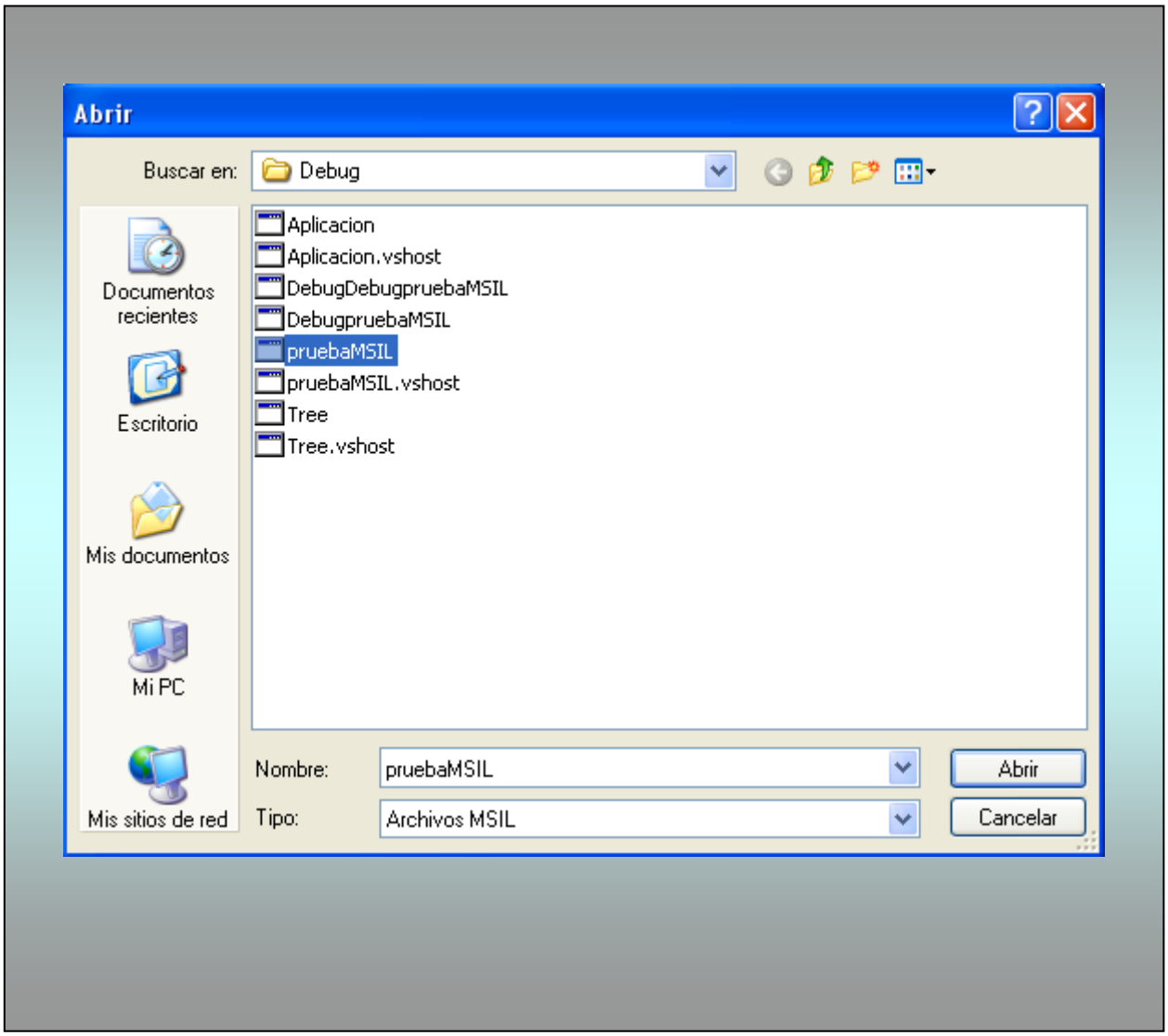


Los pasos que tendrá que seguir un usuario serán:

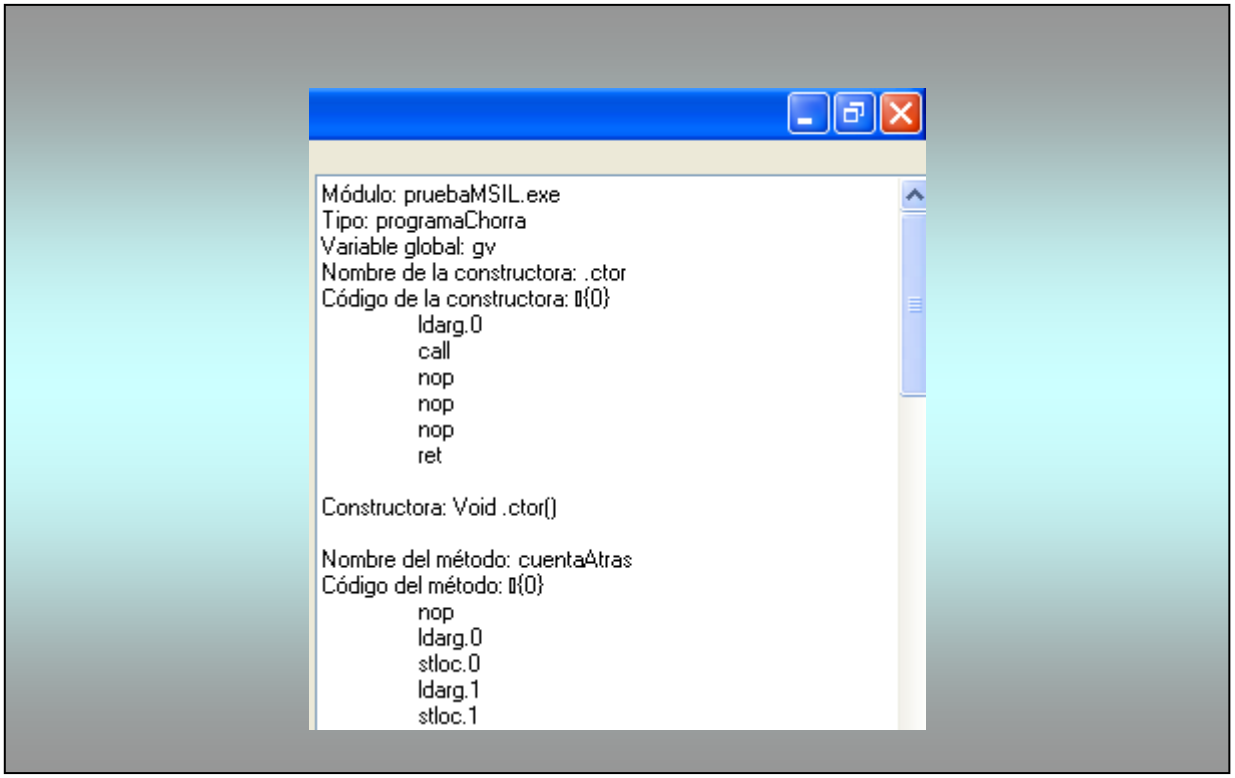
1º Seleccione el botón Abrir y Generar.



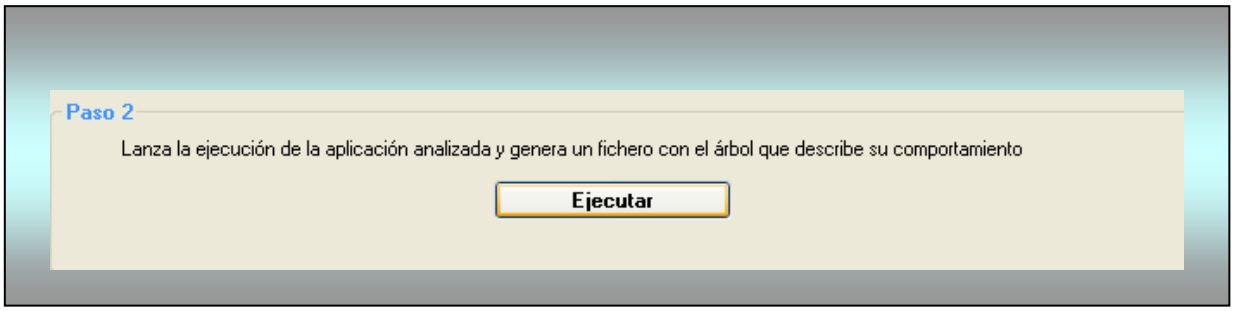
2º Seleccione el ejecutable del programa que quiere depurar.



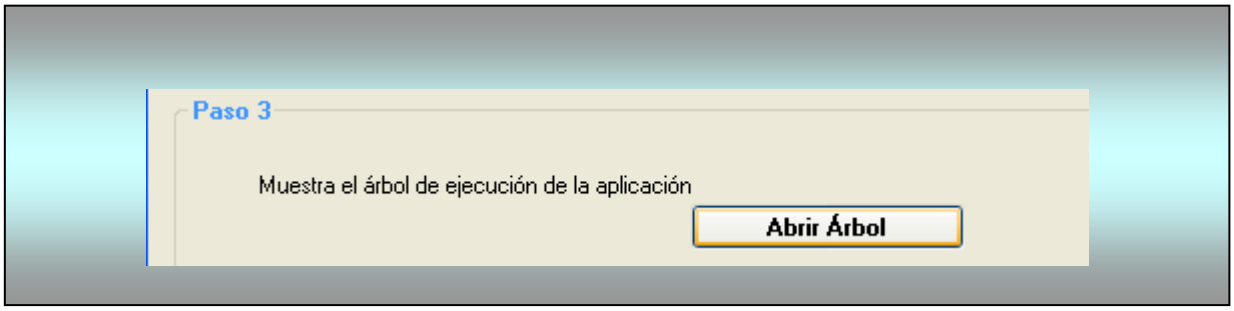
3º Una vez seleccionado el ejecutable, se realizará el análisis del código MSIL en el panel lateral derecho.



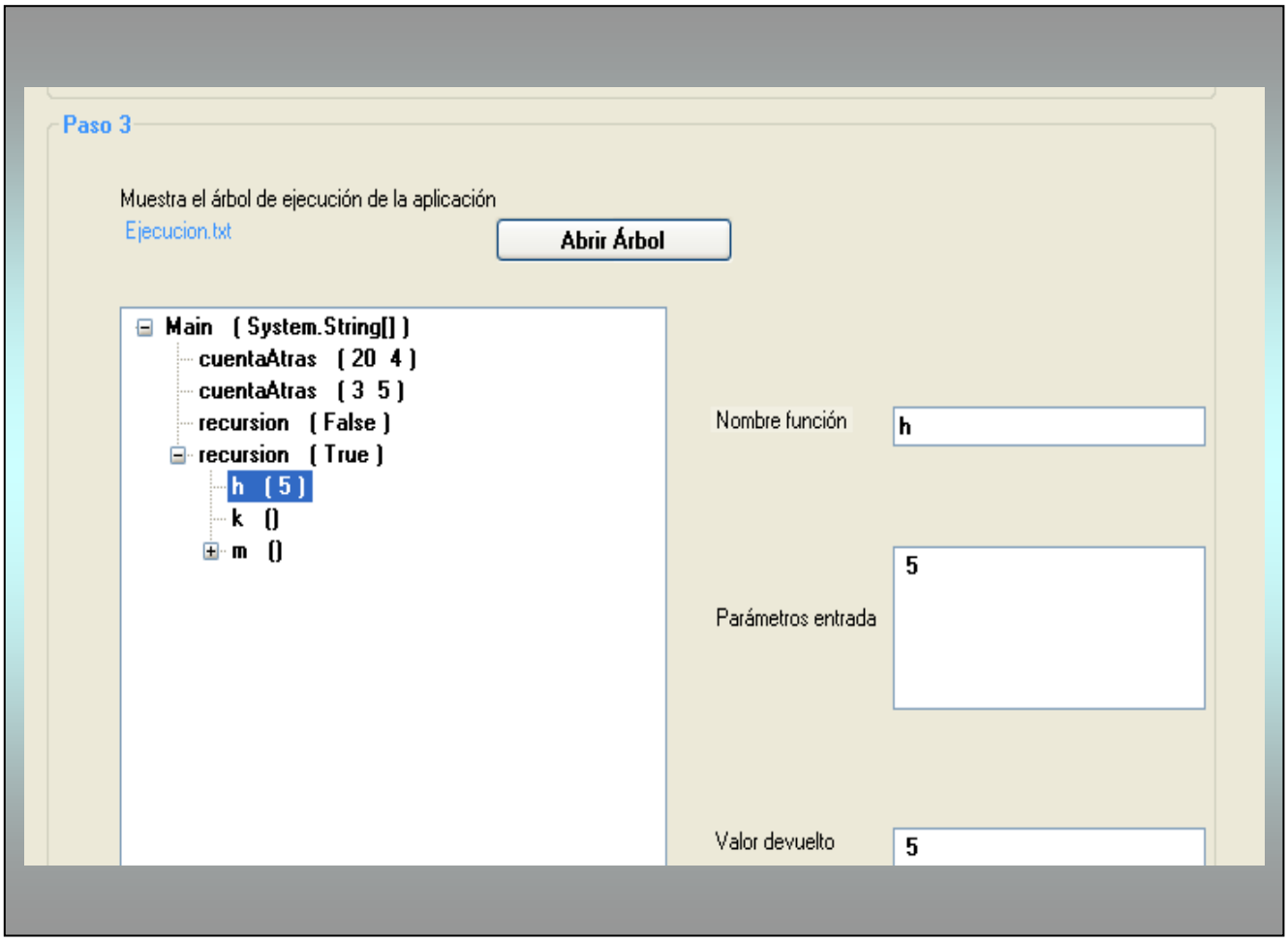
4º Seleccione el botón Ejecutar. En este momento se ejecutará el programa que quiere depurar y se generará automáticamente un fichero de texto con el árbol de cómputo de su programa. Dicho fichero de texto será abierto en el paso siguiente para visualizar el árbol de cómputo.



5º Seleccione la opción de Abrir Árbol. Al pulsar este botón se visualizará el árbol de cómputo del programa elegido para realizar la depuración declarativa.

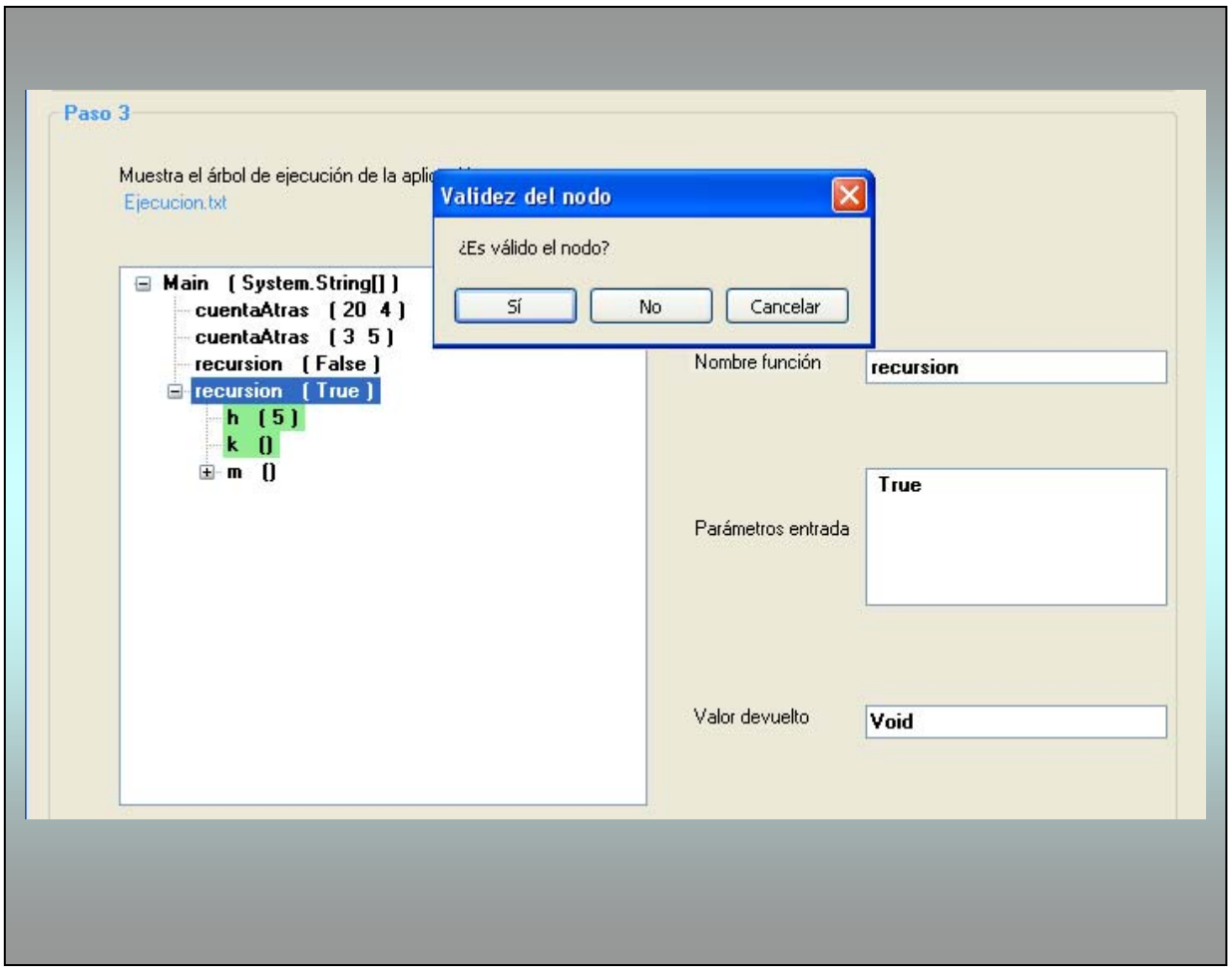


El resultado de pulsar el botón será la visualización del árbol mostrado en la siguiente figura.



6º Tras visualizarse el árbol de cómputo del programa, podrá navegar por él para poder realizar la depuración declarativa. Cuando haga doble click sobre un nodo, se le realizará una pregunta acerca de la validez del nodo. Distintas respuestas derivan en distintos casos:

-En caso de afirmar que el nodo es válido, éste aparecerá en color verde.
-En caso de negar que el nodo es válido, éste aparecerá en color rojo.
-En caso de cancelar, no se le asignará ni estado correcto ni estado erróneo, lo dejará sin asignar, dando igual si antes estaba seleccionado como nodo correcto o incorrecto.



8.Conclusiones:

8.1.Objetivos logrados

El objetivo de este proyecto ha sido crear un Depurador Declarativo Multilenguaje para la Plataforma .NET.

- El Depurador conseguido acepta aplicaciones creadas en cualquier lenguaje de la Plataforma .NET: C#, Visual Basic, C++, Fortran, Haskell, Java, Pascal, Perl, Phyton o Smalltalk.
- Las aplicaciones que acepta alcanzan una complejidad media: pueden incluir saltos, modulación en funciones, recursión o llamadas a librerías externas.
- El programa instrucmentado no requiere más memoria que el original y el tiempo no se verá apenas afectado, es decir es una solución escalable)
- A partir del código compilado el sistema genera un árbol con los valores generados durante la ejecución del programa. El usuario puede consultar cualquier valor en cualquier función sin tener que ejecutar paso a paso el código de su aplicación.
- Además el sistema cuenta con un sistema inteligente de detección de errores, por el cual, realizando preguntas al usuario, puede detectar en qué nodo se originó el error.

8.2.Limitaciones

Este Depurador está limitado a programas de complejidad media, pero está preparado para poder añadir mejoras sin tener que realizar cambios importantes en la arquitectura del programa. Las limitaciones de las que hablamos son:

- El orden de la declaración de las funciones es importante, no se puede llamar a una función que todavía no ha sido declarada. Esto es porque la aplicación funciona como un intérprete, emite cada línea de código una a una, y para emitirla necesita toda la información correspondiente a instrucción. Este problema de muy fácil solución, bastaría con realizar un recorrido previo por la estructura del programa antes de comenzar a emitir las instrucciones.
- El depurador no puede manejar objetos, esta limitación se intentó solucionar y se consiguió en parte, pero no pudimos acabar de adaptar el depurador al manejo de objetos por falta de tiempo, aunque sí conseguimos que aceptara la creación de objetos.
- No acepta programas no secuenciales. La idea en que se basa este Depurador es en incluir código instrumentado, por ello, si la ejecución no es secuencial puede haber problemas al incluir este nuevo código entre las instrucciones del programa.
- Tampoco acepta aplicaciones dónde se pueda dar una finalización no esperada. Es un caso muy parecido al anterior, si la ejecución deja de ser secuencial, por ejemplo al aceptar una interrupción, pueden darse resultados no esperados.

- Tampoco acepta programas no terminantes. El Depurador no sería capaz de terminar de ejecutar el análisis del Ensamblado.

8.3.Trabajo futuro

Pensamos que el proyecto desarrollado durante este tiempo ha cumplido sus objetivos, sin embargo, al internarnos más en el desarrollo del mismo hemos descubierto aspectos que podrían mejorar su funcionalidad.

Además de ampliarlo solventando algunas de las limitaciones antes comentadas, hay otros aspectos orientados a objetivos diferentes de la depuración, que podrían mejorar su funcionalidad. Estos otros aspectos están orientados a mejorar le rendimiento del Depurador al frente a grandes cargas de trabajo.

Por ejemplo, se podría asociar una Base de Datos al árbol de ejecución dónde se pudiera consultar el valor de los nodos sin necesidad de tener toda la información en memoria al mismo tiempo, así se mejoraría mucho el coste referido al consumo de espacio del navegador.Para efectuar esta mejora se necesitaría cargar el árbol de forma incremental. El usuario podría moverse por el árbol libremente, pero sólo se necesitaría guardar en memoria la información de los nodos de una pequeña porción del árbol.

Otra posible mejora sería incluir más tipos de estrategias para la localización de nodos críticos.

9.Bibliografía

Depuración
http://www.whyprogramsfail.com/

Depuración declarativa
[Shapiro 82] E.Y.Shapiro. "Algorithmic Program Debugging," ACM Distiguished Dissertation, MIT Press, 1982.

Depurador declarativo para Java
Algorithmic Debugging of Java Programs.
Rafael Caballero, Christian Hermanns, and Herbert Kuchen
WFLP 2006 - 15th Workshop on Functional and (Constraint) Logic Programming, Electronic Notes in Computer Science. Volume 177, (June 2007), ISSN:1571-0661.

JAVADD
JavaDD: A declarative debugger for Java (Hani Z. Girgis y Bharat Jayaraman)

MSIL y CLR
http://www.elguille.info/colabora/NET2005/Percynet_Modificando_codigo_msil.htm
http://msdn.microsoft.com/
http://www.vtortola.net/post/Modelo-de-ejecucion-del-NET-Framework-Parte-1.aspx
http://www.devjoker.com/asp/ver_contenidos.aspx?co_contenido=25

"El lenguaje de programación C#", por José Antonio González Seco.

MSIL:
http://gsyc.es/~agutierr/pfc-tecnica-html/node15.html
http://gsyc.es/~agutierr/pfc-tecnica-html/memoria.html
http://msdn.microsoft.com/es-es/library/c5tkafs1(VS.80).aspx
http://weblogs.asp.net/kennykerr/pages/My-Articles.aspx
http://www.codeguru.com/Csharp/.NET/net_general/il/article.php/c4635

CLR:
http://es.wikipedia.org/wiki/CLR

Ofuscadores:
http://www.elguille.info/colabora/NET2005/Percynet_Modificando_codigo_msil.htm

ILDasm:
http://www.codeguru.com/Csharp/.NET/net_general/il/article.php/c4635#more
http://msdn.microsoft.com/en-us/magazine/cc301368.aspx
http://msdn.microsoft.com/es-es/library/f7dy01k1(VS.80).aspx

Reflector:
http://aspnet.4guysfromrolla.com/articles/080404-1.aspx#postadlink

Reflexión:
http://msdn.microsoft.com/es-es/library/system.reflection(VS.80).aspx

Ensamblados:
http://msdn.microsoft.com/es-es/library/hk5f40ct(VS.80).aspx

Módulos:
http://msdn.microsoft.com/es-es/library/system.reflection.module(VS.80).aspx

Tipos:
http://msdn.microsoft.com/es-es/library/system.type(VS.80).aspx

Visual Studio 2005:
http://msdn.microsoft.com/es-es/vs2005/default(en-us).aspx

Página Web que explica cómo analizar el código IL de un método. Aquí encontramos el código del programa que hizo Sorin Serban y que utilizamos en nuestro proyecto:
http://www.codeproject.com/KB/cs/sdilreader.aspx

10.Palabras clave

- Depurador Declarativo
- MSIL
- Reflexión
- Árbol de cómputo
- .NET
- ILDasm
- Reflector

Derechos sobre el proyecto

Laura Alemany de Aguinaga, Víctor Manuel Arias Torrijos y Alberto Sánchez Carrillo, actuales alumnos de la Facultad de Informática de la UCM, autorizamos a la Universidad Complutense a difundir y utilizar, con fines académicos no comerciales, tanto la documentación del proyecto realizado como el código del mismo.

Fdo. Laura Alemany de Aguinaga

Fdo. Víctor Manuel Arias Torrijos

Fdo. Alberto Sánchez Carrillo