Universidad Autónoma de Querétaro

Facultad de Informática

Utilización de especificación formal para validar bloques elementales de lenguaje máquina: caso práctico procesadores x86

Tesis

Que como parte de los requisitos para obtener el grado de

Maestro en Sistemas de Información

Presenta

José Luis Carreño Arteaga

Santiago de Querétaro, 7 de octubre del 2011

Universidad Autónoma de Querétaro
Facultad de Informática
Maestría en Sistemas de Información Gestión y Tecnología

Utilización de especificación formal para validar bloques elementales de lenguaje máquina: caso práctico procesadores x86

TESIS

Que como parte de los requisitos para obtener el grado de

Maestro en Sistemas de Información

Presenta:
José Luis Carreño Arteaga

Dirigido por:
M.I.S.D Carlos Alberto Olmos Trejo

SINODALES

M.I.S.D. Carlos Alberto Olmos Trejo
Presidente

M.I.S.D. Juan Salvador Hernández Valerio
Secretario

M.S.I. Elisa Morales Portillo
Vocal

M.S.I. Ernesto Rubalcava Durán
Suplente

M. S. I. Sandra Patricia Arreguín Rico
Suplente

M.C. Ruth Angélica Rico Hernández
Director de la Facultad

Dr. Irineo Torres Pacheco
Director de Investigación y Posgrado

Centro Universitario
Querétaro, Qro.
Enero del 2012
México

Resumen

Esta tesis discute el diseño, arquitectura y aplicación de un esquema para convertir lenguaje binario a un nivel de abstracción más alto, permitiendo la creación de pruebas re-utilizables y robustas. El resultado es una serie de pasos a seguir para convertir de código máquina X86 a código de nivel intermedio. El proceso de transformación mencionado aquí es una etapa de la ingeniería inversa, la cual por fuerza es el primer paso del proceso de re-ingeniería. Las técnicas de de-compilación inicialmente fueron documentadas en [Cifuentes]. Los conceptos utilizados por [Cifuentes] están estrechamente relacionados con compiladores y teoría de la optimización. Han ocurrió muchos cambios en el campo de la construcción de compiladores desde la publicación de [Aho et Al]. Algunos de los cambios a la construcción de compiladores que merecen mención son: -DTD (Document Type Definition). Esencialmente, un DTD es una gramática libre de contexto extendida, -Grafos XML [Anders et Al], -el novel enfoque no-canónico para la construcción de compiladores [Schmitz 2005], [Schmitz, 2007]. Cuando el momento sea el adecuado, se abordará cada tópico y se propondrá un nuevo esquema que traduzca de código máquina x86 a Common Intermediate Language (CIL) [ECMA-335].

(**Palabras clave**: de-compiladores, código máquina, gramáticas)

Summary

This thesis discusses the design, architecture, and application of a schema to convert binary language to a higher level of abstraction, allowing for the creation of reusable, robust tests. The result is a series of steps to follow to convert from x86 machine code to intermediate level code. The transformations process as mentioned here is a stage of reverse engineering, which is necessarily the first step of the reengineering process. De-compilation techniques were initially documented in [Cifuentes]. The Concepts used by [Cifuentes] are closely related to compilers and optimization theory. Many changes have occurred in the field of compiler construction since the publication of [Aho et Al]. Some of the changes to compilers construction that deserve mention include: -DTD (Document Type Definition). Essentially, a DTD is an extended context-free grammar [Vianu], -XML Graphs [Anders et Al], -Novel approach to Non-Canonical compiler construction [Schmitz 2005], [Schmitz, 2007]. At the appropriate time, each topic will be addressed and a new scheme that translate from x86 machine code to Common Intermediate Language (CIL) [ECMA-335], will be proposed.

(**Key words**: de-compilers, machine code, grammars)

Agradecimientos

Es un honor y un privilegio dar las gracias a la gente que me apoyó para completar este trabajo y que me han asistido tan atinadamente durante el transcurso de mis estudios de pos-grado.

Quiero dar las gracias a mis padres y hermanos que nunca tuvieron duda de que podía realizar cualquier cosa.

Quiero agradecer especialmente a mi director de tesis MISD. Carlos Alberto Olmos Trejo, por su paciencia e impecable orientación.

También quiero agradecer a mis maestros, compañeros y amigos que de alguna forma han contribuido a mi formación. MSI Elisa Morales Portillo, MSI Ernesto Rubalcava Durán, MISD Juan Salvador Hernández Valerio.

…Construimos más computadoras para guardar más información, para producir más copias que nunca, pero nos comunicamos menos…

Sri Sathya Sai Baba

INDICE

INDICE DE CUADROS

INDICE DE FIGURAS

INDICE DE DEFINICIONES

I. INTRODUCCCION

La definición de un esquema en donde se especifique de manera formal la serie de etapas que se deben considerar para construir una herramienta que transforme el código máquina a código de un más alto nivel de abstracción con el propósito de realizar pruebas de software, es de importancia relevante en todo sistemas de software y particularmente vital en los sistemas de misión crítica.

Las justificaciones que menciona [Boyer] para realizar un análisis estático que asegure el correcto funcionamiento de código máquina se fundamenta en las siguientes premisas: -el desarrollo a alto nivel, tal como el uso de compiladores no está especificado de manera precisa-, -algunos compiladores producen código erróneo [Thompson]-, -los programas escritos en alto nivel, de todas formas requieren de código ensamblador y ninguna especificación de alto nivel hace clara la semántica de las instrucciones empotradas en el ensamblador. Los compiladores cada vez se están haciendo más complejos y es por este motivo de que son susceptibles a cometer más errores.

Como se menciona en el estudio exploratorio de [Villalobos et Al] el área de gestión y control de calidad en proyectos de software en México es relativamente nueva y las empresas si bien reconocen la importancia de la calidad, no se encuentran suficientemente preparadas para aceptar los nuevos retos que trae consigo y poner en práctica sus principios y técnicas.

Las mejores prácticas que permiten desarrollar una herramienta que pruebe el correcto funcionamiento de código máquina, se fundamentan en la construcción de de-compiladores. El trabajo pionero en este campo es [Cifuentes], que propone la serie de etapas que se deben considerar al construir un de-compilador.

Debido a que el presente trabajo considera un subconjunto del total de instrucciones de la arquitectura propietaria x86, es menester mencionar que en noviembre de 2006 Intel proporcionó un modelo de arquitectura utilizando prosa informal y el cual no

tenía soporte para realizar pruebas formales. Continuando con el desarrollo para formalizar su arquitectura, en agosto de 2007 presentó un modelo también en prosa informal pero esta vez se muestra el sustento para realizar pruebas formales.

II. LAS PRUEBAS Y SU CLASIFICACIÓN

La importancia de las pruebas

Las pruebas de software son una etapa importante en la Ingeniería de Software. Algunos esfuerzos han sido enfocados a desarrollar una especificación formal de lenguaje [Vadera et Al]. El análisis de requerimientos es una etapa en el desarrollo de software que proporciona información para su posterior uso en la fase de desarrollo de pruebas.

Con el advenimiento de lenguajes de especificación formal, se logró un avance debido a que dichos lenguajes proporcionaron un medio para verificar de manera automática la consistencia de un sistema de software. Los lenguajes de especificación formal, proporcionan también un marco de referencia para descubrir potenciales casos de prueba.

El principal problema con los lenguajes de especificación formal es al tratar de pasar los requerimientos los cuales se encuentran en un formato textual ambiguo. Cualquiera que sea la herramienta que sea utilizada para lograr este fin indudablemente fracasará debido a que es imposible construir una herramienta que elimine todas las inconsistencias del lenguaje natural.

Por otro lado, las consideraciones de tipo heurístico aunque son de gran ayuda en el aseguramiento de la calidad de software, proporcionan una aproximación y no pueden ser utilizadas para probar software de misión crítica.

El ciclo de vida de desarrollo de software no puede basarse en una única especificación de software (formal o informal). En la actualidad existe una clasificación exhaustiva de las pruebas que pueden ser utilizadas en cada una de las etapas de desarrollo de software [Utting et Al].

Una etapa desarrollo de software que obliga al uso de pruebas es el mantenimiento, debido a que cualquier software tiende a decaer en su funcionamiento si no se le es mantenido. Y es precisamente este punto que será abordado en la presente tesis. La utilización de un modelo estructural y estático que permita realizar pruebas en software de naturaleza binaria (código máquina).

Las pruebas y su especificación

Cuando se estudia el tema de pruebas en el desarrollo software debemos reconocer tres conceptos distintos: -las pruebas en sí mismo, el uso de especificaciones para la realización de esas pruebas y la instrumentación que se debe ser realizada para llevar a cabo las pruebas-.

La pruebas que permiten asegurar la calidad del software, La instrumentación que proporciona el medio y las herramientas para la realización de las pruebas y las especificaciones que define un alto nivel de abstracción para expresar lo que será probado. Las pruebas definen el qué será probado. La instrumentación especifica el *cómo, dónde* y cuándo será probado. Una especificación para realizar pruebas expresa también *cómo* realizar las pruebas. La utilización de una especificación es una etapa de la instrumentación.

Al desarrollar una herramienta para automatización de pruebas de software [Misruda et Al] sugiere considerar dos tipos de especificaciones. Una especificación gráfica informal y otra de carácter textual. La primera le permite al encargado de planear las pruebas interactuar con la herramienta. La especificación gráfica es desarrollada mediante una interface de usuario y es de carácter informal. Por otro lado, la especificación textual “testspec”, permite incrustar acciones semánticas necesarias en el proceso de prueba.

Instrumentación de las pruebas

El trabajo de [Misruda et Al] es interesante debido a que no únicamente propone una herramienta, también, plantea la forma en la cual se debe realizar la instrumentación. Como pasos de esa instrumentación se propone el flujo de acción de la figura 2.1.

La ingeniería inversa en la evaluación de software y como apoyo al mantenimiento de los sistemas de software.

El presente trabajo utiliza código de bajo nivel como lenguaje fuente para generar una especificación formal y otra semi-formal (de manera indistinta utilizaremos los términos semi-formal o informal). En el presente trabajo se utiliza el proceso de ingeniería inversa con el propósito de generar un alto nivel de abstracción que permita evaluar el software.

Dos puntos apoyan el uso de la ingeniería inversa: (1) Como se menciona en [Boyer et Al], -el desarrollo a alto nivel, tal como el uso de compiladores no está especificado de manera precisa-, -algunos compiladores producen código erróneo-, -los programas escritos en alto nivel, de todas formas requieren de código ensamblador y ninguna especificación de alto nivel hace clara la semántica de las instrucciones incrustadas de ensamblador- (2) El propósito de la ingeniería inversa es re-documentar y recuperar el diseño. Como lo menciona [Buchli], los sistemas que son desarrollados en un periodo de tiempo grande resultan difíciles de mantener.

Una solución para ahorrar costos es el mantenimiento y la re-ingeniería. El primer paso del proceso de re-ingeniería es la ingeniería inversa, debido a que es un proceso que pone en relieve la estructura del sistema y proporciona vistas a un más alto nivel de abstracción. El mantenimiento y la re-ingeniería permiten que los sistemas de software se mantengan funcionales. La ley de la entropía de software dicta que los sistemas tienden gradualmente a decaer en su calidad con el paso del tiempo si no son mantenidos y adaptados [Lanza].

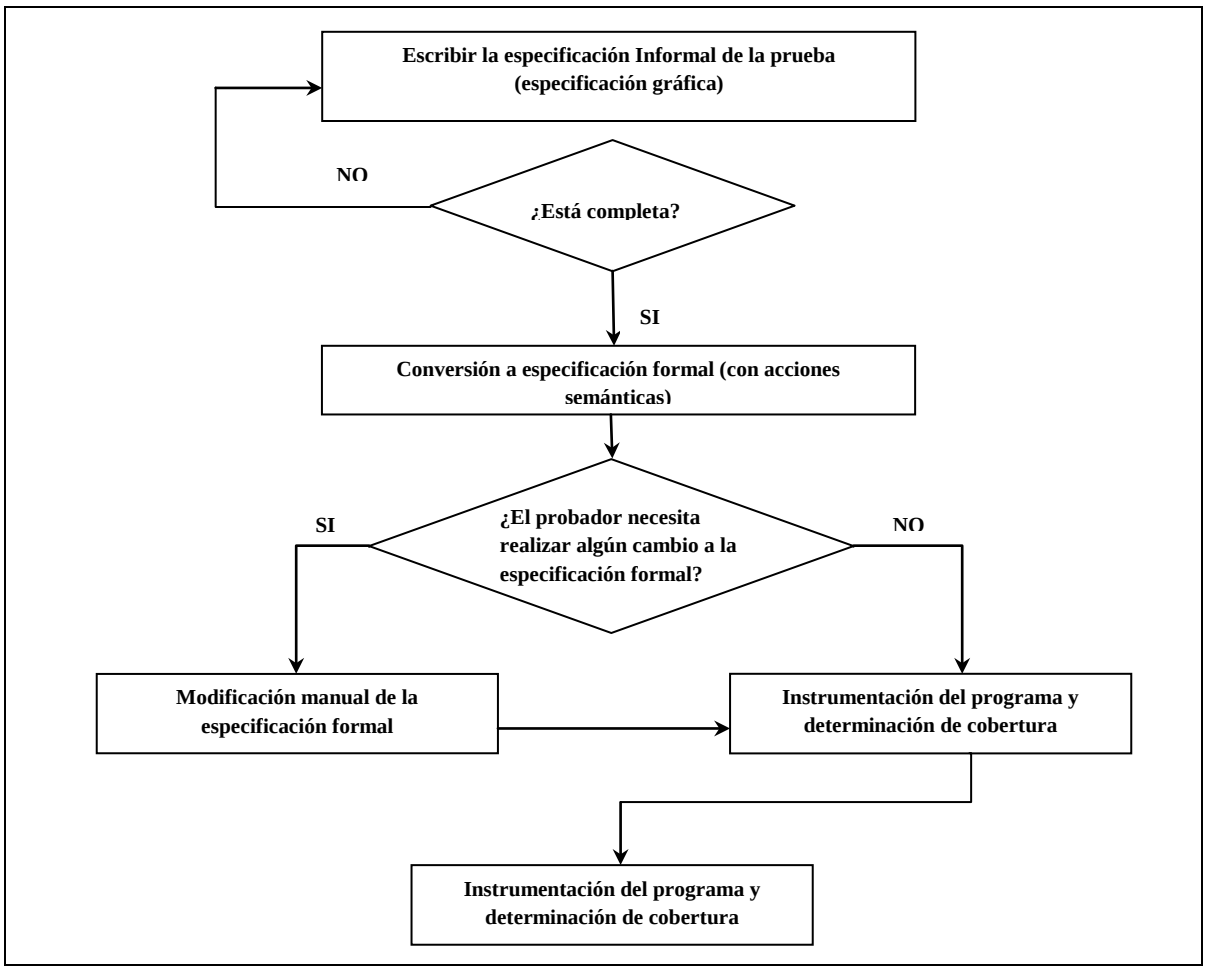

Fig 2.1 Instrumentación de las pruebas

Aunque resulta interesante (y puede ser tema de estudio en futuros trabajos), la construcción de especificaciones que permitan realizar el proceso de mantenimiento de manera eficiente, no es tema que será analizado en el presente trabajo. Y es en este punto donde acotamos que las especificaciones propuestas únicamente permitirán realizar pruebas sobre los componentes básicos de software.

Etapas que conducen a la prosecución de pruebas de software

En el trabajo de [Utting et Al] se menciona el uso de un modelo el cual será tomado como base para describir el sistema real, figura 2.2. Las pruebas fundamentadas en este modelo surgen de la necesidad de proporcionar una estructura coherente y bien

documentada para realizar la evaluación del software (validar el modelo para lograr la consistencia e integridad del sistema real). La serie de pasos o etapas que conducen a la prosecución de las pruebas de software son mencionados a continuación

- Construcción del modelo basado en los requerimientos

- Definición de los criterios prueba. Relacionada con la funcionalidad del sistema (criterios de prueba basados en requerimientos)

- Especificación de los casos de prueba formalizando de este modo la selección de los criterios de prueba. (posible uso de generadores de casos de prueba). La diferencia entre un caso prueba y las pruebas es que un caso prueba define la intencionalidad (la clase) mientras que las pruebas expresan una instancia

- Generación del conjunto de pruebas

- Los casos prueba son ejecutados respetando el conjunto de pruebas desarrolladas en el punto anterior. La ejecución aplica la parte concreta de la entrada del caso prueba y se registra la salida. La especificación de la entrada es realizado mediante el componente adaptador. Para poder ejecutar un caso prueba se utiliza un script de caso prueba. El adaptador es un concepto el cual puede ser traducido a una acción, política o regla de negocios (no necesariamente es software)

- Comparación de la salida con la salida esperada del caso prueba. El resultado puede ser: pasa, falla, no importa

Taxonomía de las pruebas

Para realizar un análisis cualitativo de las características de los métodos de especificación tanto formales como semi-formales (informales) es necesario clasificar las pruebas que se realizan durante todo el ciclo de desarrollo de software. En el trabajo

realizado por [Utting et al] se sugiere una forma de clasificar las pruebas basadas en el modelo del sistema. De manera literal, el concepto de taxonomía se encuentra expresado de manera clara en los trabajos de [Buckley et Al].

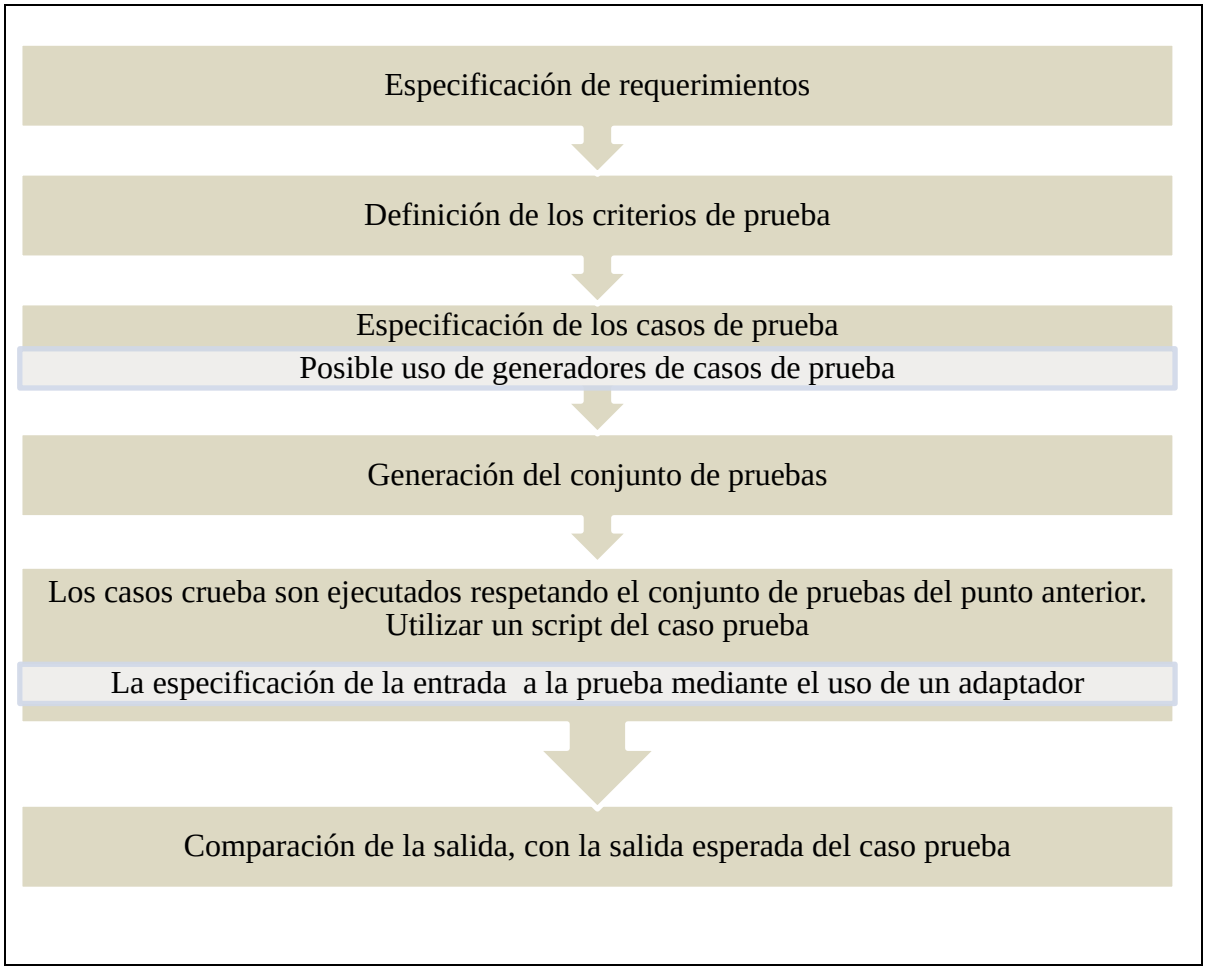

Fig 2.2 Esquema propuesto por [Utting et Al] para la realización de pruebas

Existen varias especificaciones que permiten modelar el comportamiento del sistema con el propósito de realizar pruebas [Utting et Al] sugiere la siguiente clasificación

a) Especificación basada en el estado (Pre/Post). Modela al sistema como una colección de variables las cuales expresan instantáneas del estado del sistema. Cada

operación en este tipo de especificación se encuentra limitada por una pre-condición y una post-condición. Como ejemplo mencionamos las notaciones VDM y JML

b) Especificación basada en la transición. Describe las transiciones entre los diferentes estados del sistema. La notación se expresa mediante un grafo. Mencionaremos como ejemplo las máquinas de estado finito. En esta especificación existe también la representación textual y en forma matricial (tal como las matrices de estado). Uno de los objetivos de tales especificaciones es mostrar la funcionalidad del sistema debido a su poder de expresividad

c) Especificaciones basadas en la historia. Modelan al sistema mostrando su comportamiento con el tiempo. Utiliza lógicas temporales. Existen notaciones de tipo gráfico y también textual en este tipo de especificación

d) Especificación funcional. Describe al sistema como una colección de funciones. Las especificaciones pueden ser expresadas en lógica de primero o de alto orden. Resultan más difíciles de escribir que otras notaciones

e) Especificación operacional. Procesos los cuales se desarrollan en paralelo. Utilizado en sistemas distribuidos y también en protocolos de comunicación. También las álgebras de procesos CSP o CCS y los esquemas gráficos tal como redes Petri deben ser incluidas en este tipo de especificación

f) Especificación estocástica. Describen al sistema mediante eventos probabilísticos. Como ejemplo podemos mencionar las cadenas de Markov

g) Notación de Flujo-de-datos. Su objetivo es modelar los sistemas continuos

Los modelos para la generación de código no siempre son lo ideal para la generación de pruebas. En este sentido, no existe la redundancia en este escenario y el sistema es probado contra sí mismo [Pretschner et Al]. A pesar de que este enfoque no es

adecuado para probar la funcionalidad del sistema, éste puede ser utilizado por los generadores de código y pruebas, pues refuerza la confianza sobre la hipótesis del ambiente en que se ha planteado el modelo.

III. SELECCIÓN DE LA PLATAFORMA DE DESARROLLO

El presente capítulo tiene un doble propósito. Por un lado, presentar la naturaleza y tipo de herramienta que será desarrollada y además mostrar la plataforma de desarrollo que fue seleccionada para construir el prototipo.

El de-compilador

La técnica mencionada en [Cifuentes] tiene como fundamento el área de compiladores y la teoría de la optimización.

Un de-compilador se compone de varias fases las cuales se pueden agrupar en módulos dependiendo de la arquitectura de la máquina en cuestión. La parte frontal de esta herramienta es una serie de módulo los cuales analizan el programa binario, la semántica de cada instrucción y generan una representación intermedia del programa que se encuentra originalmente en formato binario.

En el caso de un de-compilador, el avance parte de los nemónicos del lenguaje máquina (formato binario) y termina en la construcción de estructuras que definen un lenguaje de alto nivel.

[Cifuentes] menciona la generación de código intermedio que tiene como base una maquina universal. Por máquina universal, el autor se refiere un modelo de computadora tomando en cuenta instrucciones genéricas independientes a cualquier arquitectura real.

Un de-ensamblador es una etapa inicial que debe ser considerada en el proceso de múltiples pasos hasta llegar a la construcción de un de-compilador.

También como se menciona en [Cifuentes], un de-compilador puede ser llamado compilador inverso, pues intenta reproducir el proceso de compilación pero en forma inversa.

La escritura de un de-compilador es un problema que enfoca varios aspectos teóricos y también de carácter práctico. Estos problemas pueden ser resueltos por métodos heurísticos y otros no pueden ser completamente resueltos. Es por ello que la traducción se realiza únicamente para algunos programas fuente.

Si consideramos un programa en formato binario, la separación de los datos y de los programas es un problema que debe ser enfrentado al construir un de-compilador. Debido a que este es un problema parcialmente computable, se puede diseñar un algoritmo que separe datos de código, pero esto no será logrado en todos los casos.

Incluso en arquitecturas segmentadas tal como la Intel en donde los datos son almacenados en un segmento y el código en otro, las instrucciones pueden ser almacenadas en la forma de datos y posteriormente ser ejecutadas mediante interpretación. Como muestra basta un botón y podemos mencionar la técnica bien conocida por los hackers que explota la memoria de los datos locales almacenados en la pila para inyectar código la cual es denominada “shellcode”[Anley].

Problemas relacionados con la construcción de un de-compilador

Al construir un de-compilador se debe tener cuidado de algunos problemas relacionados con el código en cuestión.

- El código inyectado de manera maliciosa por algún virus. Este código puede encontrarse presente en forma cifrada. Y debido a que los virus son piezas pequeñas de código, un virus puede cifrar éste utilizando la técnica *xor*. Virus más avanzados utilizan el concepto de mutación polimórfica para cifrar el código

- Restricciones de la Arquitectura: utilización de instrucciones pre-fijadas. Una instrucción pre-fijada es almacenada en una localidad diferente a la que se encuentra en memoria principal. Las instrucciones pre-fijada son las que serán ejecutadas en turno. Estas instrucciones son difíciles de encontrar.

- Subrutinas incluidas por el compilador y eslabonador. Otro problema con el que tienen que lidiar los desarrolladores de de-compiladores son la inmensa cantidad de subrutinas que anexan los propios compiladores y eslabonadores.

Fases de de-compilación

La estructura que respeta un de-compilador es muy similar a las fases que deben ser consideradas para construir un compilador. Esta serie de fases transforman el código máquina del archivo fuente a otra de más alta abstracción. Las fases que se encuentran incluidas en un de-compilador son mostradas en la figura 3.1.

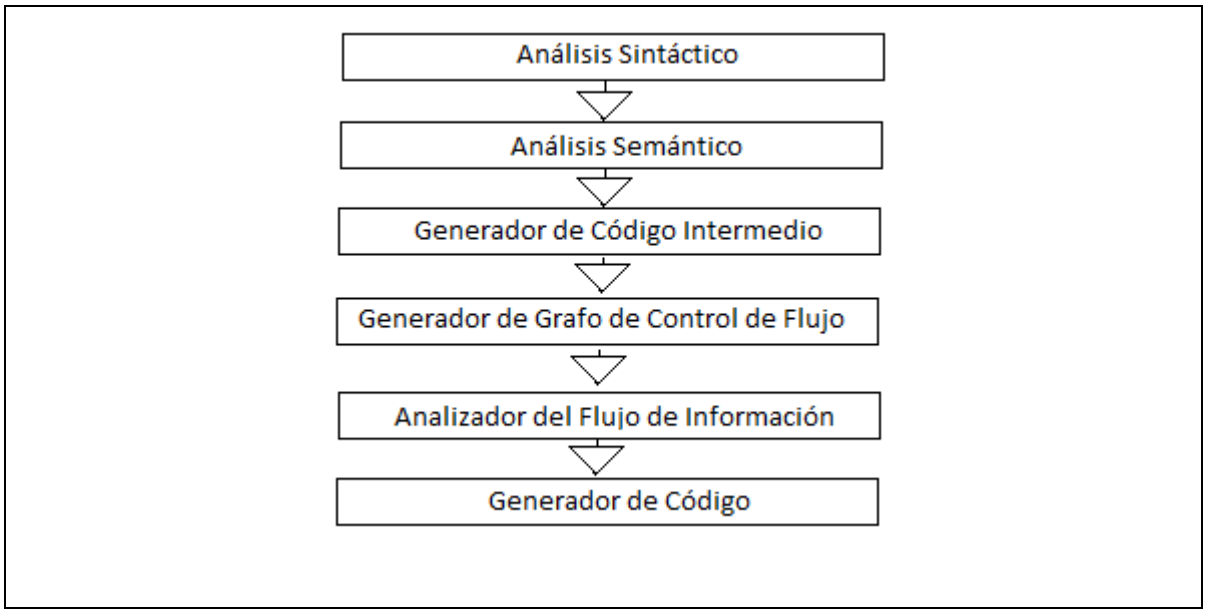

Fig 3.1 Fases de un de-compilador

Analizador sintáctico

El analizador sintáctico agrupa los bytes de un programa fuente en frases gramaticales o sentencias de lenguaje máquina [Cifuentes]. Por ejemplo, la expresión *sub cx, 50* es equivalente en un lenguaje de abstracción más alta a *cx-=50*. El principal

problema que se presenta al construir un analizador sintáctico es identificar si el símbolo que se está analizando es un dato o una instrucción.

En la construcción del prototipo se utilizó el formato COFF como base para eliminar tal ambigüedad.

Análisis semántico

En la fase de análisis semántico se determina el significado del conjunto de instrucciones, recopilando los tipos de información y la forma en que se propaga este tipo en la unidad de programación (función o subrutina). Como se menciona en [Cifuentes], es raro el caso que un programa binario contenga errores debido a que es código generado por compilador. Pero existe algo más, estos errores pueden ser disminuidos si el código generado se apega algún formato preestablecido.

La propagación de errores semánticos no son producidos por el compilador cuando se genera el código, sin embargo, se les puede encontrar cuando un programa es ejecutado en arquitecturas más avanzadas que la arquitectura que fue tomada como base (Ejemplo, se pueden notar algunas discrepancias entre la arquitectura i386 e i486).

Generación de código intermedio

Aunque se sugiere diseñar y generar código intermedio que idealice una máquina para realizar un de-compilador, este requisito debe ser superado para que la herramienta se apegue a la realidad. El código intermedio en el presente trabajo utiliza los nemónicos de del lenguaje ensamblador Intel x86, aunque también existe una fase posterior en la cual se utiliza el lenguaje CLI para instrumentar las pruebas.

De-compilador que produce lógica para pruebas de software

El enfoque propuesto por [Mayreen et Al] sugiere utilizar las mismas técnicas de construcción de un de-compilador para crear una herramienta que automatice las pruebas de código máquina. Este trabajo considera tres puntos

a) No se presuponen consideraciones de simplificación

b) No se requiere conocimientos del modelo

c) Se permite la reutilización de pruebas entre arquitecturas

El trabajo antes mencionado indica que dada una secuencia de instrucciones en lenguaje máquina el de-compilador crea una cola de funciones y prueba algunos teoremas indicando que la función creada describe de manera exacta el comportamiento del código máquina dado. De esta forma se ocultan detalles irrelevantes de la especificación del lenguaje máquina. La notación utilizada para la formalización de pruebas en este trabajo, respeta las tripletas de Hoare `{p}c{q}`.

La plataforma de desarrollo

La plataforma que fue seleccionada para desarrollar el presente trabajo es Common Language Runtime (CLR) de Microsoft. Dicha selección se sustenta en los siguientes puntos

- Proporciona apoyo para una amplia variedad de lenguajes de programación [Hamilton, Jennifer]

- Debido a su esquema de virtualización, es una plataforma que tiene el nivel de aislamiento requerido para desarrollar un plan de pruebas de software [Jeanna]

- Proporciona el apoyo necesario para el desarrollo de nuevos lenguajes [Hamilton et Al]

- Integra lenguajes que utilizan el paradigma funcional para filtrar información desde diferentes orígenes de datos [Chaudhari et Al]

CLR Soporta una amplia variedad de lenguajes de programación [Hamilton]. CLR es una plataforma de desarrollo y pruebas la cual integra un recolector de basura, un cargador de clases, un motor de metadatos y servicios de depuración y seguridad.

Todos sus servicios se encuentran ampliamente documentados en el estándar [ECMA-335]. Los desarrolladores de lenguajes de programación y de construcción de prototipos, se pueden concentrar en tareas más importantes de "parsing" y generación de código en lugar de pensar en la construcción de una plataforma para probar su producto.

También proporciona servicios para medir el desempeño de un programa considerando componentes de seguridad.

Su motor de metadatos fue diseñado para proporcionar un esquema independiente del lenguaje. Cuando algún programa es compilado con la herramienta CLR, la información que describen a las clases contenidas en el programa son emitidas como metadatos mediante el motor de metadatos y son almacenadas como el programa resultante mediante Common Intermediate Language(CIL).

Entendemos como aislamiento la característica de un sistema que no acepta interferencias externas [Wahbe et Al]. Esta característica es deseable cuando se realizan pruebas de software y es por ello que para lograrlo, se utilizará un entorno virtual.

Los entorno virtuales según [Jeanna et Al] pueden ser tipificados bajo las siguientes clases

- Virtualización completa

- Para-virtualización

- Virtualización a nivel sistema operativo

La virtualización completa ofrece aislamiento en todos los casos. La para-virtualización ofrece un beneficio cercano sin degradación. La virtualización a nivel sistema operativo es variado y muestra la complejidad de lograr el aislamiento en todos los recursos en un sistema fuertemente acoplado.

Es de interés y para propósitos del desarrollo de este trabajo, la virtualización a nivel sistema operativo. Cabe añadir, que los tipos de aislamiento tratados en [Jeanna et Al] por su consistencia en pruebas, consideran los asilamientos de memoria, de procesos, de uso de CPU, uso de disco y uso de red.

El desarrollo de nuevos lenguajes sobre la plataforma CLR también está ampliamente documentado y probado según las referencias que podemos encontrar en el documento de [Hamilton]. También esta característica es importante para desarrollar prototipos como el que se especifica en el presente trabajo.

Para seleccionar información desde diferentes orígenes de datos, CLR cuenta con lenguajes los cuales pueden ser clasificados dentro del paradigma funcional. El filtrado de información es necesario en esquemas de pruebas de software [Chaudhari et Al].

¿Por qué no se utilizó el lenguaje Java? (Comparación de CLR con Java)

A parte de disponer de una plataforma que permite el aislamiento con el propósito de realizar pruebas de software, la infraestructura que proporciona la maquina virtual CLR es única para desarrollar compiladores, como se menciona en el trabajo de [Meijer et Al]. La forma de realizar las pruebas de software es, desarrollar una serie de transformaciones de código fuente a un código intermedio en el cual se aplicarán dichas pruebas. La facilidad con la que se realizan estas transformaciones juega un papel importante para construir una plataforma de lanzamiento de pruebas estable.

Los puntos en los que sustentan sus afirmaciones [Meijer et Al] son siete

- **Portabilidad**. Se requieren de menos traductores para implementar tales lenguajes en diferentes plataformas.

- **Código intermedio compacto**. Es una propiedad que adquiere de facto la plataforma CLR debido a los años de trabajo acumulados en los laboratorios de investigación de Microsoft.

- **Eficiencia**. Se adapta de manera única a la plataforma nativa (Intel X86).

- **Seguridad**. El código intermedio resultante, obliga a imponer restricciones de seguridad y tipificación.

- **Interoperabilidad**. La interoperabilidad radica en que la intercomunicación se logra en un más alto nivel de abstracción que únicamente utilizar código binario.

- **Flexibilidad**. Conceptos tales como, tipificación segura, reflexión, generación de código dinámico, seriación y navegación en tipos son logrados mediante el uso de metadatos.

Por otro lado, JVM no proporciona una forma de utilizar estructuras de tipos inseguros, tal como punteros, descriptores inmediatos y conversión de tipos insegura. JVM carece de primitivas para desarrollar tipificación de lenguaje que no sea el propio lenguaje Java. Tipos tales como estructuras o uniones, mecanismos de retorno de múltiples valores, punteros a funciones y equivalencia estructuras de tipos. (Se menciona únicamente las estructuras no existentes en la actualidad.)

Resumen de la arquitectura del lenguaje CLI

CLI permite la administración de múltiples hilos de ejecución. Un hilo puede ser analizado como una lista enlazada de registros de activación [Grune et Al]. Los registros de

activación son creados cuando se llama a algún método o función y removidos cuando el método llamado finaliza. La máquina virtual considera los siguientes componentes

Un puntero a la siguiente instrucción la cual será desarrollada y que apunta a la siguiente instrucción CLI presente en el método

Una pila de evaluación conteniendo valores intermedios resultantes de los cálculos realizados dentro del método.

Un arreglo de variables locales (con inicio en índice 0). Cada variable en particular debe usar el mecanismo de consistencia de tipos.

Un arreglo de argumentos de entrada (con inicio en índice 0). El arreglo de argumentos y el de variables son diferentes.

Una estructura para almacenar la información del método en ejecución. La información almacenada en este receptáculo es la firma del método, sus tipos de variables locales y sus datos relacionados con el manejo de excepciones.

Un contenedor de memoria local el cual es utilizado por CLI para asignaciones dinámicas de objetos [Aho et Al]

Mecanismo para administrar el estado de retorno, el cual permite restablecer el estado del método llamador. (Eslabonamiento dinámico, según la terminología de construcción de compiladores)

Un descriptor de seguridad. Almacena las modificaciones relacionadas con la seguridad (de imposición, por permiso y de negación).

IV. FUNDAMENTACIÓN TEÓRICA

En el presente capítulo se hará mención a la teoría formal que sustenta cada una de las fases de desarrollo de la especificación. Esta serie de transformaciones utilizan como apoyo en lo general la teoría formal del lenguaje y algunas técnicas de de-compilación anteriormente mencionadas y las cuales se encuentran expresadas en los trabajos de [Cifuentes].

El código intermedio generado mediante el novel esquema NLALR(1), que aún se encuentra en desarrollo y es el estado del arte en el desarrollo de compiladores.

Como último punto en el presente capítulo, será abordado el tema de expresiones lambda para búsqueda de información en los nodos de un AST(Abstract Sintax Tree).

Gramáticas libres de contexto

Debido a que en el presente trabajo se usará el concepto de gramática libre de contexto, es pertinente que se mencione la nomenclatura utilizada figura 4.1 y también se debe mencionar la definición formal de gramática libre de contexto (CFG).

(V, P) es un sistema de re-escritura en donde el conjunto finito V expresa el vocabulario y P el conjunto de producciones P o reglas definidas sobre $V^* \times V^*$ y sus elementos son expresados como $\alpha \longrightarrow \beta$ para sistemas generativos (gramáticas) o bien $\alpha \vdash \beta$ en *sistemas* de reconocimiento (autómatas).
$\Rightarrow$ es la relación de derivación de un sistema generativo sobre $V^* \times V^*$, definido por δ y σ en V^* y $\alpha \longrightarrow \beta$ en P por $\delta\alpha\sigma \underset{\Rightarrow}{\overset{\alpha \longrightarrow \beta}{}} \delta\beta\sigma$; de manera similar se puede aplicar el símbolo $\vDash$ para sistemas de reconocimiento
$g = \langle N, T, P, S \rangle$ es una gramática libre de contexto (CFG), en donde $N, T\ y\ S$, son el conjunto de símbolos no terminales, el conjunto de símbolos terminales y el símbolo inicial en N; el sistema generativo considera $V = N \cup T$ y P está restringido a $N \times V^*$
El lenguaje generado por la CFG g es $L_g = \{x \mid S \Longrightarrow^* x\}$
El símbolo $\underset{rm}{\Longrightarrow}$ define una derivación por símbolo más a la derecha y $\underset{lm}{\Longrightarrow}$ derivación por símbolo más a la izquierda
La gramática es aumentada con la regla $S' \rightarrow S\$$ en donde \$ expresa el fin de las entradas y S' es el nuevo símbolo de inicio y se anexan las reglas $T' = T \cup \{\$\},\ N' = N \cup \{S'\}$ y $V' = T' \cup N'$
Las primeras letras mayúsculas del alfabeto $A, B, C, \cdots$ expresan no terminales en N; las primeras letras del alfabeto $a, b, c \cdots$ definen los símbolos terminales en T, las últimas letras minúsculas del alfabeto $u, v, w \cdots$ denotan cadenas en T^* , las últimas letras mayúsculas del alfabeto X, Y, Z denotan símbolos en V, los primeros símbolos del alfabeto griego $\alpha, \beta, \gamma, \cdots$ indican cadenas de símbolos en V^*, q denota un estado canónico en LR(0); s denota un estado no canónico
ε es una secuencia vacía de símbolos
$\lvert\alpha\rvert$ es el tamaño de una cadena α siendo $\lvert\varepsilon\rvert = 0$
$k{:}\,\alpha =$ es el prefijo de tamaño k de una cadena α

Fig 4.1 Nomenclatura utilizada para gramáticas

Definición 1 Gramática Libre de Contexto

[Aho et Al] Una gramática libre de contexto (CFG Context-Free Grammar) se encuentra definida por el cuádruplo ($\boldsymbol{V_t},\ \boldsymbol{V_n},\ \boldsymbol{S}, \boldsymbol{P}$) en donde

a) $\boldsymbol{V_t}$ es un vocabulario finito de terminales

b) $\boldsymbol{V_n}$ es un conjunto de símbolos intermedios diferentes, llamado el vocabulario de los no terminales $\boldsymbol{V_n}$

c) S es el símbolo de inicio el cual $\boldsymbol{S} \in \boldsymbol{V_n}$. Algunas veces este símbolo es llamado símbolo objetivo

d) P es un conjunto finito de producciones de la forma $\boldsymbol{A} \longrightarrow \boldsymbol{X_1} \cdots \boldsymbol{X_m}$, y

$$\boldsymbol{A} \in \boldsymbol{V_n},\ \boldsymbol{X_i} \in \boldsymbol{V_n} \cup \boldsymbol{V_t}, \boldsymbol{1} \leq \boldsymbol{i} \leq \boldsymbol{m}, \boldsymbol{m} \geq \boldsymbol{0}$$

Se debe notar que $\boldsymbol{A} \longrightarrow \boldsymbol{\varepsilon}$ es una producción válida

El vocabulario de la gramática libre de contexto es el conjunto de los símbolos terminales y no terminales ($\boldsymbol{V_t} \cup \boldsymbol{V_n}$). El conjunto de cadenas derivables a partir del símbolo de inicio S comprenden el lenguaje libre de contexto de la gramática.

La definición 1 expresada anteriormente fundamenta la información de los próximos párrafos y da el sustento a algunas de las siguientes definiciones.

La representación intermedia XML y los DTD

Las representaciones intermedias son un paso obligatorio para construir compiladores y herramientas de transformación como se menciona en el documento

[Grechanik et Al]. Existen varias notaciones que permiten manipular estructuras jerárquicas. Este problema se extiende y va más allá de la construcción de compiladores. Por ejemplo, la descripción de grafos dirigidos transformados a estructuras de árbol. En [Grechanik et Al] se ofrece un lenguaje para manipulación de Árboles denominado TML (por siglas en inglés) que permite tratar las estructuras jerárquicas como si fueran arreglos multidimensionales.

En el trabajo de [Grechanik et Al] se propone TML (Tree Manipulation Language) como la forma concisa y elegante para tratar con estructuras multidimensionales.

Aunque no existe una definición algorítmica clara para este menester y tal representación es considerada el estado del arte en los esfuerzos de diseño y desarrollo.

Esencialmente un DTD (Document Type Definition) es una gramática libre de contexto-. Los no-terminales de la gramática son las etiquetas de los elementos en el árbol etiquetado en el documento XML. No existen símbolos terminales. [Vianu]

Definición 2 Document Type Definition (DTD)

Sea **Σ** un alfabeto finito de etiquetas. Un **DTD** consiste de un conjunto de reglas de la forma $\boldsymbol{e} \longrightarrow \boldsymbol{r}$ en donde $\boldsymbol{e} \in \boldsymbol{\Sigma}$ y **r** es una expresión regular sobre **Σ**. Existe una regla para cada $\boldsymbol{e}$ y el **DTD** también especifica la etiqueta de la raíz. Un documento XML satisface un **DTD** si es una derivación de la gramática extendida libre de contexto.

AST y la tabla de símbolos

Las posibles representaciones internas para análisis de flujo de datos son los Árboles sintácticos abstractos (AST), Grafos a-cíclicos dirigidos (DAG), Grafos de flujo de Control (CFG), Grafos de flujo de programa (PFG), multígrafos de llamadas (CG), Grafos de Dependencia del Programa (PDG), Asignación estática simple (SSA). Aunque los AST pueden ser utilizados para análisis de control de flujo, no exhiben control de flujo de manera explícita. [Khedker et Al].

AST (Abstract Syntax Tree) es la estructura de datos más importante en el desarrollo de un compilador o un de-compilador. Contiene la información de los identificadores, así como sus tipos, sus valores iniciales y su alcance (sensibilidad de contexto).

En años recientes se han realizado algunas investigaciones en el campo de los compiladores con el objeto de construir componentes re-usables y en todas estas investigaciones [Grechanik et Al] el componente que une todas las piezas es una representación intermedia.

El flujo

Como fue mencionado en el capítulo 3, un de-compilador debe considerar la construcción de un generador de grafos de control de flujo. [Cifuentes]

De manera ordinaria, se utiliza una matriz para representar un grafo. Una matriz se llama rala si muchos de sus casilleros tienen un valor de cero. Este tipo de estructura es utilizada para aprovechar de manera más eficiente el espacio. Una matriz que ocupa todos sus espacios incluyendo los casilleros con valor cero recibe el nombre de matriz densa. La estructura matriz rala ahorra espacio debido a que únicamente son almacenados los valores diferentes de cero.

Una formato que utiliza vectores para almacenar matrices ralas se le describe como formato de almacenamiento ordenado por compresión de renglón (**CSR** row-compressed sorted storage) [Kebler et Al] y más recientemente [Khalili et Al]

En la estructura CSR la adyacencia de un grafo con n vértices y m arcos es representada utilizando dos arreglos con $xadj$ de tamaño n+1 y una $adjncy$ de tamaño $2m$ (esto se debe a que cada arco entre los vértices (v,u) y (u,v) es almacenado). Para los grafos de control de flujo de código máquina es posible tener tanto (v,u) como (u,v), según se aprecia en el ejemplo de la figura 4.2

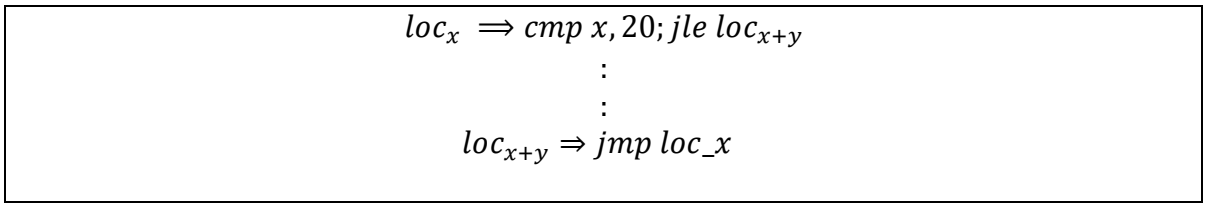

Fig 4.2 Lazos infinitos generan lazos $(u, v), (v, u)$

Grafo de control de flujo (control-flow graph CFG)

Un grafo de control de flujo (control-flow graph CFG) es una estructura la cual es utilizada por algunos compiladores con propósitos de optimización. Los CFG permiten realizar de manera eficiente el análisis de tipo estático. La ramificación y la detección de la primera y última instrucción pueden complicar el seguimiento del flujo basado en un CFG.

El algoritmo base puede ser explicado en dos pasos [Cooper et Al].

Dividir el código en un conjunto de bloques básicos (secuencias de longitud máxima de código completamente lineal). Esto son los nodos en el CFG.

Analizar las ramificaciones de código y las coloca en los arcos del CFG para representar el control de flujo.

Complejidad del algoritmo CFG

Para el primer paso se examina instrucción por instrucción tomando un tiempo $\boldsymbol{O(1)}$, esto es, que tomará un tiempo para $\boldsymbol{i}$ instrucciones de $\boldsymbol{O(i)}$.

Para el segundo paso se examina la instrucción también una a la vez. Esto toma un tiempo de $\boldsymbol{O(1)}$. Pero para las ramas se debe agregar $\boldsymbol{j}$ arcos, en donde $\boldsymbol{j}$ es el número de potenciales ramificaciones –también llamado “factor de ramificación”. Entonces, el tiempo para el segundo punto es $\boldsymbol{O(i + j \cdot b)}$, siendo $\boldsymbol{b}$ el número de ramificaciones. Para el caso

de la presente investigación se tienen saltos definidos, por lo que $\boldsymbol{b}$ es dos y entonces la etapa dos tiene un tiempo de $\boldsymbol{O(i)}$.

CFG y la representación intermedia XML

Para realizar el análisis estático de un programa se requiere una representación intermedia que proporcione una aproximación en ciertos puntos del programa. [Anders et Al] propone un modelo formal para realizar un análisis estático de un programa a partir de una serie de documentos XML o de fragmentos de ellos.

Definición 3 Grafo XML

Un grafo XML, $\boldsymbol{\chi}$ en un quíntuplo

$$\boldsymbol{\chi = (N, R, contenidos, cadenas, vacíos)}$$

El conjunto finito $\boldsymbol{N = N_E \cup N_A \cup N_T \cup N_S \cup N_C \cup N_I \cup N_G}$ consistente de nodos de diferentes tipos de conjuntos: Nodos elemento ($\boldsymbol{N_E}$), Nodos atributo ($\boldsymbol{N_A}$), Nodos de texto ($\boldsymbol{N_T}$), Nodos secuencia ($\boldsymbol{N_S}$), Nodos opción ($\boldsymbol{N_C}$), Nodos de intercalación ($\boldsymbol{N_I}$) y nodos vacíos ($\boldsymbol{N_G}$). El grafo tiene un conjunto de nodos raíz: $\boldsymbol{R \subseteq N}$

La Función para cadenas de caracteres XML se define como $\boldsymbol{N_T \cup N_A \cup N_E \longrightarrow S}$, en donde $\boldsymbol{S}$ es una familia de lenguajes regulares de cadenas sobre el alfabeto Unicode.

No todas las aplicaciones incluyen vacíos; no en todos estos los componentes vacíos son ignorados.

Para simplificar la validación se requiere que los nodos de intercalación nunca aparezcan anidados dentro del contenido o de los atributos.

Dos grafos XML se dice que son compatibles si coinciden en los valores de $\boldsymbol{N}, \boldsymbol{G}, \boldsymbol{S}, \boldsymbol{T}$ y sus contenidos para $\boldsymbol{n} \in \boldsymbol{N_E} \cup \boldsymbol{N_A} \cup \boldsymbol{N_S} \cup \boldsymbol{N_I}$

La principal razón para utilizar un grafo XML en lugar de algunas otras alternativas es que los grafos XML forman de manera natural una lattice de altura finita [Schwartzbach], la cual es la estructura común para representar el CFG de un programa.

Parsing (última fase de transformación)

Tal vez la transformación más importante que sufrirá el código es la que permite generar código intermedio a partir del código ensamblador. Existen varias formas para tal fin y entre las cuales podemos citar [Aho et Al] los parsing LL(1), utilización de configuraciones LR(1) o bien SLR(1) y LALR(1). Todas y cada una de ellas tienen ventajas y desventajas inherentes.

Existe también, algunas herramientas que permiten construir la parte frontal de un traductor de código, pero también ellas tienen su propia problemática. La tarea de transformar una gramática hasta que su representación LALR sea del tipo determinista es ardua y difícil y aún existe más –la fase de semántica puede verse afectada- [Schmitz 2005].

La claridad y poder de expresividad de los parsing $LALR(k)$ puede verse reducida si tiene $k > 1$ elementos de exploración. Aunque los parsing $LALR$ clásicos son una herramienta poderosa, no permite una sintaxis ambigua. En contraposición, el uso de parsings no-canónicos permite realizar la reducción de frases sin utilizar el esquema de reducción por el símbolo más a la izquierda. Los parsing no-canónicos son una herramienta más flexible que puede competir con los parsing clásicos.

Parsing no-canónico

El inverso de una derivación del tipo $\delta A\sigma \Longrightarrow \delta\alpha\sigma$ (ver nomenclatura en la figura 4.2) es la reducción de la frase α en la sentencia $\delta\alpha\sigma$ al no terminal A. En los parser bottom-up, incluidos los LALR, este tipo de derivaciones se logran utilizando el símbolo

más a la derecha y es por esto que la frase reducida se encuentra más a la izquierda y es llamada conductor de la sentencia.

Los parsers No-Canónicos permiten reducir frases que no pueden ser conducidas y el orden de derivación puede diferir a la del símbolo más a la derecha y adaptarse a las necesidades del parser. En este sentido, los parser No-Canónicos son una herramienta más flexible que los parsers clásicos. Los parser No-Canónicos permiten la cancelación de una decisión de reducción a diferencia de los parser canónicos los cuales son más inflexibles en ese aspecto.

Parsers LR(1) y LALR(1)

Los parsers $LALR$ fueron diseñados para lenguajes del tipo determinista. En lugar de construir muchos estados $LR(k)$, los parsing $LALR(K)$ agregan un conjunto de exploradores a las acciones de pequeños parsers $LR(0)$. El conjunto completo de exploradores son la unión de todos los conjuntos de exploradores que un parsing $LR(k)$ contiene, proporcionada cierta entrada.

Elementos y prefijos

Definición4 Elemento válido

Usando la producción punto $A\alpha \cdot \beta$ de la gramática $g\ LR(0)$ es un elemento válido para la cadena γ en V'^{*} si

$$S' \underset{rm}{\Longrightarrow^{*}} \delta A z \underset{rm}{\Longrightarrow} \delta\alpha\beta z = \gamma\beta z$$

Si tal derivación está contenida en la gramática g, entonces γ en V'^{*} es un prefijo válido. El conjunto de tales elementos válido para una cadena dada γ en V'^{*} es denotado por $Valid(\gamma)$

Definición 5 Autómata LR(0)

Sea $M = (Q \cup T \cup \{\$, \|\}, R)$ un sistema de re-escritura en donde \$ y ‖ (el marcador de fin y el componente superior de la pila respectivamente) no son encontrado ni en Q ni T (el conjunto de estados y el alfabeto de entrada respectivamente)

Una configuración de M es una cadena de la forma

$$[\varepsilon][X_1] \cdots [X_1 \cdots X_n] \| x\$$$

En donde $X_1 \cdots X_n$ es una cadena en V^{*} y x una cadena en T^{*}. Se dice que M es un autómata $LR(0)$ para la gramática g si su configuración inicial es $[\varepsilon] \| w\$$ con w como la cadena de entrada en T^{*}, su configuración final es $[\varepsilon][S] \| \$$ y que cada una de las reglas de re-escritura en R es una de las formas siguientes

- *Shift* a en el estado $[\delta]$

  $[\delta] \| a \underset{shift}{\vdash} [\delta][\delta a] \|$, la cual está definida si existe un elemento de la forma $A \to \alpha \cdot a\beta$ con $Valid(\beta)$

- *Reduce* por la regla $A \longrightarrow [X_1 \cdots X_n]$ del conjunto P en el estado $[\delta X_1 \cdots X_n]$ $[\delta X_1] \cdots [\delta X_1 \cdots X_n] \parallel \underset{A \to X_1 \cdots X_n}{\vdash} [\delta A] \parallel$, la cual está definida si $A \to X_1 \cdots X_n \cdot$ con $Valid(\delta X_1 \cdots X_n)$

Diferencia entre parsing $\boldsymbol{LALR(1)}$ y $\boldsymbol{NLALR(1)}$

La diferencia más importante entre $LALR(1)$ y la representación no canónica de un parsing $LALR(1)$ ($NLALR(1)$) es que los exploradores para $NLALR(1)$ aceptan también símbolos no terminales a diferencia de $LALR(1)$ el cual acepta únicamente terminales. Debido a que un NLALR(1) acepta tanto terminales como no terminales en sus exploradores [Schmitz 2007] propone la siguiente definición.

Definición 6 Cobertura válida

La cadena de símbolos γ es una cobertura válida en la gramática g para la cadena de símbolos δ si y sólo si γ es un prefijo válido y $\gamma \Longrightarrow^* \delta$. Se escribe $\underset{\delta}{\bigwedge}$ para expresar alguna cobertura de δ y $Cover(L)$ para expresar el conjunto de todas las coberturas válidas en el conjunto de cadenas de símbolos L.

Un símbolo explorador no-canónico es un símbolo en V' en lugar de ser un símbolo de T' como se hace con los de tipo canónico. Los símbolos de exploración no-canónicos no pueden ser nulos entonces X es no nulo únicamente si $X \Rightarrow^* ax$

La definición formal de NLALR puede ser encontrada en los trabajos de [Schmitz Sylvain 2005] y [Schmitz 2007].

Algoritmo de recorrido de un AST

El algoritmo de primero en profundidad es utilizado para realizar transformaciones de flujo tal como $CFG \longrightarrow DFST$ [Khedker et Al] figura 4.3. También, el algoritmo primero en profundidad es el algoritmo más común utilizado para recorrer árboles AST

Input: A CFG *G* with *N* nodes.
Output: A DFST *T* for *G* and an array rpo[1..*N*] representing a reverse postorder listing of nodes in the graph.

Algorithm:

```
0   function dfstMain()
1   { i = N
2     make root(G) the root of T
3     dfst(root(G))
4   }
5   function dfst(currnode)
6   { mark currnode
7     while there are unmarked successors of currnode do
8     { let child be an unmarked successor of currnode in
9       { add the edge (currnode →child) to T
10        dfst(child)
11      }
12    }
13    rpo[currnode] = i
14    i = i−1
15 }
```

Fig 4.3 Algoritmo para el cálculo de un árbol primero en profundidad

El esquema básico de transformación sintáctica

La fase de análisis sintáctico de un compilador se divide en

- Verificación de propiedades las cuales son independientes al contexto. Esta fase por sí sola recibe el nombre de análisis sintáctico. El resultado de tal análisis es una estructura AST.

- Verificación de las propiedades de contexto (sensibilidad al contexto), también llamado análisis semántico. Se parte del punto anterior en donde el programa se encuentra analizado mediante la gramática libre de contexto y transformado a una

estructura AST. En cada producción expresada en la gramática se aplican reglas semánticas. Tales reglas, definen si la sintaxis del programa es la correcta (sintaxis de la sensibilidad del contexto).

Para la verificación del programa se utiliza el algoritmo de primero en profundidad con orden izquierda a derecha. Las reglas semánticas que definen la sensibilidad de contexto son aplicadas a cada nodo del AST. [Heberle et Al].

La mayoría de nodos en el AST de transformación para las instrucciones X86 aplican algún tipo de asignación. Esto debe ser así, debido a que la naturaleza de sus instrucciones no se basa en instrucciones de pila como es el caso del lenguaje intermedio CLI. [Heberle et Al] sugiere que en un estatuto de asignación $assign::des := expr$ se requiere que el tipo del lado derecho sea el mismo que el de su lado izquierdo. Aunque el estado de suficiencia de equivalencia de tipos entre la expresión de lado derecho y del lado izquierdo es expresado por el símbolo $\rightsquigarrow$ y que reemplaza al signo =, menciona que ambos tipos no necesariamente tienen que ser completamente iguales. Entonces $\rightsquigarrow$ es el símbolo que denota coercitividad y que es dependiente del lenguaje y representa la asignación de tipos como se expresa en la semilattice de la figura 4.4

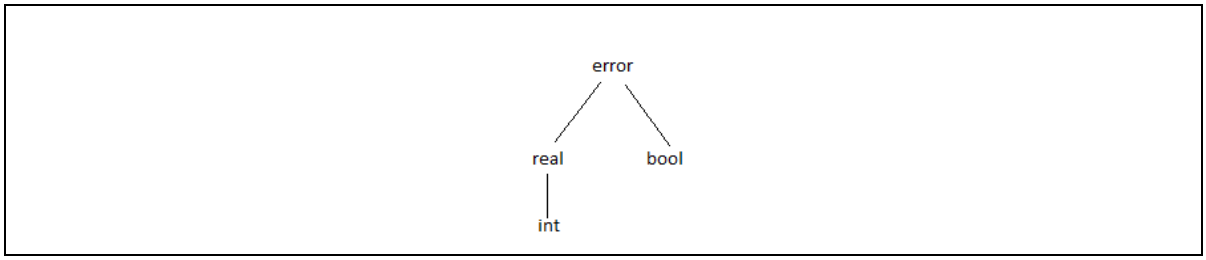

Fig 4.4 Semilattice mostrando los conceptos de error y coercitividad

Dependiendo del diseño de lenguaje, no existe error entre las conversiones de tipo *real* e *int* y tal problema debe ser resuelto por medio de coerción explícita o implícita, pero en el caso de conversión de tipos real y bool existe únicamente un estado el cual es el error, lo que indica que no puede existir coerción posible.

Es en este punto, que se debe mencionar que los tipos utilizados en el código máquina X86, de todas formas pueden ser tratados de manera coercitiva. Esto es necesario en los casos tales como las instrucciones de comparación. Existen dos juegos de instrucciones de naturaleza comparativa: comparaciones con signo y comparaciones sin signo.

Utilización de expresiones lambda para recorrer nodos en AST

Los lenguajes que permiten el uso de filtrado funcional facilitan el recorrido selectivo de nodos en el AST. Aunque las bases teóricas que sustentan tal afirmación no son nuevas [Hilzer et Al], la madurez de los lenguajes de desarrollo no permitían el uso de tales prácticas de carácter funcional en lenguajes orientados a objetos y es en años recientes que se empieza a utilizar el filtrado funcional selectivo.

Para una profundización de estos temas leer la referencia [Hilzer et Al] pero aquí mencionaremos únicamente los conceptos básicos de esta técnica.

Evaluación retrasada y evaluación inmediata

En el paradigma funcional una función $f\ x_1\ x_2\ \cdots x_n,$ siendo $n \geq 0$ se dice que es estricta sobre el argumento x_i, en donde $1 \leq i \leq n$ si siempre requiere el valor de x_i para su evaluación. En caso contrario, no es estricta sobre el argumento x_i.

En los lenguajes funcionales, se dice que se utiliza evaluación inmediata si todos sus argumentos son evaluados antes que la función sea aplicada independientemente de que sus argumentos sean estrictos o no.

Se dice que un lenguaje respeta la evaluación retrasada si la evaluación de los argumentos de una función es retrasada hasta después de la aplicación de la función que los contiene. Los argumentos con retraso son evaluados hasta que sea necesario. Si cada instancia de un argumento compartido es evaluada por separado, se dice que la

implementación tiene un retraso parcial. Si todos los argumentos son evaluados al mismo tiempo se le llama retraso total.

Si la evaluación de una expresión termina utilizando ambos tipos de evaluación, ambas formas regresarán el mismo resultado. Pero la evaluación inmediata es más lenta y consume más espacio que la evaluación retrasada si es que fueran encontrados argumentos innecesarios de manera frecuente. Pero por otro lado la evaluación inmediata es más fácil de desarrollar. Aunque la evaluación inmediata con argumentos estrictos se ejecutará más rápidamente. [Hilzer et Al]

La habilidad de administrar secuencias de datos de longitud indeterminada, (cuyo nombre técnico se les llama streams) lo cual es una de las atracciones para los programadores.

Cálculo lambda

Propuesta por [Church 1941]. Es una forma sencilla de describir las propiedades de funciones computables y trata a las funciones mediante una serie de reglas y no únicamente se aplica a lenguajes del tipo funcional, sino también todo tipo de función.

Las expresiones lambda son funciones anónimas consistentes de una lambda (λ), parámetros formales, un cuerpo de función y el valor de los argumentos que están siendo tratados.

Ejemplo $(\lambda x. + x\ x)\ 3$ en este caso λ identifica a la función como una expresión lambda, la x antes del punto identifica a x como un parámetro formal y $+ x\ x$ com el cuerpo de la expresión λ. 3 es el valor del argumento actual. La expresión lambda anterior es *reducida* a la forma siguiente $(\lambda x. + x\ x)\ 3 \longrightarrow + 3\ 3 \longrightarrow 6$.

Otro ejemplo de expresión con dos parámetros $(\lambda x, y.\ *\ y\ x) 2\ 4$ es reducido en la forma siguiente $(\lambda x, y.\ *\ y\ x) 2\ 4 \equiv \big((\lambda x.\ \lambda y.\ *\ y\ x)\ 2\big) 4 \rightarrow (\lambda y.\ *\ y\ 2)\ 4 \rightarrow * 2\ 4 \rightarrow 8$

Definiciones lambda

Expresiones reductibles (también llamadas expresiones ***redex***). Se dice que una llamada a función es reductible cuando todos sus argumentos se encuentran disponibles.

Forma normal (*NF*). Una expresión está en forma normal si ha sido reducida a un valor único o es un objeto que no puede ser reducido aún más (como una lista).

Forma normal frontal (Head Normal Form ***HNF***) es una expresión en forma normal o es una expresión λ o que es una función que no tiene uno o más de sus argumentos más externos. Como ejemplo (+ 14) se encuentra en forma HNF pero la expresión $(+(*\ 5\ \ 4))$ no es HNF debido a que existe un argumento no evaluado de +.

Forma normal frontal débil (Weak-Head Normal Form **WHNF**) es una forma la cual puede ser una NF, HNF o una expresión lambda o una función sin uno o más de sus argumentos más externos y sus argumentos más internos pueden ser evaluados o no.

Variables limitadas y variables libres Las variables en el cuerpo de una expresión lambda pueden aparecer limitadas o libres. Una variable es limitada si es un parámetro formal encerrada dentro de una expresión lambda. De otra forma la variable se encuentra libre. Por ejemplo en la expresión $(\lambda y.+\ y\ x)$ la variable y es limitada y la variable x es libre.

Currying. Es posible tratar a una función de n argumentos tal como una concatenación de n funciones de un único argumento. Ejemplo la función $(f\ x\ y\ z)$ puede ser tratada como la expresión $(((f\ x)\ \ y)\ \ z)$

Reducción de orden aplicativo (AOR). Reduce el redex más interno primeramente. Ejemplo, dada la función $(f\ \ (g\ \ arg))$, un AOR reducirá primeramente arg, y luego el argumento $(g\ arg)$ y posteriormente $(f\ \ (g\ \ arg))$. AOR es desarrollado mediante evaluación inmediata.

Reducción por orden normal (NOR) reduce el redex más externo y más a la izquierda primero. Ejemplo, en la expresión $(f\ \ (g\ \ arg))$, NOR iniciará en la función f y luego con sus argumentos $(g\ arg)$ y finalmente los argumentos de g si es necesario. NOR utiliza evaluación retrasada.

Reglas lambda

Utilizadas para cambiar su forma con el propósito de reducirla para obtener una forma más simple. Existen los siguientes tipos

1. Reducción; permite reducir una expresión a una forma más simple
2. Abstracción; transforma las expresiones a una forma más compleja
3. Conversión; cambia la expresión a una forma equivalente (como ejemplo mediante e cambio de nombre de los parámetros)
4. Asenso; elimina las variables libres de una expresión

Las reglas del cálculo lambda son las siguientes

Reducción beta. Reduce a la expresión lambda mediante la substitución de los argumentos actuales por parámetros formales en el cuerpo de la expresión lambda.

Ejemplo

$$(\lambda x.\ *\ x\ x\ )\ 7\ \xrightarrow{\beta_r}\ *\ 7\ 7$$

Abstracción beta. Es la operación inversa a una Reducción Beta.

Ejemplo

$$* \ 7 \ \ 7 \ \underrightarrow{\beta_a} \ (\lambda y.* \ y \ y)$$

Reducción delta. Reduce una función básica.

Ejemplo

$$* \ 7 \ \ 7 \ \underrightarrow{\delta} \ 49$$

Conversión alfa. Cambia los nombres de los parámetros de una función.

Ejemplo

$$(\lambda y.* \ y \ y) \ \underleftrightarrow{\alpha} \ (\lambda z.* \ z \ z)$$

Conversión eta. Convierte una representación en otra la cual es equivalente.

Ejemplo

$$(\lambda x. f \ x) \ \underleftrightarrow{\eta} \ f$$

En donde f es una función y x no es libre en f

Asenso lambda. Ocurre en expresiones con variables limitadas. Para eliminar variables libres se puede utilizar abstracciones beta y reducciones.

Ejemplo

$$(\lambda x.* \ x \ y) \ \underrightarrow{\beta} \ (\lambda y. \lambda x.* \ x \ y \ ) \ y \ \underrightarrow{\alpha} \ (\lambda w. \lambda x.* \ x \ w) y$$

Intérpretes AST

Algunos intérpretes de lenguajes funcionales utilizan también estructuras AST para traducir el código fuente a expresiones lambda y posteriormente convierten estas expresiones lambda a un formato lambda el cual puede ser recorrido por un algoritmo como el de primero en profundidad. La evaluación inmediata facilita el desarrollo de este algoritmo. Por otro lado, mediante el uso de evaluación tardía y dependiendo el lenguaje de desarrollo la implantación puede resultar más simple. (Estas y otras consideraciones de diseño para tal algoritmo serán abordadas en el capítulo siguiente).

Un AST es un árbol parse en donde cada uno de los nodos es decorado con reglas semánticas y acciones semánticas. Los operadores semánticos incluyen

- **app** aplica la expresión del hijo izquierdo del nodo que se esté analizando a la expresión del hijo derecho

- **int** el hijo es una constante entera

- **var** el hijo es una referencia a una variable

- **prim** el hijo es una función primitiva

- **lambda** el hijo izquierdo es un parámetro formal y el hijo derecho es un cuerpo lambda

- **closure** una estructura compuesta consiste de una expresión lambda y una lista de variables vinculadas

Ejemplo

Dada la expresión $(\lambda x.* \ x \ \ x)$ 6 su correspondiente árbol AST es ilustrado en la figura 4.5.

Aunque las acciones semánticas incluidas en el AST para la traducción de código X86 son diferentes, el concepto es el mismo.

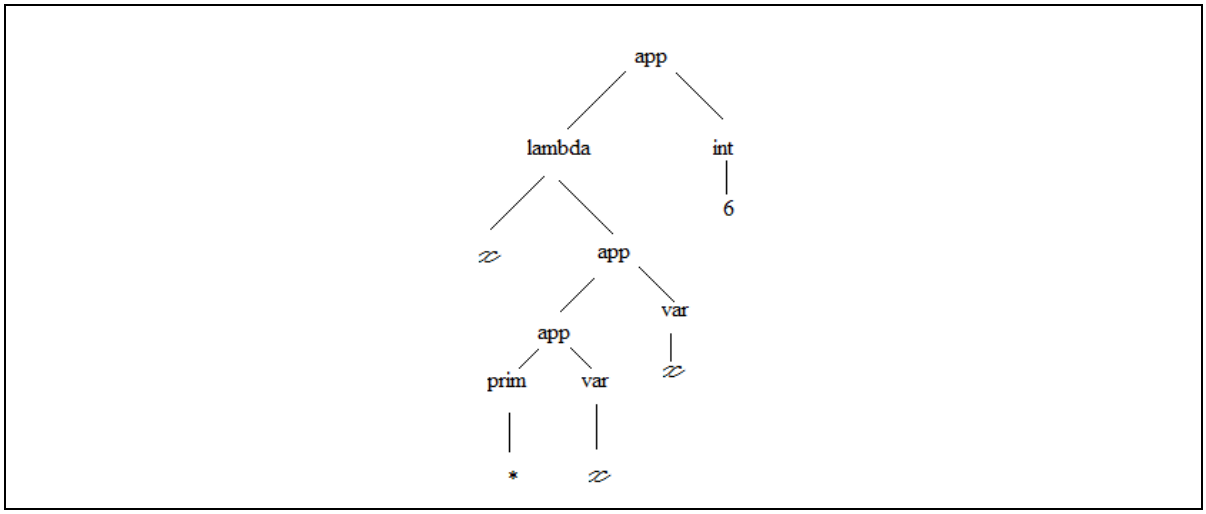

Fig 4.5 Estructura AST para la expresión $(\lambda x.* \ x \ \ x)$ 6

V. DISEÑO Y ARQUITECTURA

En el presente capítulo se menciona cada una de las fases de transformación que deberá sufrir el código binario original el cual será probado.

Se usarán los conceptos mostrados en el capítulo 4 los cuales proporcionan el soporte teórico, también se toma como base el estándar PE (Portable Executable) y Common Object File Format (COFF) y también el estándar ECMA-335.

Se mencionarán conceptos de diseño, tal como utilización de patrones y algunos otros conceptos de carácter programático.

La figura 5.1 detalla las diferentes fases de transformación originadas a partir del formato COFF. El primer paso es la transformación a código binario plano. El punto de origen es pues, el formato COFF-PE, propiedad de Microsoft [Microsoft 2010].

El prototipo fue desarrollado con la idea de proporcionar una herramienta que facilite la automatización de pruebas a partir de código binario y es por este hecho, que se consideró construir un nivel de abstracción intermedio para apoyar los conceptos relacionados con la generación del código maquina, el flujo de control y la tabla de símbolos, estructuras necesarias todas ellas para desarrollar el autómata y las acciones semánticas en posteriores etapas y que darán forma al esquema propuesto. El nivel de abstracción mencionado respeta las premisas y formalismos propios del lenguaje XML.

También es importante mencionar código ensamblador X86 y la referencia de [Sarkar et Al]. Se utilizarán algunas técnicas recientes mostradas en [Liu et Al] para manipulación de la tabla de símbolos y el paso de parámetros entre las diferentes etapas de transformación.

Aunque no es parte del presente trabajo, se mencionarán los guiones utilizados para el desarrollo de pruebas.

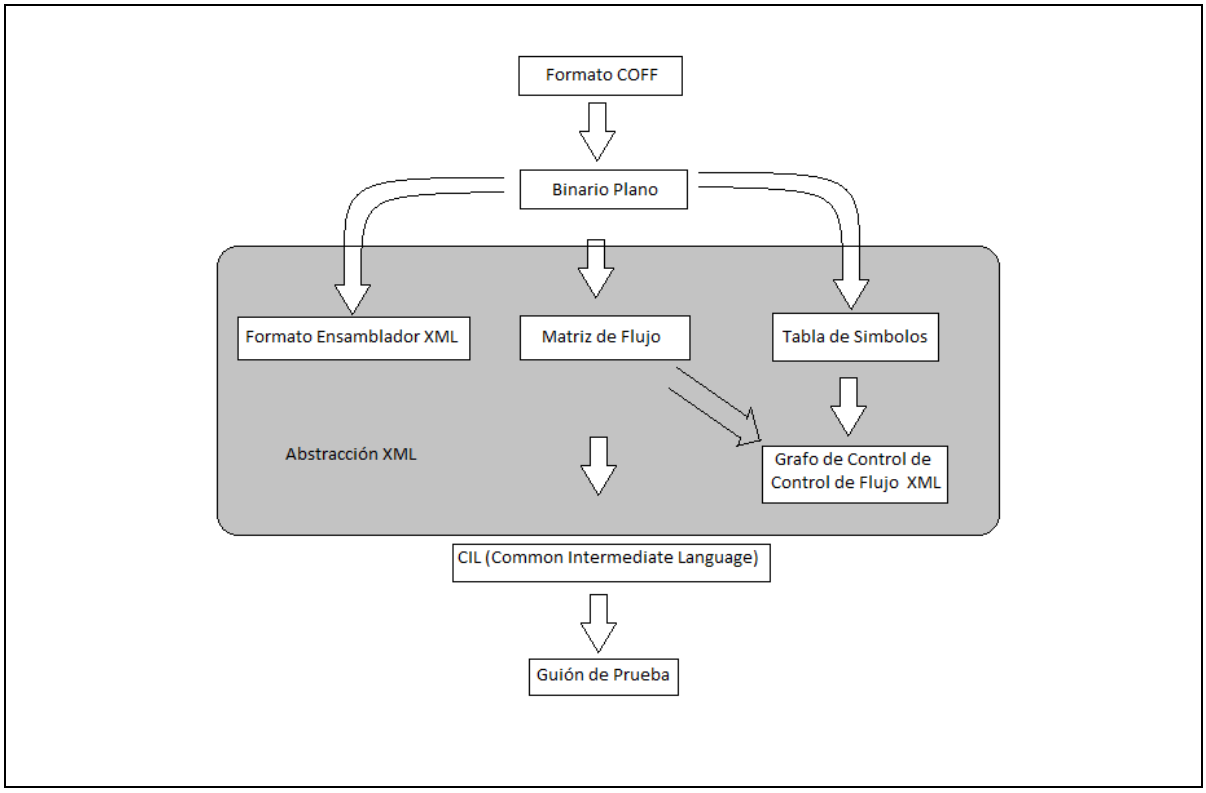

Fig 5.1 Arquitectura de las etapas de transformación

El diseño en la fase de des-ensamblado

El diseño base en la primera fase (fase de des-ensamble) considera seis (figura 5.2) paquetes cuya funcionalidad se basa en los siguientes puntos

a) Clasificación del tipo de proyecto
b) Seriación de Información
c) Reconocimiento de flujo binario
d) Clasificación de la Arquitectura
e) Manipulación de información a Nivel bit

f) Clasificación de códigos de operación
g) Transformación de código binario a lenguaje ensamblador
h) Almacenamiento de información utilizando formato XML
i) Administración de Archivos Temporales
j) Almacenamiento tabular y en forma de flujo
k) Administración de Métricas

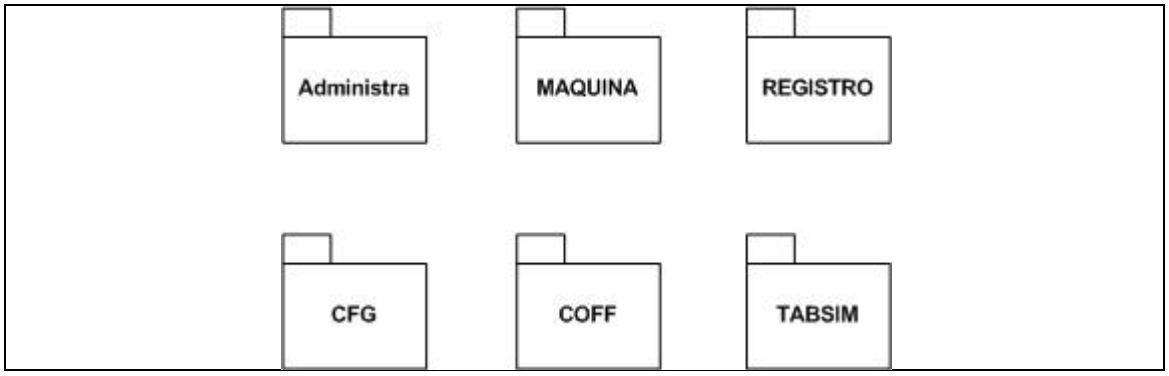

Fig 5.2 Paquetes construidos en la fase de de-compilación

El paquete Administra

El objetivo del paquete administra es reconocer el tipo de proyecto de acuerdo a las peticiones de entrada y salida. De igual forma, es el encargado de seriar la información al formato requerido. Las clases que son parte de este paquete son

- **Distribuye**: Permite clasificar el proyecto de acuerdo al formato de origen y el formato destino requeridos. Los formatos desarrollados son: COFF-FLUJO, COFF-REFLEXIÓN, BINARIO-FLUJO, BINARIO-REFLEXION y BINARIO-SPECSHARP.

- **Flujo**: realiza la seriación de flujo transformando a formato XML.

- **Lin**: realiza la seriación de código máquina a código ensamblador.

- **Proyecto**: realiza la seriación que contiene el tipo de proyecto (encabezado). La información que proporciona es: folio temporal (se utiliza para ello un registro de sistema), Fecha de creación, Fuente de origen, Nombre del Grafo (CFG).

- **SerializaFlujo**: como su nombre lo indica, se encarga de seriar el flujo de programa.

- **Sim**: realiza la seriación de los operadores de lenguaje ensamblador. Contiene el número de byte que está siendo analizado, La tabla que fue utilizada para conversión, La función utilizada para conversión. El tipo y elemento para los operadores izquierdo y derecho.

El paquete Cfg

Su objetivo es realizar la transformación del flujo de programa en ensamblador a formato GRAPHML. Sus elementos son componentes de un Grafo de flujo (CFG), esto es: la clase edge, la clase grafo, la clase graph, la clase graphml y la clase node. No mencionaremos cada uno de ellos debido a que resulta obvio para qué es utilizada cada clase en la seriación XML.

El paquete Coff

El propósito de este paquete es exponer cada uno de los componentes de un archivo binario COFF (ver el punto 5.2). Consiste de dos clases: Coff y Maquina.

- **Coff**: es la clase encargada de separar cada una de las partes del documento COFF

- **Maquina**: Clasifica cada una de las arquitecturas (hardware) junto con sus características programáticas importantes

El paquete Maquina

Es el paquete que se encarga de traducir el formato binario plano a código ensamblador y contiene las siguientes clases

- **Bits** clase que contiene la clasificación de los registros de la arquitectura Intel x86 y las funciones que permiten tratar el byte $\boldsymbol{ModRM}$ (ver sección 4.4).

- **Celda** es una clase auxiliar para clasificar la información utilizada en la clase **CodOp** (ver sección 5.4).

- Las clases **CeldaT3**, **ParLlaveT3** y **ParValorT3** son clases auxiliares para la clase **CodOpT3**. Utilizada para traducir las instrucciones tipo T3 (ver sección 5.4).

- **CodigoMaquina**: Es la clase más importante de este paquete debido a que transforma el código en código ensamblador.

El paquete Registro

Este paquete se encarga de Administrar los nombres del código generado utilizando para ello un registro de sistema operativo. Su única clase es **AdministraRegistro**.

El paquete TabSim

Este paquete se encarga de registrar la información de los símbolos y tipificación de los operandos. Registra el flujo de tipificación. También proporciona la métrica de tiempo.

- **Flujo**: Registra el flujo en una matriz Rala. (ver capítulo anterior)

- **FunEns**: contiene almacenado en forma de diccionario las funciones de ensamble utilizadas

- **ParFunEns**: Es una clase auxiliar de la clase **FunEns**

- **TabSim**: Utiliza un diccionario para almacenar la tabla de símbolos. Se auxilia de la clase ParTabSim

- **TabFlujo**: es auxiliar de la clase **FunEns** y agrega la información en la tabla de símbolos.

- **TipoOper**: Almacena el tipo de operando utilizado en la operación. Se auxilia de la clase **ParTipoOper**.

El formato COFF-PE

La especificación difundida por Microsoft [Microsoft 2010] para archivos PE (Portable Executable) y Common Object File Format (COFF) debe ser mencionada al principio y fin del presente trabajo debido a que inicia la exploración con código binario y finaliza en la traducción a código CLI.

En este estándar se especifica la estructura de los archivos objeto y ejecutables bajo la familia de sistemas operativos de Windows. Aunque se menciona que el formato PE (Portable Executable) no se dirige a una arquitectura en particular, este es utilizado exclusivamente en los Sistemas Operativos de Microsoft.

En esta misma especificación, se detalla cada uno de los campos de los formatos mencionado anteriormente, pero para el presente trabajo únicamente se analizaran las figuras 5.3 y 5.4.

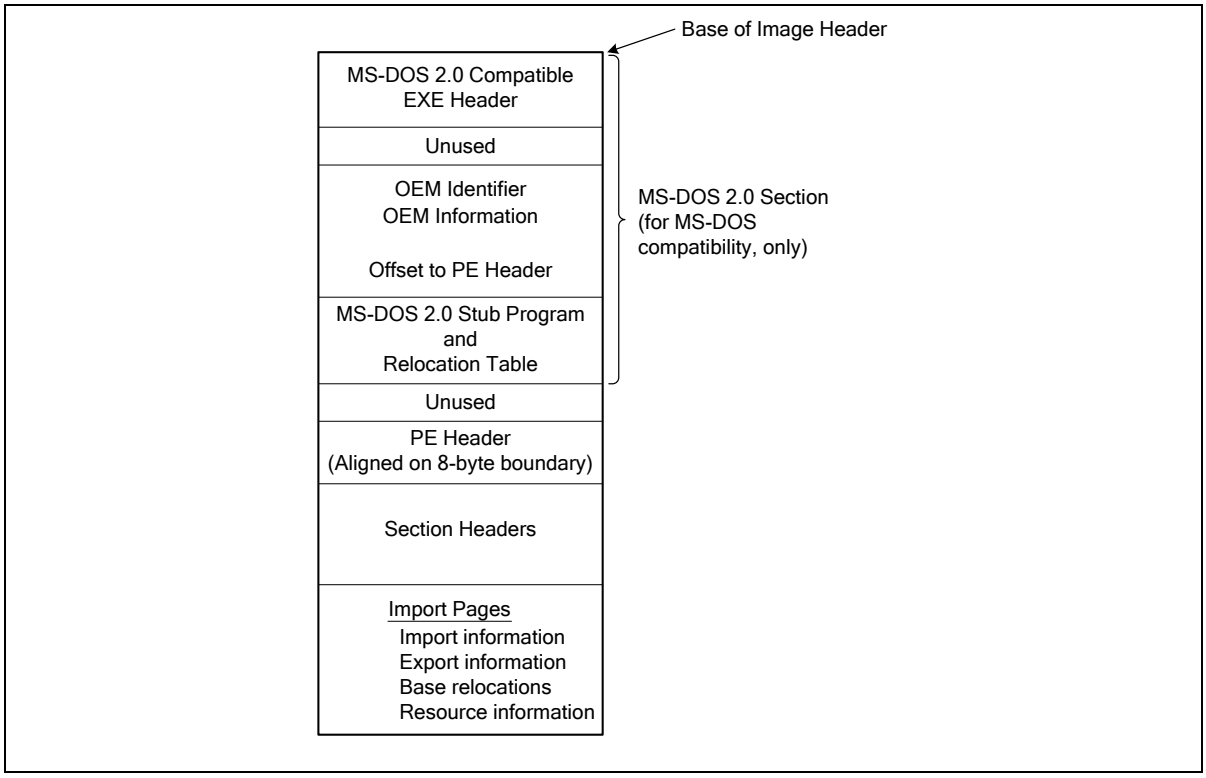

Fig 5.3 Distribución típica de un archivo EXE

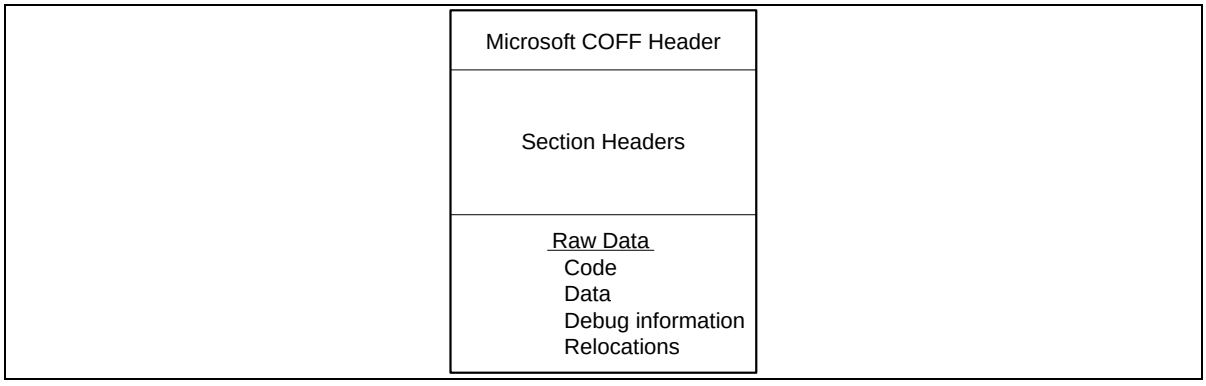

Fig 5.4 Distribución típica de un módulo objeto COFF

El inicio de un archivo PE consiste de una pieza de información MS-DOS, la cabecera COFF y una cabecera opcional. La cabecera COFF consiste de la información relacionada con la propia cabecera y una cabecera opcional.

La pieza de información MS-DOS es utilizada en aplicaciones que son ejecutadas sobre sistemas operativos MS-DOS y se encuentra colocada en la parte frontal de la imagen EXE. La pieza de información MS-DOS puede ser modificada utilizando la opción de eslabonamiento /STUB.

En la dirección 0x3c (ver figura 5.5), la pieza de información MS-DOS tiene el desplazamiento a la firma PE. Esta información permite que el archivo sea ejecutado usando el Sistema Operativo (SO) Windows aunque se tenga la cabecera MS-DOS.

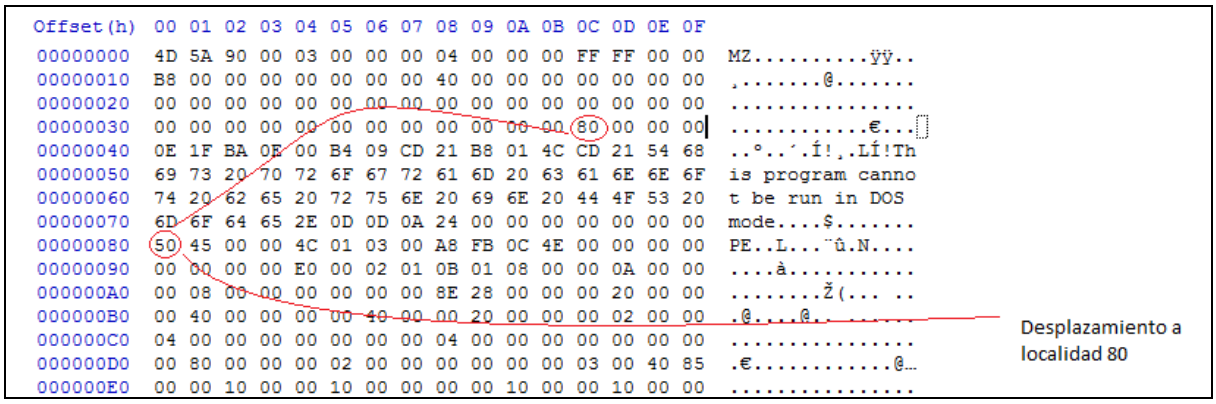

Fig 5.5 Ejemplo de archivo EXE mostrando la localidad 3c y la firma PE

La firma PE

En el desplazamiento mostrado en la localidad 3c de 4 bytes se identifica el archivo como un archivo de tipo PE. Las letras PE seguidas de dos nulos (ver figura 5.4)

Los Archivos COFF

El inicio de un archivo COFF contiene las secciones: Tipo de Máquina, Número de Secciones, Fecha de creación, Puntero a la Tabla de Símbolos, Número de Símbolos, Tamaño de la Cabecera Opcional y Características. La cabecera de un archivo COFF mantiene la información que se muestra en la tabla 5.1.

Los punteros a datos planos permiten acceder a los binarios que son utilizados como código original en el presente trabajo.

Los meta-datos y el lenguaje intermedio (Common Language Intermediate CIL) son almacenados en los archivos PE [Microsoft, Metadata].Los meta-datos son una serie de estructuras de dato que contienen la información relacionada con el programa ejecutable. Clases, métodos, información relacionada con la estructura del montículo, información de las cadenas de caracteres (Strings), las referencias a objetos binarios largos y la identificación de usuario se encuentran almacenados en forma de meta-datos.

En el momento de compilación un programa CLR (ver apartado 2.2.8 del documento de especificaciones del estándar) es convertido a archivo PE el cual consiste de tres partes como se ilustra en la tabla 5.2

Desplazamiento	Tamaño	Campo	Descripción
0	8	Nombre	Una cadena de caracteres terminada en nulo.
8	4	Tamaño Virtual	Tamaño de la sección cuando es cargada en la memoria
12	4	Dirección Virtual	Dirección del primer byte relativa a la base de la imagen cuando se carga en memoria
16	4	Tamaño de los datos planos	El tamaño de la sección
20	4	Puntero a datos planos	El puntero a la primer página de la sección dentro del archivo COFF
24	4	Puntero a Relocalizaciones	Puntero al inicio de las relocalizaciones de la sección
28	4	Puntero a los números de línea	Puntero al inicio de los número de línea
32	2	Número de Relocalizaciones	Número de relocalizaciones
34	2	Número de líneas	Número de líneas para la sección presente
36	4	Características	Banderas describiendo las características de la sección

Tabla 5.1 Descripción del archivo de cabecera COFF

Código ensamblador propietario X86

El modelo de memoria utilizado en los actuales procesadores X86 y el cual se presume que es secuencialmente consistente, aunque la realidad es que tal modelo exhibe un esquema de memoria relajado [Sarkar et Al]. Los mecanismos sofisticados para

proporcionar un alto performance son más de naturaleza ingenieril que de formalización. Los mecanismos a los que se hace referencia en el trabajo de [Sarkar et Al] son los caches locales, escritura en buffers y ejecución especulativa. El ejemplo presentado en el trabajo de [Sarkar et Al] y en el que se mencionan dos procesos ejecutándose en paralelo demuestra la fragilidad del sistema.

Sección	Contenido de la sección
Cabecera PE	El índice de la sección principal y la dirección del punto de entrada
Instrucciones CLI	Instrucciones el lenguaje intermedio. Muchas de las instrucciones son almacenadas con meta-datos
Meta-datos	Tablas de meta-dato y de montículos. En esta parte se almacena la información relacionada con los tipos y miembros del código. Incluye información relacionada con la seguridad

Tabla 5.2 Secciones en un archivo PE

Los eventos de lectura y escritura no pueden ser consistentemente incrustados en un orden lineal simple en sistemas que tienen varios procesadores x86, es por ello que no existe la noción de tiempo global. Por tal razón, existe incertidumbre en algunas técnicas de sincronización utilizadas en los Sistemas Operativos (tal como los “spinlock”, también llamados ciclos de espera).

Aunque es de gran importancia el análisis de la información en procesos en paralelo y los métodos de bloqueo a nivel binario, este tema queda fuera del alcance del presente trabajo.

En este apartado únicamente se mencionarán las clases necesarias para construir la codificación de lenguaje máquina a ensamblador y cuyas referencias están expresadas en los manuales del mismo fabricante.

Para realizar este análisis se usará en un conjunto reducido de instrucciones y al cual se hará mención como bloques básicos. El estándar de este procesador se refiere a [Intel 32 architecture]

Se utilizará la convención de la figura 5.6 para expresar los elementos que componen una instrucción X86

Nomenclatura	Descripción
X_{reg32}	Registro de 32 bits
X_{reg8}	Registro de 8 bits|
I_{s-32}	Cantidad constante con signo de 32 bits
I_{u-32}	Cantidad constante sin signo de 32 bits (Inmediato 32 bits)
I_{s-8}	Cantidad constante con signo de 8 bits
I_{u-8}	Cantidad constante sin signo de 8 bits (Inmediato 8 bits)
I_{s-16}	Cantidad Constante con signo de 16 bits
I_{u-16}	Cantidad Constante sin signo de 16 bits
M	Memoria
S_{reg}	Registro de segmento
X_{eflag}	Bandera

Fig 5.6 Nomenclatura utilizada para inmediatos y memoria

Los registros para la arquitectura Intel y sus banderas se muestran en la figura 4.7

$X_{reg32} ::= EAX \mid EBX \mid ECX \mid EDX \mid ESI \mid EDI \mid EBP \mid ESP$
$X_{reg8} ::= AH \mid AL \mid BH \mid BL \mid CH \mid CL \mid CH \mid DL$
$S_{reg} ::= CS \mid DS \mid ES \mid SS \mid FS \mid GS$

$X_{eflag} ::= X_{CF} \mid X_{PF} \mid X_{AF} \mid X_{ZF} \mid X_{SF} \mid X_{OF}$

Fig 5.7 Registros y banderas de los procesadores X86

A su vez, la memoria puede ser representada de diferentes maneras y dependiendo de estas formas el desensamblado de una instrucción en código maquina pueden ser de 1, 2 y 4 bytes de tamaño (figura 5.8).

$M ::= [X_{reg8} \pm I_{s-8}] \mid [X_{reg32} \pm I_{s-32}] \mid [I_{u-32}]$

Fig 5.8 Especificación de memoria

Tipificación de instrucciones

El repertorio de instrucciones que serán consideradas en el presente trabajo es un subconjunto del total de instrucciones y contiene inmediatos y direcciones de 32 bits y de 8 bits. De tal forma que el byte de tamaño no se usará. Tampoco el prefijo de bloqueo (LOCK) mencionado en [Sarkar et Al] y que permite que cada instrucción conserve su carácter atómico no se utilizará.

Una instrucción en lenguaje máquina puede contener cero, uno, dos e incluso tres operandos. Para el caso desarrollado aquí fue considerada la producción mostrada en la figura 5.9

$$operando ::= I_{u-8} | I_{s-8} | I_{u-32} | I_{s-32} | X_{reg8}\ M | X_{reg32}\ M$$

Fig 5.9 Operandos usados en bloques básicos

Para la traducción de instrucciones binarias se utilizó de manera primordial dos bytes. El primer byte llamado código de operación (***CodOp***), que especifica la propia instrucción y el segundo byte que especifica la naturaleza de la instrucción (***ModRM***). En este punto resulta importante mencionar que las traducción de código máquina a instrucciones ensamblador no respeta una relación uno a uno como así se menciona en muchos textos. En su lugar, se utilizan combinaciones de tablas. Aquí se mencionan únicamente las tablas ***T1*** derivada del código de operación y la tabla ***T3*** la cual es derivada del byte selector ***ModRM***. Por supuesto que existen más combinaciones, pero con estas se abarcará la mayoría de las instrucciones y que son de interés para la presente tesis.

La figura 5.10 muestra la naturaleza de las instrucciones X86 y el esquema utilizado para su primera transformación.

$Instrucción ::= (T1)CodOp \mid (T1)CodOp\ I_{u-8} \mid (T1)CodOp\ I_{u-32} \mid$
$(T1)CodOp\ X_{reg8}M\ , X_{reg8} \mid$
$(T1)CodOp\ X_{reg8}\ , X_{reg8}M \mid$
$(T1)CodOp\ X_{reg32}M\ , X_{reg32} \mid$
$(T1)CodOp\ X_{reg32}\ , X_{reg32}M \mid$
$(T3)CodOp\ X_{reg8}M\ , I_{s-8} \mid$
$(T3)CodOp\ X_{reg32}M\ , I_{s-32} \mid$
$(T3)CodOp\ X_{reg32}M\ , I_{s-8} \mid$
$(T3)CodOp\ X_{reg8}M\ , I_{u-8} \mid$
$(T3)CodOp\ X_{reg32}M\ , I_{u-32} \mid$
$(T3)CodOp\ X_{reg32}M\ , I_{u-8} \mid$
$(T3)CodOp\ X_{reg8}M\ , 1 \mid$
$(T3)CodOp\ X_{reg32}M\ , 1 \mid$
$(T3)CodOp\ X_{reg8}M\ , CL \mid$
$(T3)CodOp\ X_{reg32}M\ , CL \mid$
$(T3)CodOp\ X_{reg8}M \mid$
$(T3)CodOp\ X_{reg32}M$

Fig 5.10 Conjunto de instrucciones X86

El nivel de abstracción XML

¿Por qué utilizar XML como representación intermedia para análisis de tipo estático? Esta es una pregunta que surge al construir una herramienta para realizar análisis estático de programas. La respuesta a tal pregunta es espontánea -XML es un lenguaje que se adapta de manera extraordinaria a diferentes tipos de paradigmas. Aunque XML está fuertemente influenciado por la comunidad de Base de Datos, permite consultar orígenes de información irregulares con muy poca distinción entre esquema y datos.

Una representación intermedia que mantenga la riqueza BNF puede ser lograda si se basa en la representación XML.

La idea es proporcionar una forma de acceso uniforme a estructuras jerárquicas, que oculte la complejidad para manipular las representaciones intermedias [Muchnick].

Formato ensamblador-XML

El formato binario transformado a lenguaje ensamblador puede ser almacenado para su posterior análisis en una estructura jerárquica XML. Como ejemplo supongamos que tenemos el código máquina en formato binario plano mostrado en la figura 5.11.

```
5589E58005DDCCBBAA118005DDCCBBAA117EF0E90200000089C389D1E80400000089EC5DC35589E589EC5DC
3
```

Fig 5.11 Código máquina X86

Resultaría de utilidad para posteriores análisis tener el mismo código en un formato jerárquico como el que se muestra en la figura 5.12

Los lenguajes actuales tienen mecanismos de seriación que permiten el análisis y recorrido de código jerárquico como el que se menciona en la figura 5.12.

La tabla de símbolos

La pregunta obligada en este punto es ¿Cuál es la representación intermedia que preserva la riqueza y originalidad de las gramáticas BNF? Por su portabilidad, tal representación intermedia se debe basar en el lenguaje XML.

Pero en el presente trabajo los requerimientos son un poco más modestos y la información que debe almacenar la taba de símbolos es la localidad de la instrucción, la tabla de transformación, la función de transformación y el tipo de operadores. La figura 5.13 proporciona un ejemplo de estructura del programa

El flujo de información y su representación

A diferencia de las dos referencias mencionadas en la sección 4.4 del capítulo 4, los índices en los arreglos inician en 0 y su desplazamiento es irregular considerando el tamaño de las instrucciones en bytes. (ver figura 5.14)

Grafo de control de flujo (CFG)

Si el código que se analiza en un CFG es lenguaje ensamblador, una complicación se puede deber a la ramificación hacia los contenidos de una operación de registro. Este problema puede ser solucionado si el análisis de transformación de código se inicia a partir de código binario. Otro problema de utilizar código ensamblador son las ramificaciones a líneas vacías. En la primera transformación se debe utilizar un contador que contabilice la traza de bytes.

Ejemplo básico de ramificación

Para ilustrar el caso, el análisis parte del programa en formato binario de la figura 5.12. En el fragmento de programa se muestra que el binario utiliza un número de un byte con representación a complemento a 2. La instrucción en la posición 0028 indica una ramificación de salto cercano hacia adelante 0X0C bytes y esta cantidad debe ser sumada a la posición 0028. Las cantidades positivas indican salto adelante y cantidades negativas salto atrás. La figura 5.15 muestra este resultado

```
  <Linea numbyte="0000">
    <CodMaq>55</CodMaq>
    <Ensamblador>PUSH EBP</Ensamblador>
  </Linea>
  <Linea numbyte="0001">
    <CodMaq>89E5</CodMaq>
    <Ensamblador>MOV EBP, ESP</Ensamblador>
  </Linea>
  <Linea numbyte="0003">
    <CodMaq>8005DDCCBBAA11</CodMaq>
    <Ensamblador>ADD [0XAABBCCDD], byte  0X11</Ensamblador>
  </Linea>
  <Linea numbyte="0010">
    <CodMaq>8005DDCCBBAA11</CodMaq>
    <Ensamblador>ADD [0XAABBCCDD], byte  0X11</Ensamblador>
  </Linea>
  <Linea numbyte="0017">
    <CodMaq>7EF0</CodMaq>
    <Ensamblador>JLE  0XF0</Ensamblador>
  </Linea>
:
:
:
```

Fig 5.12 Código ensamblador auto-generado

```
<Simbolo numbyte="00000" tabla="T1" funcion="1" opIzq="-99" opDer="-99" />
  <Simbolo numbyte="00001" tabla="T1" funcion="12" opIzq="510" opDer="600" />
  <Simbolo numbyte="00003" tabla="T3" funcion="23" opIzq="340" opDer="100" />
  <Simbolo numbyte="00010" tabla="T3" funcion="23" opIzq="340" opDer="100" />
  <Simbolo numbyte="00017" tabla="T1" funcion="2" opIzq="100" opDer="-99" />
  <Simbolo numbyte="00019" tabla="T1" funcion="3" opIzq="200" opDer="-99" />
  <Simbolo numbyte="00024" tabla="T1" funcion="12" opIzq="510" opDer="600" />
  <Simbolo numbyte="00026" tabla="T1" funcion="12" opIzq="510" opDer="600" />
  <Simbolo numbyte="00028" tabla="T1" funcion="3" opIzq="200" opDer="-99" />
  <Simbolo numbyte="00033" tabla="T1" funcion="12" opIzq="510" opDer="600" />
  <Simbolo numbyte="00035" tabla="T1" funcion="1" opIzq="-99" opDer="-99" />
  <Simbolo numbyte="00036" tabla="T1" funcion="1" opIzq="-99" opDer="-99" />
  <Simbolo numbyte="00037" tabla="T1" funcion="1" opIzq="-99" opDer="-99" />
  <Simbolo numbyte="00038" tabla="T1" funcion="12" opIzq="510" opDer="600" />
  <Simbolo numbyte="00040" tabla="T1" funcion="12" opIzq="510" opDer="600" />
  <Simbolo numbyte="00042" tabla="T1" funcion="1" opIzq="-99" opDer="-99" />
  <Simbolo numbyte="00043" tabla="T1" funcion="1" opIzq="-99" opDer="-99" />
```

Fig 5.13 Símbolos almacenados en una tabla XML

Componentes de la Matriz Rala		Tipo de Desplazamiento
Arreglo origen	**Arreglo Destino**	
0	1	1 Flujo normal
1	3	2 Flujo normal
3	10	7 Flujo normal
10	17	7 Flujo normal
17	3	14 Condicional
17	19	2 Flujo normal
19	26	7 Incondicional
19	24	5 Flujo normal

Fig 5.14 Ejemplo de uso CSR para almacenar flujo de instrucciones

No Byte	Instrucción Binaria	Transformación Ensamblador	Pseudo-código
0028	7E0C	JLE 0X0C	$if(Op1 \leq Op2)\ goto\ 0042$
0030	8105DDCCBBAADDCCBBAA	ADD [0XAABBCCDD], dword 0XAABBCCDD	$M_{[0XAABBCC]} \mathrel{+}= 0XAABBCCDD$
⋮			
0042	89D1	MOV ECX, EDX	$Reg_{ECX} \leftarrow Reg_{EDX}$

Fig 5.15 Transformación de una ramificación

CFG Utilizando la representación intermedia XML

La utilización de la representación intermedia XML es relevante debido a que

- Captura las estructuras y *toda* la información importante para su análisis en posteriores etapas

- Proporciona una estructura de carácter finito la cual permite verificar flujos de dato

- Permite expresar documentos estandarizando un formalismo para todos los documentos XML

- Debe ser un esquema maduro y completamente determinado

Los grafos XML son una versión generalizada de los árboles XML debido a que

- El texto, atributos y nombres utilizan un lenguaje regular

- Se agregan nodos de opción y de intercalado

- Existen nodos vacíos los cuales son una variante de nodos opción

Por otro lado, en el trabajo presentado por [Maruyamaet Al] utiliza transformaciones a XML para representar el grafo CFG (Control Flow Graph). Una idea similar es aplicada en la presente tesis. Ver figura 5.16

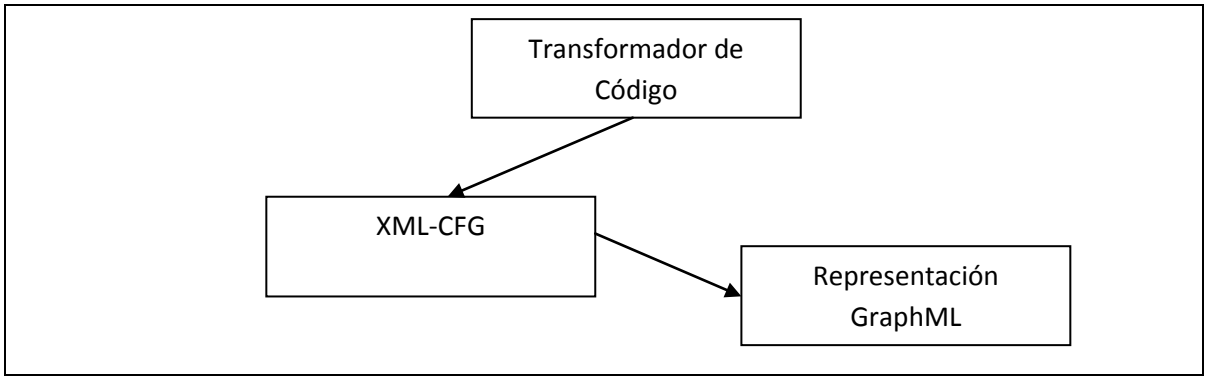

Fig 5.16 Transformación de flujo de instrucciones

Mediante el proceso de seriación se transformó el código binario original en código XML y a partir de esta transformación se generó código GraphML, como se puede apreciar en la figura 5.17. El formato GraphML desde el punto de vista semántico, puede ser analizado en [Brandes et Al]

XML-CFG	GraphML
<Flujo origen="0" destino="1" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="1" destino="3" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="3" destino="8" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="8" destino="15" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="15" destino="22" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="22" destino="24" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="24" destino="8" tipo="Ciclo" operador="Ciclo" /> <Flujo origen="24" destino="26" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="26" destino="28" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="28" destino="42" tipo="CondicionalConSigno" operador="MenorIgual" /> <Flujo origen="28" destino="30" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="30" destino="40" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="40" destino="42" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="42" destino="44" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="44" destino="46" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="46" destino="47" tipo="FlujoNormal" operador="FlujoNormal" /> <Flujo origen="47" destino="48" tipo="FlujoNormal" operador="FlujoNormal" />	<?xml version="1.0" encoding="utf-8"?> <graphml xmlns="http://graphml.graphdrawing.org/xmlns"> <graph id="03000" edgedefault="directed" parse.nodes="16" parse.edges="17" parse.order="nodesfirst" parse.nodeids="free" parse.edgeids="free"> <node id="0" /> <node id="1" /> <node id="3" /> <node id="8" /> <node id="15" /> <node id="22" /> <node id="24" /> <node id="26" /> <node id="28" /> <node id="42" /> <node id="30" /> <node id="40" /> <node id="44" /> <node id="46" /> <node id="47" /> <node id="48" /> <edge id="arcover11.TabSim.Flujo" source="0" target="1" /> <edge id="arcover11.TabSim.Flujo" source="1" target="3" /> <edge id="arcover11.TabSim.Flujo" source="3" target="8" /> <edge id="arcover11.TabSim.Flujo" source="8" target="15" /> <edge id="arcover11.TabSim.Flujo" source="15" target="22" /> <edge id="arcover11.TabSim.Flujo" source="22" target="24" /> <edge id="arcover11.TabSim.Flujo" source="24" target="8" /> <edge id="arcover11.TabSim.Flujo" source="24" target="26" /> <edge id="arcover11.TabSim.Flujo" source="26" target="28" /> <edge id="arcover11.TabSim.Flujo" source="28" target="42" /> <edge id="arcover11.TabSim.Flujo" source="28" target="30" /> <edge id="arcover11.TabSim.Flujo" source="30" target="40" /> <edge id="arcover11.TabSim.Flujo" source="40" target="42" /> <edge id="arcover11.TabSim.Flujo" source="42" target="44" /> <edge id="arcover11.TabSim.Flujo" source="44" target="46" /> <edge id="arcover11.TabSim.Flujo" source="46" target="47" /> <edge id="arcover11.TabSim.Flujo" source="47" target="48" /> </graph> </graphml>

Fig 5.17 Transformación XML-CFG a GraphML

La transformación a GraphML permite la fácil visualización del grafo. Existen varios visualizadores para grafos y en el presente estudio se utilizará Graph# (ver figura 5.18)

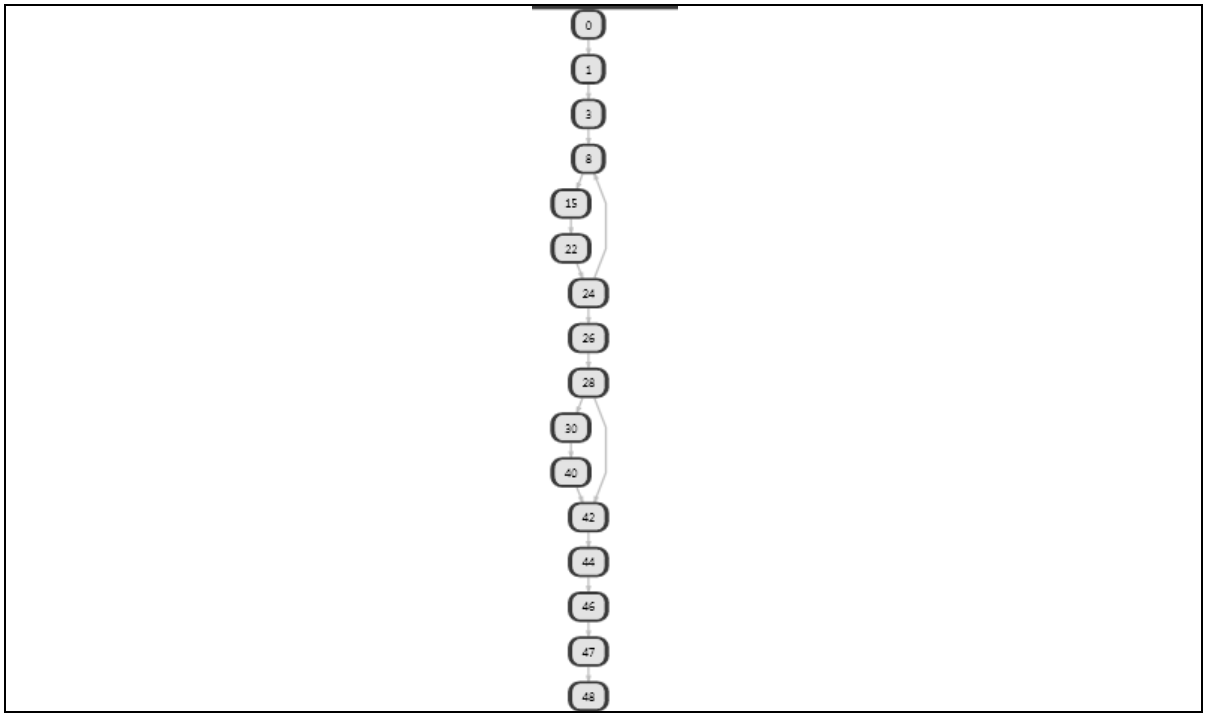

Fig 5.18 Visualización CFG utilizando Graph#. (Algoritmo Efficient Sugiyama)

La fase de generación CIL

En la última etapa de transformación se utilizó el API para construcción de compiladores *Irony .NET Compiler Construction Kit* de Roman Ivantsov's. [Ivantsov]. Aunque esta herramienta se encuentra aún en desarrollo, tiene varias ventajas

- Utiliza algoritmos de parsing NLALR
- Construye árboles sintácticos concretos y abstractos.
- Automatiza el proceso de creación del AST, logrando con esto un ahorro sustancial de tiempo
- Las reglas son expresadas de manera directa en C#

- La herramienta proporciona un explorador de gramáticas que facilitan la escritura de producciones.

AST

Como se mencionó en la sección 4.8 del capítulo anterior, un AST (Abstract Syntax Tree) es una estructura que permite almacenar información para análisis de verificación de tipos y generación de código. Un AST contiene nodos de diferente naturaleza y su complejidad es directamente proporcional al número de nodos que contiene multiplicado por el tipo de atributos que puede tener cada nodo. Esta complejidad obliga a que se desarrollen funciones también de muy alta complejidad con el objetivo de manipular estas estructuras. La construcción de componentes de un compilador de manera tradicional impone una vista monolítica de todo el sistema, haciendo difícil el re-uso de dichos componentes.

Los nodos que serán manipulados en la última fase de transformación, están contenido en el paquete Nodos. Las clases contenidas en este paquete se esquematizan en la Tabla 5.3. Las clases en el paquete Nodos se relacionan de manera directa con las reglas gramaticales que permiten generar código CIL.

Relación de recorrido de los nodos AST

La figura 5.19 detalla la forma en que se encuentran relacionados cada uno de los nodos en el AST.

La relación de nodos asegura el recorrido utilizando las reglas de la gramática que a continuación mostraremos

Nombre	Descripción
NodoEstatuto	Nodo que expresa cualquier tipo de estatuto en lenguaje X86
NodoEstDiv	Nodo que expresa cualquier tipo de estatuto de división en lenguaje X86
NodoEstProd	Nodo que expresa cualquier tipo de estatuto de producto en lenguaje X86
NodoEstSub	Nodo que expresa cualquier tipo de estatuto de substracción en lenguaje X86
NodoEstSum	Nodo que expresa cualquier tipo de estatuto de suma en lenguaje X86
NodoAsignaMemReg32	Nodo que expresa un estatuto de asignación de Memoria a Registro de 32 bits X86
NodoAsignaReg32Mem	Nodo que expresa un estatuto de asignación de Registro de 32 a Memoria en X86
NodoAsigR32R32	Nodo que expresa un estatuto de asignación de Reg de 32 bits a Reg de 32 bits en X86
NodoDivMem	Nodo que expresa división usando operando de memoria X86
NodoDivR32	Nodo que expresa división usando operando de Registro de 32 bits X86
NodoExpresion	Nodo que permite definir una expresión en instrucciones X86
NodoFinPrograma	Nodo que permite definir el fin de un programa X86 (función)
NodoGenericoCliAsm	Nodo que auxilia en la interfaz de lenguaje CIL – ASM
NodoInicioPrograma	Nodo que permite marcar el inicio de un programa X86 (función)
NodoListaEstatutos	Nodo que expresa una lista de estatutos X86
NodoProdMem	Nodo que expresa la operación de producto usando el operando de memoria X86
NodoProdR32	Nodo que expresa la operación de producto usando el operando de registro de 32 bits X86
NodoPrograma	Nodo que define un programa X86
NodoReg32	Nodo que expresa un registro de 32 bits X86
NodoSubMemReg32	Nodo que expresa una operación de resta de Memoria a registro de 32 bits X86
NodoSubR32R32	Nodo que expresa una operación de resta de Registro de 32 bits a Registro de 32 bits X86
NodoSubReg32Mem	Nodo que expresa una operación de resta de Registro de 32 bits a Memoria X86
NodoSumMemReg32	Nodo que expresa una operación de suma de Memoria a Registro 32 bits X86
NodoSumR32R32	Nodo que expresa una operación de suma de Registro 32 bits a Registro 32 bits X86
NodoSumReg32Mem	Nodo que expresa una operación de suma de Registro 32 bits a Memoria X86
NodoVariableSet	Nodo de especificación de variable X86

Tabla 5.3 Conjunto de Nodos AST

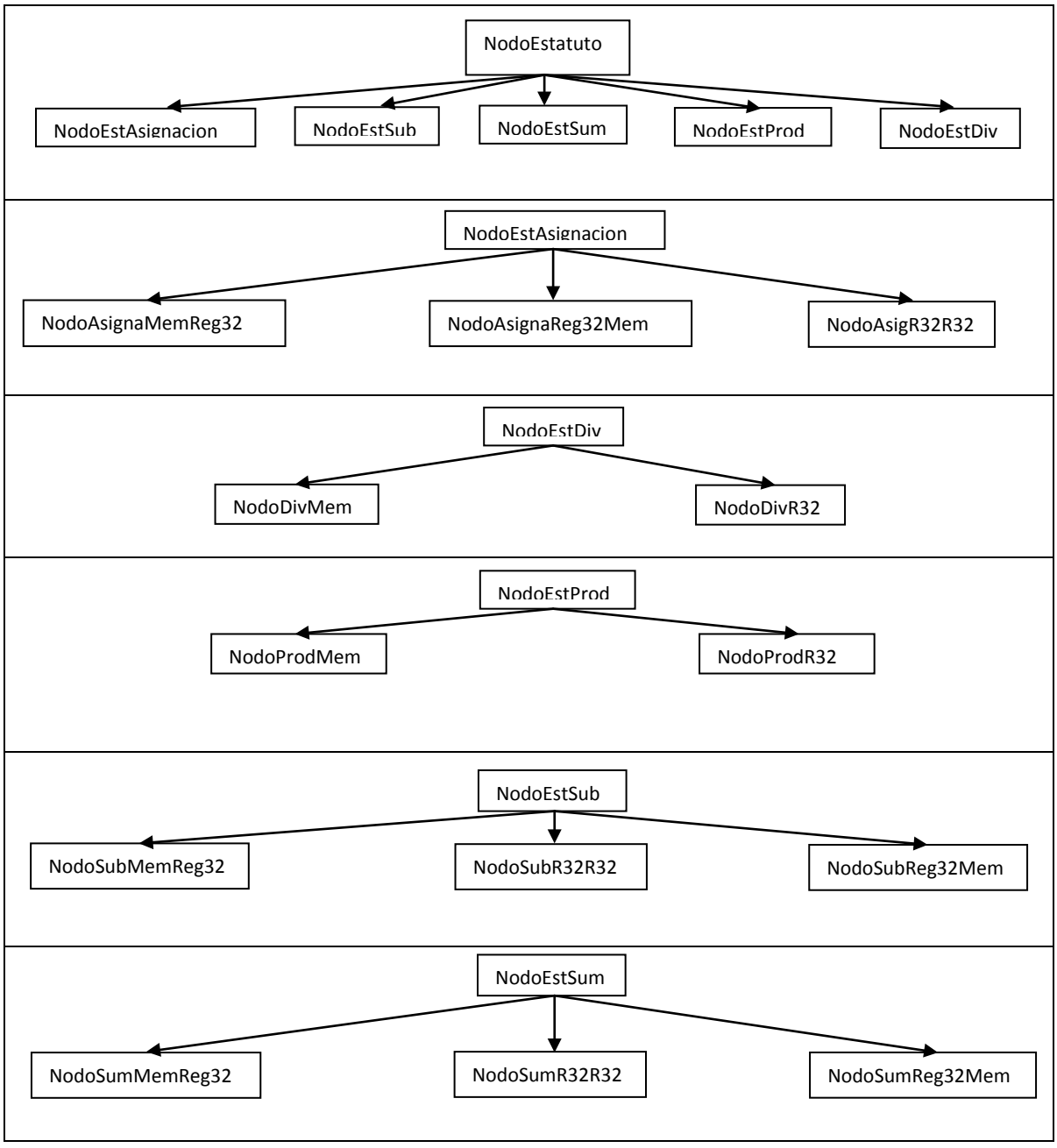

Fig 5.19 Relación de nodos del AST

Gramática

La gramática que fue utilizada se desarrollo en el lenguaje C# y está esquematizada en le figura 5.20

$estatuto ::= estAsignacion \mid estSub \mid estSum \mid estProd \mid estDiv$
$listaEstatutos ::= estatuto *$
$Programa ::= iniciaPrograma\ listaEstatutos\ finPrograma$
$estatutoAsignacion ::= asignaReg32 \mid asignaMemReg32 \mid asignaReg32Mem$
$asignaReg32Reg32 ::= \text{“}mov\text{”}\ reg32\ comma\ reg32$
$asignaMemReg32 ::= \text{“}mov\text{”}\ \text{“}[\text{“}\ numero\ \text{“}]\text{”}$
$asignaReg32Mem ::= \text{“}mov\text{”}\ reg32\ comma\ \text{“}[\text{“}\ numero\ \text{“}]\text{”}$
$estSub ::= substraeReg32Reg32 \mid substraeMemReg32 \mid substraeReg32Mem$
$substraeReg32Reg32 ::= \text{“}sub\text{”}\ reg32\ comma\ reg32$
$substraeMemReg32 ::= \text{“}sub\text{”}\ \text{“}[\text{“}\ numero\ \text{“}]\text{”}\ comma\ reg32$
$substraeReg32Mem ::= \text{“}sub\text{”}\ reg32\ comma\ \text{“}[\text{“}\ numero\ \text{“}]\text{”}$
$estSuma ::= sumaReg32Reg32 \mid sumaMemReg32 \mid sumaReg32Mem$
$sumaReg32Reg32 ::= \text{“}add\text{”}\ reg32\ comma\ reg32$

$sumaMemReg32 ::= \text{“}add\text{”}\ \text{“}[\text{“}\ numero\ \text{“}]\text{”}\ comma\ reg32$
$sumaReg32Mem ::= \text{“}add\text{”}\ reg32\ comma\ \text{“}[\text{“}\ numero\ \text{“}]\text{”}$
$estProd ::= prodReg \mid prodMem$
$prodReg ::= \text{“}mul\text{”}\ reg32$
$prodMem ::= \text{“}mul\text{”}\ \text{“}[\text{“}\ numero\ \text{“}]\text{”}$
$estDiv ::= divR32 \mid divMem$
$divR32 ::= \text{“}div\text{”}\ reg32$
$divMem ::= \text{“}div\text{”}\ \text{“}[\text{”}\ numero\ \text{“}]\text{”}$

Fig 5.20 Gramática utilizada para prueba

Construyendo el algoritmo de recorrido sobre nodos del AST

El patrón Visitor junto con el algoritmo primero en profundidad se convierte en una herramienta poderosa de exploración debido a que se puede acceder a diferentes tipos de nodos sobre un árbol AST [Egilsson].

Expresiones Lambda en LINQ para recorrer nodos en AST

Ejecución diferida en el lenguaje LINQ [Microsoft bb943859].

Esta sección tiene como base teórica la información mencionada en el punto 4.*9.1* del capítulo anterior. En aquella sección se mencionó que, por ejecución diferida se indica que la evaluación es retardada hasta que su valor sea requerido. La ejecución diferida es proporcionada por el lenguaje LINQ mediante el uso de la palabra reservada yield cuando es utilizada dentro de un bloque iterador. Cuando se escribe un método mediante ejecución diferida, también se puede utilizar la evaluación retrasada o la evaluación inmediata.

- En la evaluación retrasada un solo elemento de la colección es procesado durante cada llamada al iterador. Esta es la forma común de funcionamiento de los iteradores

- En la evaluación inmediata, resultará que en la primera llamada todos los elementos de la colección serán procesados.

La evaluación retrasada usualmente tiene mejor rendimiento debido que distribuye la sobrecarga de ejecución en la colección completa.

Construyendo el algoritmo de recorrido sobre nodos del AST

El patrón Visitor junto con el algoritmo primero en profundidad se convierte en una herramienta poderosa de exploración debido a que se puede acceder a diferentes tipos de nodos sobre un árbol AST [Egilsson].

La figura 5.21 ilustra el algoritmo primero en profundidad utilizando el NodoGenericoCliAsm. En este algoritmo también se utiliza evaluación retrasada para recorrer cada uno de los nodos.

Expresiones Lambda en LINQ

Aunque esta forma es poco expresiva, ofrece algunas ventajas en ciertos lenguajes tales como LINQ. El ejemplo en la figura 5.22 muestra la utilización de expresiones lambda.

```
public static IEnumerable<NodoAst> RecorridoPrimProf(this NodoAst nodoInicio)
    {
       yield return nodoInicio;

       NodoGenericoCliAsm portadorHijo = nodoInicio as NodoGenericoCliAsm;

       if(portadorHijo != null)
       {
          foreach(NodoAst hijo in portadorHijo.NodosHijo)
          {
             if (hijo is NodoGenericoCliAsm)
             {
                foreach (var x in RecorridoPrimProf(hijo))
                {
                   yield return x;
                }
             }
             else
             {
                yield return hijo;
             }

          }
       }
    }
```

Fig 5.21 Algoritmo Primero en profundidad utilizando el patrón Visitor y la evaluación retardada.

Expresión Lambda	Expresividad LINQ
$(\lambda x.x * x)(3+2) \longrightarrow (3+2) * (3+2)$	$lambda = x => x * x;$

Fig 5.22 Representación lambda y expresiones lambda en LINQ

Las funciones de alto nivel de la programación funcional permiten realizar el filtrado y la transformación de la información. La sintaxis de expresiones de consulta que utiliza LINQ también permite el filtrado y la transformación. Utilizando esta sintaxis, el recorrido sobre el AST resulta simple como se puede apreciar en la figura 5.23

```
var NodosSuma = from es in nodoRaiz.RecorridoPrimProf()
                where es is NodoEstSum
                select (NodoEstSum)es;
```

Fig 5.23 Sintaxis declarativa LINQ para recorrer nodos sobre el AST

El código generado CIL en la última fase de trans*formación*

El código generado en la última transformación respeta el estándar [ECMA-335]. Los componentes elementales se encuentran encapsulados en un DLL como formatos de función (ver sección 5.13) y las llamadas a estas funciones también se encuentran en formato CIL. Debido a que los registros son declarados como campos de tipo estático, se utilizan las instrucciones **ldsfld** y **ldsflda**.

- La instrucción **ldsfld** se utiliza para pasar el argumento como *valor* a la llamada a función

- La instrucción **ldsflda** se utiliza para pasar el argumento como referencia a la llamada a función

La utilización de memoria también respeta el estándar [ECMA-335], y se utilizan las siguientes instrucciones

- **ldsfld ds**, para enviar la dirección de inicio del segmento de datos

- **ldsfld offset**, para marcar la dirección del desplazamiento sobre el segmento de datos

- **ldelema [mscorlib] System.Int32**, para indicar la utilización de la biblioteca mscorlib sobre ese elemento

Encapsulación de los bloques elementales

Al igual que el código de llamadas a los bloques básicos (ver sección 5.13) los mismos bloques elementales se encuentran codificados y encapsulados en un DLL respetando también el estándar [ECMA-335].

La encapsulación utiliza muchas de las producciones de este estándar, pero aquí se hace mención únicamente a las más importantes Figura 5.24.

Ensamble $Decl :: = .assembly\ DottedName$ '{' $AsmDecl *$ '}'
Ensamble Externo $Decl :: = .assembly\ extern\ DottedName$ '{' $AsmRefDecl *$ '}'
Clase $Decl :: = ClassHeader$ '{' $ClassMember *$ '}'
Campo $Decl :: = .Field\ FieldDecl$
Método $Decl :: = .method\ MethodHeader$ '{' $MethodBodyItem *$ '}'
Módulo $Decl: := .module\ [Filename]$

Fig 5.24 Componentes del estándar ECMA-335 usados en la construcción de los bloques elementales

Herramientas utilizadas

Como se mencionó en las dos secciones anteriores, tanto las acciones semánticas como el código generado fueron desarrolladas en CIL. Para realizar esto se utilizó una herramienta de desensamblado ILDASM la cual es proporcionada de manera gratuita por Microsoft. En la figura 5.25 se puede apreciar la utilización de esta herramienta que muestra las acciones semánticas.

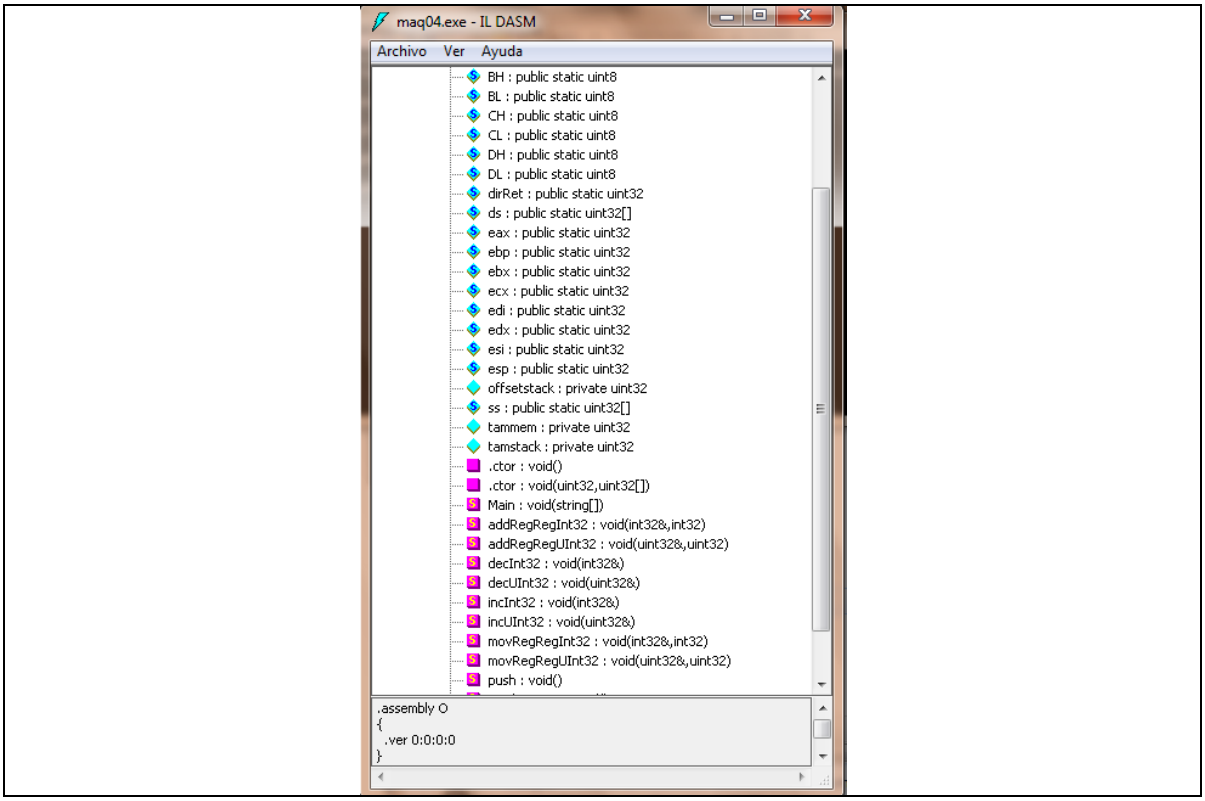

Fig 5.25 Acciones Semánticas uso de la herramienta ILDASM

Para el desarrollo de código CIL se utilizó ILASM. Esta herramienta permitió probar cada uno de las acciones semánticas de manera inmediata.

VI. CONCLUSIONES

Proponer un esquema que permita probar código binario es una tarea larga y laboriosa y la cual requiere de abordar varios conceptos que versan desde el uso de estándares hasta aspectos relacionados con la construcción de compiladores y técnicas de transformación de código. Durante el año en que se desarrollo la presente tesis, salieron a la luz una serie de trabajos que también analizaban tal tema. Todos ellos permitieron enriquecer el material que fue presentado aquí.

Cada aporte a este tema tiene un significado especial debido a que se parte del uso de herramientas de Ingeniería Inversa las cuales hasta hace poco eran consideradas prohibidas en el ámbito académico. En la actualidad Ingeniería Inversa es una etapa substancial en la Ingeniería de Software y es el primer paso a considerar en el proceso de re-Ingeniería.

Ciertamente la gramática que fue utilizada para construir el prototipo no considera el total de las instrucciones de los procesadores x86, sin embargo, el juego de instrucciones analizadas consideran los bloques elementales a los que se hace mención en el título del presente trabajo.

La traducción a código intermedio en la última fase, tiene sus ventajas debido a que permite enfocarse en aspectos de parsing, evitando tener que construir una plataforma para realizar pruebas.

Al igual que otras tesis de la misma naturaleza, pero de diferente objetivo [Egilsson], se puede concluir que .NET es una buena plataforma para desarrollar de-compiladores. El principal beneficio es su facilidad para interactuar con otros lenguajes de la misma familia. Es fácil anexar estructuras adicionales mediante la escritura de bibliotecas en lenguajes tales como el C#.

La escritura de construcciones semánticas puede ser fácilmente representada en lenguaje IL [Oudheusden]. El analizador semántico resultante utilizando bloques IL se divide en tres partes: extractores, meta-modelo y mecanismos de búsqueda.

Por otro lado, LINQ es una herramienta que al permitir el filtrado funcional mediante el uso de expresiones lambda, se puede realizar con ella búsquedas eficientes en los AST [Robinson].

Debido a todo el cúmulo de conocimientos acumulados sobre la teoría de compiladores y las herramientas existentes para construcción de estas estructuras, se puede concluir que en la actualidad se puede diseñar un esquema de especificación (tema VI) el cual puede ser utilizada para desarrollar una herramienta (tema V) que permita traducir de código binario a código de más alto nivel con el propósito probar la exactitud de su funcionamiento.

Trabajos relacionados

En Septiembre de 2010, se presentó una disertación doctoral cuyo nombre es Static Analysis of x86 Executables [Kinder] y en la cual se abordan conceptos relacionados con el análisis estático. Esta tesis tiene como finalidad la transformación de código binario a código intermedio, para su posterior uso en pruebas de software. Aborda el proceso completo utilizando algunas herramientas heurísticas. Aunque este enfoque es interesante tiene un punto débil –no puede ser utilizada con total certidumbre en software de misión crítica. El enfoque es parecido al de la presente tesis pero en el trabajo de [Kinder] se utilizaron algunas técnicas de tipo heurístico para analizar el flujo de programa.

La plataforma utilizada para desarrollar el prototipo fue la máquina virtual de Java y la traducción final a lenguaje CIL.

Como se mencionó en el capítulo 2, la plataforma .NET tiene varias ventajas cuando se le compara con la máquina virtual de Java.

Trabajo a futuro

Las arquitecturas reflexivas son utilizadas para permitir a los sistemas de software definir su comportamiento en tiempo de ejecución basado en información la cual puede ser almacenada en metadatos. La reflexión es una tecnología que está estrechamente ligada a la meta-información. La meta-información describe la estructura, semántica propósito y uso de la información. La tecnología de la meta-información proporciona una solución eficiente para administrar y controlar el uso de información compleja. [Juan-feng et Al] Teniendo esta información en cuenta, es factible desarrollar una arquitectura que permita realizar pruebas de software de código binario a un más alto nivel de abstracción. Las ventajas de realizar esto saltan a la vista, debido a que se puede tener un mayor control desde el punto de vista programático.

REFERENCIAS BIBLIOGRÁFICAS

Aho, A. V., Sethi, R., Ullman J D. Compilers: Principles, Techniques and Tools. Addison-Wesley, Mass., 1986

Anders, Moller, et Al. XML Graphs in Program Analysis. · Proceeding PEPM '07 Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. ACM New York, NY, USA ©2007

Anley, Chris. Creating Arbitrary Shellcode In Unicode Expanded String. January 2002

Boyer, Robert S., Yu Yuan 1992. Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor. Proceeding of the 11th International Conference on Automated Deduction: Automated Deduction. Pages: 416-430. (1992). ISBN:3-540-55602-8

Brandes, Ulrik., Pich, Christian. GraphML Transformation (2004). Proceedings of the 11TH International Symposium on Graph Drawing.

Buchli, Frank 2003. Detecting Software Patterns using Formal Concept Analysis. Master Thesis. University of Bern.

Chaudhari, Mahesh B. et Al. A Distributed Event Stream Processing Framework for Materialized Views over Heterogeneous Data Sources. Arizona State University 2010.

Cifuentes, Cristina. Reverse Compilation Techniques. Submitted to the School of Computing Science in partial fulfilment of the requirements for the degree of Doctor of Philosophy. QUEENSLAND UNIVERSITY OF TECHNOLOGY, July 1994.

Cooper, Keith D. Cooper, et Al. Building a Control-flow Graph from Scheduled Assembly Code. 2002. CiteSeer

ECMA-335. Standard ECMA-335 Common Language Infrastructure (CLI). 5th edition (December 2010)

Egilsson, Einar. A Process Language Runtime for the .NET Platform. Technical University of Denmark Kongens Lyngby 2009.

Grechanik, Mark, et Al. XML-based Intermediate Representation (XIR). 2002

Grune D. et Al. Modern Compiler Design. Wiley, 2001

Hamilton, Jennifer. Language integration in the common language runtime. ACM SIGPLAN Notices. Volume38 Issue 2, February 2003

Heberle, Andreas et Al. Names, Types, and Static Semantic Analysis. CiteSeer 1997

Hilzer, Ralph C. et Al. A Survey of Sequential and Parallel Implementation Techniques for Functional Programming Languages. Oregon State University Corvallis, OR, USA ©1995. ACM Digital Library

Intel 32. Intel 32 architecture memory ordering white paper 2007.

Ivantsov, Roman. Lang .NET Symposium 2009. Microsoft Corporate, Redmond, WA. April 2009

Jeanna Neefe Matthews, et Al. Quantifying the performance isolation properties of virtualization systems. ExpCS '07 Proceedings of the 2007 workshop on Experimental computer science. ACM New York, NY, USA ©2007

Juan-feng, Yao. Et Al. Reflective Architecture Based Software Testing Management Model. Management of Innovation and Technology, 2006 IEEE International Conference. June 2006

Kebler Christoph W., Smith Craig H. The SPARAMAT Approach to Automatic Comprehension of Sparse Matrix Computations. 1999 IEEE

Khalili, Hamid, et Al. Web-graph pre-compression for similarity based algorithms. Third International Conference on Modeling, Simulation and Applied Optimization, Sharjah, U.A.E, 2009, 20-22 .January 2009

Khedker, Uday et Al. Data Flow Analysis: Theory and Practice. CRC Press; 1 edition (March 27, 2009). ISBN-10: 0849328802

Kinder, Johannes. Static Analysis of x86 Executables. Fachbereich Informatik Technische Universit¨at Darmstadt. September 2010.

Lanza, Michele 2003. Object-Oriented Reverse Engineering - Coarse-grained, Finegrained, and Evolutionary Software Visualization. PhD thesis, University of Berne, 2003.

Liu, Shih-His, et Al. A SOA Approach for Domain-Specific Language Implementation, 2010 6th World Congress on July 2010.

Maruyama, Katsuhisa et Al. Design and Implementation of an Extensible and Modifiable Refactoring Tool, Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on. Issue Date: 15-16 May 2005

Mayreen O., Magnus, et Al. Machine-code verification for multiple architectures. –decompilation into logic-. Feb 2009

Meijer, Erik, et Al. Technical Overview of the Common Language Runtime. Microsoft, 2000

Microsoft 2010. Microsoft Portable Executable and Common Object File Format specification. Revision 8.2 – September 21, 2010

Microsoft, Ejecución aplazada. msdn.microsoft.com, Microsoft 2010

Microsoft, Metadata. http://msdn.microsoft.com/en-us/library/8dkk3ek4%28v=vs.71%29.aspx.

Microsoft bb943859. http://msdn.microsoft.com/en-us/library/bb943859.aspx

Misurda, J. Clause, J. Reed, J. Childers, B. R. Soffa, M. L. Jazz: A Tool for Demand-Driven Structural Testing. LECTURE NOTES IN COMPUTER SCIENCE. Publisher SPRINGER-VERLAG. ISSN 0302-9743. 2005

Muchnick, S. Advanced Compiler Design and Implementation. Morgan Kaufmann

Oudheusden M.D.W van. Automatic Derivation of Semantic Properties in .NET. Enschede, August 28, 2006.

Pretschner A., Prenninger W., Wagner S., K¨uhnel C., Baumgartner M., Sostawa B., Z¨olch R., Stauner T. One evaluation of modelbased testing and its automation, in: Proc. ICSE'05, 2005

Robinson R. Aaron. Integrating a Universal Query Mechanism into Java. September 8, 2007

Sarkar, Susmit, et Al. The Semantics of x86-CC Multiprocessor Machine Code. POPL '09 Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM New York, NY, USA ©2009

Schmitz, Sylvain 2005. Noncanoical LALR(1) Parsing. CiteSeer 2005

Schmitz, Sylvain 2007. Approximating Context-Free Grammars for Parsing and VerificationLaboratoire I3S, Universit´e de Nice - Sophia Antipolis & CNRS, France, 2007

Schwartzbach, Michael I. Lattice Theory Control Flow Graphs Dataflow Analysis. Static Analysis 2009. Computer Science, University of Aarhus

Thompson, Ken. Reflections on Trusting Trust. Communications of the ACM. vol. 27, no. 8 (August 1984)

Utting Mark, Pretschner, Legerad Bruno. A taxonomy of model-based testing. Working Papers 2006. Department of Computer Science, The University of Waikato (New Zealand), April 2006

Vadera Suni, Meziane Farid. Tools for producing formal specifications: a view of current architectures and future Directions. March 12, 2003

Vianu, Victor. A Web Odyssey: from Codd to XML. U.C. San Diego. ACM New York, USA 2001

Villalobos M. y Gutiérrez A. Investigación sobre las prácticas de ingeniería de software en México. Asociación Mexicana de Calidad para la Ingeniería de Software (AMCIS) y el Laboratorio de Sistemas de Información del CIC-IPN. (Agosto de 2001).

Wahbe, Robert, et Al. Efficient software-based fault isolation**.** Proceeding SOSP '93 Proceedings of the fourteenth ACM symposium on Operating systems principles. ACM New York, NY, USA ©1993