INSTITUTO POLITÉCNICO NACIONAL

UNIDAD PROFESIONAL INTERDISCIPLINARIA DE INGENIERÍA Y CIENCIAS SOCIALES Y ADMINISTRATIVAS

SECCIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

PROCEDIMIENTO DE ANÁLISIS PARA SISTEMAS HEREDADOS EN LA PLATAFORMA MAINFRAME

T E S I S

QUE PARA OBTENER EL GRADO DE MAESTRO
EN CIENCIAS EN INFORMÁTICA
P R E S E N T A :

ANTONIO VEGA ELIGIO

DIRECTOR:
M. EN C. EDUARDO RENÉ RODRÍGUEZ ÁVILA

MÉXICO, D.F. AÑO 2011.

SIP-14

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 12:00 horas del día 27 del mes de junio del 2011 se reunieron los miembros de la Comisión Revisora de Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación de UPIICSA para examinar la tesis titulada:

"PROCEDIMIENTO DE ANÁLISIS PARA SISTEMAS HEREDADOS EN LA PLATAFORMA MAINFRAME"

Presentada por el alumno:

VEGA	ELIGIO	ANTONIO
Apellido paterno	Apellido materno	Nombre(s)

Con registro: B 0 7 1 6 6 7

aspirante de:

MAESTRO EN CIENCIAS EN INFORMÁTICA

Después de intercambiar opiniones, los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Director de tesis

M. en C. EDUARDO RENE RODRIGUEZ ÁVILA

DR. MAURICIO JORGE PROCEL MORENO

DR. NICOLÁS RODRÍGUEZ PEREGO

M. en C. EMILIA ÀBAD RUÍZ

M. en C. GUILLERMO PÉREZ VÁZQUEZ

LA PRESIDENTA DEL COLEGIO

DRA. MARÍA ELENA TAVERA CORTÉS

API/mpcs

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de México, Distrito Federal, el día 10 del mes de junio del año 2011, el que suscribe Antonio Vega Eligio, alumno del Programa de Maestría en Ciencias en Informática con número de registro B071667, adscrito a la Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, manifiesta que es autor intelectual del presente trabajo de Tesis bajo la dirección del M. C. Eduardo René Rodríguez Ávila y cede los derechos del trabajo titulado **PROCEDIMIENTO DE ANÁLISIS PARA SISTEMAS HEREDADOS EN LA PLATAFORMA MAINFRAME**, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección avegael@hotmail.com. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Antonio Vega Eligio

A mi hijo, te tengo en mi memoria.

A mis padres y hermanos que son fuente de amor y apoyo incondicional.

AGRADECIMIENTOS.

A mi director de tesis por su guía durante la incepción, desarrollo y conclusión del presente estudio, así como por sus revisiones sobre el contenido y sus acertadas correcciones.

A mi comité tutorial por sus revisiones y comentarios.

A los profesores que me impartieron varios de los cursos de la maestría en informática. El conocimiento que recibí tiene un valor muy apreciado.

A mi profesor de la materia Seminario Departamental 1, quien proporcionó una guía sólida para elegir, iniciar y desarrollar el presente estudio con un método.

A mis ex-compañeros de trabajo analistas de sistemas y líderes de proyecto en la plataforma mainframe con quienes tuve la fortuna de trabajar durante la implementación de proyectos informáticos. Gracias por ayudarme a responder la encuesta que realicé y por sugerir ideas de contenido.

A todas las demás personas que me apoyaron de una manera u otra me durante el desarrollo del presente proyecto académico.

Al Instituto Politécnico Nacional por ser una institución que hace posible la investigación y mejora académica continua en México.

Antonio Vega
Junio de 2011.

RESUMEN.

Debido a sus características y otras causas, gran parte del análisis de aplicaciones que se hace actualmente en los ambientes mainframe es en la mayoría de los casos manual. Cuando un error es reportado, uno o varios un analistas de sistemas examinan el código fuente del programa junto con los mensajes de error para determinar cuál puede ser y dónde está el problema. De la misma forma se analiza dónde debe modificarse el código para agregar una nueva funcionalidad o bien qué debe hacerse para integrar un nuevo módulo o aplicación. En ambos casos esto requiere de una cantidad considerable de tiempo, es altamente dependiente de un grupo específico de personas (las que están familiarizadas con el aplicativo) y tiene la posibilidad de introducir nuevos errores.

El presente trabajo de investigación es de naturaleza aplicativa. Presenta la utilización de diferentes técnicas de ingeniería de software y de análisis sintáctico empleadas en la construcción de compiladores en el planteamiento de un procedimiento semiautomático de análisis de código, en particular para un sólo componente de compilación que forme parte de sistemas heredados en la plataforma descrita, si bien el procedimiento a plantear no libera del todo al analista de hacer el trabajo manual, sí se anticipa como una ayuda importante para el análisis y así minimizar el riesgo asociado y tiempo empleado. Como una consecuencia directa e inmediata de esta iniciativa se prevé la reducción de horas de trabajo y por tanto de costos asociados a este rubro.

El procedimiento tiene en la medida de lo posible un enfoque general, sin embargo donde así lo requiere el caso aplicativo, el enfoque es específico al lenguaje dominante de la plataforma. Se eligió el caso de estudio durante la experiencia profesional del autor, al detectar varios problemas con relación a la etapa de análisis de código en los proyectos en que éste participó. El planteamiento de la solución de algunos de estos problemas es una oportunidad para generar conocimiento aplicativo en el tema, como caso particular, para el lenguaje *COBOL* de la plataforma mainframe de *IBM®* donde gran parte de los productos tienen licencia de uso con precios elevados y donde el costo de capital humano para desarrollo también es caro.

ABSTRACT.

Due to its own characteristics and many other reasons, application analysis on mainframe environments is mainly performed manually. When an error is reported one or several analysts examine program's source code and all associated error messages to determine which and where the problem could be in. In the same way, source code is analyzed to know where it should be modified to introduce a new functionality or to add a new module or application. In both cases, this is an activity that requires a lot of time, it is extremely dependent on specific groups of persons (those who are familiar with the applications) and it could introduce new errors.

This research describes how to apply different techniques used in Software Engineering and Compilers Theory to propose a semiautomatic code-analysis procedure on one single unit of compilation, to be used for the mainframe platform, although this procedure does not eliminate manual work completely, it is anticipated as an important tool to help the analyst in its labor and therewith an effort to minimize time usage and associated risk. As a direct consequence of this initiative a man-hours reduction is expected and therefore of all of associated costs.

As far as possible the procedure was developed attempting a general approach; however in some stages of the development, focus in some specific properties of the prevailing language in the platform was required. The practical application was chosen based on the author's professional experience and participation on several projects when several issues at source code analysis were detected. The solution proposed to some of those problems is a good opportunity to generate practical knowledge on analysis matters, as in this particular case for *COBOL* language on the *IBM®* mainframe platform where software licenses' prices are high and the same can be said in regards to the analysis and development work.

Keywords: Functional Analysis, *COBOL*, mainframe, Legacy Systems.

CONTENIDO.

LISTA DE FIGURAS.

ABREVIATURAS Y SÍMBOLOS.

.NET. .NET es un framework, APIs de soporte de desarrollo, de Microsoft que hace un énfasis en la transparencia de redes, con independencia de plataforma de hardware y que permita un rápido desarrollo de aplicaciones.

ACM. Acrónimo de "Association for Computing Machinery". Fue fundada en 1947 como la primera sociedad científica y educativa acerca de la Computación.

ACS. Affiliated Computer Services Inc. (ACS) es una empresa que proporciona servicios de tecnología de la información y soluciones de manufactura externa de procesos de negocio (business process outsourcing solutions) a empresas, agencias de gobierno y organizaciones sin fines de lucro. ACS tiene su sede en Dallas, Texas.

AJAX. Ajax, acrónimo de Asynchronous JavaScript And XML (JavaScript asíncrono y XML), es una técnica de desarrollo Web para crear aplicaciones interactivas o RIA (Rich Internet Applications). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano.

AN/FSQ-7. Fue un modelo de computadora desarrollada y construida en la década de los 1950s por IBM® en asociación con la fuerza aérea de los Estados Unidos de Norteamérica.

Apache Tomcat. Tomcat (también llamado Jakarta Tomcat o Apache Tomcat) funciona como un contenedor de servlets desarrollado bajo el proyecto Jakarta en la "Apache Software Foundation". Tomcat implementa las especificaciones de los servlets y de JavaServer Pages (JSP) de Sun Microsystems.

API. Una interfaz de programación de aplicaciones o API (del inglés Application Programming Interface) es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

AS/400. El sistema AS/400 es un equipo de *IBM®* de gama media y alta, para todo tipo de empresas y grandes departamentos. Se trata de un sistema multiusuario, con una interfaz controlada mediante menús y comandos (CL Control Language) intuitivos que utiliza terminales y un sistema operativo basado en objetos y bibliotecas, denominado OS/400.

ASG. ASG Software Solutions proporciona servicios de administración de servicios de negocio (business service management), entre otros.

ASG-Existing Systems Workbench (ESW) Suite. Es un conjunto de productos de la empresa ASG® Software Solutions que se utilizan para administrar sistemas ya existentes. La suite ASG-ESW es una herramienta de entendimiento integrada que soporta aplicaciones heredadas,

reingeniería y mantenimiento de procesos al proporcionar una solución de tipo ciclo de vida de sistemas para administrar sistemas ya existentes.

BMC. BMC Software, Inc. (NASDAQ: BMC) es un corporativo multinacional especializado en software de administración de servicios de negocio (Business Service Management BSM), su casa matriz se encuentra en Houston, Texas.

C. Es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell. Es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones.

C++. C++ es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.

CA. CA Technologies (NASDAQ: CA), anteriormente CA, Inc. y Computer Associates, Inc., es una compañía del grupo Fortune 500 y una de las corporaciones de software independiente más grandes en el mundo. Su corporativo matriz se encuentra en Islandia, New York.

CASE. Las herramientas CASE (Computer Aided Software Engineering, Ingeniería de Software Asistida por Computadora) son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software reduciendo el costo de las mismas en términos de tiempo y de dinero.

CICS. *CICS*, acrónimo en inglés de Customer Information Control System (en español, Sistema de control de información de clientes), es un gestor transaccional, o monitor de teleproceso, que se ejecuta principalmente en mainframes *IBM*® con los sistemas operativos OS/390 o z/OS. También existen versiones de *CICS* para otros entornos, como OS/400, OS/2, etc. La versión para entornos Unix recibe el nombre de TX Series.

CMMI. Integración de Modelos de Madurez de Capacidades o Capability Maturity Model Integration es un modelo para la mejora y evaluación de procesos para el desarrollo, mantenimiento y operación de sistemas de software. Las mejores prácticas CMMI se publican en los documentos llamados modelos. En la actualidad hay tres áreas de interés cubiertas por los modelos de CMMI: Desarrollo, Adquisición y Servicios.

COBOL. Acrónimo del lenguaje de programación COmmon Business Oriented Language, Lenguaje común orientado a negocios.

COBOL z/OS V3.1 Dialecto *COBOL* usado en sistemas operativos de 64 bits *IBM*®.

COPY COPYBOOK. Un Copybook es una sección de código escrito en un lenguaje de alto nivel o lenguaje ensamblador que puede ser tomado (desde una pieza maestra - archivo) y ser insertado en diferentes programas (o múltiples lugares en un solo programa). Se usa para definir formatos de datos físicos, piezas de código de tipo procedimientos y prototipos. El término "copybook" pudo haber sido originado del uso de *COBOL* sobre sistemas operativos de

mainframes *IBM®*: *COPY* es su palabra reservada en *COBOL* y el artefacto fue almacenado como un “book” o libro dentro de una librería más grande de código fuente.

CORBA. En computación, CORBA (Common Object Request Broker Architecture, arquitectura común de intermediarios en peticiones a objetos); es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos.

COTS. En los Estados Unidos de Norteamérica Commercially available Off-The-Shelf (COTS), disponible comercialmente en el mercado es un término en la regulación federal de adquisiciones (Federal Acquisition Regulation, FAR) que define a un producto o artículo no producible (nondevelopmental ítem, NDI) de suministro que es comercial y vendido en cantidades importantes en el mercado y que no puede ser procurado o usado bajo contrato gubernamental en la misma forma en que está disponible al público en general.

CPU. La unidad central de procesamiento o CPU (por el acrónimo en inglés de central processing unit), o simplemente el procesador o microprocesador, es el componente de una computadora y otros dispositivos programables, que interpreta las instrucciones contenidas en los programas y procesa los datos.

CRUD. En computación CRUD es el acrónimo de Crear, Obtener, Actualizar y Borrar (Create, Read, Update y Delete en inglés). Es usado para referirse a las funciones básicas en bases de datos o la capa de persistencia en un sistema de software.

Cubos OLAP. Un cubo OLAP, OnLine Analytical Processing o Procesamiento Analítico En Línea, término acuñado por Edgar Frank Codd de EF Codd & Associates, encargado por Arbor Software (en la actualidad Hyperion Solutions), es una base de datos multidimensional, en la cual el almacenamiento físico de los datos se realiza en un vector multidimensional. Los cubos OLAP se pueden considerar como una ampliación de las dos dimensiones de una hoja de cálculo.

DOM. El Document Object Model o DOM (Modelo de Objetos del Documento es esencialmente una interfaz de programación de aplicaciones (API) que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos y una interfaz estándar para acceder a ellos y manipularlos.

EDS. Electronic Data Systems (EDS) (NYSE: EDS, LSE: EDC) es una empresa estadounidense de consultoría de tecnologías de la información que definió al negocio "outsourcing" o manufactura externa, cuando se fundó en 1962 por Ross Perot. Con sede central en Plano (Texas), la compañía fue adquirida por General Motors en 1984, mientras que en 1996 se convirtió nuevamente en una empresa independiente. EDS es una de las mayores empresas de servicios, según la lista Fortune 500. Actualmente es subsidiaria de Hewlett-Packard.

EJB. Los Enterprise JavaBeans (también conocidos por sus siglas EJB) son una de las API que forman parte del estándar de construcción de aplicaciones empresariales J2EE (ahora JEE 5.0) de Oracle Corporation (inicialmente desarrollado por Sun Microsystems). Su especificación detalla cómo los servidores de aplicaciones proveen objetos desde el lado del servidor que son, precisamente los EJB, los cuales manejan entre otras cosas el mapeo con la base de datos, la persistencia, la concurrencia, las transacciones, la seguridad, la comunicación remota, eventos

controlados por mensajes, servicios y nombres de directorio y ubicación de componentes en un servidor de aplicaciones.

ENIAC. Es una sigla que significa Electronic Numerical Integrator And Computer (Computadora e Integrador Numérico Electrónico), utilizado por el Laboratorio de Investigación Balística del Ejército de los Estados Unidos de Norteamérica.

ERP. Los sistemas de planificación de recursos empresariales, o ERP (por sus siglas en inglés, Enterprise Resource Planning) son sistemas de información gerenciales que integran y manejan muchos de los negocios asociados con las operaciones de producción y de los aspectos de distribución de una compañía comprometida en la producción de bienes o servicios.

ESA. ESA/390 (Enterprise Systems Architecture/390) fue introducida en septiembre de 1990 y es el último mainframe *IBM®* con 31 bits de direccionabilidad y 32 bits para datos en diseño para computadora tipo mainframe.

Ext JS. Ext JS es una biblioteca de JavaScript para el desarrollo de aplicaciones Web interactivas usando tecnologías como AJAX, DHTML y DOM. Fue desarrollada por Sencha.

FiOS. Verizon FiOS (Fiber Optic Services), es un servicio de comunicaciones integradas (Internet, telefonía y televisión) que opera en una red de fibra óptica, ofrecido en algunas áreas de Estados Unidos de Norteamérica por la compañía Verizon.

Hibernate. Hibernate Framework es un conjunto de APIs para soporte de Mapeo objeto-relacional (ORM) en la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones.

HTML. Siglas de HyperText Markup Language (Lenguaje de Etiquetación de Hipertexto), es el lenguaje de marcas predominante para la elaboración de páginas Web. Es usado para describir la estructura y el contenido en forma de texto, así como para complementar el texto con objetos tales como imágenes.

IBM®. International Business Machines, empresa multinacional norteamericana de tecnología y consultoría con cuartel general en Armonk, Nueva York.

IEEE. Siglas de Institute of Electrical and Electronics Engineers, en español Instituto de Ingenieros Eléctricos y Electrónicos, una asociación técnico-profesional mundial dedicada a la estandarización, entre otras cosas.

IMPACT. Software de administración de liberaciones, de versiones, de control de cambios, integración y soporte de pruebas patrocinado por NSF (Fundación Nacional de Ciencias EEUU) ACM y el IEE (UK).

ISO 9000. Designa un conjunto de normas sobre calidad y gestión continua de calidad, establecidas por la Organización Internacional de Normalización (ISO).

Java. Java es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de apuntadores o memoria.

JB. Los JavaBeans son un modelo de componentes creado por Sun Microsystems para la construcción de aplicaciones en Java. Se usan para encapsular varios objetos en un único objeto (la vaina o Bean en inglés), para hacer uso de un solo objeto en lugar de varios más simples.

JCL. JCL, acrónimo de Job Control Language, se traduce al español como Lenguaje de Control de Trabajos. Es un conjunto de especificaciones de morfología y sintaxis requeridas para la redacción de instrucciones de ejecución de programas informáticos por parte del sistema operativo de un equipo informático. Principalmente usado en la plataforma mainframe de *IBM*®., en analogía con otras plataformas, es un script del tipo .BAT de Windows o un C/Bourne Shell script de Unix.

JDK. Java Development Kit o (JDK), es un software que provee herramientas de desarrollo para la creación de programas en java. Puede instalarse en una computadora local o en una unidad de red.

JNI. Java Native Interface (JNI) es un framework (conjunto de APIs de soporte) de programación que permite que un programa escrito en Java ejecutado en la máquina virtual java (JVM) pueda interactuar con programas escritos en otros lenguajes como C, C++ y ensamblador.

JRE. El ambiente de ejecución de Java (Java Runtime Environment), es un conjunto de utilidades que permite la ejecución de programas Java. En su forma más simple, el entorno en tiempo de ejecución de Java está conformado por una Máquina Virtual de Java (Java Virtual Machine JVM), programa ejecutable en el sistema operativo huésped, un conjunto de bibliotecas Java y otros componentes necesarios para que una aplicación escrita en lenguaje Java pueda ser ejecutada. El JRE actúa como un "intermediario" entre el sistema operativo (el microprocesador) y Java.

JSP. JavaServer Pages (JSP) es una tecnología Java que permite generar contenido dinámico para Web, en forma de documentos HTML, XML o de otro tipo.

MDA. La arquitectura dirigida por modelos (Model-Driven Architecture o MDA) es un acercamiento al diseño de software, propuesto y patrocinado por el Object Management Group. MDA se ha concebido para dar soporte a la ingeniería dirigida a modelos de los sistemas de software. MDA es una arquitectura que proporciona un conjunto de guías para estructurar especificaciones expresadas como modelos.

MDD. Desarrollo orientado a modelos (Model Driven Development) es un paradigma de desarrollo de software que se centra en la creación y explotación de modelos de dominio (es decir, representaciones abstractas de los conocimientos y actividades que rigen un dominio de aplicación particular), más que en conceptos informáticos (o algoritmos). Este paradigma se concibió cuando el Object Management Group (OMG) desarrolló la arquitectura de diseño orientado a modelos Model-driven.Arquitecture.

MIPS. MIPS es el acrónimo de "millones de instrucciones por segundo". Es una forma de medir la potencia de los procesadores. Sin embargo, esta medida sólo es útil para comparar procesadores con el mismo juego de instrucciones y usando *benchmarks* (programas, utilerías de prueba) que fueron compilados por el mismo compilador y con el mismo nivel de optimización.

Modelo de Cascada. En Ingeniería de software el desarrollo en cascada, también llamado modelo en cascada, es el enfoque metodológico que ordena rigurosamente las etapas del ciclo de vida del software, de tal forma que el inicio de cada etapa debe esperar a la finalización de la inmediatamente anterior. El primer uso de fases similares se planteó por parte de Herbert D. Benington en 1956. La primera descripción formal se atribuye a Winston W. Royce en 1970.

MQ Series. *IBM®* WebSphere MQ es una familia de productos de software para redes lanzado por *IBM*® en Marzo de 1992. Fue previamente conocido como MQSeries, una marca registrada que *IBM*® adaptó en 2002 para agregar los productos WebSphere. WebSphere MQ, que es conocido como "MQ" por sus usuarios es la oferta de *IBM*® para conectividad de aplicaciones de plataforma heterogénea orientada a mensaje (Message Oriented Middleware).

MySQL Server. MySQL es un sistema de gestión de base de datos relacional, multihilo y multiusuario. MySQL AB desde enero de 2008 una subsidiaria de Sun Microsystems y ésta a su vez de Oracle Corporation desde abril de 2009 desarrolló MySQL como software libre en un esquema de licenciamiento dual.

OMG. El Object Management Group u OMG (de sus siglas en inglés Grupo de Gestión de Objetos) es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA. Es una organización sin ánimo de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas.

OS/390. OS/390 es un sistema operativo de *IBM*® para los mainframes *IBM*® System/370 y System/390. Es básicamente una versión renombrada de MVS (Multiple Virtual Storage) que añade los servicios de sistema UNIX.

Perot Systems. Actualmente parte de Dell Services, Perot Systems era un proveedor de servicios de tecnología de la información con sede en Plano, Texas, USA.

PIA 2000. El Problema Informático del Año 2000 (PIA 2000), se originó porque al economizar memoria, los programadores utilizaron en sus rutinas un formato para identificar los años de sólo dos cifras y no cuatro; por ejemplo: a 1998 le atribuyeron sólo "98" o a 1999 le atribuyeron sólo "99".

Platinum. Platinum Technology Inc. fue fundada por Andrew Filipowski en 1987 para comercializar y dar soporte en instalaciones a administración de productos de software de base de datos. En Mayo de 1999, Platinum fue adquirido por Computer Associates (CA).

RACF. Resource Access Control Facility de forma acortada RACF es un producto de *IBM*®. Se trata de un sistema de seguridad que proporciona control para z/OS y z/VM.

RUP. El Proceso Racional Unificado (Rational Unified Process en inglés RUP) es un proceso de desarrollo de software, junto con UML constituye una de las metodologías estándar más utilizadas para el análisis, implementación y documentación de sistemas orientados a objetos.

SAGE. El Semi-Automatic Ground Environment fue un sistema de control automatizado para rastrear e interceptar bombarderos aéreos enemigos usado por el NORAD North American Aerospace Defense Command.

Sistema Heredado. Un sistema heredado (o legacy system) es un sistema informático (equipos informáticos o aplicaciones) que ha quedado anticuado pero continúa siendo utilizado por el usuario (típicamente una organización o empresa) y no se quiere o no se puede reemplazar o actualizar de forma sencilla.

SOS. System Of Systems, Sistema de Sistemas es una colección de sistemas dedicados u orientados a tareas que agrupan sus recursos y capacidades para obtener un nuevo "meta-sistema" más complejo que ofrece más funcionalidades y rendimiento que la simple suma de los sistemas que lo constituyen.

Spring. El Spring Framework (también conocido simplemente como Spring) es un conjunto de APIs de código abierto de soporte de desarrollo de aplicaciones Web empresariales en capas para Java y también para .NET.

SQL. El lenguaje de consulta estructurado o SQL (por sus siglas en inglés Structured Query Language) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en éstas.

SSEC. Acrónimo de Selective Sequence Electronic Calculator computadora electromecánica construida por *IBM*® terminada en enero de 1948.

Struts2. Struts Framework es un conjunto de APIs de soporte para el desarrollo de aplicaciones Web bajo el patrón Modelo Vista Controlador bajo la plataforma Java EE (Java Enterprise Edition).

SW-CMM. El Modelo de Madurez de la Capacidad para el desarrollo de Software (Capability Maturity Model for Software, SW-CMM) es un modelo de procesos para el desarrollo y mantenimiento de sistemas de software, diseñado sobre los criterios de que la calidad de un producto es consecuencia directa de los procesos que se siguen para obtenerlo y las organizaciones que desarrollan software tienen un atributo llamado madurez, que es la medida de la capacidad e institucionalización de sus procesos de manufactura.

Tata Consultancy. Tata Consultancy Services Limited (TCS) es una compañía India de servicios de TI, soluciones de negocio y outsourcing (manufactura externa) que tiene su casa matriz en Mumbai, India.

UML. Lenguaje Unificado de Modelado (LUM o UML, por sus siglas en inglés, Unified Modeling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema.

UNIX Unix (registrado oficialmente como UNIX®) es un sistema operativo portable, multitarea y multiusuario; desarrollado, en principio, en 1969 por un grupo de empleados de los laboratorios Bell de AT&T, entre los que figuran Ken Thompson, Dennis Ritchie y Douglas McIlroy.

Visual Basic. Visual Basic es un lenguaje de programación orientado a eventos, desarrollado por el alemán Alan Cooper para Microsoft. Este lenguaje de programación es un dialecto de BASIC, con importantes agregados.

VS COBOL II. Dialecto *COBOL* usado en sistemas operativos de 31 bits *IBM®.*

VSAM. Virtual Storage Access Method (VSAM) es un esquema de almacenamiento de *IBM®* del sistema operativo OS/VS2, utilizado también en la arquitectura MVS y ahora en z/OS. Es un sistema de archivos orientado a registros que pueden estar organizados de cuatro maneras diferentes: Key Sequenced Data Set (KSDS), Relative Record Data Set (RRDS), Entry Sequenced Data Set (ESDS) y Linear Data Set (LDS). Mientras los tipos KSDS, RRDS y ESDS contienen registros, el tipo LDS (añadido después a VSAM) contiene una secuencia de bytes sin ningún orden de organización intrínseco. Un "Data Set" en la plataforma mainframe es coloquialmente conocido como lo que es: un archivo almacenado en disco o en cinta.

Whirl Wind. Computadora desarrollada en el MIT Massachusetts Institute of Technology en 1947. Es la primera computadora que operó en tiempo real, usó salida de video y la primera que no fue creada solo como reemplazo de un dispositivo electromecánico previo.

XML. Siglas en inglés de eXtensible Markup Language (Lenguaje de Etiquetación extensible), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C) su principal propósito es servir de estructura de contención --las etiquetas-- para transmisión de datos.

Z/OS. Z/OS es el sistema operativo actual de los mainframes de *IBM®.* Del sistema MVT (de 1967) se pasó al MVS en 1974 añadiéndole múltiples espacios de memoria virtual, agregándole a éste compatibilidad UNIX se pasó al OS/390 en 1995 y ampliando a éste el direccionamiento de 64 bits se pasó a z/OS en el año 2000.

Z10. El sistema Z10 es una línea de mainframes de *IBM®.* El z10 Clase Empresarial (Enterprise Class EC) fue anunciado el 26 de Febrero de 2008. El 21 de Octubre de 2008, *IBM®* anunció el z10 Clase de Negocios (Business Class BC), una escala reducida del z10 EC. El sistema z10 representa la primer familia de modelos soportada por la máquina de procesamiento z10 quad core y el primero en implementar la Architectura/z 2 (ARCHLVL 3 - "Zero down-time" en 64 bits "Architecture Level Sets" 3).

INTRODUCCIÓN.

La conceptualización y desarrollo de proyectos informáticos desde cero o innovadores tiene ocurrencia mínima en nuestro país por diversas causas. Cuando se habla del desarrollo de software en México, el tema se limita la gran mayoría de las veces al uso o manufactura de tecnologías y productos o aplicativos conceptualizados en otros lugares. Sobre los mismos se basan soluciones tipo extensión o de mantenimiento para resolver problemas locales, esto es lo que comúnmente llamamos nuevos desarrollos. Asumimos pues, que el entorno tecnológico actual en México en materia de proyectos informáticos tiene marcada tendencia hacia el mantenimiento y escalamiento de aplicativos/productos ya existentes. Esta naturaleza "seguidora" en los proyectos informáticos desarrollados en nuestro país es fuente de problemas con ciertas particularidades.

El problema sobre el cual se centra la presente investigación fue identificado dentro del entorno descrito. Cuando un nuevo proyecto tiene como principal fundamento o punto inicial la tarea "obvia" de entender y adaptarse a lo que otros hicieron, esto puede ser un arma de dos filos. Hacer que este hecho sea una ventaja o desventaja es el grado de capacidad para dominar en el aspecto *cognitivo* las características de lo ya existente, para poder hacer uso de las mismas en la solución de nuevos problemas.

Desarrollar sobre *.NET*, *J2EE*, *Frameworks* de código abierto, *SAP*, *ORACLE*, etcétera es el ejemplo más benéfico de la situación descrita y en este caso está probado que las ventajas superan con mucho las aparentes desventajas. La filosofía de no reinventar la rueda en cada desarrollo nuevo ayudándose de bases ya existentes no tiene ninguna desventaja real. Se podría alegar como desventaja que se deben aprender especificaciones gigantescas para poder desarrollar usando estas tecnologías como base (los desarrollos que no usan las mejores características de los productos existentes pueden terminar en desastre). Sin embargo dominar una especificación de manera razonable es algo perfectamente manejable y totalmente preferible que tener que codificar algoritmos ya implementados por alguien más desde cero. La clave de este tipo de productos es que normalmente tienen una documentación fidedigna además de exacta y que las funciones implementadas son muy enfocadas a aspectos de carácter técnico puntual, también se puede decir que las funciones implementadas no son ambiguas y que su implementación está estandarizada dentro del producto.

Existen sin embargo otro tipo de aplicativos de software que implementan funcionalidades más enfocadas a negocio, los cuales solucionan problemas operativos específicos de las organizaciones. Estos aplicativos muchas veces se basan en el tipo de productos mencionados en el párrafo anterior, pero también existen otros en esta categoría que se desarrollan sin utilizar soluciones del tipo ambiente de desarrollo/ejecución pre-construido. El primer problema observado es que estos sistemas informáticos tienen naturaleza evolutiva que no siempre se apega a métodos formales de desarrollo (estándares), de hecho la situación crucial se centra muchas veces en la documentación de lo que el software hace tanto funcional como técnicamente. En el mejor de los casos la documentación existente tiene una aproximación aceptable al describir las características del sistema actual. Que este tipo de productos "a la medida" de una organización

tengan una documentación exacta es casi una utopía y los desarrolladores que implementan nuevas funciones o corrigen errores aquí casi siempre confían más en el análisis del código fuente que en la documentación. El peor de los casos es que no exista la documentación ni funcional ni de diseño en absoluto o qué ni siquiera el código fuente exista, sino únicamente el ejecutable en formato binario para el procesador de la computadora de ejecución.

A este último tipo de aplicativos pertenecen los llamados *sistemas heredados*, que además de las características mencionadas anteriormente, tienen otra más que los distingue: que fueron desarrollados hace mucho tiempo y se siguen usando en la actualidad, por lo que normalmente usan tecnología acusada de no reciente, arcaica o desactualizada.

Si se debe dar mantenimiento o se debe implementar un nuevo proyecto en un *sistema heredado*, uno de los problemas cruciales a resolver es el análisis funcional (el aspecto *cognitivo* mencionado anteriormente), de éste depende gran parte del éxito de la implementación. Cuando existen dentro de la organización personas que dominan funcionalmente los *aplicativos heredados*, es muy bajo el riesgo de que los nuevos proyectos fracasen si estas personas se involucran en la ejecución. El problema grave aparece cuando las personas expertas no están disponibles por diversas causas y aún así se deben implementar nuevos proyectos. Este tipo de contingencias se da de manera muy frecuente en organizaciones grandes, donde hay demasiados empleados, mucha rotación de personal, donde el número de requerimientos para modificar los *sistemas heredados* rebasa el número de personas experimentadas que pueden atenderlos o aunque suene poco creíble: donde haya mucha pugna política inter-organizacional.

La experiencia profesional del autor de esta investigación ha sido en su gran mayoría como arquitecto de soluciones en la plataforma *mainframe*. En la plataforma *mainframe* y sus tecnologías asociadas se generan continuamente proyectos en nuestro país y los mismos están asociados a la extensión de funcionalidades o bien al mantenimiento de sistemas implementados hace mucho tiempo. La plataforma *mainframe* está presente en organizaciones grandes (empresas de telecomunicaciones, gobierno, bancos, etcétera) con departamentos de informática con tamaño del orden de miles de empleados y la misma se caracteriza por la presencia dominante de *sistemas* informáticos *heredados* en las funciones principales del núcleo organizacional. Es en este contexto donde se identificó el problema tratado en esta investigación: para dar mantenimiento o evolucionar un *sistema heredado* en la plataforma *mainframe* es necesario analizar de manera pronta y expedita la funcionalidad del mismo. Estos sistemas son complejos en extremo y muchas veces los expertos funcionales no están disponibles por diversas causas durante la ejecución de los proyectos. Aunado a todo lo anterior, se observó que en la mayoría de las organizaciones el análisis funcional en esta plataforma se hace de forma manual. El trabajo manual tiene muchos riesgos y estos se materializan en errores de manera más frecuente de lo que se piensa. Los errores de análisis materializados tienen normalmente costo elevado en diversos rubros: calidad de vida de los empleados, dinero, tiempo, desprestigio de la organización, afectación a clientes, etcétera. *La problemática* identificada es, pues: El riesgo de aparición de errores y afectaciones asociados al análisis manual de sistemas informáticos en las plataformas *mainframe*.

La investigación aplicativa que se desarrolla a continuación tiene como principal *objetivo* generar un procedimiento de análisis funcional para *aplicativos heredados* en la plataforma *mainframe*, las características con las que se planeó dotar al procedimiento propuesto es que debe ser: ágil, fácil de seguir, con una parte importante basada en la automatización, que incluya la presentación de conocimiento en formato amigable, que tenga funciones para introducir conocimiento

generado manualmente y que esté basado en tecnologías de código abierto. Cubierto este objetivo se anticipa que varios problemas que tienen su origen en el análisis manual de aplicativos se verían disminuidos utilizando los resultados del presente estudio.

El proyecto está *justificado* porque se ha observado que las organizaciones que tienen entre sus activos *sistemas heredados* en la plataforma *mainframe* tienen problemas serios asociados a errores de análisis durante la implementación de proyectos. No se observa en esta industria en México el uso popularizado de herramientas automáticas de análisis tal vez por el alto costo al ser productos con licencia o tal vez por lo complejo de su interpretación. Ante este panorama, la presente propuesta incluye el uso de tecnologías de código abierto, simplificación de resultados, confiabilidad y reducción de tiempos en las tareas de análisis funcional, lo cual se traduce en reducción de costos y disminución en la incidencia de errores durante la implementación de proyectos, que son atribuidos al trabajo manual realizado bajo diferentes situaciones de presión y volumen. El resultado entregado no es un tema cerrado, el mismo puede ser extendido o incluido en otro tipo de investigaciones como herramientas de análisis cruzado, reingeniería o repositorios de conocimiento funcional. Debido a lo anterior, el desarrollo del proyecto se anticipa como una aportación interesante.

El problema identificado y el objetivo fijado para contribuir a su solución tienen un ámbito muy amplio. Se propuso un diseño del tipo especificación de alto nivel, el cuál define el *alcance* de la investigación al perfilar los componentes y módulos que contribuyen a la solución del problema. Entre los principales entregables se propusieron: la documentación de las buenas prácticas que se realizan actualmente en la plataforma *mainframe* de forma manual en las organizaciones, la documentación del estado del arte en materia de análisis de software, la implementación de un analizador sintáctico como base de una herramienta automática extractora de conocimiento, la implementación de varios algoritmos de extracción del flujo imperativo de programas, la propuesta de varios algoritmos para extraer reglas funcionales del código fuente, la presentación de un resumen de análisis (seudo-especificación) generado automáticamente, la posibilidad de modificar manualmente la seudo-especificación generada, las pruebas unitarias ejecutadas con el entregable y una evaluación e interpretación de los resultados obtenidos.

La implementación del diseño se cerró con varias partes que no fueron terminadas lo cual se debió principalmente a *limitantes* de calendario y algunas otras causas. Sin embargo se entregó un diseño de alto nivel que es viable de construir en un periodo extendido de tiempo y además la primera iteración implementada contribuye de manera importante a la solución del problema identificado.

Entre los *resultados esperados*, se anticipa una reducción de riesgo y tiempo en las actividades de análisis funcional de aplicativos, como consecuencia directa se espera una reducción de costos asociados al tiempo a invertir y de manera indirecta y no cuantificable se espera que el uso del procedimiento propuesto disminuya la incidencia de errores que se presentan cuando personas no expertas en la funcionalidad de un aplicativo implementan proyectos evolutivos/correctivos en los *sistemas heredados* de las organizaciones que cuentan con plataformas *mainframe*.

La *manera* de resolver el problema identificado, fue la de asumir inicialmente que la problemática que tiene la forma actual de hacer el análisis de *aplicativos heredados* puede ser mejorada por medio del uso de los resultados de una investigación aplicativa. Esta investigación debió implementar diferentes teorías informáticas enfocadas a automatizar el análisis funcional. El procedimiento usado tuvo varios pasos: Primero se documentó la manera actual de hacer las

cosas, con base en esta revisión se llegó a una identificación de los puntos que pueden mejorarse por medio del uso de la tecnología y teorías ya desarrolladas. Esto da el perfil de la investigación aplicativa. Se realizó la investigación teórica del estado del arte en las áreas de conocimiento involucradas en la solución, esta colección de investigaciones, muchas veces enfocadas hacia otro tipo de problemas o entornos tecnológicos, se adaptó al entorno de estudio y en base al análisis de las mismas se propuso un diseño enfocado a la plataforma *mainframe*. Se detalló el procedimiento de solución propuesto, el cual incluyó la generación de un entregable de software. Se implemento el diseño en varias de sus partes más importantes y una vez que el software quedó terminado, se documentó finalmente la evaluación del producto, que fue sometido a varias pruebas unitarias.

El desarrollo de la investigación se ha conducido en *4 capítulos***.** *En el primero* se proporciona un contexto del problema detectado que se quiere solucionar: los riesgos asociados al análisis funcional de aplicativos en la plataforma *mainframe*, que con mucha frecuencia se hace de forma manual. Se define y detalla el problema en este apartado, se delimita el alcance inicial de la solución propuesta y se justifica el enfoque de esfuerzos para contribuir con este resultado.

En el segundo capítulo se presenta el estado del arte en materia de teorías que pueden ser aplicadas a la solución del problema, se secciona el problema en las diferentes teorías de conocimiento que requieren ser revisadas para extraer de ellas una propuesta aplicativa y se señala en diferentes puntos del capítulo qué teorías podrían ser aplicadas a la solución. Primero se documenta un resumen de las consideraciones necesarias a tener en cuenta durante el análisis funcional de aplicativos en *mainframe*. Después se documentan las principales teorías e investigaciones actuales involucradas en la solución del problema, estas son: el análisis automático de flujo de datos y el de control de flujo imperativo de los programas, la extracción automática de reglas de negocio desde el código fuente y la expresión de conocimiento en formatos simbólicos o gráficos.

El capítulo 3 es la propuesta de solución, se expone aquí el diseño del autor de manera conceptual y descriptiva en un alto nivel, ¿cómo es posible mejorar la manera actual de hacer las cosas en el entorno descrito para el problema identificado? Se enuncia y detalla el procedimiento que es el título de la tesis y se perfila el entregable principal: un producto de software que pretende mejorar el estado actual de hacer las cosas, este producto tiene un diseño inicial que se basa en teorías de ingeniería de software, de compiladores y de tecnologías de código abierto. Se incluye además la propuesta opcional de inclusión de algoritmos básicos de extracción de conocimiento a partir de los datos del compilador y de igual manera se detalla el prototipo inicial.

El capítulo 4 por último, es la evaluación de la propuesta. El producto desarrollado es sintetizado en cuanto a alcance final, se describen sus principales características funcionales, la manera de utilizarlo y las pruebas ejecutadas sobre el mismo para emitir una evaluación inicial. Se documentan en este capítulo también las lecciones aprendidas en materia de administración del proyecto informático asociado a la implementación del diseño.

CAPÍTULO I. PANORAMA DEL ENTORNO Y DEFINICIÓN DEL PROBLEMA.

En este capítulo se expone un contexto del problema detectado que se quiere solucionar: los riesgos asociados al análisis funcional de aplicativos en la plataforma mainframe, que con mucha frecuencia se hace de forma manual. Se define y detalla el problema en este apartado, se delimita el alcance inicial de la solución propuesta y se justifica el enfoque de esfuerzos para contribuir con este resultado.

1.1 LA EVOLUCIÓN DE LA FORMA DE DESARROLLAR SOFTWARE.

El aprovechamiento adecuado de los recursos tecnológicos e informáticos en una organización es un tema de vital importancia, hoy en día muchas organizaciones no dudan en clasificar el tema como de alta prioridad, pues de esto depende gran parte de su éxito o fracaso en el mundo competitivo actual.

El presente trabajo se enfoca en un campo relacionado con el desarrollo de software, en particular en un problema detectado durante el mantenimiento de aplicaciones que se han liberado a ambientes de producción, cuando las mismas fueron desarrolladas hace varios años resistiendo el reemplazo y operando como “sistemas núcleo” (porque dan soporte eficiente a las organizaciones en procesos críticos) y debido a esto, con el paso del tiempo no ha sido tema de discusión su reemplazo sino que incluso han crecido y evolucionado.

El desarrollo de software con el paso del tiempo ha tenido ciertas tendencias, es útil tener una idea general de cómo ha ido evolucionando esta industria y cuáles son los problemas a los que se

han enfrentado los desarrolladores durante la implementación de proyectos [1], lo cual es un preámbulo contextual para el problema que se describirá posteriormente.

La programación de las primeras computadoras fue en gran parte integrada a los circuitos o al cableado de la máquina, si se considera a las primeras computadoras de funcionalidad limitada, como la ENIAC (*Electronic Numerical Integrator and Computer*, Departamento de Defensa EUA 1947) o a la SSEC (*Selective Sequence Electronic Calculator*, *IBM®* 1948), para estos equipos todavía no se observaba una clara diferenciación entre el software y el hardware [2].

En la década de los 50 la manera de desarrollar software estuvo fuertemente ligada a la manera de desarrollar hardware. El hardware dictaminaba como se debía desarrollar el software y la manera de desarrollar sistemas de información seguía en la medida de lo posible métodos de ingeniería de construcción de computadoras. Se requería de matemáticos o ingenieros orientados al conocimiento de la operación de la máquina para desarrollar los programas aplicativos. Es por esto que se puede decir que el desarrollo de software tenía un paradigma influenciado por el hardware. Un indicativo de la influencia de los herrajes sobre la lógica de los programas es el nombre de las organizaciones líderes de profesionales del software: *ACM - Association for Computing Machinery* y *IEEE - Institute of Electrical and Electronics Engineers Computer Society*. Un proyecto representante de esta tendencia en el desarrollo de software ocurrió en julio de 1958 cuando el primer búnker de la red *SAGE* (*Semi-Automated Ground Environment*, sistema de defensa norteamericano - canadiense) fue puesto en operación. La computadora era la *AN/FSQ7* (cuyo prototipo era el WhirlWind de 1951) y cada búnker era capaz de administrar 400 aviones simultáneamente. El sistema *SAGE* cumplió con sus especificaciones aproximadamente con un año de atraso, el comentario más destacado relativo al proyecto fue dado por Herbert D. Benington “Es fácil para mí señalar el único factor que nos ha llevado a este éxito relativo: todos somos ingenieros y hemos sido entrenados para encaminar nuestros esfuerzos a través de líneas de ingeniería” [3]. El último búnker de la red *SAGE* cerró en enero de 1984 [4].

En la década de los 60 los dos campos empezaron a diferenciarse, el software era mucho más fácil de modificar que el hardware y no requería de costosas líneas de producción para generar el producto final, esto llevó a una filosofía de “codifica y repara” en el desarrollo de programas, en oposición a las exhaustivas revisiones que los ingenieros de hardware hacían antes de comprometer líneas de producción y doblar metales (mide dos veces y corta una sola vez). El desarrollo de aplicaciones se enfocó en las personas más que al hardware. La demanda de software rebasó la capacidad del mercado educativo para ofertar ingenieros y matemáticos y es entonces cuando se inició la contratación y entrenamiento de personas egresadas de carreras “no-relacionadas”, tales como humanistas, ciencias sociales, finanzas y otras áreas para desarrollar software, lo mismo ocurrió con otras posiciones relacionadas al proceso y manejo de la información, con personas de estos nuevos perfiles profesionales se incrementó la tendencia “codifica y repara después”.

Un claro ejemplo de la tendencia de permitir la entrada de no-matemáticos y no-ingenieros al desarrollo de software fue la creación del lenguaje *COBOL*, resultado del alto nivel de aislamiento entre el programa y la máquina. En esta época muchos proyectos trataron de enfocarse únicamente en el propósito o misión sin preocuparse mucho por cómo lograrlo, lo cual llevó a muchos fracasos, retrasos y trabajo desperdiciado ante constantes correcciones [5]. Esto se puede concebir como la era artesanal del desarrollo de software.

En la década de los 70 el desarrollo de software se enfocó a una respuesta directa al bosquejo de procesos o soluciones basados en "codifica y repara después". Se crearon iniciativas para intentar crear código más organizado y precedido de un diseño: este movimiento tuvo dos ramas principales: los métodos formales enfocados a la exactitud de los programas (ya sea por prueba matemática o por construcción vía "programación basada en cálculo" [6]) y la mezcla menos formal de técnicas y métodos de administración, programación estructurada *top-down* (desarrollo descendente), con equipos de programadores liderados por un jefe. Esta etapa se caracterizó por el éxito de la programación estructurada, la fortaleza del concepto de modularidad y ocultamiento de información e inclusive del diseño estructurado. Una fuerte síntesis del paradigma basado en hardware y en el de elaboración manual o artesanal fue reflejada en la versión del modelo de cascada de Royce [7], quién agregó el concepto de iteración entre 2 fases sucesivas del desarrollo y una actividad de elaboración de prototipos ("constrúyase 2 veces") antes de comprometer un desarrollo de gran escala. Una versión subsecuente enfatizaba en la validación de artefactos en cada fase antes de iniciar la siguiente, esto con la intención de encontrar defectos y solucionarlos dentro de la misma fase en la medida de lo posible. Esto se hizo con la finalidad de ahorrar los costos relativos de encontrar defectos temprano versus tarde. Desafortunadamente el modelo de cascada fue interpretado como un proceso secuencial donde el diseño no iniciaba hasta que hubiese un conjunto completo de requerimientos y la codificación no iniciaba hasta tener una revisión exhaustiva del diseño, lo cual resultó en proyectos administrativamente inmanejables. Otra tendencia importante de la década de los 70 fue la aproximación cuantitativa (medible) de la ingeniería de software, el análisis del factor persona, muchas empresas se empezaron a dar cuenta que el costo del software empezaba a rebasar el costo del hardware.

Junto con las buenas prácticas desarrolladas en la década de los 70, la década de los 80 llevó a un buen número de iniciativas para solucionar los problemas de la década anterior y para mejorar la productividad de la ingeniería de software y la escalabilidad. Cuando los métodos cuantitativos pusieron en evidencia aspectos de productividad se pudieron identificar muchos puntos de mejora. Por ejemplo que alrededor del 70% del trabajo en la etapa de pruebas era en realidad trabajo de corrección de errores. La identificación de aspectos de este tipo llevó a reducciones de costo. En esta década se desarrollaron modelos de desarrollo de software, así como varios estándares como por ejemplo *SW-CMM* el *Software Capability Maturity Model* que evalúa la madurez de los procesos de desarrollo de software en una organización [8] o bien estándares semejantes aplicados en Europa, como el estándar de calidad *ISO 9000*. Muchos desarrolladores de software buscaron cumplir con estas normas para evitar quedar fuera de las ofertas del mercado, la mayoría reportó buen retorno de inversión al tener como resultado reducción de los re-procesos, esto hizo que la tendencia se tomara al nivel interno de las organizaciones. Esta década también se vio incrementado el desarrollo de herramientas, por ejemplo para la ejecución de pruebas o para la configuración de la administración (proyecto *IMPACT* [9]) o herramientas de soporte para el desarrollo, como Ingeniería de software con ayuda de computadora (*Computer-Aided Software Engineering CASE*). Otro avance observado en el desarrollo de aplicaciones durante la década de los 80 se reflejó en los sistemas expertos, lenguajes de alto nivel, orientación a objetos, máquinas más potentes y la programación visual, también se dio mucha importancia al concepto de reuso o bien lograr un ambiente de colaboración y evitar trabajar en problemas ya resueltos.

Durante la década de los 90's se acrecentó la fortaleza de los métodos orientados a objetos lo cual se vio reforzado por la mejora continua en patrones de diseño [10], arquitecturas de software y lenguajes de descripción de arquitecturas. El lenguaje *UML* fue creado en esta época. También se debe destacar el crecimiento continuo de la Internet. Debido a la urgencia con la que ciertos sistemas eran demandados en el mercado hubo un cambio del modelo de cascada hacia una

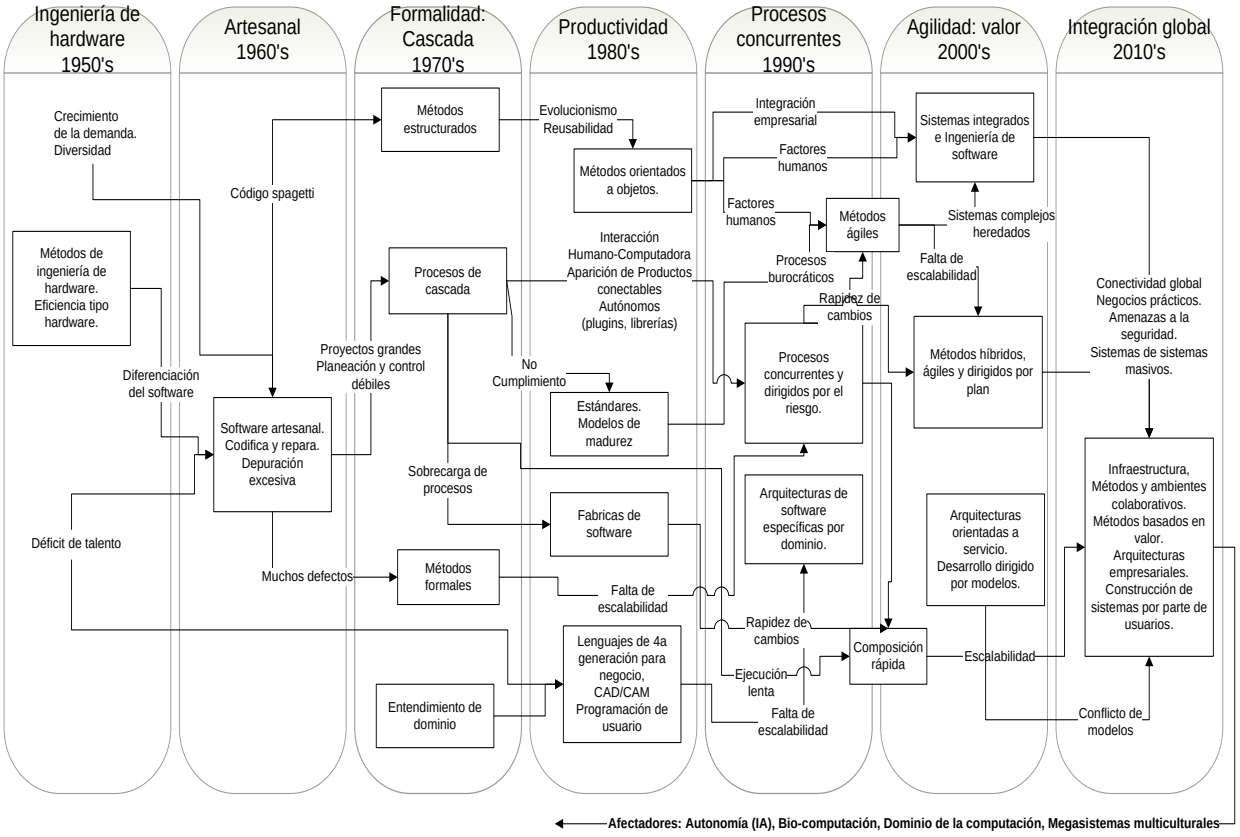
orientación de ingeniería de requerimientos concurrentes, diseño y codificación de productos, procesos, software y sistemas. Otro factor detonante de esta tendencia fue el cambio de productos interactivos de usuario, en vez de requerimientos pre-especificados. Un ejemplo de ataque a este tipo de requerimientos fue la creación del *Rational Unified Process* de *IBM®* [11], otro avance significativo fue el inicio del desarrollo de código abierto. En general hubo una tendencia a incrementar la utilización de productos de software por no programadores y así mismo investigaciones serias en la interacción humano-computadora.

En la primera década del siglo XXI se observó una continuación a la tendencia del desarrollo rápido de aplicaciones: se observa un cambio continuo en el aspecto tecnológico (Google, aplicaciones Web de soporte colaborativo), en organizaciones (fusiones, adquisiciones, creaciones), en disposiciones oficiales (nueva leyes, seguridad nacional) y en el medio ambiente (globalización, patrones de demanda de consumo). Esto causa frustración cuando se trata de cumplir con los patrones inerciales actuales de madurez en el desarrollo, con planes rígidos o inamovibles, especificaciones y demás tipo de documentación requerida. Por ejemplo: existen registros de aprobación de certificaciones de *CMM* en volúmenes gigantescos solo para un nivel. Es evidente la necesidad de diseño de métodos más ágiles. Como el software ha pasado a ser un activo muy crítico en las organizaciones, la tendencia actual es la de diseños orientados al servicio, el uso de componentes prefabricados de venta en el mercado (*COTS Commercial off-the- Shelf*), métodos ágiles y confiables de desarrollo, la tendencia a utilizar cada vez menos software con licencia y métodos como el desarrollo y arquitecturas dirigidos por modelos o arquitecturas empresariales de software (*MDD*, *MDA*) [12].

Para la década de los 2010 se mencionan ciertas tendencias, debido al abaratamiento de la Internet de banda ancha, tecnologías como *FiOS®* (*Fiber Optic Service*) que pretenden llevar redes de fibra óptica al público en general, se habla de servicios y proveedores independientes a la ubicación física, por ejemplo el abaratamiento de costos por desarrollo off-shore (contratistas y proveedores globales ubicados en varios lugares alrededor del mundo donde la diferencia de salarios es un arma competitiva), surgen con esto nuevos retos como el de acoplar equipos de personas en desarrollo de software fragmentados alrededor del mundo, equipos humanos multiculturales, retos de administración, semántica de comunicación, visibilidad y control. La habilidad de las organizaciones para integrar sistemas independientes relacionados en "sistemas de sistemas" *SOS* (Systems of Systems) será vital para su competitividad [13].

Para la década del 2020 si la ley de Moore se mantiene, estaríamos en un escenario de mejora en rendimiento de 1:10,000 en el hardware, se anticipan nuevos tipos de plataformas como "polvo inteligente –Smart Dust" (sensores micro electrónicos conectados en una red inalámbrica) o nuevos tipos de aplicaciones como redes de sensores, prótesis humanas, materiales adaptables; lo cual generaría nuevos retos a la ingeniería de software, para especificar su configuración y comportamiento. Otro tipo de especulaciones más ambiciosas se enfocan en la autonomía, donde se predicen las plataformas de verdadera inteligencia artificial, administración de conocimiento, máquinas que dados ciertos escenarios sean capaces de tomar decisiones sin la intervención de un ser humano o mejorar de manera autónoma, en este caso se puede clasificar a los generadores automáticos de código o lenguajes naturales interpretados por sistemas informáticos.

En la figura 1 se muestra la tendencia de la ingeniería de software a través del tiempo con una estimación hasta la década de los años 2010.



Rango completo de tendencias en ingeniería de software.

Figura 1 Tendencias en el desarrollo de software.

1.2 SISTEMAS HEREDADOS.

El escenario planteado en el punto 1.1 nos da una idea de una evolución acelerada en la forma de desarrollar software, sin embargo, en la actualidad se observa un fenómeno informático en las grandes empresas, donde existen ambientes con sistemas los cuales no fueron creados recientemente y son además de mediana complejidad o muy complejos. Estamos al inicio de la década de los 2010 y muchas empresas grandes en México y al nivel mundial tienen sistemas estratégicos muy importantes, los cuáles no son de creación reciente. Al nivel administrativo y tecnológico, la tendencia que se perfila como la dominante para el futuro, la de servidores distribuidos o la de código abierto, no está reemplazando del todo a los sistemas centralizados y de software con licencia sino por el contrario: muchas compañías optan por la opción más fácil de administrar un solo equipo con su software estratégico, o por la seguridad que da tener soporte por parte de una empresa global de desarrollo de software. Es una forma de tener control, incrementar la seguridad e invertir menos en personal de planta. Las tecnologías de código abierto, de plataforma Web o de ambientes gráficos dan soporte a empresas que buscan innovación constante, que son de reciente creación, pero sobre todo que manejan volúmenes

medianos o pequeños de información, donde una migración hacia sistema informático novedoso podría ser relativamente manejable o viable. Sin embargo, muchas organizaciones no han migrado a nuevas tecnologías con el paso del tiempo, o lo han hecho, pero solo en una parte de los sistemas, esto se debe a razones de negocio, de costo o de riesgo elevado ante una migración total. En general se dice que estas compañías tienen **sistemas heredados** [14].

¿Qué es un sistema heredado? Un sistema heredado (o "*legacy system*") es un sistema informático (equipos informáticos y/o aplicaciones de software) que usa tecnología no reciente y que continúa siendo utilizado por las una organizaciones o empresas, el cual no puede ser reemplazado o actualizado de forma sencilla, ya sea por voluntad o por necesidad.

Las compañías gastan mucho dinero en sistemas informáticos y, para obtener un beneficio de esa inversión, el software o el hardware debe utilizarse por varios años. El tiempo de vida de los sistemas informáticos es muy variable, pero muchos sistemas grandes se pueden llegar a utilizar hasta por más de 20 años. Estos sistemas antiguos aún son importantes para sus respectivos negocios, es decir, las empresas cuentan con los servicios suministrados por los mismos y cualquier falla en estos servicios tendría un efecto serio en el funcionamiento de la organización. Un ejemplo de gasto en sistemas heredados se tuvo con el problema del año 2000, conocido como efecto 2000, error del milenio, problema informático del año 2000 (*PIA2000*) o *Y2K*, un error de software causado por la costumbre que habían adoptado los programadores de omitir el siglo para el almacenamiento de fechas (generalmente para economizar memoria), asumiendo que el software sólo funcionaría durante los años cuyos valores comenzaran con 19. Lo anterior tendría como consecuencia que después del 31 de diciembre de 1999, sería el 1 de enero de 1900 en vez de 1 de enero de 2000. La corrección del problema costó miles de millones de dólares en el mundo entero, sin contar otros costos relacionados, como los errores que ocurrieron realmente afectando a los usuarios [15].

Lo habitual es que los sistemas heredados, no sean los mismos sistemas que originalmente se empezaron a utilizar. Muchos factores externos e internos, como el estado de las economías nacional e internacional, los mercados cambiantes, los cambios en las leyes, los cambios de administración o la reorganización estructural, conducen a que los negocios experimenten cambios continuos. Estos cambios generan o modifican los requerimientos del sistema de información, por lo que éstos van sufriendo cambios conforme cambian los negocios. Por esta razón, los sistemas heredados incorporan un gran número de actualizaciones hechas a lo largo de su vida útil. Muchas personas diferentes pueden haber estado involucradas en la realización de estas modificaciones a lo largo del tiempo y es inusual para cualquier usuario o administrador del sistema tener un conocimiento completo del mismo, sobre todo cuando éste tiene una cierta importancia o tamaño.

Los negocios por lo general reemplazan sus equipos y maquinaria con sistemas más modernos, pero desechar un sistema informático y reemplazarlo con hardware y software moderno conduce a riesgos de negocio significativos. Reemplazar un sistema heredado es una estrategia arriesgada por varias razones:

- Rara vez existe una especificación completa de los sistemas heredados. Si existe una especificación, no es probable que tenga los detalles de todos los cambios hechos en el sistema. Por lo tanto, no existe ninguna forma directa de especificar un nuevo sistema que sea funcionalmente idéntico al sistema que se utiliza.

- Los procesos de negocios y las formas en que los sistemas heredados operan a menudo están entrelazados de manera compleja. Estos procesos se diseñaron para aprovechar los servicios del software y evitar sus debilidades. Si el sistema se reemplaza, estos procesos también tendrán que cambiar, con costos y consecuencias impredecibles.

- Las reglas de negocio importantes están contenidas en el software y no suelen estar anotadas en ningún documento de la empresa. Una regla de negocio es una restricción que aparece en algunas funciones del negocio y romper esa restricción puede tener consecuencias impredecibles para éste. Por ejemplo, las reglas para valorar el riesgo de la aplicación de una política de una compañía de seguros pueden estar contempladas en su software. Si a estas reglas no se les da mantenimiento, la compañía puede aceptar políticas de riesgo altas que conduzcan a pérdidas operativas (reclamaciones costosas por ejemplo).

Seguir utilizando los sistemas heredados evita los mencionados riesgos del reemplazo pero hacer cambios al sistema existente en vez de cambiarlo por uno más moderno puede ser costoso también puesto que éste es cada vez más viejo. Las razones de este costo de mantenimiento de sistemas que ya tienen una cierta antigüedad son:

- Las diversas partes del sistema pueden haber sido implementadas por diferentes equipos. Por lo tanto, existen estilos de programación no consistentes a lo largo del sistema.

- Parte del sistema, o todo, pudo implementarse utilizando un lenguaje de programación que ahora es obsoleto. Es difícil encontrar personal que tenga conocimiento de estos lenguajes de programación, por lo que se requiere consultoría externa costosa para dar mantenimiento al sistema.

- A menudo, la documentación del sistema no es adecuada y no está actualizada. En algunos casos, la única documentación existente es el código fuente del sistema. En los casos más graves el código fuente puede haberse perdido y sólo está disponible la versión ejecutable del sistema.

- Por lo general, muchos años de mantenimiento dañan la estructura del sistema, haciéndola cada vez más difícil de comprender. Tal vez se agregaron nuevos programas que interactúan con otras partes del sistema de una forma adyacente.

- El sistema se pudo optimizar para la utilización del espacio o para la velocidad de ejecución más que para comprenderlo del todo. Esto provoca dificultades importantes a los programadores que conocen las técnicas modernas de ingeniería de software pero no conocen los trucos de programación utilizados por quienes desarrollaron el sistema original (quienes probablemente desconocían los conceptos de la ingeniería de software actual).

- Los datos procesados por el sistema se conservan en diferentes archivos que pueden tener estructuras o formatos incompatibles. Puede existir duplicación de datos y los datos mismos pueden no estar actualizados, ser imprecisos o estar incompletos.

Una solución a este tipo de casuísticas tan complejas podría ser la implementación de una arquitectura orientada a servicios o SOA (por sus siglas en inglés), dónde las aplicaciones de los sistemas heredados podrían ser publicadas como servicios. El carácter modular de una SOA,

también hace que sea fácil poder adaptarse a los cambios de mercado con la simple creación o publicación de servicios sin tener que depender de una macro-estructura tan compleja. Aun así habría que tener en cuenta los posibles riesgos que entraña y mientras tanto seguir absorbiendo costos elevados y enfrentando también los riesgos asociados.

1.3 CASO APLICATIVO. LA PLATAFORMA MAINFRAME.

Como se ha mencionado, en materia de avance en el desarrollo de software y también de la tecnología en computadoras, se pudiera pensar que sistemas informáticos con 10 años de antigüedad o más, deberían estar en desuso, no tendrían porque existir, si la tecnología es antigua ¿no es acaso lógico reemplazar los sistemas por soluciones modernas?

La respuesta a la pregunta anterior es sencilla: no es un tema de lógica, o de moda o de reducción de costos a corto plazo. Una de las principales razones para justificar la existencia de sistemas heredados es la utilidad de los mismos. Un sistema informático es concebido como un sistema que ayude a la organización a conseguir sus objetivos, sean estos de posición en el mercado, de competitividad, de supervivencia, de dominio, etc. Desde este punto de vista, pasa a ser relativamente secundario qué tipo de tecnología se utilice para construir el sistema, es decir, si el mismo se visualiza como una herramienta que ayuda a la organización en el cumplimiento de sus metas y maneja funciones críticas con eficiencia, ayudando a producir utilidades o retorno de la inversión. Bajo esta perspectiva, inclusive el costo pasa a ser un tema con ciertos niveles de tolerancia. Si las empresas cuentan con presupuesto, el sistema elegido es el que de seguridad, eficiencia y confiabilidad en la administración y explotación de los datos, sobre todo si las aplicaciones son críticas o si los datos son muy sensibles o confidenciales. De igual manera la decisión de tener un sistema resulta de una posición favorable en un análisis costo-beneficio. Las razones anteriores justifican la tecnología de implementación, sin importar su antigüedad.

Los mainframes, coloquialmente conocidos como "*big irons*", son computadoras con 40 años de antigüedad, las cuales se distinguen por las grandes dimensiones, ya que están orientadas al procesamiento de grandes volúmenes de datos, estas computadoras son usadas principalmente por empresas grandes para aplicaciones críticas, procesamiento de grandes volúmenes como censos, estadísticas de la industria y de consumo, *ERP* (*Enterprise Resource Planning*), aplicaciones bancarias, tributarias, compañías telefónicas o de telecomunicaciones y mercado de valores, aerolíneas y tráfico aéreo, en general como de centro principal de procesamiento de grandes empresas con un volumen de facturación elevado. La capacidad de un mainframe se define en términos de los siguientes factores: la velocidad de su *CPU*, su memoria interna, su capacidad de almacenamiento externo, sus resultados en los dispositivos E/S rápidos y considerables, la calidad de su ingeniería interna que tiene como consecuencia una alta fiabilidad y soporte técnico caro pero de alta calidad. Es común que los mainframes soporten miles de usuarios de manera simultánea, los cuales se conectan mediante terminales. Algunas de estas computadoras pueden ejecutar muchos sistemas operativos y por lo tanto, no funcionan como una computadora sola, sino como varias computadoras virtuales. En este papel, un mainframe por sí solo puede remplazar docenas o cientos de pequeñas computadoras personales, reduciendo los costos administrativos y de gestión al tiempo que ofrece una escalabilidad y fiabilidad mucho

mejor. Es importante aclarar que un mainframe no es una supercomputadora [16] sino una arquitectura orientada a la solución de problemas de negocio y por lo tanto, más orientada al procesamiento masivo de datos, soporte de transacciones de negocio, almacenamiento de bases de datos grandes y acceso optimizado a dispositivos de entrada/salida, mientras que lo que comúnmente se conoce como supercomputadora, está orientada al procesamiento en paralelo y optimización de cálculos por medio de la *CPU*, como es requerido en aplicaciones de tipo científico, es por eso que la programación de un mainframe es sencilla en comparación a una supercomputadora. Actualmente, los mainframes de *IBM*® dominan el mercado, junto con Hitachi, Amdahl, Fujitsu, *NCR* y Unisys. Los precios no suelen ser menos de varios cientos de miles de dólares, por ejemplo el más reciente mainframe, el *Z10* [17] de *IBM*®, la empresa dominante en el sector.

En términos de costo monetario la plataforma es cara [18-22], los clientes son cautivos, únicamente *IBM®* (o el fabricante del mainframe) fija los precios del software/hardware de sistema, de base de datos, herramientas de conectividad e interacción con usuarios. El mantenimiento y desarrollo de aplicaciones es caro, las empresas gastan grandes sumas en consultoría cada año.

Hablando en general, ya que esta información es confidencial en la mayoría de las empresas, se pueden enunciar los siguientes datos, relacionados al desarrollo de aplicativos en mainframe, esto de acuerdo a experiencia laboral, consultas con administradores de proyectos y diferentes artículos en la Internet o bolsas de trabajo que publican puestos con la oferta económica para trabajar en estas tecnologías:

- Un proyecto de desarrollo de software catalogado como menor o “micro”, está valuado en unos 30 mil dólares americanos.
- Un proyecto mediano está valuado en unos 50 mil o 70 mil dólares americanos.
- Un proyecto grande cuesta desde 1 millón hasta los 5 millones de dólares americanos o más (Por ejemplo la conversión del año 2000 en los bancos mexicanos o la integración del banco Serfin a Santander, así como el de Banamex que migrará sus aplicaciones en mainframe Unisys a mainframe *IBM®*).

En este escenario se visualizan otros problemas: Actualmente en las universidades ya no se enseñan lenguajes de desarrollo para plataforma mainframe (como por ejemplo *COBOL*, o ensamblador), lo cual sería el menor de los requisitos. De lo que verdaderamente carece este sector del mercado es de profesionales conocedores de todo el ambiente asociado, es decir el uso de herramientas de administración de los datos, de edición y manejo de archivos, conectividad a las bases de datos, coordinadores de transacciones, comunicaciones, etcétera. Se observa otra característica: Para el mainframe en México, el promedio de edad de los profesionales dedicados a dar mantenimiento o crear nuevos aplicativos, es alto, comparado con el promedio de edad de personas que se dedican a explotar otras tecnologías. Si se ingresa a la página de la *OCC* [23], una de las más populares actualmente en materia de búsqueda de empleo en México y se teclea “*COBOL*” aparece una cantidad considerable de vacantes, lo cual es una señal de que la plataforma, tal vez acusada de arcaica, no necesariamente está desapareciendo, inclusive varias empresas de desarrollo mexicanas están incursionando en el mercado norteamericano (off-shore o proveedores de servicios de desarrollo de software en ubicación remota), donde en últimos años también se ha notado un déficit de profesionistas dedicados a esta área.

Hablando de los productos o herramientas de mainframe, los mismos son demasiado robustos y cada uno maneja características propias de configuración y explotación, una persona que desconoce los mismos puede tardar un tiempo considerable en familiarizarse con su manejo y también con los errores más comunes que se cometen cuando se trabaja en esta plataforma.

Para los lenguajes o plataformas de tecnología más reciente, en muchos de los casos, estos se han beneficiado de los avances en materia de metodologías de desarrollo y prácticas recomendadas, existe actualmente todo un marco de desarrollo, de ingeniería y métodos asociados a la creación de software como los que se mencionaron en el punto 1.1. Basadas en experiencias y fracasos de proyectos previos, las organizaciones han notado la importancia de seguir un método y documentarlo, lo cual ahorra muchos dolores de cabeza y sobre todo costo cuando hay que dar mantenimiento, es por eso que actualmente es muy difícil imaginar que un sistema nuevo se desarrolle sin una metodología y un proceso de documentación asociado, inclusive en la plataforma mainframe.

Sin embargo, muchos de los sistemas en mainframe con los que actualmente operan las organizaciones en México, no fueron diseñados ni construidos con estricto apego a una metodología, esta es una característica inherente a sistemas con 10 o más años de antigüedad, donde la ingeniería de software y metodología estaban en etapas de crecimiento e implementación. En muchas ocasiones estos sistemas son cruciales para las organizaciones, es decir, manejan operaciones críticas del negocio y también, como ya se mencionó en las características de los sistemas heredados, muchas veces no están documentados. Existe una dependencia riesgosa hacia el grupo de personas que conoce el funcionamiento de tales sistemas, lo cual han logrado a través de muchos años de trabajar en los aplicativos. De igual forma se observa que en el mainframe existen programas que no siguen los patrones de estructuración sino que están codificados basados en saltos (*GO TO*), lo cual hace complejo su seguimiento o mantenimiento cuando presentan errores o se quiere modificar su funcionamiento.

Existe otra característica observada durante la experiencia profesional en esta plataforma: aún cuando las empresas tratan de implementar metodologías o modelos, como *CMMI* (y tienen departamentos de personal completamente dedicados a implementar y auditar las mejores prácticas de desarrollo) en fechas recientes se observa todavía renuencia o descuido de parte de los administradores y ejecutores de un proyecto: siguen presentes los viejos vicios como: modificar sin tener conocimiento técnico y funcional adecuado, codificar al vuelo, iniciar sin un requerimiento bien definido, construir y reparar después, agregar correcciones a destiempo y con diseño deficiente, no monitorear que la documentación sea generada adecuadamente, etcétera.

En referencia a la documentación de los proyectos es muy común que los documentos se generen como requisito y sean elaborados por personas casi ajenas a la implementación del mismo (personas que no diseñaron la aplicación, por ejemplo recursos emergentes o becarios). El resultado de esto es que se cumple con el requisito que exige la metodología pero muchas veces la misma no es de mucha utilidad para el mantenimiento o evolución futura del sistema, solo se revisa que los documentos sean generados, pero no se audita el contenido de los mismos como parte del cierre del proyecto.

Si se toma como antecedente la situación actual descrita, la definición previa proporcionada en el punto 1.2 de sistemas heredados encaja perfectamente. Es de esperarse que las consultorías con el conocimiento en el área cobren fuertes sumas por dar mantenimiento o desarrollar nuevas funcionalidades a sistemas heredados, la justificación reside en lo que se ha mencionado, el

tiempo a invertir en análisis representa una gran parte de las facturas, pero las compañías lo absorben actualmente, invierten fuertes sumas de dinero en el mantenimiento y renovación de estos sistemas.

Recapitulando, hay tres costos asociados en la plataforma de estudio: el costo de la computadora, el costo del software de sistema y el costo del desarrollo de aplicaciones.

Ante esta perspectiva, numerosos competidores y críticos del mainframe han sugerido la posibilidad de re-construir estos sistemas en otras plataformas más modernas, amigables y baratas, algunas empresas lo han hecho, como el centro de operaciones de NYSE (*New York Stock Exchange*) [24], el cual se cuenta entre los casos de éxito en las migraciones a aplicativos cliente servidor. Hacer esto implica una serie de pasos que van desde el análisis y documentación de la funcionalidad actual, empleando consultores de mainframe, estudio de viabilidad de la nueva arquitectura, diseño del nuevo sistema empleando consultores especialistas en la nueva plataforma, planeación de liberación del nuevo aplicativo, construcción y mapeo del nuevo sistema contra el viejo, migración de los datos, pruebas de funcionamiento al nivel técnico, operativo y funcional, pruebas de estrés, de regresión, liberación piloto, instalación final y eliminación del sistema viejo. Por supuesto, lo anterior es un bosquejo de lo que debería de hacerse, porque un proyecto de estos demanda muchas cosas más, como por ejemplo: recursos que no son de sistemas (usuarios operativos), equipo de hardware nuevo, evaluación de costo, rentabilidad y viabilidad, logística, cultura organizacional y capacitación, etcétera. Hablando en términos de negocio: es muy costoso. Por último, si una organización ha considerando invertir en un proyecto así, lo que definitivamente ha detenido a muchos, hasta el momento, es un factor crucial: el riesgo. Muchos directores de negocio ven innecesario tomar este riesgo ¿Por qué cambiar, si no hay garantía de que el nuevo sistema proporcionará la misma funcionalidad? Al nivel de negocio, ¿es tan importante migrar a la tecnología de moda como para poner en riesgo la continuidad del negocio?

Otro sector de mercado ni siquiera pone en tema de discusión un proyecto de migración como algo a considerar en su portafolio de proyectos, sencillamente el mainframe es lo único que da el soporte en volumen, facilidad de administración, seguridad informática y eficiencia, como ya se mencionó, el volumen de negocio y el retorno de la inversión justifican los gastos asociados.

Como última descripción de la plataforma, en diversos artículos de la Internet, el lenguaje *COBOL* (el más usado en la plataforma mainframe) es descrito como los cables feos y fierros que existen debajo del motor de un auto, nadie los ve pero están ahí, en cantidades que sorprenden y hacen que el automóvil se mueva, una proyección emitida en el año 2007 es que para el año 2025 al nivel global habrá 1 trillón de líneas de código y un poco menos de la mitad seguirá siendo *COBOL* [25].

1.4 DELIMITACIÓN Y JUSTIFICACIÓN DEL PROBLEMA.

Si tomamos el marco de referencia mencionado en el punto 1.3, se pueden observar muchos campos de investigación y mejora en la plataforma mainframe, primeramente en el mercado

mexicano y después en el mercado mundial. La decisión de tomar como caso aplicativo a esta plataforma es la de generar conocimiento en un ambiente donde todo tiene licencia, es demasiado caro, hay déficit de personal y en el cual no está muy extendido el uso de herramientas de análisis automatizadas. Son demasiados problemas los que se han identificado durante el desarrollo del presente capitulo y no se pretende ni es posible resolverlos todos, por lo cual es necesario delimitar el problema que se pretende resolver y con esto generar conocimiento que ayude a solventar algunos de los problemas identificados.

1.4.1 OBJETIVO.

La propuesta del presente trabajo se enfocará en el análisis de aplicativos en mainframe. En un procedimiento que disminuya el riesgo asociado al análisis manual en dicha plataforma y que a la vez reduzca el tiempo de análisis. El resultado debería ser una herramienta de utilidad para los analistas de sistemas. Se basará la propuesta en técnicas de ingeniería de software y de teoría de análisis sintáctico de compiladores. En resumen, el objetivo es: Elaborar una herramienta de análisis para sistemas de información desarrollados en *COBOL* en ambiente mainframe como un auxiliar para el analista de sistemas en la documentación, mantenimiento y depuración de errores.

El enfoque es sobre la versión *VS COBOL II* de *IBM®.* Además el alcance de la investigación de limita a un solo elemento de compilación dejando el tema de extrapolación a todo un sistema y componentes no *COBOL*, para futuras propuestas de investigación. Esto es: el problema a resolver se limitará a un solo elemento en el lenguaje descrito por lo que no se incluye la solución a todo un sistema o múltiples unidades de compilación ni tampoco a componentes externos heterogéneos (diferentes tecnologías en interacción).

1.4.2 JUSTIFICACIÓN DEL PROBLEMA.

El problema es originado por la siguiente situación, lo cual justifica su planteamiento: Debido a que durante experiencia profesional del autor se ha observado en muchas ocasiones que el mantenimiento de sistemas heredados en la plataforma mainframe carece de especificaciones de diseño confiables, las razones se han discutido previamente, cuando un equipo de desarrollo de software debe atender requerimientos que impliquen dar mantenimiento (corregir errores detectados durante la operatividad) o bien introducir funcionalidad nueva, es necesario que personal con conocimiento elevado del aplicativo analice y diseñe la solución. El análisis realizado es completamente manual, en la práctica no existe algún método o herramienta confiable que garantice recuperar las reglas de negocio implementadas en los componentes. Esta actividad implica un riesgo, de la misma depende un diseño adecuado de la solución, la facilidad para soportar correcciones futuras, la conclusión de la codificación a tiempo y el cumplimiento del requerimiento de negocio solicitado. Se observan actualmente muchos problemas debido a la carencia de personal experimentado y la demanda constante de requerimientos que producen las organizaciones, cuando personas que desconocen la funcionalidad efectúan las tareas de análisis y diseño sin una base confiable de inicio (documentación del diseño del sistema actual) se producen situaciones no deseables: tiempo excesivo de análisis y diseños inadecuados, esta materialización del riesgo se refleja en muchos problemas en cascada: muchas veces proyectos ya

avanzados en la construcción tienen que iniciar desde cero, todo este re-trabajo tiene costos elevados, un diseño de solución inadecuado en el aspecto técnico y funcional desemboca muchas veces en errores de sistema que se ponen de manifiesto durante la operativa en ambientes de producción y por último: si el requerimiento fuese mal interpretado debido a un análisis inadecuado (qué entendió el ejecutor contra lo que el usuario realmente quiere), el costo derivado a este error corresponderá principalmente a la etapa de implementación en la que sea detectado. El peor escenario es que el usuario detecte que el sistema hace cosas que él no pidió tiempo después de que el sistema ha sido liberado.

Además del objetivo principal descrito, un segundo escenario aplicativo es que el resultado de esta investigación puede ser usado como una herramienta auxiliar para facilitar el cambio de plataforma, la cuál puede ser otro lenguaje que se ejecute en máquinas más baratas (Visual Basic, C) o bien un paradigma distinto, como el de orientación a objetos (Java, C++).

En cualquiera de los 2 escenarios mencionados, la investigación se plantea como auxiliar al problema de la curva de aprendizaje en el análisis de aplicativos existentes, ayudando a reducir el costo y el riesgo asociado al análisis manual que normalmente se hace en el mainframe.

1.4.3 PLANTEAMIENTO DEL PROBLEMA.

El problema en el que se enfoca este trabajo de investigación está delimitado a temas relacionados con el análisis de un aplicativo. Se requiere el planteamiento de un procedimiento semiautomático de análisis, haciendo uso de algunas técnicas de ingeniería de software y teoría de análisis sintáctico de compiladores, enfocado a primordialmente a programas en *COBOL* ejecutados en ambiente mainframe. Se selecciona como primera instancia a este lenguaje para el caso de aplicación. El procedimiento semiautomático no será ejecutado necesariamente en ambiente mainframe.

Se desarrollará una herramienta que, a través de una presentación gráfica o en forma de texto proporcione la suficiente información sobre un programa en *COBOL* para que éste pueda considerarse documentado y servir de base para análisis que lleve a un rápido diagnóstico o decisión esencial para resolver un problema (un incidente en producción por ejemplo) o entender su propósito y funcionamiento (para implementar un nuevo requerimiento de negocio). Junto con esta herramienta se presentarán una guía sobre su utilización.

Este trabajo no pretende crear una herramienta de mantenimiento automatizado o metodología de diagnóstico. Únicamente se entregará un resultado que servirá de ayuda a una persona con el conocimiento suficiente para entenderlo y tomar una decisión más rápidamente en el estudio de un programa o conjunto de programas.

Los productos a obtener se listan de manera general:

- La herramienta o componente semiautomático.
- Instrucciones sobre su utilización (breve manual de usuario).
- Directrices de aplicación para la correcta interpretación de resultados.

Se puntualiza que el resultado se desarrollará de la manera más generalizada posible. Debido al tiempo asociado, tal vez la implementación del procedimiento no se alcance completamente, pero se dejaría como tema a resolver ya sea utilizando o generando nuevas áreas de conocimiento, esto también aplica a la comprobación experimental de la utilidad del mismo.

Como ya se mencionó, la naturaleza propuesta para la investigación es inicialmente genérica, el conocimiento generado debería ser aplicable a cualquier lenguaje imperativo, sobre esta premisa se tratará de conducir la investigación. Se eligió como caso práctico de aplicación de la solución a *COBOL* por ser el lenguaje más utilizado en la plataforma tecnológica de estudio. Conforme la investigación se vaya desarrollando se determinará si la misma conserva su carácter genérico, en caso de no ser posible, por encontrar muchas restricciones a un lenguaje específico o por tiempo dedicado a la investigación, la intención genérica podría ser encaminada a un solo lenguaje.

CAPITULO II. DEFINICIÓN DEL MARCO TEÓRICO Y ANÁLISIS DEL PROBLEMA.

En este capítulo se presenta el estado del arte en materia de teorías que pueden ser aplicadas a la solución del problema, se secciona el problema en las diferentes teorías de conocimiento que requieren ser revisadas para extraer de ellas una propuesta aplicativa y se señala en diferentes puntos de este apartado qué teorías podrían ser aplicadas a la solución. Primero se documenta un resumen de las consideraciones necesarias a tener en cuenta durante el análisis funcional de aplicativos en mainframe. Después se documentan las principales teorías e investigaciones actuales involucradas en la solución del problema, estas son: el análisis de flujo de datos y el de control de flujo de los programas, la extracción automática de reglas de negocio desde el código fuente en aplicativos *COBOL* y la expresión de conocimiento en formatos simbólicos o gráficos.

2.1 CARACTERÍSTICAS DEL PROBLEMA A RESOLVER.

El problema descrito al final del capítulo I ha sido ampliamente estudiado, muchas investigaciones relacionadas serán referenciadas durante el desarrollo del presente capítulo; algunas presentan soluciones parciales, otras no están implementadas del todo y las que son implementadas de forma más extensa tienen la característica de que son productos comerciales con licencia, manejan salidas aún demasiado complejas y además tienen costo elevado.

No es una sola teoría o área de conocimiento la que se involucra en la solución del problema en tratamiento, son varias (análisis de flujos, extracción y reconocimiento de patrones, compiladores, transformaciones de paradigma, ingeniería de software, expresión gráfica del conocimiento, etc.). Las soluciones conocidas hasta la fecha sirven como herramienta auxiliar para un analista informático. Como se ha mencionado anteriormente, el autor ha observado durante su experiencia profesional que las organizaciones dependen mucho del trabajo manual, que efectivamente hoy en día existen muchos errores asociados al análisis manual de aplicativos y que el único producto observado en la plataforma de estudio (la familia de productos *ASG*™), maneja resultados muy complejos y tiene costo elevado, dichas características se asumen como las causas principales de su poca popularidad en los lugares donde se ha visto implementado.

La solución aplicativa que se propone en este trabajo se diseñará para tener las siguientes características: fácil de interpretar en la práctica, debe ser una herramienta ágil que presente resultados sencillos de manejar y que además acepte la inclusión de un complemento asociado al trabajo manual, interpretaciones abstractas y conclusiones que hasta este momento sólo el cerebro humano puede procesar o inferir, lo cual significa que el trabajo manual estará relacionado a la interpretación de los resultados y ajustes de carácter abstracto, no se pretende que sean utilizadas tareas manuales para extraer los componentes de micro y macro estructura del programa a analizar, ni tampoco para analizar el control de flujo y de datos, pues esto será analizado de manera automática, así mismo se anticipa exactitud y confiabilidad en las tareas automáticas realizadas. El diseño y la construcción son resultado de la compilación de diferentes tecnologías e investigaciones en materia de análisis de código y extracción de conocimiento, se pretende, con base en las mismas, presentar un procedimiento de naturaleza práctica y aplicativa sobre un solo elemento de compilación.

El presente análisis inicia con un resumen de los puntos a considerar cuando de analiza un aplicativo heredado en lenguaje *COBOL*, el primero de ellos es una breve descripción del lenguaje, posteriormente las implicaciones de análisis en estos aplicativos, las cuales han sido extraídas de algunas investigaciones y también se incluyen algunas observaciones del autor que han sido colectadas durante su experiencia profesional. El siguiente punto es la revisión de algunas teorías de análisis formal de software, las cuales son conceptos que utilizan algoritmos formales que se ven materializados pocas veces y cuando esto sucede, se implementan de manera parcial o ligeramente relajados con algoritmos más prácticos. En general y durante el desarrollo de la solución del problema, estos conceptos teóricos se deben tener en consideración para que la esencia de los mismos se refleje en el producto final a obtener. Enseguida se presenta una revisión de algunas técnicas de análisis de flujos, sobre dichas técnicas se planea dar soporte a una parte de la construcción de la parte automática, que tendrá como punto de inicio un analizador sintáctico de *COBOL* que ayude a extraer los componentes de la micro y macro estructura del lenguaje y que ayude además a la extracción del control de flujo del programa, lo cual es un resultado parcial de conocimiento. Una vez extraída la información en crudo de un programa, el siguiente paso es extraer conocimiento, para este punto, el marco teórico son investigaciones relacionadas a la extracción de la funcionalidad de los programas por medio de técnicas que se valen de reglas heurísticas aplicadas de manera automática para extraer funcionalidad. La última parte del capítulo se refiere a investigaciones relacionadas a la presentación de los resultados y las de particular interés son las que versan sobre la expresión de conocimiento en forma gráfica o simbólica.

El presente capítulo es pues una colección de investigaciones que sirven de marco teórico. Sobre este marco teórico se tomarán algunos puntos para proponer un diseño aplicativo, que forma parte de la solución al problema propuesto en el capítulo anterior. Este diseño de alto nivel servirá como guía durante la construcción de la solución, que una vez implementada deberá ser acompañada de documentación que ayude a un analista usuario a interpretar los resultados.

2.2 El ANÁLISIS DE APLICATIVOS EN COBOL PARA LA PLATAFORMA MAINFRAME.

Cuando existe un requerimiento de modificación sobre un sistema heredado en *COBOL*, es necesario tener identificado qué cosa es lo que tiene implementado el sistema informático actual. Esta información puede estar disponible, estar presente solo de manera parcial o completamente ausente. Los involucrados en la implementación y administración de proyectos la obtienen basados en la documentación disponible, el conocimiento humano (experiencia de los desarrolladores del aplicativo afectado) y el análisis manual a un alto nivel. Dicha información es usada para la toma de decisiones de viabilidad, valoración y estimado de recursos, calendario de implementación y limitación de alcances o restricciones. Si la información no es correcta se presentarán desviaciones importantes en la ejecución e implementación de los proyectos. Para el caso específico de la plataforma mainframe el análisis o identificación de los puntos a implementar en un proyecto evolutivo o de mantenimiento, así como las características técnicas y funcionales del sistema actual, es un tema muy importante, si se posee esta información de manera exacta y confiable, muchos de los problemas que ocurren durante la ejecución o después de la implementación de los proyectos podrían evitarse o por lo menos reducirse su impacto.

2.2.1 DESCRIPCIÓN DE LAS CARACTERÍSTICAS DEL LENGUAJE DE PROGRAMACIÓN COBOL.

COBOL es un lenguaje antiguo de programación que es usado en la actualidad, su nombre son las siglas de *COmmon Business-Oriented Language* (lenguaje común orientado a negocios), lo cual define su principal dominio en los sistemas de negocios finanzas y de administración en las compañías de la iniciativa privada y gubernamentales. La primera aparición del lenguaje se remonta a 1959 y de ahí a la fecha se han creado varios dialectos que se ejecutan en diversa cantidad de máquinas. El paradigma de programación es imperativo orientado a procedimientos, actualmente este tipo de programación es lo que más se usa cuando se desarrolla en este lenguaje, aunque en 2002 la Organización Internacional para la Estandarización (*ISO*) emitió un estándar para incluir programación orientada a objetos la cual está disponible desde ese año. Desde su creación el Instituto Americano de Estándares Nacionales (*ANSI*) produjo varias revisiones del estándar de *COBOL*, a partir de 1985 los estándares ANSI fueron adoptados por la *ISO* y de ahí a la fecha todos los reportes técnicos han sido emitidos por la *ISO* y adoptados por la *ANSI*. El último estándar fue publicado el 23 de julio de 2009. Los programas *COBOL* se usan de manera global en dependencias de gobierno y militares, en empresas comerciales y en sistemas operativos tales como *IBM®* z/OS, AS/400, Microsoft Windows y las familias *UNIX* (Linux, *SUN OS*, *AIX*, *HP-UX*) [26].

El lenguaje tiene un manejo de tipos de variables muy riguroso, muy fuerte, seguridad fuerte en tipos de variables, revisión de tipos de variables en tiempo de compilación (*static typing*), a finales del siglo XXI el problema del año 2000 fue un ejemplo de enfoque de esfuerzos significativos en programación *COBOL*, en algunas ocasiones la solución a este problema fue ejecutada por las mismas personas que habían diseñado los sistemas originales décadas atrás, la razón principal a la cual se atribuye este problema es que todas las aplicaciones de negocio usan el campo de la fecha de manera recurrente y la cláusula *PICTURE* facilita el manejo de campos numéricos de longitud fija, incluyendo campos para almacenar años con dos dígitos, que fue la principal corrección de este problema. *COBOL* tiene demasiadas palabras reservadas (del orden de 400), llamadas *keywords*. Con referencia al rendimiento en tiempo de ejecución, varios análisis sugieren que dos áreas de *COBOL* estándar que consumen la mayor parte del tiempo de

ejecución son la entrada/salida (accesos a disco) y los ciclos (formados por instrucciones *PERFORM* y *GO TO*). Los accesos de entrada/salida son la parte más crucial, sin embargo, una vez que se ha direccionado debe manejarse paralelismo en el código para explotar el paralelismo en los accesos entrada/salida. La mayoría de las aplicaciones *COBOL* encajan dentro de dos categorías, *OLTP* y por lotes. En el procesamiento en línea de transacciones (*OLTP*), se explota un paralelismo entre transacciones para permitir varias copias de un programa en ejecución de manera concurrente en transacciones independientes (un tema de consideración aparte es el bloqueo de bases de datos, archivos, etc., como resultado de esto). Los programas *OLTP* se ejecutan en paralelo típicamente por monitores de procesamiento de transacciones (e.g. *CICS - Customer Information Control System*, Tuxedo - *Transactions for Unix, Extended for Distributed Operations*, *TPMS - Transaction Processing Management System*) dentro de los cuales se controla su ejecución. En estos casos el paralelismo ínter transacción muchas veces está restringido por parte de un tercer producto, por ejemplo un *RDBMS* (Sistema manejador de bases de datos relacional) subyacente. Como resultado, existe poco mejoramiento en el desempeño para dicho tipo de aplicaciones, no obstante siempre existen potenciales mejoras. En el procesamiento por lotes (*Batch*) donde programas más grandes tienden a manejar de manera serial un número grande de registros, sometiéndolos a un procesamiento mayoritariamente homogéneo. Este es un ciclo leer-modificar-escribir que es usado en muchos programas *COBOL*. En este caso la mayor parte del tiempo de ejecución se emplea en estos ciclos, como resultado, los mismos representan un punto importante sobre los cuales implementar mejoras en rendimiento.

En algunos análisis de *benchmark* de *COBOL* [27, 28] se ha señalado que siete verbos contabilizaban el 95% de las instrucciones en la ejecución, mientras que los mismos 7 contabilizaban casi el 90% de las instrucciones almacenadas. Estos verbos son *MOVE*, *IF*, *GO TO*, *ADD*, *PERFORM*, *READ* y *WRITE*, los cuales pueden ser agrupados como sigue:

Acceso a memoria: *MOVE*, *IF* y *ADD*, con el 70% de uso tanto estático como dinámico;
Ciclos: *PERFORM* y *GO TO*, con el 18% de uso dinámico y 27% de estático;
I/O: *READ* y *WRITE*, con el 7% de uso dinámico y el 5% de estático.
Los ciclos y las instrucciones I/O juntos cubren el 25% del tiempo de ejecución. Obviamente dentro de los ciclos un gran porcentaje del tiempo se emplea en accesos a memoria.

¿Cuáles son las ventajas de *COBOL*? Los partidarios del lenguaje resaltan varias de sus ventajas. Se hizo una revisión en foros de programación en la Internet para extraer opiniones favorables acerca del lenguaje, las cuales se listan a continuación [29-31].

- *COBOL* es robusto. Existen productos de diversos fabricantes orientados a la interacción con *COBOL* y otros que han sido desarrollados para ayudar a los programadores en áreas críticas de pruebas, depuración, análisis de aplicaciones, soporte de productos y reutilización de código.
- Es adecuado para aplicaciones de negocio. Los programas son relativamente fáciles de desarrollar, usar y mantener, es un lenguaje de alto nivel, con semejanza al idioma inglés, cuando es utilizado correctamente, puede parecer a una novela bien estructurada con apéndices tablas de referencia cruzada, capítulos pies de página y párrafos, esto lo hace el más fuerte cuando el mismo se adapta a casi todas las aplicaciones relacionadas a negocios.

- Auto documentado y fácil de aprender. A diferencia de otros lenguajes, se sabe que personal no técnico lo ha aprendido en unas cuantas semanas sin entender la arquitectura interna del ambiente operativo.
- Es fácil de mantener. *COBOL* es fácil de mantener, su sintaxis parecida al idioma inglés y semántica permiten que el mantenimiento sea realizado por personas diferentes a los programadores originales, aún más, el código fuente puede ser referido por personas que no tienen una formación en programación.
- Portabilidad de plataforma cruzada. *COBOL* es uno de los lenguajes que tiene la característica de portabilidad. Los usuarios del mismo pueden llevar sus aplicaciones a diferentes plataformas de hardware sin modificar o inclusive sin compilar el código fuente.
- Es escalable. Las aplicaciones codificadas en versiones anteriores pueden ser recompiladas con modificaciones mínimas a nuevas versiones. Actualmente se permite el llamado a componentes Java de tipo *JNI* (*Java Native Interface*), inclusión de código *XML* incrustado, el compilador tiene un analizador sintáctico de *XML*, soporte para la base de datos *DB2* (*Data Base 2*) *V9* con nuevos tipos de datos e instrucciones *SQL*, parecen simples las mejoras pero con esto se puede migrar a las aplicaciones sin afectar de manera significativa la lógica de negocio implementada para que puedan ser utilizadas como servicios Web, eliminando la necesidad de volver a crear aplicaciones tipo Java o *.NET* para tener la conectividad con interfaces con el usuario en la Web [32].

Por otro lado están los detractores, una cita muy célebre del científico ganador del premio Turing, Edsger Dijkstra quién en una carta a un editor en 1975 mencionó que "el uso de *COBOL* paraliza la mente, por lo tanto su enseñanza debe ser considerada una ofensa criminal" [33]. Principalmente se le atribuye una sintaxis verbosa que contribuye al crecimiento exagerado de los programas, esto a expensas del proceso de razonamiento necesario para el desarrollo de software. Otra característica que dificulta la estructuración de los programas es la ausencia de variables de uso local, una gran parte del código es definición de variables. No podemos dejar de lado el hecho de que el lenguaje es considerado obsoleto en el ámbito educativo, esto trae como consecuencia poca oferta de personas dedicadas a desarrollar en este lenguaje, lo cual se relaciona directamente con los costos, coloquialmente se dice que los programadores *COBOL* forman parte de una élite que cobra sueldos por encima del promedio en el mercado y cuyo rango de edad es en general elevado [23].

Como cualquier otro lenguaje, el código *COBOL* se puede hacer más verboso de lo necesario por ejemplo una de las raíces de la ecuación cuadrática $ax^2 + bx + c = 0$, se puede codificar de la siguiente manera en *COBOL*, utilizando la instrucción *COMPUTE*:

```
COMPUTE X = (-B + SQRT (B ** 2 - (4 * A * C))) / (2 * A)
```

La misma fórmula se puede escribir de manera menos concisa de la siguiente forma:

```
MULTIPLY B BY B GIVING B-SQUARED.
MULTIPLY 4 BY A GIVING FOUR-A.
MULTIPLY FOUR-A BY C GIVING FOUR-A-C.
SUBTRACT FOUR-A-C FROM B-SQUARED GIVING RESULT-1.
COMPUTE RESULT-2 = RESULT-1 ** .5.
SUBTRACT B FROM RESULT-2 GIVING NUMERATOR.
MULTIPLY 2 BY A GIVING DENOMINATOR.
DIVIDE NUMERATOR BY DENOMINATOR GIVING X.
```

Cual formula usar es tema de preferencias. En algunos casos la forma menos concisa es la más fácil de leer, por ejemplo:

```
ADD YEARS TO AGE.
MULTIPLY PRICE BY QUANTITY GIVING COST.
SUBTRACT DISCOUNT FROM COST GIVING FINAL-COST.
```

El lenguaje *COBOL* es probablemente uno de los más difíciles lenguajes para el cual escribir un compilador. No tiene una gramática que facilite el análisis sintáctico. Uno de los analizadores más sencillos de implementar es el descendente por la izquierda con análisis del símbolo más a la izquierda y con 1 símbolo predictivo, el cual es conocido como analizador de gramáticas *LL (1)*, es probable que para la fecha en que fue creado dicho lenguaje no muchos de sus creadores tuvieran mucho interés en implementar una gramática *LL (1)*, lo cual hace necesario un reacomodo de las producciones de la gramática para adecuarlo a un analizador del tipo mencionado. Al tener el lenguaje demasiadas palabras reservadas, hay demasiados formatos de entrada y demasiadas variantes, el análisis léxico es altamente dependiente del contexto. En términos coloquiales es a la vez un reto como una pesadilla. Un *parser* (analizador sintáctico) de *COBOL* puede ser utilizado como punto de partida para escribir herramientas de análisis, herramientas de formato de código fuente entre varios dialectos de *COBOL*, documentación automática o traducción a otros lenguajes como C o Java.

Si tomamos como referencia el momento de su creación, el lenguaje fue dotado de capacidades de auto documentación, una buena gestión de archivos y de gestión de los tipos de datos, a través de la palabra *PICTURE* para la definición de campos estructurados. Para evitar errores de redondeo en los cálculos que se producen al convertir los números a binario y que son inaceptables en temas comerciales, *COBOL* puede emplear y emplea por defecto números en base diez. Para facilitar la creación de programas en *COBOL*, la sintaxis del mismo fue creada de forma que fuese parecida al idioma inglés, evitando el uso de símbolos que se impusieron en lenguajes de programación posteriores. Pese a esto, a comienzos de los ochenta se fue quedando anticuado respecto a los nuevos paradigmas de programación y a los lenguajes que los implementaban. En la revisión de 1985 parte del problema de obsolescencia se solucionó, incorporando a *COBOL* variables locales, recursividad, uso de memoria dinámica y programación estructurada. En la revisión de 2002 se le añadió orientación a objetos, aunque desde la revisión de 1974 se podía crear un entorno de trabajo similar a la orientación a objetos y un método de generación de pantallas gráficas estandarizado. Antes de la inclusión de las nuevas características en el estándar oficial, muchos fabricantes de compiladores las añadían de forma no estándar. En la actualidad este proceso se está viendo en aumento con la integración de *COBOL* con la Internet. Existen varios compiladores que permiten emplear *COBOL* como lenguaje de generación de scripts y de servicios Web. También existen compiladores que permiten generar código *COBOL* para la plataforma .NET y EJB.

En lo que respecta a la identificación de los componentes del lenguaje. La estructura de un programa *COBOL* consiste de 4 divisiones:

- Identification Division
- Environment Division
- Data Division

- Procedure Division.

La *Identification Division* es obligatoria. Las otras tres son opcionales, pero si se las codifica, estas deben ser invocadas en orden. Cada división tiene un propósito específico y usa cláusulas, frases, instrucciones y declaraciones para cumplir con dicho objetivo. En esta división se especifica el nombre del programa y se proporciona información adicional, tal como el autor del programa y la fecha de compilación, debe comenzar con las palabras *IDENTIFICATION DIVISION* o *ID DIVISION*. El párrafo *PROGRAM-ID* es obligatorio y debe ser el primero de la división. Los restantes párrafos son opcionales y pueden ser escritos en cualquier orden.

La *Environment Division* de un programa *COBOL* brinda información sobre las características físicas específicas del ambiente en el que correrá el programa. Especifica información sobre la computadora en que se generó el programa y aquella en la que se ejecutará. También relaciona los archivos de entrada y salida del programa con los dispositivos de hardware específicos. La *Environment Division* se compone de dos secciones "*section*": La *Configuration Section* y la *Input-Output Section*.

La *Data Division* describe las estructuras de datos a ser procesadas por el programa *COBOL*. Puede contener hasta cinco secciones, lo cual depende de la versión o dialecto de *COBOL* en uso. Las primeras dos secciones son comunes a todas las versiones, pero de manera universal, todas las secciones son opcionales:

- File Section
- Working-Storage Section
- Local-Storage Section
- Linkage Section

Estas secciones, en orden, están compuestas por la descripción de los archivos, las descripciones de entrada de datos necesarias para ejecutar el programa, las descripciones de áreas de datos para recibir parámetros, las descripciones de las áreas para describir el formato de los datos a ser mostrados y recibidos desde pantalla y la descripción de las entradas de datos que permitirán generar distintas instancias de las variables en aquellos programas recursivos.

La *Procedure Division* Contiene los procedimientos, secciones, párrafos, oraciones, e instrucciones necesarias para ejecutar la funcionalidad para la que fue creado el programa y con ello resolver el problema de procesamiento de los datos.

La especificación del lenguaje y la estructura de las instrucciones en detalle, que servirá para construir el analizador sintáctico del dialecto utilizado en este trabajo se encuentra disponible en la Internet [34]. El dialecto es *VS COBOL II* para arquitectura *ESA*, sistema de arquitectura empresarial, las siglas *VS* significan sistema virtual y el *II* indica que es un dialecto del estándar *ANSI85*.

Enseguida se muestra el programa "*Hola mundo*" en *COBOL*:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Saludos.
ENVIRONMENT DIVISION.
```

```
DATA DIVISION.
*(DECLARACION DE VARIABLES QUE VA USAR EL PROGRAMA)
WORKING-STORAGE SECTION.
* SE PODRA ESCRIBIR HASTA 10 CARACTERES
77 CHARLA PIC X(10).

PROCEDURE DIVISION.
INICIO.
DISPLAY "Hola, como estas"
ACCEPT CHARLA
DISPLAY "Me alegro, un abrazo"
ACCEPT CHARLA
STOP RUN.
```

2.2.2 CONSIDERACIONES PARA EL ANÁLISIS MANUAL DE APLICATIVOS EN COBOL EN LA PLATAFORMA MAINFRAME.

Durante la experiencia profesional del autor se han observado las siguientes características e implicaciones relacionadas al análisis de aplicaciones cuando las mismas son sistemas heredados en lenguaje *COBOL* en la plataforma mainframe.

Si bien es cierto que el lenguaje es legible aún para personas que no tuvieron una formación en informática, la facilidad de obtención de conocimiento es real si los programas son pequeños en cuanto al número de líneas y la programación es razonablemente estructurada, sin el uso de instrucciones poco comunes, como por ejemplo apuntadores o nombres de variables demasiado apegados a formalismos matemáticos, en este caso estaríamos hablando de sistemas pequeños con pocas reglas de negocio implementadas en el mismo y cuyo costo de mantenimiento no debería ser alto, "cualquier" persona podría fácilmente navegar en el código fuente aún sin documentación formal y encontrar un error reportado o introducir una nueva funcionalidad.

Sin embargo el verdadero costo monetario en mantenimiento que se mencionó en la introducción de este trabajo, viene asociado con el mantenimiento de sistemas con complejidad mediana o alta. Si en la creación del mismo estuvieron involucrados cientos de programadores (o miles) y las reglas de negocio implementadas en un sistema no son manejables en cuanto a cantidad, aparecen problemas que rebasan las características de legibilidad y auto documentación, inclusive de lenguajes como *COBOL*.

Por ejemplo en algunos bancos, el sistema de abonos y cargos a cuentas de dinero de sus clientes es soportado por cientos de programas que realizan validaciones segmentadas en diferentes programas, algunos de los cuales cuentan con un número de líneas de código que va de 4 mil a 20 mil, las validaciones y reglas implementadas en una aplicación de este tipo se cuentan por miles. Para un cargo o cobro de un cheque, el sentido común dicta que la validación más importante es el saldo disponible, sin embargo además de esta, muchas otras reglas aplican cuando una de estas operaciones ocurre en tiempo de ejecución: no tener adeudos, ser un cliente con datos correctamente alimentados en diferentes catálogos, existencia de valores en bases de datos corporativas (tasa de impuesto, tipo de cambio, fecha contable, firmas autógrafas, catalogo de

cheques, etc.), validaciones de lavado de dinero, planes de comisiones, impuestos de las operaciones, promociones como sorteos, otros beneficios de los productos involucrados tales como descuentos, bonificaciones, cuentas mancomunadas, catálogos de cheques robados, etc., etc. Si además de estos requerimientos de negocio se agrega que muchos de los programas implementan reglas parciales para simular paralelismo o tienen optimizaciones de carácter más técnico para mejorar el rendimiento (una organización bancaria en México hace normalmente operaciones de cargo/abono en un orden de 100,000 por día ya sea en línea o en procesamiento por lotes y durante la vida del sistema las mejoras en rendimiento no son cosa trivial, porque son demasiadas las operaciones que se ejecutan en el *CPU* de la computadora), el resultado es que se tienen programas enormes en tamaño, con un largo historial de modificaciones, coloquialmente conocidos como "parches", programación con diferente estilo de estructuración, reglas parciales y varias instrucciones poco legibles. Si el programa no tiene una documentación perfectamente actualizada (lo que ocurre casi siempre), aún la persona más experimentada invierte una gran cantidad de tiempo en identificar donde está implementada cierta funcionalidad, digamos para corregir un error asociada a la misma o para introducir nuevos módulos derivados de nuevos requerimientos de negocio.

Es en este tipo de sistemas donde el trabajo de análisis manual se torna en muy crítico, existe además otro problema: generalmente un sistema como el descrito arriba no cuenta con tiempo de holgura en la implementación de los proyectos, si por ejemplo, en la misma organización bancaria mencionada arriba, una nueva disposición legal obliga a poner un nuevo dato informativo en los estados de cuenta de los clientes para una cierta fecha, no hay cabida para holgura porque los costos de retrasos asociados con multas, interrupción del servicio, desprestigio y demás son muy altos. En estos casos debe ser realizado un análisis rápido, derivado de esto, un diseño y una implementación. Parece un poco trivial y sencillo el proceso, pero las interrupciones de servicio, el no cumplimiento de fechas y normas, los errores en ambiente productivo y la afectación a los clientes son más comunes de lo que se piensa y de manera inequívoca un mal análisis de los requerimientos es muchas veces la causa de las fallas.

¿Qué implica analizar un aplicativo en *COBOL*? El análisis de código heredado en la práctica requiere de mucha lectura de código y trabajo manual. En la mayoría de los casos el mismo se realiza de manera empírica y está basado en la experiencia profesional o habilidades de análisis de la persona que ejecuta dicha tarea, normalmente esto es un proceso personal, no existe una manera institucional de hacerlo en las organizaciones, cada analista tiene sus propios métodos y herramientas que desarrolla de manera individual, se expone a continuación la descripción de las tareas que son desarrolladas comúnmente para auxiliar y obtener resultados de valor durante un análisis, que es una colección de las principales técnicas no formales observadas entre los analistas de sistemas durante la experiencia profesional del autor, posteriormente, durante el diseño de la solución del caso aplicativo, se identificará qué puntos de las técnicas de carácter práctico pueden ser utilizadas para la construcción de dicha solución, aunado a esto se considerarán los resultados de investigaciones previas de carácter más formal, las cuales de describen más adelante en este capítulo.

Entre las tareas repetitivas que se ejecutan de forma manual cuando se analizan aplicativos heredados en *COBOL* se encuentran las siguientes:

Listar lo que se tiene. Se elaboran listas de componentes del sistema en análisis, entre ellas destacan la lista de nombres de archivos, de las bases de datos, de los módulos, o de las clases, funciones y procedimientos dentro de ellos. Lista de interfaces: procedimientos públicos, datos

públicos, etcétera. Lista de variables globales y de constantes. Si no se cuenta con uno, normalmente se debe elaborar un diccionario alfabético de todos los nombres usados en el sistema o en un programa en análisis. Las listas son útiles si el código es completamente nuevo o visto por primera vez por un analista, esta tarea se ve aligerada si las listas ya existen.

Entender procedimientos. Los procedimientos contienen la implementación de las reglas de negocio. Se debe revisar un procedimiento para ver qué es lo que hace. Se debe identificar que es lo que se calcula o procesa dentro de los mismos y sus llamadas a otros procedimientos o a otros módulos y al mismo tiempo evaluar si el análisis de un camino lógico de ejecución es relevante para el objetivo que dio origen a la tarea o no lo es (como por ejemplo tareas corporativas de validación y conversión de formatos, consulta de catálogos para obtener descripciones, etcétera). Se identifica también a las variables que son leídas y escritas. Otra actividad que normalmente se hace es identificar y entender el uso de los parámetros, identificar cuál es un parámetro de entrada y cual devuelve un valor de salida. La respuesta a la pregunta ¿Tiene el procedimiento efectos colaterales? Es muy importante, es decir, la identificación de si el mismo escribe en estructuras de datos de salida.

Identificar la secuencia de ejecución de los procedimientos. Se identifican en esta tarea los llamadores de procedimientos, quién llama a quién, si un procedimiento o una función son llamados por otras funciones, si es una función única o es reutilizada en todo el sistema, proyecto o programa.

Determinar el uso de las variables de salida. Identificar si los procedimientos o funciones regresan un valor utilizado en algún lugar, así como también si los llamadores toman alguna acción dependiendo del valor retornado por otra función o el valor retornado es esencialmente un dato muerto.

Identificar procedimientos o variables no usados. Que casi siempre existen en sistemas de producción y no deben ser considerados para el análisis esencial de una aplicación.

Elaborar diagramas o esquemas de flujo de datos. Una vez que se ha identificado información en bruto, una buena práctica consiste en consolidar el conocimiento obtenido en forma de diagramas o gráficos, donde se visualicen las ramas de ejecución, las instrucciones condicionales ciclos y saltos, el flujo de la información, las entidades de datos de entrada y de salida, identificación de los puntos del sistema donde se tienen implementadas reglas de negocio importantes, etcétera. El hecho de obtener un diagrama de este tipo a partir del código representa un avance importante, con este diagrama se tiene mucha abstracción de lo que pasa al nivel de detalle dentro del sistema. En lo que se refiere al tema de árboles de control de flujo, esta parte es importante, pues aquí se identifican los procedimientos llamadores y llamados, ¿cómo fluye el control de la ejecución entre módulos y procedimientos? ¿Son todos los procedimientos llamados alguna vez? En esta identificación se analiza el paso de estructuras de datos y también si el flujo es en una sola dirección o en ambas.

Entender las variables. Ciertas variables globales o arreglos pudieran ser significativos para la aplicación (variables de dominio) cuando se analiza código heredado, una de las tareas es resaltar las estructuras de datos y variables clave y determinar cómo están siendo usadas. Una vez que se ha identificado a las variables importantes, se deben clasificar, por ejemplo en variables de tipo lectura, localizar la ubicación donde es requerido el valor de una variable, identificar la escritura y actualización de variables, comprender dónde y cómo una variable obtiene su valor y como el

mismo es modificado, se deben resolver interrogantes como ¿La variable es modificada solamente en ese punto? ¿Puede cambiar inesperadamente como efecto colateral de alguna función o procedimiento exótico? Es recomendable también identificar información cruzada de las variables. En el caso de las constantes se revisa que tipo de valores les son asignados y en donde son utilizados los mismos.

Dependencias de módulos. Es muy común que un programa consista en un gran número de módulos interdependientes. Algunas veces entender un módulo es imposible si no se ha comprendido su relación con otros módulos. Sin embargo esto es resuelto a criterio del analista pues muchas veces al tratar de entender todos los módulos se puede perder el objetivo principal del análisis.

Rol de las entidades de entrada y de salida. En la medida de lo posible, durante el análisis de los componentes de debe identificar qué papel juegan las entidades de entrada y salida identificadas y si las mismas dependen de otros archivos, tablas de bases de datos, parámetros de entrada salida o bien estructuras de datos internas manejadas en código duro.

En la práctica, los siguientes puntos pueden ayudar a identificar código importante.

- *Ubicar los módulos y procedimientos más grandes*, se los puede simplemente ordenar por el número de líneas o instrucciones y tomar los más grandes, usualmente por experiencia se sabe que gran cantidad de código hace cosas grandes.
- *Ubicar los procedimientos más usados*, si un procedimiento tiene demasiados llamados, es candidato a ser importante.
- *Ubicar los procedimientos más complejos.* Un procedimiento con un gran número de ramificaciones o anidamiento condicional muy profundo es complejo. Frecuentemente un procedimiento complejo contiene mucha lógica interna, la obtención del grado de complejidad basado en el anidamiento y ramificación es recomendada.
- *Encontrar los procedimientos que ejecutan la mayoría del código*, un procedimiento en la parte alta del código ejecuta casi todos los demás procedimientos en el árbol de ejecución. Esto porque invoca la ejecución de cada subprocedimiento y el mismo.
- *Al encontrar módulos que son muy utilizados en el sistema*, estos deben ejecutar funciones importantes y la identificación de su funcionalidad es recomendable. Es recomendable también el cálculo o estimación de métricas relacionadas con el código.

Se aplicó una encuesta (**Anexo A**) a algunos analistas de la plataforma mainframe y programadores experimentados en *COBOL*, para confirmar que las actividades mencionadas son ejecutadas en promedio por las personas que analizan código en la plataforma mainframe, el contenido de la misma se anexa al final de esta tesis. En general se observa coincidencia entre lo que se ha expresado en este punto y lo que los encuestados respondieron.

Gran parte del conocimiento mencionado anteriormente es información que de alguna manera obtiene y analiza el compilador del lenguaje, por lo que una herramienta automatizada, auxiliar en las tareas de análisis, bien se puede basar en el analizador sintáctico del compilador, la información se puede recuperar de donde el compilador la almacena temporalmente o bien crear un nuevo almacén, posteriormente dicha información se puede utilizar para proyectar una seudo-especificación que tendrá limitaciones en cuanto a la abstracción e interpretación (algunas teorías formales revisadas en este capítulo o parte de ellas se deben aplicar para aproximar una

interpretación), pero tiene la gran ventaja de que el contenido es exacto, confiable y se obtuvo de manera automática, ahorrando el trabajo manual anteriormente expuesto.

2.3 EL ENFOQUE FORMAL DEL ANÁLISIS DE SOFTWARE.

En los últimos 15 años se ha visto un cambio acelerado hacia el razonamiento algorítmico formal esto es lo que algunos autores llaman *análisis formal de software*, esta forma de análisis tiene características particulares que la distinguen de otras formas de hacerlo. A pesar de los mejores y bien intencionados esfuerzos de los desarrolladores de software y de los investigadores, las técnicas disponibles para validar sistemas de información no son adecuadas aún para los retos de que presentan los sistemas modernos. La tendencia de conducirse hacia la complejidad del software no disminuye y nos aproximamos de manera rápida al punto en que las organizaciones liberan software el cual no entienden completamente como funciona. El análisis formal de software puede ser visto como descendiente de las metodologías basadas en matemáticas desarrolladas aproximadamente 4 décadas atrás en los trabajos de Hoare y Floyd [35] [36].

Mientras estos métodos comprensivos y de mucho trabajo manual han fallado para ganar uso amplio entre los desarrolladores, los análisis automáticos que se enfocan en nociones restringidas de validez han encontrado uso en nichos específicos de desarrollo de software, por ejemplo en el desarrollo de manejadores de dispositivos [37].

En el siguiente punto se describen de manera breve tres modelos formales de análisis de software. Su carácter riguroso, teórico y de difícil implementación, como se mencionó anteriormente sigue siendo la principal razón de poca popularidad entre los desarrolladores y analistas, sin embargo se debe mencionar que los conceptos en nivel general son importantes y deben ser tomados en cuenta de forma parcial por lo menos cuando se implementan herramientas de análisis. Esta es la razón por la que se expone su existencia.

2.3.1 ANÁLISIS FORMAL DE SOFTWARE.

Un análisis, $A : \delta \times \mathcal{P} \rightarrow \{\text{verdadero, falso}\}$ es una función booleana definida sobre un conjunto de sistemas δ y especificaciones $\mathcal{P}$. Hagamos que $\delta \vdash \mathcal{P}$ sea traducido como la prueba de que el software $S \in \delta$ satisface la especificación $P \in \mathcal{P}$.

En un mundo ideal, $\forall\ S \in \delta\ ,\ P \in \mathcal{P} : A(S, P) = \delta \vdash \mathcal{P}$, (los sistemas pueden ser deducidos o inferidos de las especificaciones) pero esto no es práctico, no es definitivo. Los análisis prácticos hacen ciertos sacrificios.

Un análisis es *sólido* si
$\forall\ S \in \delta\ ,\ P \in \mathcal{P} : A(S, P) \Rightarrow \delta \vdash \mathcal{P}$
y es *completo* si

$\forall S \in \delta, P \in \mathcal{P} : S \vdash P \Rightarrow A(S, P)$.

Cuando un análisis *sólido* produce un resultado positivo, esto puede ser considerado como evidencia *decisiva* de que la especificación soporta al software. Resultados negativos pueden indicar violación de la especificación o quizás no. Cuando un análisis completo produce resultados negativos, este es evidencia *decisiva* de que el software tiene un error con respecto a la especificación, esto es conocido también como una *detección de error sólida* esto es un análisis sólido para $A(S, \neg P)$.

Un análisis formal de software es una técnica matemática bien fundamentada para razonar sobre la semántica del software con respecto a la especificación precisa o comportamiento deseado para el cual las fuentes de poca solidez son definidas.

Esta definición resalta los temas de todos los métodos formales. Los fundamentos matemáticos permiten razonar no solo acerca de los resultados producidos por un análisis, sino que también acerca del análisis en sí mismo, e.g., su elaboración correcta. El enfoque en semántica y especificaciones está referenciado en trabajos anteriores de investigación [36]. El requerimiento de automatización es esencial, si uno espera escalar al tamaño y complejidad del software actual. La característica más distintiva de la definición es la insistencia de que un análisis debe de ser explícito al describir las formas en que un sistema es poco sólido. Este último punto amerita una discusión más profunda.

Con un análisis sólido se sabe cuando se obtiene un resultado positivo. Sin embargo muchas técnicas de análisis hacen sacrificios mayores para mayor escalabilidad al restringir δ y $\mathcal{P}$ por los cuáles son sólidos. Esto está bien mientras las restricciones estén claramente definidas, pero frecuentemente no lo están. El riesgo de tener poca solidez no especificada es que el desarrollador de $S' \in \delta$ puede interpretar erróneamente a $A(S', P)$ como una indicación de que S' satisface la especificación P, cuando esto no es así. Para que los análisis formales de software sean confiables deben definir claramente cuando son sólidos y cuando no. Este requerimiento tiene soporte en general solamente cuando el análisis formal muestra que una propiedad soporta al sistema. En contraste, si el análisis tiene el objetivo primario de encontrar una violación a la propiedad entonces se requiere una detección de errores sólida. Desafortunadamente en este caso si no se detecta error entonces típicamente no se sabe a qué conclusión llegar.

Existen actualmente 3 técnicas sobre las que descansan los avances recientes y avances futuros prometedores en lo que respecta a análisis formales, se mencionan en este trabajo como un estado actual de los avances teóricos, sin embargo es importante aclarar que por sí mismas estas teorías no constituyen una solución definitiva y que su materialización no es algo que tenga popularidad en los ambientes reales de trabajo.

La revisión de modelos [38] toma como entrada un sistema modelo *M* de estado de transición , representando el comportamiento del sistema *S'* y una propiedad *P* que debe ser revisada contra el sistema, entonces se exploran exhaustivamente todos los flujos lógicos a través de *M* mientras se verifica que *P* es verdadero en cada estado alcanzable. En sistemas concurrentes esta exploración exhaustiva de flujos lógicos considera todos los posibles entrelazados de transacciones concurrentes. El sistema de estado de transición *M* podría ser derivado del comportamiento de un modelo de diseño de alto nivel, el código fuente del sistema o la presentación ejecutable del sistema (código máquina o *byte code*). La propiedad *P* podría ser una

propiedad temporal (representada como una formula de lógica temporal, una expresión regular o un autómata) esto requiere de código con cierto orden en eventos particulares del sistema o una propiedad de estado tal como la de invariante o de aserción. El algoritmo de exploración estado-espacio puede encargarse de la búsqueda utilizando una representación explícita de las transiciones del sistema, propiedades y conjunto de estados alcanzables [39] o una representación simbólica. El espacio de estados alcanzables de *M* debe ser finito si el análisis debe terminar.

La interpretación abstracta [40] es un ambiente de trabajo (*framework*) basado en teorías de retículas (*lattice*) para diseñar y calcular sistemáticamente "sobre" y "sub" aproximaciones del comportamiento de un sistema *S*. Mientras muchos análisis formales diferentes incluyendo análisis de flujo tradicionales, cálculo de precondiciones débiles/post-condiciones fuertes, ejecución simbólica y revisión de modelos pueden ser proyectados en el ambiente de trabajo de interpretación abstracta, el mismo es más comúnmente asociado con análisis de flujo de datos diseñados rigurosamente estáticos [41]. En el contexto de análisis simple de flujo de datos simples, la formación de una interpretación abstracta sobre aproximada empieza por considerar un sistema *S* que contiene algunas variables de programa o celdas de memoria x_i de interés y algunas propiedades de retícula (*lattice*) $L = (P, \sqsubseteq_L)$ donde los elementos del conjunto de *P* pueden ser pensados como elementos simbólicos "*tokens"* que caracterizan valores de datos que pueden fluir en x_i durante la ejecución de *S* y donde $\sqsubseteq_L$ ordena las propiedades de *P* de acuerdo a la precisión/aproximación de la información representada (los *tokens* que son sobre aproximados están en la parte alta de la estructura *lattice*). El uso de la interpretación abstracta es amplio: muchos análisis estáticos de código que habilitan optimizaciones del compilador son interpretaciones abstractas [40]. La interpretación abstracta forma la base de sobre aproximaciones que hacen la revisión de modelos manejable y es utilizada directamente en la revisión de propiedades en ambientes de trabajo (*frameworks*). Sin importar si la interpretación abstracta es usada o no directamente es muy importante entender los conceptos de la misma porque los mismos predominan en casi todos los análisis formales.

Los métodos deductivos para análisis de software tienen sus orígenes en la lógica de Floyd y Hoare [35] [36] la cuál caracteriza el comportamiento de una instrucción de programa C usando tripletas de la forma:

$$\{P\}C\{Q\}$$

Donde *P*, las precondiciones y *Q*, las post-condiciones son fórmulas que describen propiedades de estados. La tripleta es válida si y solo si para cualquier estado *s* que satisface *P*, al ejecutar *C* sobre *s* esto da como resultado un estado *s'* que satisface a *Q*. Una formula *P* puede ser también visualizada como la caracterización de un conjunto de estados $[\![P]\!]$ a saber un conjunto de estados para los cuales *P* es soportado y una tripleta puede ser visualizada como un resumen del comportamiento de la entrada/salida de *C* en términos del conjunto de estados de salida $[\![Q]\!]$ que resultan del conjunto de entradas en $[\![P]\!]$. Una formula *Q* es más débil que *P* si $P \Rightarrow Q$ (*P* conlleva a *Q*). Intuitivamente esto significa que *Q* es menos restrictiva y más aproximativa que *P* y *Q* representa un resumen de estados menos preciso, un hecho que se aprecia más fácilmente cuando consideramos que $P \Rightarrow Q$ se soporta cuando $[\![P]\!] \subseteq [\![Q]\!]$. Haciendo referencia a lo mencionado sobre Interpretación Abstracta, las formulas de estado pueden ser visualizadas como abstracciones que resumen información de estado del programa y pueden ser arregladas en una aproximación natural de estructura entrelazada basada en una relación de acarreo y ordenamiento. Las reglas de inferencia de la lógica son estructuradas en forma de composición, en la cual el comportamiento de *C* es establecido en términos del comportamiento de sus componentes. Las

herramientas asociadas a los métodos de lógica Floyd/Hoare han requerido tradicionalmente de un alto grado de intervención manual para construir apropiadas pre/post condiciones. Sin embargo, una cantidad significativa de automatización puede obtenida si se usa el operador de la precondición más débil *wp*(C, Q) que, dada una instrucción *C* y post-condición *Q* automáticamente construye una pre-condición tal que {P}C{Q} es válida y *P* es la fórmula más débil que puede establecer a *Q* como una post-condición para *C*. Retomando la discusión de "la más débil", la precondición retornada por *wp* (C, Q) es la "mejor" en el sentido de que impone las menos restricciones sobre las entradas de *C* que pueden garantizar el soporte de *Q*. Un cálculo contiene reglas para calcular *wp* tales que la regla de asignación es de la forma:

$$wp\ (x := E, Q) = Q\ [E / x]$$

Esto es, para *Q* (vista como un predicado que enuncia una propiedad de *x*) que es soportado después de la asignación de *E* a *x*, *Q* debería ser soportado por el valor de *E* antes de la asignación. Un cálculo *wp* puede recorrer un camino largo para de manera automática razonar sobre lenguajes realistas, pero una intervención importante del usuario (por ejemplo construcción manual de ciclos invariantes) o compromisos sólidos para obtener un análisis completo. Su mayor beneficio puede ser su combinación sinérgica con otros análisis formales que es una tendencia actual.

A partir de estas 3 formas fundamentales de enfocar el análisis formal de software, existen otras técnicas derivadas, entre las cuáles se pueden mencionar las siguientes: Análisis sensible al camino de ejecución, abstracción de predicados [43], abstracción de la pila del programa (*heap*) [44] [45], ejecución simbólica , simplificación del control, reducciones de orden parcial [34] [46].

2.4 ANÁLISIS DE FLUJOS.

El análisis de flujo es una técnica tradicional que utilizan los compiladores para obtener información útil de los programas en tiempo de compilación, sin embargo el conocimiento que se puede obtener de este análisis se considera parte importante para describir la funcionalidad del mismo, por lo cual lo relacionado al tema, es de especial interés para el diseño de la herramienta automática.

El análisis de programas proporciona métodos automáticos, útiles para la toma de decisiones y para el análisis de las propiedades de los programas. Desde que la mayoría de ellos implícitamente involucran preguntas sobre la finalización del programa, las propiedades están enfocadas a detectar errores en el momento de su desarrollo. Para cada análisis se impone un orden a las propiedades, por ejemplo estipulando que una propiedad es más grande que otra si más valores satisfacen a la primera. Las propiedades son entonces interpretadas de tal manera que un análisis permanece correcto incluso cuando el mismo produce una propiedad más grande que la idealmente posible. Esto corresponde a producir una inferencia válida en la lógica de un programa para correcciones parciales. Sin embargo el análisis de los programas es generalmente más eficiente que la verificación del mismo y por esa razón más aproximada, porque el enfoque de verificación requiere de un proceso automático completo en programas grandes.

El análisis de control de flujos es una técnica estática para calcular y predecir de manera segura aproximaciones de un conjunto de valores que los objetos de un programa pueden tomar durante la ejecución del mismo [47]. El análisis de programas se enfoca a las propiedades analizables que son soportadas durante la ejecución sin importar cuales son los datos reales sobre los cuales opera el programa y también sin importar el ambiente sobre el cual se ejecuta el mismo. Tradicionalmente el análisis de programas ha sido utilizado por los compiladores para optimizar la implementación de los mismos. Varias clases de lenguajes de programación tienen desarrolladas técnicas específicas. Por ejemplo el análisis del flujo de datos [48] fue desarrollado principalmente para lenguajes de paradigma imperativo y sin embargo es utilizado también para lenguajes orientados a objetos y el análisis de control de flujo [49] fue desarrollado principalmente para lenguajes de paradigma funcional, pero puede ser utilizado en lenguajes orientados a objetos [50] y en lenguajes con concurrencia [51].

2.4.1 LOS PROBLEMAS DE ANÁLISIS DE FLUJO.

El análisis de flujo determina hechos invariantes al camino lógico de ejecución en puntos específicos de un programa. Un problema de análisis de flujo es una pregunta de la forma:

"¿Qué se cumple en un punto determinado del programa *p*, independientemente del camino de ejecución que se tome desde el inicio del programa hasta *p*?"

Los siguientes pueden ser dominios de interés:

- Análisis de rango. ¿Qué rango de valores es una referencia dada para restringir el valor de una variable de tipo entero a estar dentro de ese rango?
- Detección de ciclos que no varían. ¿Todas las asignaciones previas a una variable de referencia quedan fuera del ciclo que las contiene?

En los últimos 30 años se han desarrollado técnicas estándar para responder a estas preguntas en lenguajes de paradigma imperativo [52] [53]. El análisis de flujo es quizá la herramienta dominante en utilizada para la optimización de código, algunos ejemplos de las optimizaciones que usan esta técnica son: Alojamiento global de registros, eliminación global de subexpresiones comunes, detección de ciclos que no cambian de estado, eliminación de código muerto, propagación de valores constantes, análisis de rangos, colocación de código en memoria alta, eliminación de inducción de variables, propagación de copias, análisis de variables vivas, expansión de ciclos, desbloqueo de ciclo e inferencia de tipos. Los algoritmos tradicionales de análisis de flujo se enfocan en las instrucciones de asignación, consideremos el siguiente fragmento de código *COBOL*:

```
SET I TO 0
PERFORM UNTIL I > 30
  S = A (I)
  IF S < 0
    COMPUTE A (I) = (S + 4) ** 2
  ELSE
    COMPUTE A (I) =  FUNCTION COS (S + 4)
```

```
  END-IF
  COMPUTE B (I) = S + 4
  ADD 1 TO I
END-PERFORM
```

El análisis de flujo tradicional requiere la construcción de una gráfica de control de flujo para dicho fragmento, como se muestra en la figura 2.

Cada vértice (o nodo) de la gráfica representa un bloque de código: una secuencia de instrucciones tales que solamente las ramas que van al bloque son las ramas dirigidas al inicio del bloque y las únicas ramas que salen de cada bloque son las que ocurren al final del bloque. Las aristas de la gráfica representan posibles transferencias de control entre bloques básicos. Cuando se ha construido la gráfica de control de flujo se pueden utilizar algoritmos de gráficas para determinar factores invariantes en la ruta lógica de ejecución en los vértices.

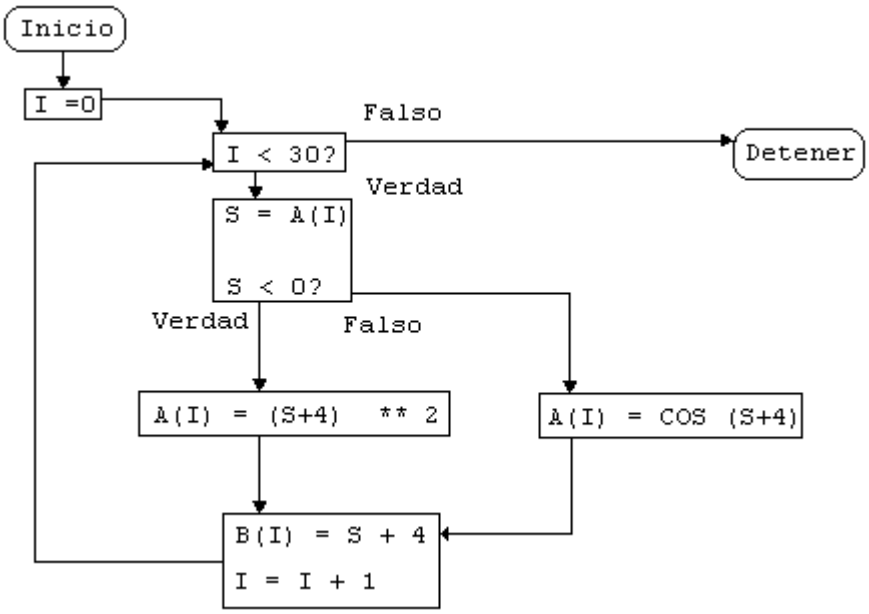


Figura 2 Gráfica de control de flujo en seudo-código.

En este ejemplo, se puede determinar que en todas las variantes de flujo de ejecución desde *Inicio* hacia el bloque *B (I) = S + 4* y *I = I + 1*, la expresión *S + 4* se calcula sin ninguna asignación previa a *S*. Por lo tanto al asignar el valor de *S + 4* a una variable temporal, se puede eliminar la instrucción redundante de suma en *B (I) = S + 4*. Esta información se obtiene a través de la consideración de los caminos de ejecución a través de la gráfica de control de flujo y es usada muchas veces por los compiladores para optimizar el código.

La identificación de ciclos es un paso necesario para la transformación a arquitecturas de código de alto rendimiento. Algunos compiladores detectan ciclos porque estos son parte de la sintaxis del lenguaje (*FOR, WHILE*) mientras que otros compiladores los detectan utilizando gráficas de flujo.

En lenguajes de nivel intermedio la técnica de utilizar gráficas puede inclusive detectar ciclos con la instrucción *GO TO*. Uno de los algoritmos para detectar ciclos es el algoritmo de Tarjan [54].

Los intervalos de Tarjan son de entrada simple y fuertemente relacionados a subgráficas, existen otras propuestas que manejan ciclos con entradas múltiples, los cuales son llamados ciclos sin reducción.

Una gráfica de flujo es una gráfica conectada y dirigida *G = (N, E, INI, FIN)*, donde *N* es un conjunto de nodos (o vértices), *E* es un conjunto de aristas (o arcos), *INI* ϵ *N* y es el nodo identificado como inicial que no tiene ninguna arista entrante y *FIN* ϵ *N* que es identificado como el nodo de salida sin ninguna arista saliente. La figura 3 muestra un ejemplo de gráfica de flujo.

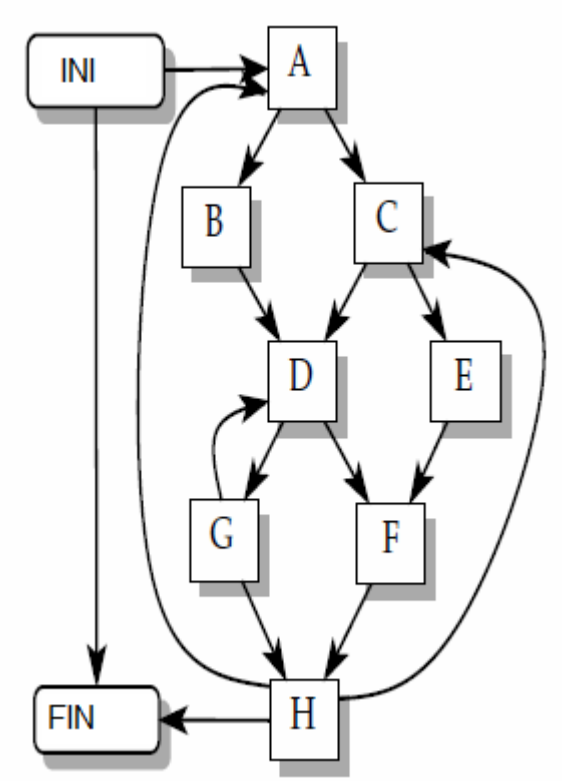


Figura 3 Gráfica de flujo en bloques.

Si $x \rightarrow y \in E$, entonces *x* y *y* son llamados los nodos origen y destino de la arista respectivamente. Un camino (o trayectoria) de longitud *n* es una secuencia de aristas $(x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n)$ donde cada $x_i \rightarrow x_{i+1} \in E$. Una notación alterna es $P: x_0 \xrightarrow{*} x_n$ para representar un camino de longitud cero o más y $P: x_0 \xrightarrow{+} x_n$ para representar un camino de longitud 1 o más. En la definición de gráfica de flujo se asume que para cualquier nodo $x \in E$ existe un camino desde *INI* hacia *x* y un camino desde *x* hasta *FIN*.

Si *S* es un conjunto, la notación |*S*| representa el número de elementos del conjunto. El nodo *x* domina a *y* si y solo si todos los caminos de *INI* hacia *y* pasan a través de *x*. Una forma de expresar esto es *x dom y*. Un nodo *x* domina a *y* de manera estricta si *y* solo si *x dom y* y $x \neq y$, lo cual puede ser expresado como *x stdom y*. Un nodo *x* domina a *y* de manera inmediata, *y = itdom*(*x*), si *y dom x* y no existe un nodo *z*, tal que *y stdom z stdom x*. La relación de dominio en este enfoque es reflexiva y transitiva y puede ser representada por un árbol llamado árbol

dominador. Para cada nodo en un árbol dominador se asocia un número de nivel el cual indica la profundidad del nodo con referencia a la raíz.

Los conceptos anteriores son los que se utilizan para optimizar la generación de código en compiladores [52, 55], se identifica con uno o varios algoritmos si existen ciclos en el flujo de ejecución del programa, posteriormente se identifica si el ciclo tiene aristas y nodos redundantes o bien si puede ser reducido a una gráfica con menos nodos y aristas, lo cual resulta en menos código a generar y ejecutar.

2.4.2 EL CONTROL DE FLUJO EN LOS SISTEMAS HEREDADOS EN COBOL.

En general, los aplicativos de optimización de código en sistemas heredados tienen como característica principal la de eliminar de manera global el código no estructurado [56]. En lo que respecta al problema tratado en este trabajo, el uso de tales algoritmos no es deseable, esto es debido a que el objetivo aquí es el de facilitar el análisis y el mantenimiento del código, mientras que los optimizadores se enfocan en el rendimiento del mismo, en tiempo de ejecución. Otra diferencia con dichos algoritmos es que la salida de su control de flujo normalizado afecta al código no estructurado. Esto no es deseable en nuestro resultado, en la práctica el código que contiene determinadas instrucciones de salto podría estar bien estructurado, por ejemplo debido al uso sistemático de convenciones en el nombre de las variables, mientras que el código estructurado podría no estarlo en buen grado si se le mira desde la perspectiva de mantenimiento. Por lo tanto para resolver este tipo de problemas algunos investigadores han propuesto un enfoque donde se tiene el control completo sobre las partes del código que se desea reestructurar para mejorar la facilidad de mantenimiento y dejar sin modificar otras partes no estructuradas en el sentido práctico. Las técnicas propuestas en este caso son el reconocimiento de patrones y las transformaciones del programa para hacer normalizaciones parciales en el control de flujo del código, donde se detecta que el mismo debe ser modificado.

En *COBOL* se tienen dos tipos de instrucciones de salto de flujo en el programa, los explícitos y los implícitos. Los explícitos son las instrucciones *GO TO*. Los implícitos están de alguna manera ocultos dentro del código, como por ejemplo las bifurcaciones a otros módulos como *CALL* o llamadas al monitor de transacciones o a las macros de acceso al motor de una base de datos, estos tipos de salto pueden ser el manejo de un error o bien una terminación anormal del programa. En los primeros programas que fueron creados en *COBOL*, existen instrucciones *GO TO*, dicha instrucción es usada frecuentemente para simular ciclos que pueden ser implementados con instrucciones estructuradas modernas (en este caso *PERFORM*), también para simular construcciones condicionales, llamadas a procedimientos, instrucciones de terminación del programa y muchas otras construcciones que hoy en día son estándares en los lenguajes de programación, inclusive en los dialectos *COBOL85*. Hay investigaciones que se han enfocado en el reconocimiento de patrones para identificar la simulación de tales construcciones con la instrucción *GO TO*. En este caso dichos patrones pueden ser utilizados para visualizar la simulación "estructurada" de bloques de código, los cuales fueron implementados con instrucciones *GO TO*. La finalidad es presentar la traducción de la forma estructurada de manera amigable a un posible analista que deba interpretar el contenido del código para una hipotética modificación.

La forma de identificar el flujo de un programa de manera automatizada inicia con un análisis sintáctico del programa, el cual entrega como resultado un árbol abstracto de la sintaxis (también llamado código base). Los procesos modulares son transformaciones condicionadas sobre el código base, luego el código base se convierte en un listado del programa [57-59]. El análisis adecuado del control de flujo del programa se facilita si se interpretan los saltos no estructurados como si lo fueran, lo cual se puede lograr aplicando técnicas de reestructuración de código utilizadas en otras investigaciones [60], dichas técnicas se listan a continuación.

- Simular que la palabra *END-IF* se ha agregado para equilibrar una estructura condicional. Todas las instrucciones condicionales que terminan con elementos implícitos como un punto o una palabra *ELSE* de más alto nivel, se recomienda esta simulación para tener más claro el alcance de las instrucciones y hacer el análisis sobre símbolos uniformes.

- La estructura *GO TO* es un salto sin condicionamiento en el flujo del programa. La restructuración de este tipo de ciclos se logra por medio de la identificación de patrones predefinidos los cual se puede lograr durante el análisis sintáctico o bien durante una fase de análisis posterior. El primer patrón a buscar es la simulación de un ciclo de tipo *WHILE*, que no existe en el dialecto *COBOL* en el cual se basará el desarrollo de la solución, inclusive la instrucción análoga *PERFORM* no era muy popular cuando muchos programas de mainframe fueron creados, en este caso un ciclo muchas veces fue implementado con código que sigue el siguiente patrón:

```
PARRAFO-CICLO.
   IF expresión-lógica
     Instrucciones
     GO TO PARRAFO-CICLO
   END-IF.
```

La instrucción equivalente a asumir es:

```
PARRAFO-CICLO.
    PERFORM UNTIL NOT expresión-lógica
        Instrucciones
    END-PERFORM
```

La etiqueta destino del salto *GO TO* es un nombre de párrafo, la expresión lógica es cualquier expresión lógica que devuelve falso/verdadero e instrucciones es al menos una (o varias) instrucción (es).

Este tipo de estrategia queda limitado por el número de patrones conocidos, los cuales pueden ser alimentados al analizador como conocimiento existente. En el caso de que un patrón no sea incluido, el ciclo irregular no sería detectado ni interpretado, en este caso el analizador se limitaría a almacenar instrucciones tal y como aparecen en el programa original. La herramienta automática debe ser diseñada para seguir procesando en el modo normal sin que esta posible eventualidad le haga detener el análisis. Posteriormente se generaría un procedimiento para que el analista alimente un nuevo patrón en un archivo de definiciones y el ciclo sea interpretado correctamente. Enseguida se muestra otra variante de un ciclo implementado por saltos no condicionados y su correspondiente interpretación en código estructurado.

```
PARRAFO-CICLO.
   Instrucciones1
   IF expresión-lógica
     Instrucciones2
   ELSE
     Instrucciones3
     GO TO PARRAFO-CICLO
   END-IF.
   Instrucciones4.

PARRAFO-CICLO.
    Instrucciones1
    PERFORM UNTIL expresión-lógica
        Instrucciones3
        Instrucciones1
    END-PERFORM
    Instrucciones2
    Instrucciones4.
```

Se muestra un tercer ejemplo que utiliza la instrucción *GO TO* para salir de un ciclo estructurado, pudiera parecer extraño pero es la facilidad que da esta peligrosa instrucción.

```
PARRAFO-CICLO.
  Instrucciones1
  PERFORM UNTIL expresión-logica1
    Instrucciones2
    IF expresión-logica2
      Instrucciones3
      GO TO PARRAFO-EXTERNO
    END-IF
    Instrucciones4
  END-PERFORM
  Instrucciones5.
PARRAFO-EXTERNO.
```

En este caso el nivel de abstracción humano debe reestructurar el bloque de código para dejarlo como sigue en las líneas de abajo (Se crea una bandera auxiliar, que sirve únicamente para propósitos de presentar el control de manera entendible a un analista)

```
77  BANDAUX  PIC X(1).
  88 FINAUX       VALUE 'S'.
  88 NO-ES-FINAUX VALUE 'N'.
...
PARRAFO-CICLO.
  Instrucciones1
  SET NO-ES-FINAUX TO TRUE
  PERFORM UNTIL expresion-logica1 OR FINAUX
    Instrucciones2
```

```
    IF expresión-logica2
      Instrucciones3
*     GO TO PARRAFO-EXTERNO  Se elimina este salto y se enciende el interruptor
      SET FINAUX TO TRUE
    ELSE
      Instrucciones4
    END-IF
  END-PERFORM
  IF NO-ES-FINAUX
    Instrucciones5
  END-IF
PARRAFO-EXTERNO.
```

Si en esencia se utiliza este tipo de notación, se puede implementar un segundo analizador sintáctico para el flujo con producciones parecidas a los ejemplos de arriba, las cuales tienen un nivel de detalle por encima de las instrucciones unitarias. Como mejor se pudo observar en el último ejemplo un salto sin condición rompe la continuidad de ejecución, entonces se debe revisar manualmente qué secciones de código son saltadas y a donde regresa (o continua) el control de ejecución. Utilizar la técnica de alimentar a una herramienta automática con patrones predefinidos y clasificados es preferible que complicar el algoritmo del análisis autómata de control de flujo del programa, sobre todo porque se estima que hay pocos patrones dentro de los que cae el manejo de la instrucción *GO TO*, los cuales pueden ser estructurados de manera inmediata y también porque se espera que sean pocos los programas que estén codificados con verdadera lógica de lenguaje ensamblador con saltos que no siguen un patrón legible.

Un proceso similar debería ser implementado para estructuras de condición o también para macros que incluyan saltos de manera implícita, como por ejemplo cuando está incrustado el manejo de errores de *CICS* (monitor de transacciones en la plataforma mainframe), lo anterior porque dicho producto es en realidad un ambiente de ejecución que gobierna y controla en un nivel lógico superior el llamado de módulos *COBOL* y las macros que se comunican con este monitor pueden inducir saltos no visibles pero que funcionan con la filosofía *GO TO*.

- Ignorar símbolos *punto* innecesarios, cuando este símbolo no es un terminador implícito de estructuras, de hecho, dejar los puntos cuando se está interpretando un ciclo no estructurado puede inducir al error, pues el mismo se podría considerar erróneamente como terminador de estructuras, por ejemplo ser interpretado como un *END-IF* o cortar o extender sin control una instrucción *NEXT SENTENCE*.

- Cuando un ciclo es reestructurado, muchas veces es útil asumir que hay bloques vacíos de código para facilitar la interpretación de saltos sin condición.

- La normalización de estructuras condicionales es útil también para interpretar el código, muchas veces es mejor reagrupar el resultado dentro de los bloques *IF* o *ELSE*, lo cual se logra por medio de un reacomodo de los operadores lógicos y relacionales. Muchas veces la interpretación humana es más fácil si la prueba condicional se expresa de manera natural y no en su forma de lógica negada.

Si dentro del análisis del flujo del programa, se detecta que un segmento de control debe ser presentado de manera estructurada, no se debe perder la ubicación de los comentarios, independientemente al control de flujo, la herramienta de análisis de flujo debería incluir a los comentarios como si fueran un componente de la gramática, ya sea para que los mismos de manera íntegra formen parte de la especificación de salida o bien para que se aplique sobre ellos un algoritmo de desecho, este último punto es deseable porque no sería conveniente incluir código comentado sino texto que tiene el código que aporta conocimiento sobre la funcionalidad, el cual muchas veces si está presente.

Una manera sencilla de ubicar los comentarios es por línea de código, como los párrafos con bloques bien definidos de instrucciones con una línea de inicio y otra de fin, los correspondientes comentarios a un párrafo son los que se ubican entre el rango de líneas del párrafo y si existiesen en las líneas inmediatamente anteriores al mismo, estos tienen alta probabilidad de describir al párrafo, por lo que en una primera aproximación el analizador automático los debería considerar.

2.4.3 IDENTIFICACIÓN DE LA ESTRUCTURA DE PROCEDIMIENTOS EN COBOL.

Como ya se mencionó, el principal mecanismo de estructuración de un programa *COBOL* es todavía la instrucción *PERFORM*, normalmente la instrucción es usada de manera directa para definir procedimientos sin parámetros (donde las variables globales son usadas para pasar datos hacia y desde el cuerpo del procedimiento). En su forma más básica la instrucción especifica rangos etiquetados de código (secciones y párrafos) dentro de un programa para ser ejecutados, por ejemplo:

```
PERFORM  SECCIONX THRU SECCIONX-SALIR
```

Especifica que todos los párrafos incluidos dentro de las dos etiquetas que delimitan la sección *SECCIONX* y la correspondiente etiqueta *–SALIR* deben ser ejecutados cuando la línea del programa sea ejecutada. Después de que este rango de código es ejecutado, el control pasa a la línea que se encuentra inmediatamente después de la instrucción *PERFORM*, algo análogo a haber ejecutado *RETURN* en un procedimiento en otros lenguajes. El lenguaje tiene cierto número de variantes para construcciones *PERFORM*, sin embargo las mismas tienen en esencia el mismo comportamiento con ciertas variantes.

A simple vista pudiera parecer que la instrucción es simplemente una manera poco elegante en la sintaxis para invocar procedimientos, si embargo la disposición de la misma puede resultar en construcciones que tengan diferentes secciones que comparten código común. En la figura 4, en el dibujo A, el rango de código C está anidado en el rango B – C y es ejecutado 2 veces. En el dibujo B se observan rangos e instrucciones anidadas, en el dibujo C las instrucciones de dos rangos se traslapan de manera inexacta y finalmente el dibujo D ilustra un programa donde una etiqueta de salida de la instrucción *PERFORM* en significado léxico sucede a la etiqueta de entrada. Todos estos ejemplos son válidos en *COBOL*. Los ejemplos anteriores tienen implícito un problema de semántica, pero la respuesta a cuál ejemplo es inválido semánticamente, es poco clara si se toman como referencia diferentes manuales de referencia del lenguaje.

Según la especificación de *COBOL* de 1965 [61], la instrucción *PERFORM* puede incluir en su rango a ejecutar una o más instrucciones *PERFORM*, cuando esto ocurre, cada instrucción que

queda dentro del bloque se dice que está anidada en la siguiente instrucción externa *PERFORM*, se deben observar las siguientes reglas:

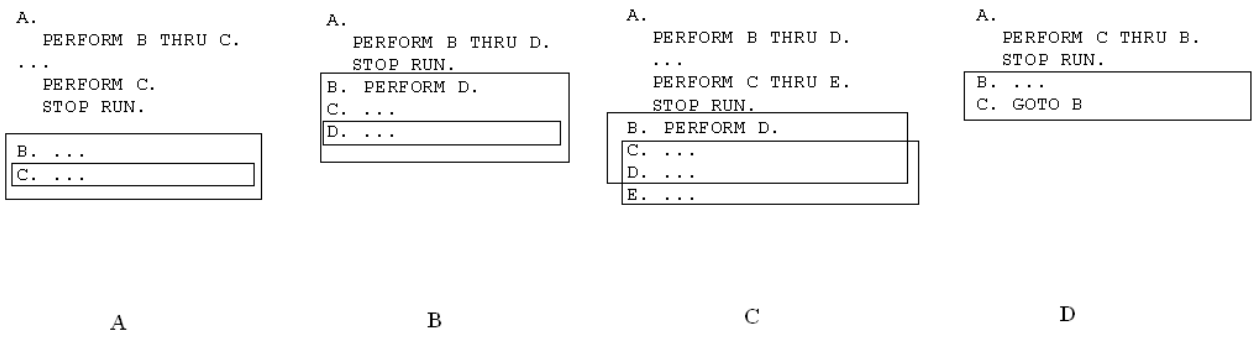


Figura 4 Diferentes construcciones de la instrucción PERFORM.

1) "El rango de cada instrucción debe terminar con un nombre diferente de sección o párrafo."

2) "Una instrucción incrustada debe tener su rango ya sea totalmente dentro o totalmente fuera del rango de cada una de las instrucciones *PERFORM* del rango que la contienen. Esto es una instrucción dentro del rango de una instrucción *PERFORM* no se puede continuar no se puede continuar dentro del rango de la siguiente instrucción *PERFORM* incrustada…"

3) "Los rangos de 2 instrucciones *PERFORM* se pueden traslapar, si se cumple que la segunda instrucción no se encuentre invocada dentro del rango de la primera."

El estándar de *COBOL* de 1985 [62] expandió esta definición como sigue (la numeración corresponde al número presente en el estándar):

7) "El rango de una instrucción *PERFORM* consiste, en nivel lógico, en todas aquellas instrucciones que son ejecutadas como resultado de la ejecución de la instrucción *PERFORM* a través de la ejecución de la transferencia implícita de control y hasta el fin de la misma. El rango incluye todas las instrucciones que se ejecuten como resultado de la ejecución de otras instrucciones *PERFORM* (sus rangos) dentro del rango exterior original. Las instrucciones que están dentro del rango de una instrucción *PERFORM*, no necesariamente deben aparecer de manera consecutiva en el código fuente…"

9) "Los resultados de ejecutar la siguiente secuencia de instrucciones *PERFORM* no está definido y no se han definido que existan condiciones de excepción cuando dichas secuencias sean ejecutadas:

- Una instrucción *PERFORM* está en ejecución, aún no ha terminado y:
- Dentro del mismo rango de tal instrucción, otra instrucción *PERFORM* inicia su ejecución, entonces:
- La ejecución de la segunda instrucción pasa por la salida de la primera…"

90) "Una salida común para instrucciones activas *PERFORM* múltiples está permitida."

Con la actualización en la definición, algunas restricciones se han agregado, sin embargo inclusive la definición de 1985 deja asuntos abiertos, por ejemplo si los PERFORM recursivos

están permitidos [61, 62]. La respuesta relativa a cuál semántica es válida es útil para examinar su implementación. Si se considera de nuevo el dibujo B de la figura 4, la implementación por parte de *IBM®* de esta construcción en el compilador es descrita en [63] (en términos generales no se espera que la misma no sea muy diferente en otros dialectos o versiones), en la figura 5 se muestra dicha implementación con seudo-código.

Cada salida de párrafo de una instrucción *PERFORM*: *e*, tiene una dirección de "continuación" almacenada en un área especial asociada a *e*, continuación *(e)*. Todas las continuaciones son inicializadas al inicio del programa con la dirección del sucesor sintáctico de la salida del párrafo (su sucesor "normal").

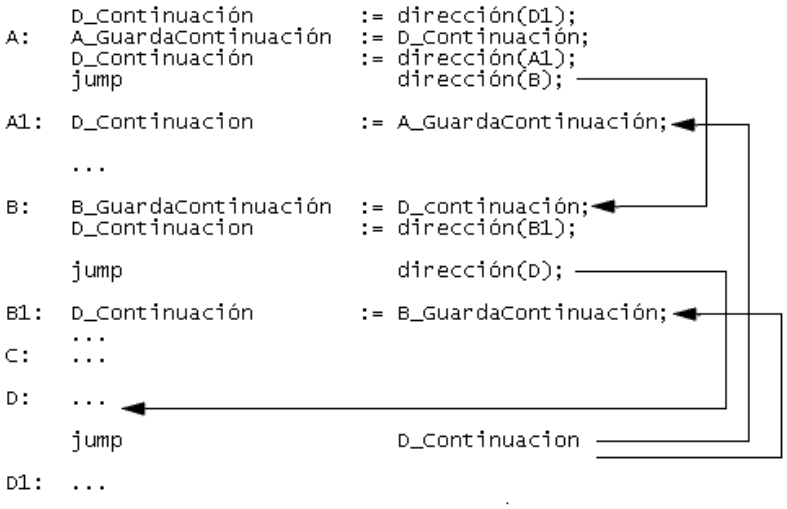


Figura 5 Implementación de instrucción PERFORM por IBM®.

Cuando una instrucción *PERFORM u* se invoca para un cierto rango con salida *e*, el valor actual de continuación (*e*) se guarda primero en el área de almacenamiento asociada a *u*, *GuardaContinuación* (*u*), después, continuación (*e*) se actualiza con la dirección de un bloque de código concluyente que restablece el valor previo de continuación (*e*) a partir de *GuardaContinuación* (*u*); el control del sucesor del bloque de código concluyente es el sucesor sintáctico de la instrucción *PERFORM u*. Finalmente, el control se transfiere a la entrada del rango de *u*. El esquema anterior tiene las siguientes características:

1. La instrucción devuelve las direcciones asociadas con las salidas de los párrafos del rango a ejecutar.
2. Cada instrucción *PERFORM* dentro del rango de otra exterior con salida de párrafo *e*, puede almacenar exactamente una continuación pendiente para *e*.
3. Después de la ejecución de un cierto rango con salida *e*, la continuación almacenada para e, no se actualiza, a no ser que el control del programa alcance nuevamente a *e*.

Como consecuencia directa de 2, en general los *PERFORM* recursivos claramente no son posibles. Y de manera más importante, a causa de 3, el comportamiento adecuado de una secuencia de instrucciones *PERFORM*, es dependiente de que el control del programa alcance las salidas de los rangos en orden inverso a su llamado, esto es que el último rango que entra debe ser el primero que salga (*LIFO*, Último en entrar, Primero en salir). Aparte del problema con la

recursión, la técnica de implementación descrita parece estar de acuerdo con la semántica proporcionada en [62].

Si se considera ahora el ejemplo de la figura 6, en este caso, hay un punto de control en el párrafo *D* con un condicionante (por ejemplo una condición de error), el cual si es verdadero, permitirá que la línea *PERFORM C THRU F* sea ejecutada antes de que la salida del párrafo *E* de la instrucción previa se alcance. Como resultado la continuación de la salida *E* tendrá la dirección de *B*, en lugar de su sucesor "normal", *F*. Esta continuación "activa" puede ser concebida como una "mina". Como resultado, cuando el segundo *PERFORM* sea invocado, si la condición de error ya no se presenta, entonces la ejecución llevará la mina a E, causando que el control otra vez de pase a B, en vez de llegar a F como presumiblemente el programador intentaba.

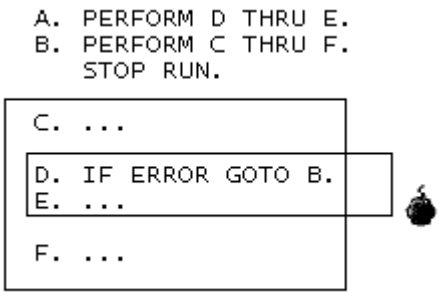


Figura 6 Instrucción PERFORM mal implementada.

De los ejemplos precedentes, debe ser claro que la semántica inusual y las capacidades para compartir código de *PERFORM* la vuelven problemática tanto para las personas que traten de entender el código como para las personas que implementan los algoritmos de funcionalidad que deben de ser correctos y eficientes cuando se aplican a *COBOL*. Existen investigaciones que han propuesto soluciones al problema, la más recurrente es la de reconocer construcciones que siguen ciertos patrones (como los mostrados en los dibujos de la figura 4) y después se sustituyen dichas construcciones por código que realiza una funcionalidad equivalente y es estructurado.

2.4.4 REPRESENTACIÓN DEL CONTROL DE FLUJO.

Se pueden plantear dos representaciones del flujo de control de un programa *COBOL* un diagrama de control de flujo implícito *ICFG* (*Implicit Control Flow Graph*) y otro explícito *ECFG* (*Explicit Control Flow Graph*). El *ICFG* y el *ECFG* comparten el mismo conjunto de vértices (o nodos), pero representan el flujo de control de procedimientos internos, o la transferencia de control ocasionada por instrucciones *PERFORM* en diferentes formas, utilizando diferentes formas de bordes. En ambas representaciones los vértices (o nodos) se dividen en 4 clases.

- Vértices de entrada. Representan inicios de rangos *PERFORM*.
- Vértices de salida. Representan el final de rangos *PERFORM*.
- Vértices *PERFORM*, representan instrucciones *PERFORM*.
- Vértices de Cálculo, representan otros tipos de instrucciones.

Será conveniente tratar al programa principal como si fuera un rango *PERFORM* con vértices de entrada y salida diferenciados S_m y E_m. Un rango *PERFORM* es representado por un *par (s, e)* que consiste en un vértice de entrada *e* y uno de salida *s*. Para cualquier vértice *u* de un *PERFORM* se define el correspondiente rango de instrucciones como *rango (u)*. El conjunto de todos los rangos en el programa *RangoPerform* es:

$$\{\langle s_m, e_m \rangle\} \cup \{rango(u) \mid u \text{ es un vértice } PERFORM\ \}$$

Las aristas que salen de los vértices de cómputo y de entrada a *PERFORM*, forman el flujo de control de procedimientos internos tanto para la *ICFG* como para la *ECFG*. En las dos representaciones un vértice de salida tendrá también una arista saliente, lo que significa una transferencia de control que ocurre de dicho vértice de salida y fluye hacia su sucesor sintáctico si el vértice es alcanzado por el flujo cuando la salida está inactiva. En la *ICFG*, cada vértice *u* tiene un sucesor único, el sucesor sintáctico de la instrucción *PERFORM*. En contraste, en la *ECFG*, *u* tiene una arista saliente de llamada hacia la entrada del vértice de rango *(u)*. El vértice de salida del rango *(u)* tiene una arista de tipo regreso hacia el sucesor sintáctico de *u*. En la figura 7 se muestra un programa de ejemplo y su representación combinada *ICFG*, *ECFG* [64].

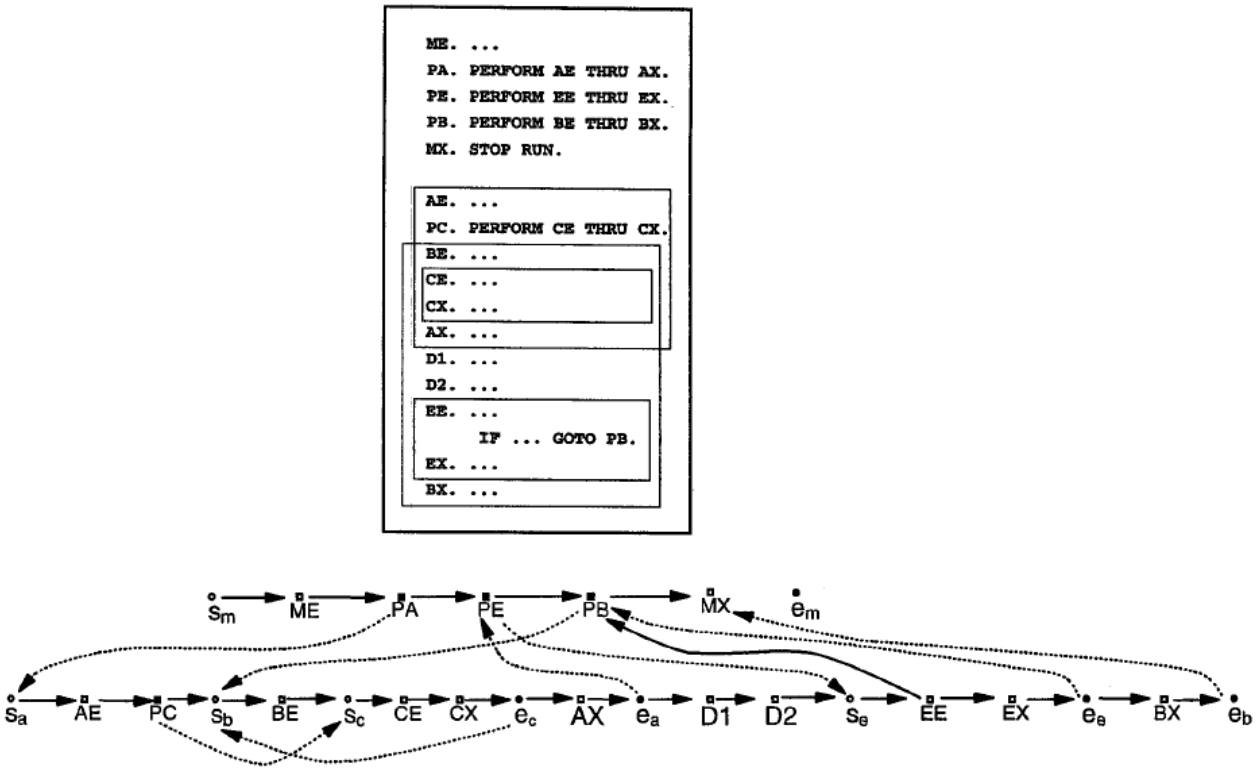


Figura 7 Gráficas ICFG y ECFG para un programa COBOL.

El objetivo primario de un análisis de control de flujo es el de determinar el conjunto de funciones que pueden ser llamadas en cada aplicación (por ejemplo *e (x)*, donde *x* es un parámetro formal de la función), este problema ha sido estudiado de manera amplia en [49, 65, 66].

Si se concibe al programa en términos de caminos lógicos de ejecución, se trata de evitar trabajar con una gráfica con el flujo completo del programa donde todos los lugares con llamadas se ligan

a todas las entradas de funciones y donde todas las salidas de funciones se ligan a sus lugares de retorno. Frecuentemente esto es complementado por medio de contornos [67] cadenas de llamadas [68] o de elementos léxicos del lenguaje [69] para mejorar la precisión de la información obtenida. Una manera de especificar el análisis es mostrar como generar un conjunto de restricciones[70-73] cuya solución más sencilla se calcula utilizando ideas basadas en gráficas. Sin embargo, la mayoría de los artículos sobre Análisis de Control de flujo [49, 65-67] no consideran efectos colaterales, siendo una excepción [74] .

2.4.5 ANÁLISIS DEL FLUJO DE DATOS.

La parte de análisis de flujo de datos en procedimientos internos ignora en la mayoría de los casos las llamadas a módulos externos y trata la llamada a estos y sus retornos como si fueran instrucciones *GO TO* desde un punto de llamada: si se debe analizar un procedimiento con ciertas piezas de información, la información resultante que se procesa a la salida del procedimiento debería ser idealmente propagada hacia atrás hacia el punto de retorno que corresponde a la llamada original en el análisis, como se muestra en la figura 8.

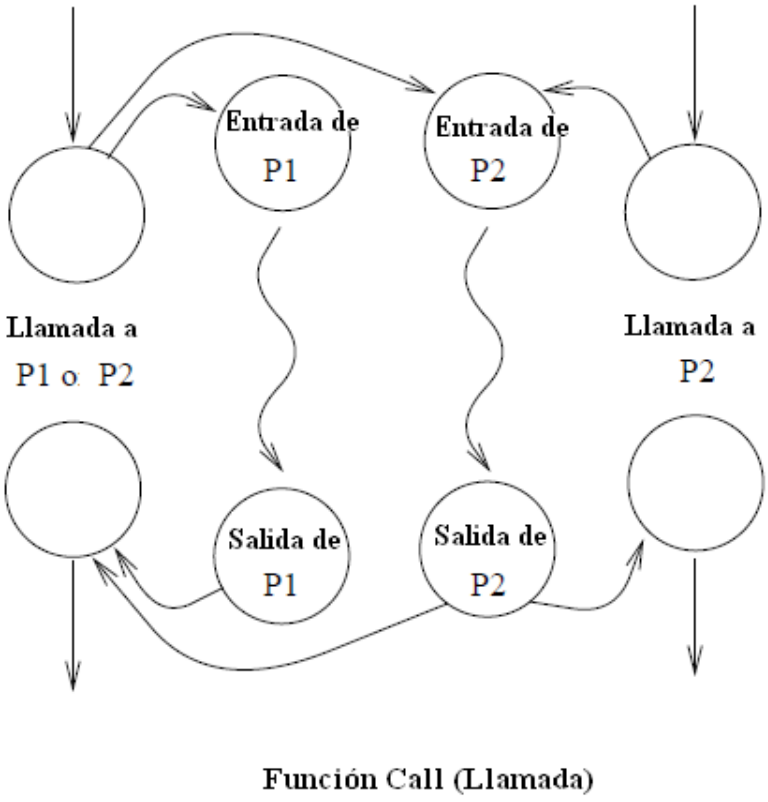


Figura 8 Función de llamada.

En otras palabras, en la vista interior de procedimientos se considera a todos los caminos lógicos de ejecución que sean válidos (y este conjunto de caminos es un lenguaje regular), en tanto que la vista de interconexión de procedimientos considera solamente aquellos caminos que son válidos, aquellos en donde las entradas y salidas de los mismos hacen juego al estilo de la codificación

con paréntesis (y este conjunto de caminos es propiamente un lenguaje libre de contexto). La mayoría de artículos que hablan sobre análisis de flujo de datos [68, 75] no consideran los procedimientos de primera clase y por lo tanto no tienen necesidad de un componente que se asemeje al análisis de control de flujo, una excepción notable a lo anterior es [76].

Una propuesta tiene que ver con el análisis de procedimientos internos para obtener funciones de transferencia para todas las instrucciones *CALL* [68, 75] (y algunas extensiones [77]).

Existen dos entornos de análisis de flujo de datos sensibles al contexto entre procedimientos. Sin pérdida de generalidad se puede considerar únicamente el análisis hacia delante de los problemas de flujo de datos [48]. Dado un programa para analizar, un análisis completo construye un entorno $\langle G, L, F, M, \eta \rangle$, donde

-G = (N, E, ρ) es una gráfica dirigida con un conjunto de nodos *N*, un conjunto de aristas *E* y un nodo de inicio ρ ε *N* (en este caso *G* se considera la gráfica de llamada entre procedimientos internos del programa).
- $\langle L, \leq, \wedge \rangle$ es una semi-retícula (red, *lattice*) de recorrido [48] con orden parcial $\leq$ y recorrido $\wedge$. Por simplicidad se considera que *L* es finito y que tiene un elemento de alto nivel $\top$.
- $F \subseteq \{ f \mid f : L \to L \}$ es una función espacio cerrada bajo composición y recorridos arbitrarios, se asume que F es monótona [48].
- $M : N \to F$ es una asignación de funciones de transferencia hacia los nodos en *G* (sin pérdida de generalidad, se asume que no hay funciones de transferencia hacia las aristas). La función de transferencia para el nodo *n* se denota f_n.
- $\eta \in L$ es la solución al fondo de ρ.

El programa se representa por medio de una gráfica de control de flujo entre procedimientos internos (*ICFG Interprocedural Control Flow Graph*). Cada procedimiento tiene un nodo simple de entrada (el nodo ρ es el nodo de entrada del procedimiento de inicio) y un nodo simple de salida. Cada instrucción *call* se representa por un par de nodos, un nodo de llamado y un nodo de retorno. Hay una arista, desde el nodo de llamada hasta el nodo de entrada del procedimiento llamado, también hay una arista desde el nodo de salida del procedimiento llamado hacia el nodo de retorno en el procedimiento llamador.

Un camino desde el nodo n_1 hasta el nodo n_k es una secuencia de nodos $p = (n_1, ..., n_k)$ tales que $(n_i, n_{i+1}) \in E$. Sea $f_p = f_{n1} \circ f_{n2} \circ ... \circ f_{nk}$, un *camino realizable* es un camino sobre el cual cada procedimiento regresa al sitio de llamada que lo invocó [68, 78, 79], solo tales caminos representan pasos potenciales de secuencias de ejecución. Un *camino realizable del mismo nivel* es un camino realizable cuyo primer y último nodo pertenecen al mismo procedimiento y sobre el cual el número de nodos de llamada es igual al número de nodos de retorno. Tales caminos representan pasos de secuencia de ejecución durante los cuales la pila de llamadas puede crecer de manera muy profunda temporalmente, pero nunca menos profunda que su profundidad original, antes de regresar de manera eventual a su profundidad original [79]. El conjunto de todos los flujos realizables (*Realizable Paths*) de *n* a *m* será denotado como *RP (n, m)*; el conjunto de todos los flujos realizables del mismo nivel de *n* a *m* se denota como *SLRP (n, m), Same-Level Realizable Paths*.

Para cada $n \in N$ la solución de *recorrido de todos los flujos realizables (MORP, Meet Over all Realizable Paths)* en *n* se define como $MORP(n) = \wedge_{p \in RP(\rho,n)} f_p(\eta)$

Análisis no sensible al contexto. Después de construir $\langle G, L, F, M, \eta \rangle$, un análisis de programa calcula una solución $S : N \to L$; la solución al flujo de datos en el fondo del nodo *n*, será denotada como S_n. La solución es *confiable* si y solo si $S_n \leq MORP(n)$ para cada nodo *n*, un *análisis confiable* calcula la solución confiable para cada entrada válida del programa. Tradicionalmente se construye un sistema de ecuaciones y después se resuelve utilizando iteraciones de punto fijo. En el caso más simple, un análisis no sensible al contexto construye un sistema de ecuaciones de la forma:

$$S_\rho = \eta, \quad S_n = \bigwedge_{m \in Pred(n)} f_n(S_m)$$

Donde *Pred (n)* es el conjunto de nodos predecesores para *n*. La solución inicial tiene a $S_\rho = \eta$ y a $S_n = \top$ para cada $n \neq \rho$; la solución final es un punto fijo del sistema y es también una aproximación confiable de la solución MORP.

Análisis sensible al contexto. El problema con la propuesta de arriba es que la información se propaga desde la salida de un procedimiento hacia todos sus llamadores. El análisis sensible al contexto se usa para lidiar con esta potencial fuente de imprecisión. Una propuesta es la de propagar elementos de L junto con etiquetas las cuales aproximan el contexto de la llamada al procedimiento. En la salida del procedimiento, estas etiquetas se consultan en orden de propagar la información hacia atrás solamente hacia los sitios en los cuales el contexto de la llamada existe. La "propuesta funcional" de sensibilidad al contexto [68] utiliza una nueva retícula que es la función espacio de las funciones que trazan un mapa de L a L. Intuitivamente, si la solución en el nodo *n* es un mapa $h_n : L \to L$, entonces para cada $x \in L$ se puede utilizar $h_n(x)$ para aproximar $f_p(x)$ para cada camino $p \in SLRP(e, n)$, donde *e* es un nodo de entrada del procedimiento que contiene a *n*. En otras palabras, $h_n(x)$ aproxima la parte de la solución en n que ocurre durante el contexto de llamada *x* en *e*. Si el contexto de llamada *x* nunca ocurre en *e*, entonces $h_n(x) = \top$. La solución al problema original se obtiene como $\wedge_{x \in L} h_n(x)$.

En general esta propuesta requiere una representación compacta de las funciones y composiciones y recorridos funcionales explícitos, los cuales son en general no viables. Cuando *L* es finito, una versión factible del análisis se puede diseñar [68]. La figura 9 representa una descripción simplificada de esta versión. $H[n, x]$ contiene el valor actual de $h_n(x)$; la lista de tareas contiene pares (*n, x*) para los cuales $H[n, x]$ ha cambiado y tiene que ser propaganda a los sucesores (*Sucr*) de *n*. Si *n* es el nodo de llamada, en la línea 7 del diagrama, el valor de $H[n, x]$ se propaga hacia el nodo de entrada del procedimiento llamado. Si *n* es un nodo de salida, el valor de $H[n, x]$ se propaga únicamente a los nodos de retorno a los cuales los nodos de llamada *x* correspondientes ocurren (líneas 8-10).

Para entornos distributivos [48], este algoritmo termina con la solución MORP; para entornos monótonos no distributivos, produce una aproximación confiable de la solución MORP [68], Cuando una retícula es la alimentadora de algún conjunto básico finito *D* de hechos del flujo de datos (por ejemplo el conjunto de todos los potenciales alias o el conjunto de todas las

definiciones de variables), el algoritmo puede ser modificado para propagar elementos de *D*, en lugar de los elementos de 2^D. Para entornos distributivos, esta propuesta produce una solución precisa [79]. Para entornos monótonos no distributivos, al restringir el contexto a un conjunto de un solo elemento necesariamente se introduce alguna aproximación; el análisis de programas de alias de apuntadores cae en esta categoría [78].

Entrada $\langle G, L, F, M, \eta \rangle$; L es finita
Salida S: arreglo [N] de L
Declarar H : arreglo [N, L] de L; valores iniciales $\top$
W : lista de *(n, x)*, $n \in N$, $x \in L$; inicialmente vacía
[01] $H[\rho, n] := \eta$; $W := \{(\rho, \eta)\}$;
[02] Mientras W ≠0 repetir ciclo:
[03] borrar *(n, x)* de W; $y := H[n, x]$;
[04] Si *n* no es un nodo *call* o un nodo de salida, entonces
[05] Para cada $m \in Sucr(n)$ *propagar* $(m, x, f_m(y))$;
[06] Si *n* es un nodo *call*, entonces
[07] $e := Entrada\_Llamada(n)$; $propagar(e, y, f_e(y))$;
[08] Si *n* es un nodo de salida, entonces
[09] Para cada $r \in Sucr(n)$ y $l \in L$ ejecutar:
[10] Si $H[nodo\_call(r), l] = x$ Entonces *propagar (r, l, $f_r(y)$)*;
[11] Para cada $n \in N$ ejecutar
[12] $S[n] := \wedge_{l \in L} H[n, l]$;
[13] Procedimiento *propagar (n, x, y)*
[14] $H[n, x] := H[x, n] \wedge y$; Si *H[n, x]* cambió, entonces *agregar (n, x)* a *W*

Figura 9 Algoritmo del análisis de flujo de datos de un programa sensible al contexto en seudo-código.

2.5 EXTRACCIÓN DE FUNCIONALIDAD DE APLICATIVOS.

Los conceptos de Minería de Conocimiento en reglas de negocio heredadas pueden sonar abstractos o amenazadores para las personas, pero los mismos son muy importantes. Cualquier organización con una base importante de sistemas heredados instalados, debe considerar el uso de la extracción de reglas de negocio, así mismo considerar el reciclado o reuso de tales reglas de negocio como el componente clave de iniciativas de proyectos de tecnología de información.

2.5.1 CRITERIOS DE EXTRACCIÓN DE REGLAS DE NEGOCIO.

Como ya se explicó en el capítulo I, los sistemas heredados tienen como una de sus principales características su antigüedad y la carencia de la documentación de diseño o que la misma no está actualizada. Ahora bien, ¿qué es una regla de negocio?, según el Grupo de Administración de Objetos (*OMG*), las reglas son declaraciones de alguna política o bien de condiciones que se deben cumplir necesariamente, las reglas de negocio son las reglas que gobiernan la manera en que un negocio opera. Para el propósito de la presente investigación aplicativa, una regla de negocio es una combinación de lógica imperativa y condicional que cambia el estado de un elemento de datos. Como ejemplo ilustrativo una regla puede ser algo similar a:

"Pagar una factura de proveedor solamente si la misma ha sido aprobada".

La extracción de reglas de negocio identifica y captura cualquier combinación de instrucciones condicionales o imperativas las cuales son aisladas para facilitar el entendimiento y la reutilización del conocimiento contenido dentro de los sistemas informáticos heredados. Las reglas pueden invocar a otras reglas y ser representadas dentro de una estructura jerárquica de reglas. La extracción de reglas de negocio documenta cuales reglas están implementadas o son ejecutadas por un conjunto de programas y sistemas informáticos. La consolidación y reutilización de las reglas extraídas puede extenderse al hacer dichas reglas disponibles a los equipos de diseño y desarrollo. La extracción de reglas de negocio requiere una evaluación preliminar de alto nivel del ambiente existente para que los analistas puedan segmentar dichos sistemas como preparación para la extracción de reglas. Los analistas deben segmentar un inventario de un subconjunto de componentes para ser alimentados en el proceso de extracción. Al nivel de arquitectura los analistas deberían determinar cómo se comparten los datos entre sistemas para refinar los sistemas a incluir en un proyecto de esta naturaleza, por ejemplo cualquier sistema que actualice una base de datos compartida o un archivo maestro. Después los analistas deberían refinar el conjunto de componentes basados en las funciones que los mismos desempeñan. El propósito es el de reducir un conjunto relativamente grande de componentes en un conjunto más manejable.

Este análisis preliminar debe identificar alguna debilidad en el sistema a analizar, la cual se recomienda corregir antes de ejecutar la extracción de reglas. La eliminación de fallas en el código, tales como recursividad, reestructuración, lógica demasiado compleja o la división de módulos demasiado grandes y complejos simplifican la extracción de reglas, lo mismo aplica para la reestructuración de los nombres de los datos, lo cual ayuda a determinar cuáles datos impactan las reglas.

Las reglas de negocio están sujetas a cambios cuando el mercado y la tecnología cambian. Las reglas de negocio usualmente son escritas en documentos (por ejemplo manuales de políticas) antes de ser codificadas en el software. Con el paso del tiempo, cuando las reglas evolucionan y se agregan nuevas funcionalidades, tanto el software como los documentos se vuelven demasiado grandes y difíciles de entender y mantener. Los analistas y programadores del software que inicialmente ayudaron a transformar las reglas de negocio expresadas en texto en su presentación correspondiente en el software, gradualmente tienden a concentrarse únicamente en el software y gradualmente pierden confianza en los documentos de texto. Esto crea un problema de mantenimiento.

En grandes sistemas heredados en *COBOL*, como los que se describieron en el capítulo I, cuando ocurre una actualización de alguna política de negocio, las personas encargadas de dar mantenimiento al software, deben entender las reglas de negocio relevantes que podrían

implementar dicha actualización en particular. Puesto que los analistas usualmente no trabajan con documentos de texto debido a que los consideran poco entendibles o confiables, entonces las actualizaciones de vuelven muy difíciles. Es aquí donde se visualiza que los analistas requieren de una aproximación automática que extraiga las reglas de negocio del código. Con esto surge la pregunta: ¿Es posible extraer las reglas de negocio incrustadas en el código heredado de manera que ellas pudieran ser controladas en los documentos de negocio? Esto es un asunto crítico e importante porque una vez que las reglas de negocio son conocidas, es fácil para una organización desarrollar nuevas reglas o modificar las existentes. Además, las reglas de negocio contienen con frecuencia información útil de propietario que podría no estar disponible en cualquier otra forma debido a una gran variedad de razones, tal como la partida de personal clave. Cuando se habla de extracción de reglas de negocio, se pueden resumir los siguientes requerimientos basado en diferentes experiencias profesionales y en diversos artículos leídos [80]

Criterio I. *Representación fiel*. Cualquier regla de negocio extraída debe reflejar el estado actual del software. Este es el criterio más importante y si este criterio entra en conflicto con algún otro este criterio debe prevalecer. También se debería tener una representación estructurada de las reglas de negocio, haciendo una transformación de las mismas si estas aparecen de manera poco estructurada dentro del código.

Criterio II. *Representación múltiple y abstracción jerárquica*. Diferentes personas requieren de diferentes representaciones de las reglas de negocio. Los gerentes de mantenimiento de software podrían requerir de una vista muy global del software, tal como cuáles reglas de negocio están implementadas en qué módulos. Sin embargo un programador de mantenimiento es el responsable de agregar, borrar o modificar en tiempo real el código que implementa las reglas de negocio. Por eso un programador típicamente desea ver expresadas las reglas de negocio como segmentos de código, mientras que los gerentes prefieren tablas de decisión, árboles de decisión o gráficas estructuradas.

Las reglas de negocio deberían ser representadas de manera jerárquica. Las reglas de negocio son frecuentemente complejas porque deben cumplir varias restricciones, tales como consideraciones legales, de mercadeo y tecnológicas. También, las reglas de negocio cambian con el tiempo. La mayoría de programas heredados contienen almacenadas reglas de negocio viejas, siendo la principal razón de esto la liberación de nuevas versiones. Un cliente de un programa existente pudiera continuar siendo atendido con una versión antigua del software usando las reglas anteriores en vez de las nuevas, también las reglas de negocio anteriores pudieran no haber sido removidas del código heredado inclusive si las reglas no son requeridas. Las reglas de negocio pudieran ser extensas (incluso miles de páginas) detalladas e implantadas en múltiples procesos. Por eso podría ser extremadamente difícil rastrear las reglas sin alguna forma de abstracción o descomposición.

Criterio III. *Políticas de negocio específicas por dominio*. Las reglas de negocio son diferentes de los comentarios o documentación en los programas. La documentación en los programas incluye tanto conocimiento de la aplicación como conocimiento de programación. Sin embargo la mayoría del personal de mantenimiento de software prefiere que las reglas de negocio sean expresadas en el vocabulario del dominio al que corresponden, o con exactitud, esto por medio del uso de variables que representen un concepto en el dominio de la aplicación, por ejemplo una variable tal como *I* o *LOOP-INDEX* es usada frecuentemente como un contador en un ciclo del programa, pero la variable *INVENTORY-RECORD-IN* podría representar un concepto de dominio porque es posible que represente registros del inventario actual utilizados en la generación de

reportes de facturación. Una herramienta que ayude a extraer las reglas debería proporcionar los medios para identificar conceptos importantes incluidos datos y algoritmos relacionados a las reglas de negocio inclusive si estos vinieran de una entidad o elemento externo.

Criterio IV. *Automatización asistida por el factor humano*. Para agilizar el proceso de mantenimiento de software la extracción de reglas de negocio debería ser automatizada tanto como sea posible. Sin embargo en la práctica ha sido demostrado muchas veces que es extremadamente difícil si no imposible concebir un programa o herramienta completamente automática que extraiga las reglas de negocio. Dicha herramienta debería involucrar intervención humana porque las reglas de negocio son complicadas y evolucionan con el software que los rodea con el paso de los años. Inclusive si la extracción automática fuese posible, las reglas extraídas pudiesen no ser adecuadas para ser usadas en el mantenimiento del software. Por eso es preferible que el mantenimiento se auxilie de herramientas interactivas que permitan la extracción de las reglas, simplificar su representación y permitan una relación con el código, en lugar de proporcionar una caja negra que extrae las reglas de negocio automáticamente. El sistema debería también sugerir pistas para ayudar al programador o analista durante el proceso interactivo de extracción.

Criterio V. *Herramienta de mantenimiento*. Las reglas de negocio extraídas deberían ser útiles para ayudar en otras tareas de mantenimiento. Por eso una herramienta de extracción de reglas de negocio debería ser parte de un paquete o conjunto de herramientas de mantenimiento de software. La herramienta debería proporcionar el mapeo de cualquier regla de negocio con su correspondiente implementación en el código y viceversa. Esta capacidad permitiría a los programadores y analistas concentrar su atención únicamente en los segmentos de código o funciones del software que son relevantes para una tarea de mantenimiento en particular.

Según algunas investigaciones, en general una regla de negocio puede ser definida como una función, restricción o transformación de las entradas de una aplicación en sus salidas. Esto es así porque eventualmente una regla de negocio debe producir una salida para los usuarios adecuados y debe tomar algunas entradas de sus usuarios para iniciar el procesamiento.

Formalmente una regla de negocio *R* puede ser expresada como un segmento de programa *F* que transforma un conjunto de variables de entrada *I* en un conjunto de variables de salida *O*.

$$O = F (I)$$

Por ejemplo *Utilidades* = (*Ingresos* - *Gastos*) es una regla de negocio simple donde *Utilidades* es la salida, *Ingresos* y *Gastos* son entradas y la resta es la función implementada por el segmento de programa *F*. En aplicaciones de negocio los datos juegan un papel predominante, por eso una regla de negocio es usualmente centrada alrededor de ciertos datos ya sea de entrada o de salida. Por ejemplo la regla de negocio simple de arriba puede ser ligada tanto a la variable de salida *Utilidades* o a cualquiera de las dos variables de entrada o a cualquier combinación de esos datos en la documentación del sistema.

2.5.2 CLASIFICACIÓN DE VARIABLES.

Basados en los criterios anteriores algunos autores recomiendan usar una aproximación centrada en datos para extraer las reglas de negocio, entonces para ser compatible con la definición de regla de negocio, se deberían identificar primero datos de entrada/salida ó ambos para iniciar la *BRE* (*Business Rule Extraction*). Entonces, por ejemplo si se desea extraer reglas de negocio que tengan que ver con llamadas gratuitas en un programa que calcula facturas de llamadas telefónicas primero se deberían identificar las variables que representen cargos de números gratuitos y después extraer segmentos de código que de manera directa o indirecta manipulen esas variables y con esto extraer las reglas de negocio relacionadas a los cargos de números gratuitos. Este primer paso es para identificar las variables importantes del código las cuales pueden ser utilizadas después para expresar reglas de negocio

En una instalación típica se pueden encontrar miles incluso cientos de miles de variables en el código. Para esto se deben aplicar técnicas de clasificación de variables. Cuando ya se han identificado las variables de dominio es necesario seleccionar un subconjunto de esas variables relacionadas a un conjunto específico de reglas y después extraer los segmentos de código que usan o son afectados por las variables de dominio seleccionadas. Las reglas de negocio extraídas deberían expresarse únicamente en términos de las variables de domino identificadas. Para esto es útil valerse de algunas reglas heurísticas.

La extracción de segmentos de código relevante plantea problemas como el criterio de extracción, las variables a contener, el punto de inicio de la extracción. Una vez que el código que contiene la regla de negocio haya sido aislado viene el problema de la presentación a diferentes usuarios esto por medio del uso de diferentes esquemas jerárquicos. La clasificación de variables involucra la categorización de variables en diferentes tipos, por ejemplo variables de dominio, de programa, constantes o globales, esta clasificación es de utilidad debido a que muchos programas manejan demasiadas variables y conocer el papel que juega cada variable en el programa, reduce el tiempo que lleva entender el código. Una clasificación útil de variables podría ser la siguiente:

Datos de dominio: Son variables inherentes al dominio de la aplicación y no dependen de una implementación específica de algún programa.

Datos de programa: Variables dentro del programa que no son variables de dominio, sino datos del programa, por ejemplo una variable definida como elemento *COBOL* de nivel 77 y usada dentro de un procedimiento de análisis.

Datos locales: Datos con alcance de un solo elemento *COBOL*.

Datos globales: Las variables globales son referenciadas en múltiples unidades de compilación *COBOL* o elementos que no son *COBOL* en múltiples procedimientos. Esta clasificación es muy importante para programas escritos en *COBOL* donde hay muy pocos mecanismos para limitar el acceso a una variable.

Datos de entrada: Las variables de entrada son los datos involucrados en eventos de entrada, como la lectura de un archivo o de una base de datos.

Datos de salida: Son variables involucradas en eventos de salida tales como la actualización de un archivo o una base de datos.

Datos Constantes: Son variables que no cambian de valor durante la ejecución del programa.

Datos de control: Son variables usadas en el cálculo de predicados. Esta clasificación le da al programador una idea en cuanto a los objetivos del programa respecto de los datos.

2.5.3 EXTRACCIÓN DE FUNCIONALIDAD EN COBOL POR MEDIO DE REGLAS HEURÍSTICAS.

Una vez que se tiene una idea de cómo clasificar las variables en diferentes categorías, es necesario identificar a las variables de dominio relevantes entre todas las variables de dominio encontradas en el código. De acuerdo a Huang y Tsai [80] la guía pueden ser reglas heurísticas como las que a continuación se listan:

Regla Heurística 1.
Se seleccionan las variables generales de entrada y salida como variables de dominio. Es decir el sistema puede ser visto en general como una caja negra que mapea sus entradas como salidas.

Regla Heurística 2.
Se seleccionan las variables de entrada y salida de cada procedimiento como variables de dominio, dado que los procedimientos son los componentes funcionales de un sistema. En un software *COBOL* a diferencia de otros lenguajes de programación los procedimientos pueden ser implementados como instrucciones, frases, párrafos, secciones y programas o partes de ellos.

A continuación se ilustra un ejemplo de selección de variables utilizando un ejemplo de Grauer [81]. Es un programa que calcula las cuotas de colegiatura en una universidad. De acuerdo a la regla heurística 1, se deberían utilizar todas las variables de entrada y salida del sistema como punto de inicio.

En este ejemplo las entradas del sistema son *EST_HORAS_MATERIAS*, *EST_BECA*, *IND_CUOTA_INSCRIPCION* y *IND_CUOTA_ACTIVIDADES* y la salida es *TOTAL_IND_FACTURA*. La identificación de estas variables puede ser realizada de forma manual o utilizando algoritmos automatizados [82]. Se puede iniciar la extracción de las reglas de negocio en las instrucciones de entrada o de salida del programa. En este ejemplo en particular se realizó el trazado desde la instrucción de salida y hacia atrás, al final del proceso se obtiene la siguiente relación, donde el asterisco (*) indica multiplicación y tiene precedencia sobre la adición y la substracción.

```
TOTAL_IND_FACTURA = Σ (EST_HORAS_MATERIAS * 127.50 +
IND_CUOTA_INSCRIPCION + IND_CUOTA_ACTIVIDADES - EST_BECA)
```

La regla de negocio extraída nos dice que la facturación total de las colegiaturas es la suma de las facturas de todos los estudiantes. Como la factura de un estudiante es la suma individual de sus cuotas de colegiatura, inscripción y actividades menos cualquier beca disponible. Una colegiatura individual es determinada al multiplicar el número de horas tomadas en sus materias por $127.50. La regla heurística 2 sugiere seleccionar las variables de entrada y salida de cada procedimiento como variables de dominio. Si en este ejemplo, las instrucciones *COBOL* se tratan como procedimientos, entonces las variables intermedias *IND_FACTURA* y *IND_COLEGIATURA*

pueden ser referidas como variables de dominio. Entonces la regla de negocio puede ser representada jerárquicamente como:

```
TOTAL_IND_FACTURA = Σ (IND_FACTURA);
IND_FACTURA = IND_COLEGIATURA + IND_CUOTA_INSCRIPCION +
              IND_CUOTA_ACTIVIDADES - EST_BECA
IND_COLEGIATURA = EST_HORAS_MATERIAS * 127.50
```

Una vez que se conoce como seleccionar variables de dominio para extraer las reglas de negocio y como seleccionar un punto de inicio para rastrear segmentos de código, se puede ya sea rastrear el código manualmente o bien hacer uso de una herramienta de software para recuperar segmentos de código relevantes. En este caso hay una técnica llamada segmentación de código. La segmentación de código y sus extensiones [83-88] son técnicas para descomponer programas por medio del análisis de su flujo y control de datos. Al iniciar en un determinado punto del programa, la segmentación de programas automáticamente recupera código relevante utilizando dependencias de datos y/o control de flujo. Las técnicas de segmentación de programas por sí mismas no son capaces de obtener reglas de negocio, ya que la segmentación es algo similar a una máquina de búsqueda que no puede ejecutarse sin un dato de entrada que la alimente, en este caso el criterio de segmentación. Por eso la extracción de código debería ser dividida en dos pasos: (1) identificación del criterio de segmentación y (2) la ejecución de la segmentación del programa. También muchas veces es necesario reducir el espacio de búsqueda en el proceso.

Tradicionalmente un criterio de segmentación es un par *(i, V)* donde *i* es una instrucción del programa en *P* y *V* es un conjunto de variables que están relacionadas o conciernen a la instrucción *i*. Pero el par *(i, P)* no es suficiente para ejecutar la segmentación porque no proporciona información acerca del espacio de búsqueda. Por lo tanto la segmentación a veces produce segmentos muy grandes para poder ser considerados útiles [88]. Además, una regla de negocio puede extenderse solamente en una porción particular del programa, por ejemplo desde un procedimiento de aceptación de algunas entradas hasta la generación de algunas salidas. Sin restricciones, la segmentación puede tratar de buscar relaciones en el programa entero y producir segmentos que contengan código innecesario o irrelevante.

La identificación del criterio de segmentación consiste en tres partes: (1) un conjunto de variables *V*; (1) la instrucción *i*; y (3) la restricción *C*. El conjunto de variables *V* es seleccionado previamente para contener variables de dominio. La instrucción del programa *i* se puede seleccionar usando tres reglas heurísticas.

Regla Heurística 3.
Las instrucciones de entrada y de salida del programa son buenas candidatas para realizar la extracción de reglas de negocio (*BRE*), ya que estas indican usualmente el inicio y el fin de cierto procesamiento. Al rastrear desde una instrucción de entrada (o de salida) hacia adelante (o hacia atrás) se tendrá la capacidad de recuperar segmentos de código relevantes. Por ejemplo si se considera la siguiente instrucción *WRITE*:

```
WRITE REP-LINEAIMPRESION FROM LINEA1-PRINCIPAL AFTER ADVANCING 1
LINES
```

Se puede iniciar la segmentación hacia atrás en la instrucción de arriba. El segmento que se obtenga es una regla de negocio que gobierna a *LINEA1-PRINCIPAL*. Cuando el proceso dirigido

por el la interacción entre las variables de entrada y de salida es complejo, se pueden utilizar algunas variables intermedias para simplificar las reglas de negocio. Es importante notar que las variables intermedias deberían ser variables de dominio solamente. De otra forma la expresión de la regla de negocio se puede volver complicada. Debido a esto las siguientes reglas heurísticas aplican para seleccionar variables intermedias.

Regla Heurística 4.
Un lugar que es un centro de distribución en el programa es un candidato a punto de inicio para hacer segmentación hacia adelante. Un punto de entrega en el programa delega datos de entrada de diferentes tipos a los procedimientos correspondientes. Dada una entrada, el procedimiento de entrega correspondiente lleva las reglas de negocio que regulan el procesamiento de entradas de diferente tipo. La figura 10 ilustra la situación.

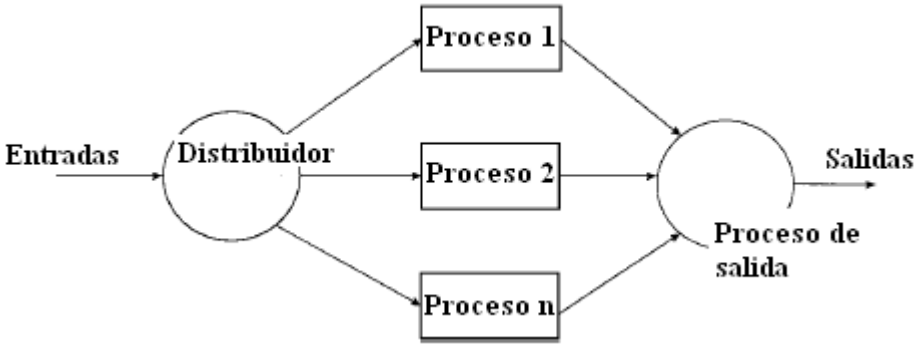


Figura 10 Punto de bifurcación (distribución) en un programa.

Por ejemplo si se considera el siguiente procedimiento extraído de un programa *COBOL*. El mismo envía registros de entrada a las secciones *E01*, *E11*, *E02* ó *E04* en término de los valores de la variable *TARJETA-NUM* en el registro. Claramente si se inicia la segmentación hacia adelante a partir de la línea *055000*, *055200*, *055400* ó *055800*, se tiene la capacidad de separar el programa en 4 partes, donde cada parte soporta solo un tipo de registro.

```
054600 PROCESA-UN-REGISTRO SECTION.
054700 PROCESA-UN-REGISTRO-010.
054800  EVALUATE TARJETA-NUM
054900   WHEN 'E01'
055000     PERFORM E01-PROCEDIMIENTO
055100   WHEN 'E11'
055200     PERFORM E11-PROCEDIMIENTO
055300   WHEN 'E02'
055400     PERFORM E02-PROCEDIMIENTO
055500   WHEN 'E03'
055600     CONTINUE
055700   WHEN 'E04'
055800     PERFORM E04-PROCEDIMIENTO
055900   WHEN 'E05'
056000     CONTINUE
056100 END-EVALUATE.
056200 PERFORM LEER-ENTRADA.
056300 PROCESA-UN-REGISTRO-SALIR.
```

```
056400 EXIT.
```

Regla Heurística 5.
Un punto de finalización en un procedimiento es un candidato para punto de inicio para segmentar hacia atrás. Algunos procedimientos están dedicados a producir salidas. Puede haber múltiples ocurrencias de eventos de salida para la misma variable de salida. La secuencia de esas ocurrencias de salida es una regla de negocio que muestra cómo organizar las salidas para formar un archivo, un reporte, una tabla o una presentación de una pantalla de salida. Empezando desde el final del procedimiento, un trazado hacia atrás a lo largo de todas las posibles rutas lógicas de ejecución puede identificar todos los posibles eventos de salida con respecto a una variable de salida dada y de este modo formar un segmento. Tal segmento es una regla de negocio. Las restricciones en el espacio de segmentación son de 2 tipos: en profundidad y en frontera. La limitación en profundidad es una restricción que limita el espacio de búsqueda por medio de distancia en dependencia. Un límite en profundidad puede reducir de manera efectiva el espacio de búsqueda en una gráfica de dependencia y por lo tanto reducir el tamaño del segmento a obtener.

La segmentación de frontera es un conjunto de puntos en un programa que especifican una región del mismo que debe ser segmentada. Estos puntos del programa son implementados como un conjunto de marcas en los nodos de una gráfica de control de flujo (*CFG*), tal como una gráfica de llamadas a módulos externos. El entendimiento del programa es incremental. Un programador usualmente se concentra en una porción particular del código a la vez. El programador puede entender algunas de las funciones con esta porción del código y dichas funciones se pueden tratar como componentes intrínsecos que no necesitan ser analizados otra vez y por eso pueden ser puestos fuera de la frontera del segmento a obtener.

Elegir el algoritmo de segmentación depende del criterio de segmentación elegido y de la estructura del código. Para esto se sugieren las siguientes otras reglas heurísticas.

Regla Heurística 6.
Escoger segmentación hacia adelante si la variable del criterio de segmentación es una variable de entrada.

Regla Heurística 7.
Escoger segmentación hacia atrás si la variable del criterio de segmentación es una variable de salida. Una entrada es un punto de inicio de dependencias y por lo tanto escoger segmentación hacia atrás no tiene sentido. De manera similar desde que una salida es un punto final de las dependencias, la segmentación hacia adelante desde una variable de salida es de manera muy probable inútil. Una variable de entrada puede ser utilizada para producir diferentes salidas.

Regla Heurística 8.
Escoger la segmentación hacia adelante si un punto de inicio de un análisis es un punto de distribución. Desde que un punto de distribución hace descender las entradas hacia diferentes unidades de procesamiento, basado en el tipo de las entradas, la segmentación hacia adelante puede separar de manera eficiente el procesamiento asociado con un tipo particular de la entrada.

No es suficiente tener únicamente algoritmos de segmentación hacia adelante y hacia atrás porque las reglas de negocio son a veces muy complejas. Puede ser necesaria una segmentación

interactiva e iterativa, además de la segmentación de una sola pasada, para obtener un procesamiento cíclico intangible, lo cual se muestra en la figura 11.

La segmentación recursiva [88] permite que segmentos obtenidos previamente sean analizados más a detalle y descompuestos. La segmentación tradicional de programas puede dividir programas únicamente, pero los segmentos consistentes con las reglas heurísticas mencionadas anteriormente no necesariamente son programas. La segmentación recursiva facilita el entendimiento de programas de manera interactiva e iterativa. Por ejemplo la segmentación recursiva puede ser utilizada para ordenar las relaciones complejas entre múltiples entradas y múltiples salidas.

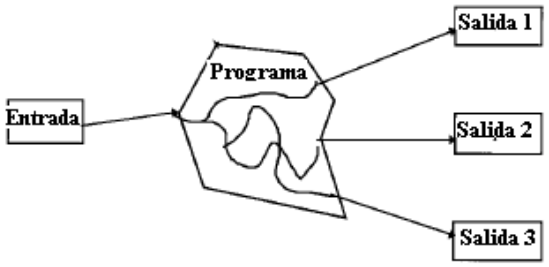
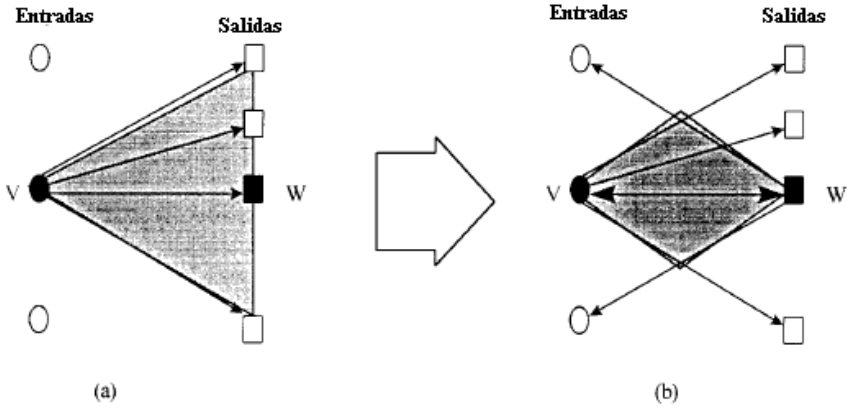


Figura 11 Funcionalidad compleja mezclada en un programa.

Usualmente una entrada contribuye a generar más de una salida y una entrada depende de una o más entradas. La segmentación recursiva puede ser utilizada de tal manera que primero se hace una segmentación hacia delante empezando desde un punto para encontrar todo el código afectado por la entrada y luego escoger una salida para ejecutar segmentación hacia atrás en el segmento. El segmento resultante es el código que se relaciona a una específica entrada y una salida específica. La figura 12 ilustra lo anterior.



(a) El área sobreada es la segmentación hacia adelante desde la entrada V

(b) El área sombreada es el segmento recursivo producido por la segmentación hacia atrás del segmento previo, utilizando la salida W como punto de inicio

Figura 12 Segmentación recursiva de un programa.

La segmentación tradicional de programas extrae las reglas de negocio al nivel de instrucción. Esto no es apropiado para sistemas grandes. De acuerdo al criterio de representación múltiple y

abstracción jerárquica, es deseable extraer reglas de negocio en niveles de abstracción altos, tales como gráficas de invocación a módulos externos o gráficas de dependencia de módulos representadas al nivel del programa. Segmentación al nivel de módulo o de procedimiento puede ser usada para extraer relaciones entre módulos y procedimientos. Por ejemplo un segmento de módulo consiste en módulos y de variables entre módulos, a través de las cuales los módulos interactúan entre ellos. Sin embargo depende del usuario determinar cual nivel a usar es el más apropiado para derivar el segmento bajo diferentes situaciones.

2.5.3.1 Representación de las reglas de negocio.

Cuando la segmentación del programa está lista, con las variables de dominio de entrada y salida y los puntos extremos del segmento, ahora el objetivo es como visualizar esta estructura de la vista, esto es qué formato de representación para las reglas de negocio es el más significativo para el usuario. Existen 3 elecciones de vista.

Vista de código.
En este formato se representan las reglas de negocio como fragmentos de código. La vista al nivel de código es la representación de las reglas de negocio de más bajo nivel. Desde que el personal dedicado al mantenimiento confía más en el código en operación que en cualquier otro documento y desde que durante el proceso de mantenimiento de software se prefiere una vista detallada de las reglas de negocio, la disponibilidad de la vista al nivel de código es esencial. La vista al nivel de código puede ser obtenida al presentar simplemente el segmento de código obtenido anteriormente.

Vista de fórmula.
Algunos usuarios encuentran deseable ver las reglas de negocio implementadas en el código como fórmulas en lugar de como segmentos de código. Una fórmula puede ser derivada al atravesar una gráfica de dependencia que represente el segmento de código. Durante el proceso de atravesar dicho segmento se hacen substituciones algebraicas hasta que las variables de dominio son alcanzadas o bien se alcanzan tramos de control.

Vista de dependencias de entrada/salida.
Un segmento de programa es un procedimiento que liga entradas a salidas. Una vista de dependencia de entrada-salida es un modo de trazar el flujo de los datos, ya sea de entradas a salidas o viceversa. La misma caracteriza las relaciones del flujo de datos entre todas las variables involucradas en el segmento de código. Representa reglas de negocio en forma abstracta. Una vista de dependencias de entrada-salida de un segmento de un programa se puede derivar al trazar la gráfica de dependencia correspondiente.

2.5.4 ARQUITECTURA GENERAL DE UNA HERRAMIENTA DE EXTRACCIÓN DE FUNCIONALIDAD.

Un proceso de extracción de reglas de negocio se puede basar en un árbol sintáctico abstracto de análisis (*AST*). Un *AST* es un árbol, donde los nodos internos son etiquetados con operadores y

los nodos hoja representan los operandos de los operadores. Este árbol es obtenido por medio del análisis sintáctico del código

El analizador sintáctico requerido para construir el *AST* que ayude en esta tarea tiene que ser específicamente diseñado para analizar e incluir todos los elementos del código fuente, incluidos los comentarios. Los analizadores sintácticos diseñados para propósitos de verificación de sintaxis o transformación de código no son apropiados para cumplir el objetivo mencionado.

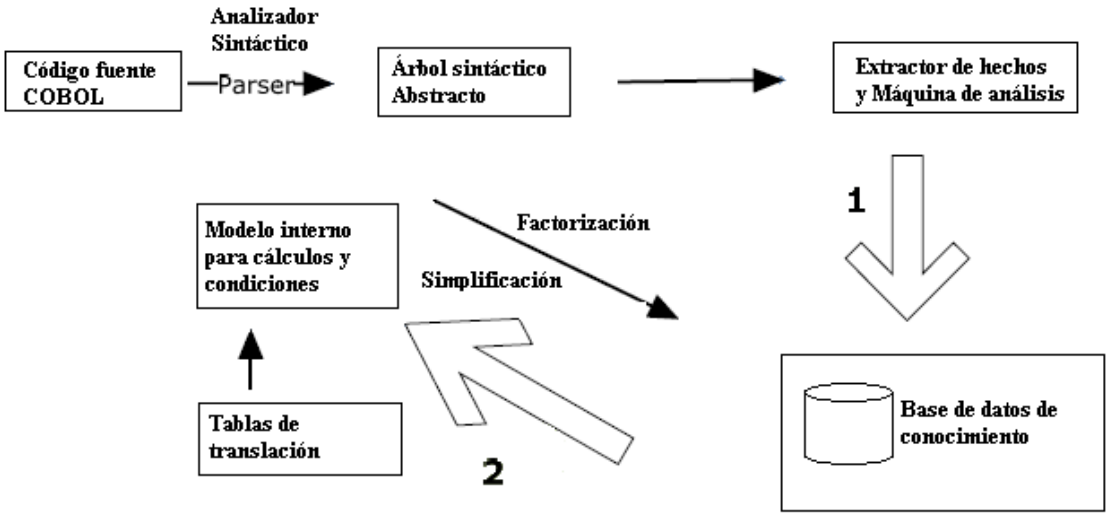


Figura 13 Arquitectura general propuesta para un extractor de funcionalidad a partir de código fuente.

La extracción de reglas de negocio se divide en dos pasos. Primero se analiza el *AST* y se construye una base de datos de conocimiento. Segundo, una vez que la base de datos es construida, se simplifican los datos recolectados y se enlazan los artefactos, donde sea posible con la documentación existente del sistema (Ver figura 13).

Los elementos extraídos de la base de datos de conocimiento son:

- **Reglas de producción**: condiciones *IF*, acciones
- **Condiciones:** Una o más expresiones booleanas con variables y constantes unidas por operadores lógicos
- **Reglas de negocio:** Una acción, posiblemente una condición o algún comentario del código fuente; donde la acción puede ser ya sea una instrucción de bifurcación o cálculos efectuados con identificadores y constantes.
- **Las acciones:** Pueden ser instrucciones de bifurcación o de cálculo con identificadores y constantes.
    - Identificadores: Variables utilizadas en condicionamientos y cálculos.
    - Bloques de código: Representan una o más líneas de código del programa original.
- **Dependencias de reglas de negocio**: Algunos cálculos en otras reglas de negocio que son ejecutados con anterioridad pudieran estar ligados a alguna o algunas otras reglas de negocio; Información de ciclos y bifurcaciones: El párrafo en el cual se ubica a una regla de negocio pudiera ser invocado desde un ciclo desde otra ubicación del programa;
- **Ciclos e información de bifurcación**: El párrafo en el cual se localiza la regla de negocio y puede ser llamada en un ciclo desde otra parte del programa.

- **Excepciones:** Son todas las operaciones que llevan a un mensaje de error o a una terminación anormal.

El siguiente ejemplo ilustra las reglas de negocio extraídas de un fragmento de código *COBOL*. Es importante notar que las condiciones *IF* se obtienen sin expansión y que las dos instrucciones *COMPUTE* se tratan como separadas (no son parte de una instrucción *IF*). Las reglas extraídas en este ejemplo no son validadas en un análisis semántico.

```
019872   IF M39PEN-CD = 18
019873     IF FUND2-PREV-YR-202 > 0
019874       COMPUTE FUND2-5B7-202 =
019875               FUND2-5B7-202
019876             + (FUND2-GROSS-202
019877              * CORRCTNL-FACTOR)
019878     END-IF
019879     IF FUND3-2PYR-GROSS-202 > 0
019880         COMPUTE FUND3-5B7-202 =
019881                 FUND3-5B7-202
019882               + (FUND3-GROSS-202
019883                * CORRCTNL-FACTOR)
019884     END-IF
019885   END-IF
```

Y las reglas extraídas:

• BR-1: Si M39PEN-CD = 18 Y FUND2-PREV-YR-202 > 0, calcular
FUND2-5B7-202 = FUND2-5B7-202 + (FUND2-GROSS-202 * CORRCTNL-FACTOR)

• BR-2: Si M39PEN-CD = 18 Y FUND3-2PYR-GROSS-202 > 0, calcular
FUND3-5B7-202 = FUND3-5B7-202 + (FUND3-GROSS-202 * CORRCTNL-FACTOR)

2.5.5 ENFOQUE DE LA INGENIERÍA INVERSA.

A pesar de tener tamaño de millones de líneas de código, algunos de los programas heredados en *COBOL* que se encuentran en ciertos dominios aplicativos están razonablemente estructurados con secciones de *DATA DIVISION* bien desarrolladas y numerosos párrafos unidos. Para cuantificar estas características algunos investigadores han analizado ejemplos de código y se han realizado métricas que pudieran ayudar a comprender la estructura y significado del código: Se realizan conteos del número de variables el número de variables de nivel 1 y se ha encontrado que dichas variables tienen un cierto número de variables tipo campo asociadas a ellas [89].

Existen investigaciones en el pasado encaminadas a documentar de manera automática el código *COBOL*, lo anterior debido a que la inexistencia de la misma es un problema que se encuentra frecuentemente en los sistemas heredados. Entre las principales técnicas utilizadas se encuentran el uso de diccionarios, traducción de código en seudo-código, ligado del código con la documentación existente, etc.

El análisis de flujo es una técnica tradicional de la optimización de compiladores para extraer información útil de un programa en tiempo de compilación. El análisis de flujo determina factores invariantes a la trayectoria de ejecución en puntos específicos del programa.

Un proceso de extracción de funcionalidad a partir del código fuente tiene como objetivo expresar de manera formal: La funcionalidad del programa. Si bien muchos paquetes comerciales generan una gran cantidad de documentación e información sobre las estructuras de datos y el control de flujo del programa a partir del código fuente, lo que realmente importa es la verdadera funcionalidad del programa, esto tanto a clientes como al personal de mantenimiento y esto es verdaderamente difícil de descubrir. Existen actualmente muchos avances tanto en métodos como en automatización para lograr este objetivo [90]. Normalmente las personas involucradas en el mantenimiento de programas pueden obtener algunas pistas acerca del funcionamiento de un programa al examinar las entradas y salidas del programa y al relacionar las mismas a diagramas de las estructuras de los datos contenidas en la *DATA DIVISION* del programa, esto es que se ve una variable *A* convertida en una *B* por el segmento de código *P*, solo al mirar la lista de declaraciones de datos, hacer un razonamiento abstracto sobre los nombres de las variables utilizadas e identificando donde es usada qué variable.

Sin embargo la funcionalidad detallada solo puede ser obtenida por medio de un análisis de instrucción por instrucción que se encuentran en la *PROCEDURE-DIVISION*, una vez que se obtiene una descripción precisa, se puede extraer una documentación más descriptiva y exacta incluidas explicaciones en lenguaje natural que conectan la funcionalidad con el problema original del dominio al que pertenece el programa.

Otra aproximación consiste en aplicar algoritmos para generar especificaciones de bajo nivel, esto para manejar aplicaciones que son menos dóciles a métodos que pueden detectar algún tipo de patrón. En estos casos la acumulación de información sobre un programa para tratar de bosquejar una especificación sin un conocimiento profundo previo parece ser una tarea con poca garantía de éxito. Sin embargo en muchas ocasiones es posible desenmarañar el código hasta una especificación aceptable inclusive si no se conoce cuál es el significado en un sentido profundo. Una técnica a aplicar en la solución de este problema es la de la simplificación de especificaciones funcionales generadas de manera automática en una forma formal que ha probado por sí mismas ser accesibles a la lectura del ojo humano y a la mente.

El proceso de ingeniería en reversa, puede basarse en 3 fases: Limpieza, especificación y simplificación (figura 14), lo cual puede tener una cierta analogía con el proceso de compilación.

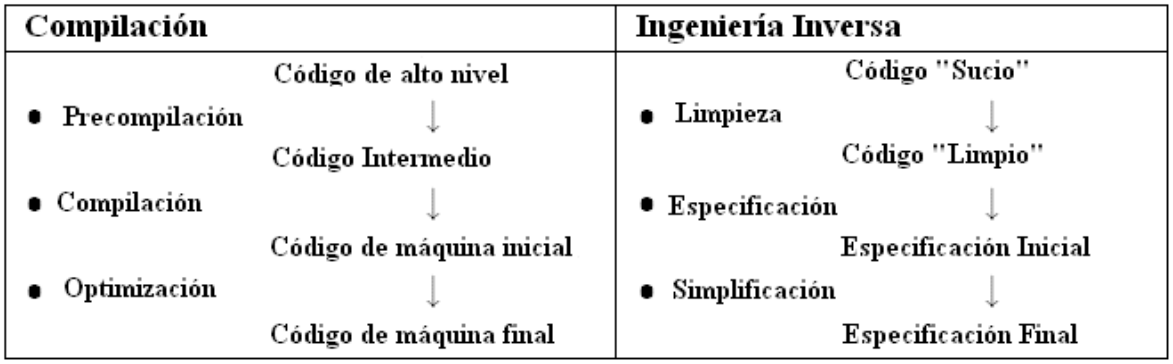


Figura 14 La ingeniería inversa visualizada de manera análoga al proceso de compilación.

La idea es la de obtener una aproximación al diseño original a partir del código que aparentemente no tiene conexión con el diseño entonces un esfuerzo encaminado a obtener una forma estándar, la cual provea una descripción clara del programa en un formato legible y entendible provea además una base sólida para la aplicación de técnicas más profundas: ingeniería en reversa, ingeniería estándar y documentación.

En todas las fases del proceso de extracción de la funcionalidad para la aplicación total o parcial existe la necesidad de la guía y razonamientos humanos, pero también es posible conducir el proceso de ingeniería en reversa en forma completamente automática. En particular la fase de limpieza, que es la parte de reestructuración de código, puede ser llevada a cabo de manera automática [90]. Sin embargo, la fase de especificación se puede facilitar por medio de decisiones de inteligencia acerca del nivel de detalle a ser incluido. Por ejemplo una operación de escribir archivo generalmente también modifica el campo de fecha de grabado, en este caso dicha operación de actualización del campo puede ser ignorada para propósitos de análisis.

Además, si bien hay claramente espacio para la aplicación de razonamiento humano en la fase subsecuente de simplificación, de manera frecuente parece ser satisfactorio dejar que una gran parte de este trabajo sea automático. Las decisiones de inteligencia parecen consistir en decidir cuales detalles deben ser dejados fuera, con lo cual si es utilizada dicha inteligencia o no, se tiene que los métodos empleados proporcionan una especificación que es más legible que el código fuente.

Para efectos de la comprensión automática de un programa es preferible asumir que se tiene un programa razonablemente estructurado, es decir, que el mismo no contiene instrucciones de salto no controlado *GO TO* o otras construcciones poco estructuradas, con sus entradas y salidas plenamente identificadas. Entonces el proceso de ingeniería en reversa inicia con la reorganización de los datos y del código para facilitar su análisis, esto encaminado a producir objetos de funcionalidad bien definida.

El proceso para obtener objetos de funcionalidad abstracta es el siguiente:

1.- Identificación. El primer paso es recolectar todas las banderas y estructuras de datos asociadas en el área de almacenamiento *WORKING-STORAGE* las cuales registren información importante acerca de las principales estructuras de datos (archivos o tablas). Estas banderas podrían registrar cuando un archivo es inválido o está vacío. El proceso de ingeniería en reversa tiene entonces que hacer conjeturas y declarar una relación entre dichas variables y la estructura de datos principal, esto se considera como una parte invariable del programa, que se asume así durante todos los puntos significativos en la ejecución del mismo. Revisar la validez de estas conjeturas de hecho no es parte significativa del trabajo, que solo requiere de la revisión de algunas instrucciones relacionadas que son obvias.

2.- Ensamblado. El siguiente paso es tomar cualquier variable destino *f* que es del tipo arreglo o archivo y tratar de acumular ahí a todas las variables que están relacionadas a ella de manera lógica o conceptual. Se buscan operaciones en el código que cambien la estructura principal, tales como leer o escribir el archivo y determinar si otras variables son actualizadas en instrucciones cercanas (en términos del control de flujo), de tal forma que las propiedades conjeturadas se mantengan. Para cada variable subordinada *x* y una propiedad propuesta $\theta(x)$, se agrega *x* como una variable local del objeto *F* la cual se trata de interpretar que extiende el tipo declarado de *f*.

Se hace conjunción de *θ(x)* con las relaciones invariantes conocidas de la estructura y se agrega cualquier operación que modifique o haga acceso de esta variable y sea usada dentro del código para la lista de operaciones. Las operaciones ya descubiertas puede ser necesario extenderlas para incorporar efectos colaterales sobre *x* que le den mantenimiento a *θ(x)*. Se termina esta tarea con una lista de operaciones *ACT* y una invariante lógica simple *INV* de *f* la cual se conserva para cada operación en la lista y la cual es verdadera después de la inicialización de las variables locales *x* de la clase *F*.

Revisar estas condiciones tiene que ser hecho con la ayuda de una herramienta de razonamiento basada en el lenguaje de representación del código.

3.- Detección de errores. Si la revisión de invariabilidad falla, entonces se necesita ya sea flexibilizar o cambiar por completo la propiedad o concluir que *x* no realmente un componente de la estructura construida en *f*. En la práctica habrá imperfecciones en la relación entre las variables subordinadas y las principales, en estas el programador pudiera no haber actualizado variables asociadas en alineación con las variables principales cuando esto es de hecho necesario de manera lógica. Si se restringe la atención a los puntos en el código donde estas variables subordinadas se acceden, sin embargo, entonces se espera con certeza que se conserve una relación consistente y si esto no ocurre, entonces algo está mal en algún punto, ya sea el proceso de ingeniería en reversa o el código original. Si es lo último y se intenta reestructurar el código a un alto nivel, son este tipo de fallas lógicas las que deberían ser tomadas en cuenta para una corrección y este método de análisis las identifica.

4.- Reorganización del código. El siguiente paso es separar el programa en procedimientos distintos (materialización de procesos) lo cual ejecuta una función simple de entrada/salida, posiblemente en varios archivos o bien en estructuras internas de datos e identificar esta funcionalidad de manera exacta, se buscan dos tipos de cosas:

a) *Límites de proceso.* Los mismos se obtienen al examinar los lugares donde los archivos se abren o se cierran, cuando se interrumpen las rutinas de entrada o salida, o con más trabajo al examinar los patrones de acceso a las estructuras. De manera lógica se trata a estas operaciones como llamadas a nuevos objetos, los cuales han sido definidos previamente como relaciones invariantes.
b) *Funcionalidad.* Estas secciones de código (o funcionalidad simple) *P* son analizadas para obtener su funcionalidad, ya sea por métodos que infieren la funcionalidad o por postulados de que la función implementada es de la forma $f_P(in_1,...,in_n) = (out_1,...,out_m)$, donde la lista de argumentos incluye todas las entradas y la lista de resultados a todas las salidas (esta forma excluye cualquier efecto colateral no declarado) y después derivando f_P al usar la herramienta de razonamiento para encontrar un predicado que exprese el efecto de *P*.

Utilizar una herramienta de razonamiento lógico de esta manera es un procedimiento bien documentado e involucra trabajar haciendo referencia a las últimas instrucciones lógicas del proceso, postulando y revisando relaciones invariantes donde sea necesario, ayudado por heurísticas. Cada instrucción no compuesta tendrá un predicado estándar asociado a la misma (esto es que sea automáticamente generada por la herramienta) pero se deben detectar de manera manual ciclos de condiciones invariantes. Si no es posible una detección manual simple por parte de un ingeniero, la herramienta genera una instrucción genérica más bien carente de información (pero que tiene exactitud), por lo tanto es en principio un proceso totalmente automático pero es

mucho más efectivo si es utilizado en forma interactiva. El ingeniero inicia con un conocimiento resumido del programa, hasta tener un conocimiento detallado del mismo usando la precisión de las técnicas de verificación para revisar y hacer más precisa la intuición creciente acerca del programa.

2.6 EXPRESIÓN DEL CONOCIMIENTO EN FORMATOS VISUALES.

Existe una gran variedad de técnicas de visualización para facilitar las tareas de ingeniería en reversa. Las cuales extraen de manera estática información del flujo de ejecución de los procedimientos o funciones y los presentan de manera gráfica [91, 92].

La visualización gráfica de la ejecución de procedimientos se puede elaborar a partir de un conjunto de datos que tenga información referente a los nombres de los procedimientos y la información de los procedimientos que se ejecutan de manera previa o posterior. Es deseable también tener información del nivel lógico de abstracción para presentar información parcial de alto nivel o más granular. Dependiendo del nivel de abstracción de interés, los procedimientos se pueden conceptualizar como figuras colocadas en un plano de trabajo. La figura que represente a cada procedimiento puede ser homogénea o diferenciada en color (o por ejemplo una figura diferente) para los diferentes niveles lógicos en la profundidad de ejecución [93].

La visualización puede ser considerada como un proceso que transforma datos en crudo en vistas. Ha habido dos categorías importantes de modelos de procesamiento de datos que han sido propuestos para modelar el proceso de transformación visual. La visualización transforma datos o información a una forma gráfica que se puede desplegar. Por lo tanto la visualización trata con temas tanto de transformación como de representación. La transformación es el proceso que convierte datos en gráficas primitivas. La representación son las estructuras de datos que son utilizadas para soportar y almacenar las múltiples salidas de estos procesos.

Muchos trabajos de investigación se han concentrado en las transformaciones que son necesarias para generar un mensaje gráfico desplegado. Estos trabajos relacionados al flujo de datos generan gráficas dibujando una red con nodos que representan al proceso de transformación de datos y bordes direccionales que representan como fluyen los datos de un proceso a otro. La experiencia en el campo de la visualización ha mostrado que el modelo de flujo de datos es un modelo de programación efectiva que permite a los usuarios construir una aplicación al integrar los módulos que la componen.

2.6.1 LA HERRAMIENTA GRAPHVIZ.

La complejidad de algunos programas ha comprobado ser un serio impedimento para entender lo que hacen los aplicativos, la visualización del software es una forma de atacar el problema. La idea es la de modelar algunos aspectos del software en forma de gráfica y la de presentar la gráfica como un dibujo que haga más fácil de entender el modelo. Las gráficas son convenientes

para describir modelos de tipos de datos, funciones, variables, estructuras de control, archivos e inclusive defectos en el código fuente de los programas o bien la estructura de máquinas de estado finito y gramáticas. Las gráficas pueden ser creadas a partir de análisis estático, monitoreo dinámico (en tiempo de ejecución) y de algunas otras fuentes.

Graphviz es una herramienta de código abierto que implementa varios algoritmos de presentación gráfica de información, entre sus principales características están la optimización de área de gráficas con número de elementos limitado, la presentación de dibujos con diferente formato estético y por último, que tiene una interfaz de implementación para diferentes lenguajes de programación. Se considera que la arquitectura de esta herramienta ilustra de manera adecuada como se expresa conocimiento en formato gráfico, por lo cual se describe de manera básica la implementación de sus algoritmos.

Los algoritmos en los cuales se basa el producto abarcan desde gráficas estándar y algoritmos de dibujo, implementados para dar robustez y flexibilidad, hasta modificaciones novedosas de los algoritmos estándar y utilizados de maneras nuevas. Parece más natural describir dichas técnicas en el contexto del modelo gráfico de dibujo donde son utilizadas y se deja para el final a los algoritmos multipropósito.

Dibujos estáticos en capas. Para dibujos en capa, Graphviz se basa en el enfoque de estilo Sugiyama [94]. El objetivo es hacer dibujos estéticos de gráficas de tamaño moderado acercándose al estilo de elaboración manual. El primer paso en la creación de un formato con diseño Sugiyama es colocar los nodos en rangos discretos, dando preferencia a la dirección de las aristas. Hay muchas maneras de hacerlo dependiendo de cuales aspectos de la asignación de rangos son considerados más importantes. Graphvitz modela la asignación de rango a los nodos según el siguiente programa linear entero:

Minimizar $$\sum_{(u,v)\in E} w(u,v)(y_u - y_v) \quad (1)$$

Sujeto a $$y_u - y_v \geq \delta(u,v) \text{ para todo } (u,v) \in E \quad (2)$$

Donde y_u denota el rango del nodo u y por lo tanto es un entero no negativo y δ (u, v) es la longitud mínima de la arista. Por defecto, δ se toma como 1, pero el caso general soporta aristas planas, donde los nodos están colocados en el mismo rango ($\delta = 0$) o cuando es importante asegurar una separación mayor ($\delta > 1$). El factor de peso w (u, v) permite especificar la importancia de tener el rango de separación de dos nodos tan cerca del valor mínimo como sea posible. El propósito de asignar un rango a un nodo es conseguir aristas de longitud mínima lo cual es relevante desde el punto de vista estético y cognitivo. El problema que representa el algoritmo linear para asignación de rangos tiene varias soluciones polinomiales en el dominio del tiempo, el cual depende del número de nodos. En el algoritmo de red que se utiliza en Graphvitz, la asignación de rangos es un proceso pensado como cualquier asignación de coordenadas (y) a los nodos, este proceso es factible si satisface las restricciones de longitud de las aristas (2). Dado cualquier rango, no necesariamente factible, la *holgura* de una arista es la diferencia entre su longitud y su longitud mínima, por lo tanto una asignación de rangos es factible si la holgura de todas las aristas es no negativa. Una arista es *ajustada* si su holgura es cero. Un árbol expandido de una gráfica induce la asignación de rango o bien una familia de rangos equivalentes (el árbol expandido se define sobre una gráfica sin raíz ni dirección y no es necesariamente un árbol dirigido) Esta asignación de rango se genera al tomar un nodo inicial y darle un rango, luego para

cada nodo adyacente al nodo con rango asignado en el árbol expandido, asignarle el rango, incrementado o disminuido en la longitud mínima de la arista que los conecta, dependiendo de si el mismo es origen o destino con referencia al nodo ya calificado. Un árbol expandido es factible si este induce una asignación de rangos factible. Por la forma de construcción, todas las aristas en un árbol factible son ajustadas. Cuando ya se tiene un árbol de expansión factible, se puede asociar un valor entero de *corte* a cada rama del árbol como sigue: Si dicha rama del árbol es borrada, el árbol se divide en dos componentes conectados (disjuntos entre sí), la parte "trasera" que contiene al nodo que era origen de la arista y el componente "delantero" que queda del lado del nodo destino de la arista borrada. El valor de corte se define como la suma de los pesos de todas las aristas desde el componente trasero del árbol hasta el componente delantero incluyendo el peso de la arista borrada, menos la suma de los pesos de todas las aristas desde el componente delantero hasta el componente trasero. Típicamente (pero no siempre) un valor negativo de corte indica que la suma de pesos de la longitud de la arista puede ser reducida al alargar la rama de la arista tanto como sea posible hasta que alguno de los componentes de la parte delantera con dirección hacia la parte trasera se vuelva estrecho. Esto significa reemplazar una rama de un árbol expandido con una nueva rama ajustada, teniendo como resultado un nuevo árbol expandido factible. Un ejemplo se muestra en la figura 15, se tiene una gráfica de 8 nodos y 9 aristas, la longitud mínima de arista es 1 y las aristas que no pertenecen al árbol de expansión están punteadas. Los números en las ramas son valores de corte. En el dibujo (a) la rama (g, h) tiene un valor de corte de -1. En la parte (b) de la figura la misma se borra del árbol y se remplaza por la rama (a, e) reduciendo de manera estricta la longitud de la arista. Para efectos de claridad en el efecto del algoritmo, los árboles de expansión no se muestran, solo el resultado obtenido.

Es simple ver que una asignación óptima de rangos en el sentido del programa linear (1-2), puede ser utilizada para generar otra asignación de rangos inducida por medio de árboles de expansión factibles. Estas observaciones son la clave para resolver el problema de asignación de rangos de manera gráfica en lugar de un contexto algebraico. Las aristas de los árboles con valores negativos de corte se reemplazan por aristas que no son del árbol de expansión hasta que todas ellas no tengan valores de corte negativos. El árbol de expansión resultante es el que corresponde a la asignación de rangos óptima.

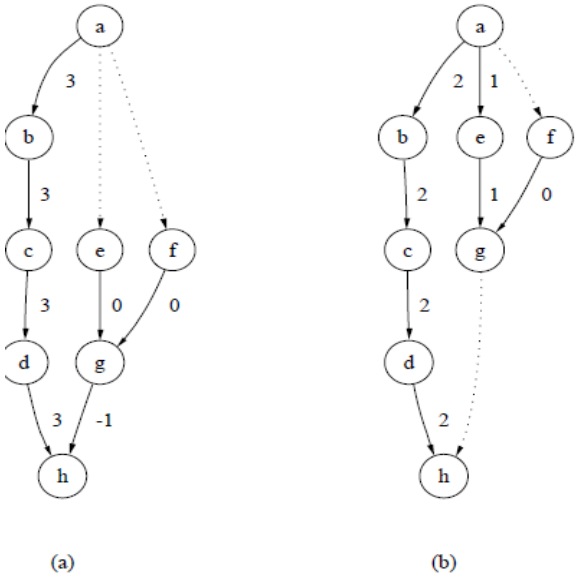


Figura 15 Reducción de gráficas con el algoritmo Network Simplex.

Se lista a continuación el algoritmo de optimización de gráficas utilizado por Graphviz.

Algoritmo: Network Simplex

Entrada: Una gráfica dirigida sin ciclos $G = (V, E)$
Salida: La asignación óptima de rangos para V
Se crea un árbol de expansión factible inicial T

Mientras que la arista $a \in T$ tenga valor de corte negativo, hacer:
Tomar la arista $f \in E \setminus T$ con la mínima holgura
Poner $T = T \cup \{f\} \setminus \{a\}$
Fin de ciclo Mientras

Asignación de coordenadas. La asignación de coordenadas *(y)* para dibujos descendentes es básicamente trivial. Pero por otro lado, tomar las coordenadas *(x)* adecuadas para minimizar las vueltas en las aristas, obtener una gráfica compacta y un formato estético es un poco más complejo. En este tipo de problemas se resuelven utilizando un programa no lineal para colocar los nodos.

$$\text{Minimizar} \sum_{(u,v)\in E} \Omega(u,v) w(u,v) \mid x_u - x_v \mid \qquad (3)$$

Sujeto a $x_a - x_b \geq \rho(a,b)$ para toda a y b

Donde a es el vecino de la izquierda de b en el mismo rango. En este programa no linear, $\rho(a,b)$ da la separación mínima horizontal de a y b. la cuál es tomada usualmente como la suma de la mitad de sus respectivas longitudes, más alguna constante en el espacio entre nodos. Ω es una función adicional de peso que favorece la permanencia en recta de las aristas de longitud mayor. Específicamente Ω es máxima donde los 2 vértices son artificiales, menor cuando solo uno de los vértices es real y mínima cuando ambos lo son. Una transformación linear introduce variables adicionales para quitar el valor absoluto, gráficamente esto corresponde a crear una nueva gráfica G' como se ilustra en la figura 16, en este caso se ignoran nodos planos en G. La nueva gráfica tiene el mismo conjunto de vértices que G más un nuevo vértice n_e para cada arista. Hay dos clases de aristas en G', la primera definida al crear dos aristas $e_u = (n_e, u)$ y $e_v = (n_e, v)$ para cada $e = (u, v)$ en G. Estas aristas tienen $\delta = 0$ y $w = w(e)\ \Omega\ (e)$ y de ese modo codifican el costo de la arista original. El otro tipo de nodos separa nodos adyacentes en el mismo rango. Si v se encuentra inmediatamente a la izquierda de w en su rango se puede añadir una arista $f = e_{(v,w)}$ en G' con $\delta(f) = \rho(v,w)$ y $w = 0$. Se debe notar que esta arista no tendrá efecto en el costo del formato.

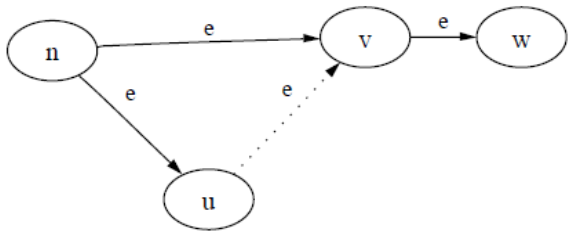


Figura 16 Construcción de G'.

Con esta construcción, la solución del problema original de optimización se convierte en equivalente a encontrar un rango óptimo en la gráfica derivada *G'* y se puede reutilizar el algoritmo Network Simplex. Este planteamiento tiene una ventaja adicional, al poner apropiadamente la mínima longitud a las aristas, en vez de utilizar el valor por defecto de cero, la gráfica derivada puede codificar intercambios horizontales en puntos finales para permitir nodos puerto. Esto permite el dibujo de flechas entre campos en los registros, como se muestra en la figura 17.

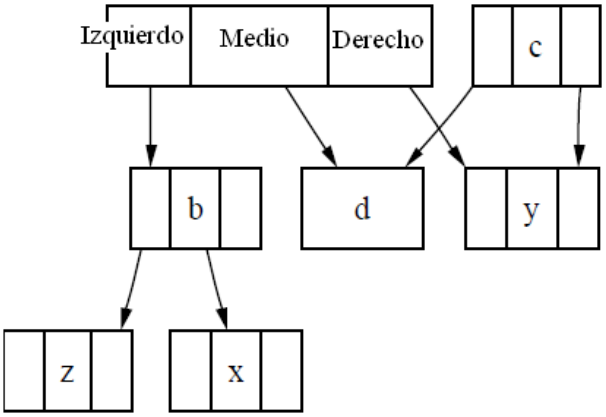


Figura 17 Registros, campos y nodos de origen (puertos).

Si $e = (u, v)$ es una arista, sea Δ_u y Δ_v el desplazamiento horizontal deseado para los extremos de una arista desde el centro de *u* y *v* respectivamente, un valor negativo de Δ corresponde al puerto tomado a la izquierda del centro del vértice. Se puede modificar el problema de optimización (3) al hacer el costo de una arista $\Omega(e)w(e)|x_u - x_v + d_e|$. Donde $d_e = |\Delta_u - \Delta_v|$ y con $\delta(e_u) = d_e$ y $\delta(e_u) = 0$ y asumir sin pérdida de generalidad que $\Delta_v \geq \Delta_u$.Al aplicar la construcción de *G'* y utilizar el algoritmo Network Simplex se obtienen las coordenadas horizontales y desplazamientos de puertos deseados.

El dibujo de las aristas, como paso final, las cadenas de nodos artificiales se utilizan para guiar la construcción de líneas, las cuales los reemplazarán. Aunque se usan algunas técnicas especiales en la elaboración de gráficas por capas, en particular para manejar aristas planas y paralelas, la esencia del enfoque que utiliza Graphviz se describe más adelante.

Para los formatos simétricos Graphviz implementa dos algoritmos que utilizan modelos físicos virtuales, uno es el algoritmo de producción de formato Kamada-Kawai [95] el cual es una variante del algoritmo de escalamiento multidimensional concebido en la comunidad estadística en la década de los 50s y 60s y que fue propuesto por primera vez como algoritmo de gráficas por Krustal y Seery en 1978 [96]. Además del modelo estándar para utilizar longitudes de caminos en la gráfica para la matriz de diferencias. Graphviz implementa también un modelo de circuitos basado en la ley de Kirchoff, el cual fue propuesto por Cohen [97] y codifica el número de caminos entre dos nodos con base en el cálculo de la distancia y tiene el efecto de hacer los grupos de nodos más ajustados. Además un segundo formato simétrico es proporcionado, el cual implementa varios modelos basados en saltos. Para gráficas grandes, se basa en bandejas dinámicas, una extensión de la técnica propuesta por Fruchterman y Reynold [98] para aproximar

fuerzas repulsivas de larga distancia y de este modo reducir el tiempo de ejecución a casi linear. Además esta técnica proporciona grupos de gráficas al utilizar recursividad.

Eliminado de traslape. La asunción general es que una gráfica se dibuja representando a los nodos como puntos y a las aristas como líneas, sin embargo generalmente esto no es así, ya que normalmente un nodo es representado por una forma que tiene un área, lo cual puede ocasionar que haya traslapes entre nodos, lo cual es generalmente indeseable.

Graphviz tiene tres estrategias para eliminar el traslape de nodos, una los elimina al escalar de manera uniforme las escalas entre los centros de los nodos mientras conserva el tamaño de los nodos [42]. Esto conserva en general las relaciones entre nodos pero puede desperdiciar un área considerable. Una segunda propuesta es el método de escaneo de fuerza bruta de Misue y coautores [94], aquí el formato de la gráfica se construye por medio de pasos de recorrido horizontales y verticales y se ejecutan movimientos rígidos de subconjuntos de nodos en la dirección de recorrido. Esto conserva el ordenamiento ortogonal mientras que (pero no siempre) requiere de menos espacio que el escalamiento. La tercera técnica es un sofisticado método heurístico iterativo que utiliza diagramas Voronoi, lo anterior basado en las investigaciones de Lyons y coautores [99]. El sustento detrás de esta técnica es que al mover el nodo en su celda Voronoi el mismo está todavía más cerca de su posición anterior que cualquier otro nodo lo está, lo cual ayuda a mantener la estructura de la forma., el método requiere mínima cantidad de espacio extra pero es más destructivo del lado de la forma de la gráfica y de la posición relativa de los nodos.

Algoritmo: Ajuste de Voronoi
Entrada: Conjunto de vértices V con un formato
Salida: Un nuevo formato de V tal que $v \cap u = \varnothing \forall v, u \in V$
Se construye un rectángulo con bordes que contiene a todos los nodos
Sea C el número de pares de vértices con intersección
Mientras que $C > 0$ Hacer:
    Construir un diagrama de Voronoi utilizando centros de vértices como elementos de sitio.
    Anexar celdas sueltas al rectángulo
    Mover cada vértice al centroide de su celda
    Sea C' el nuevo número de intersecciones
    Si $C' \geq C$ entonces
        Expandir el rectángulo
    Fin Si
    Asignar $C = C'$
Fin Ciclo Mientras

Otros algoritmos implementados por Graphviz son el de formato radial [100, 101]. El cual tiene alto desempeño con gráficas grandes, típicamente representa los nodos como puntos y codifica atributos adicionales por medio del uso de colores.

Para el dibujo de aristas en forma de curvas suaves, Graphviz tiene un módulo planificador de rutas curvas formadas con parámetros de polinomios, es una técnica heurística de dos fases, que utiliza los nodos extremos de la arista a dibujar y toma en cuenta los límites de una región de tipo polígono P, la primera fase consiste en determinar el camino más corto para unir los dos extremos (L), el algoritmo tiene un costo $O(n^3)$ que depende del número de nodos a unir n y es práctico solo para gráficas de tamaño pequeño. La segunda fase toma el camino más corto L y

calcula una curva candidata *C* para dicho camino utilizando el algoritmo de Schneider [102] Si la curva candidata queda dentro de la región *P* el cálculo termina, si no, se calculan ajustes en las tangentes de los nodos de inicio y fin de la arista arqueando o aplanando la curva y se detiene hasta que una de estas variantes funciona, en caso de que estas variantes no funcionen, se aplica un procedimiento iterativo: se escoge un punto *v* sobre el camino *L* a la altura de el punto más lejano con respecto a la curva *C* y se divide el camino en 2, L_1 y L_2, tomando como referencia este punto, en seguida se resuelve el problema de calcular la curva para cada camino por separado, las curvas resultantes se unen en *v* para conservar una sola curva resultante continua, no existe garantía de que la curva final concuerde topológicamente con el camino original o que algún punto del camino original, exceptuando el inicial y final estén incluidos en la curva final.

La mayoría de algoritmos de formato de gráficas asumen que la gráfica está conectada, pero dada una gráfica desconectada se puede ya sea aplicar el algoritmo básico de dibujo a cada componente conectado y después acomodar los componentes o bien hacer que la gráfica sea conectada, para solucionar este problema se hace uso de la técnica de formato jerárquico, el de asignación de rangos, se coloca el rango más alto de cada componente en una línea, esto si no existen restricciones de rango adicionales. La segunda propuesta es la implementación del algoritmo Kamada-Kawai, se asigna la distancia deseada entre cada par de nodos en componentes separados a $L/(|E|+\sqrt{|V|}+1)$, donde L es la suma de las longitudes de todas las aristas. La fórmula anterior da la garantía de que componentes disjuntos no tendrán traslape. Cuando se grafican demasiados componentes con los algoritmos anteriores, la aplicación del primero resulta en una gráfica que tiene aspecto de radio pobre y el segundo produce un aspecto más atractivo pero desperdicia una cantidad importante de espacio en el área de dibujo. Para evitar las situaciones mencionadas y para tener una técnica de propósito general de combinación de gráficas desconectadas, Graphviz tiene una librería de empaquetado basada en el algoritmo de empaquetado de polinomio [103] de Freivalds y coautores. Hay un beneficio adicional al usar esta propuesta en conjunto con el algoritmo Kamada-Kawai, que tiene un costo computacional de $O(n^2)$. Si la gráfica es de tamaño moderado, digamos de unos 1,000 vértices, pero con muchos componentes pequeños al aplicar el algoritmo a cada componente y después empacar los formatos juntos, se puede mejorar el tiempo de cómputo en órdenes importantes de magnitud.

El software Graphviz está disponible de manera gratuita en una licencia de código abierto. La dirección es www.graphviz.org y en www.research.att.com/sw/tools/graphviz. Además del software, el segundo sitio proporciona también documentación, ejemplos de formatos y ligas a otros sitios donde se describen las librerías o paquetes que incorporan el uso de Graphviz.

2.6.2 OTROS ALGORITMOS PARA GRAFICAR Y DIBUJAR.

Las gráficas que se representan en un plano por lo general tienen vértices (o nodos) representados por símbolos tales como círculos o rectángulos y las aristas (*u, v*) son representadas por una curva simple abierta entre los símbolos asociados a los vértices *u* y *b*.

Un dibujo en el cual cada arista sea representada como una cadena poligonal es un dibujo de líneas de polígono, existen dos variantes de este tipo de estándar, que son los dibujos de líneas rectas y los ortogonales, en el cual la unión de vértices se hace exclusivamente con líneas verticales y horizontales (ver figura 18). En algunas otras variantes las aristas pueden tener un

aspecto curvo que les da mayor estilo de presentación. Un dibujo es de tipo plano si ninguna de sus aristas se intercepta con otra, si las aristas y los lugares donde presentan curva tienen coordenadas, se dice que se trata de un dibujo de retícula.

Un algoritmo de dibujo de gráficas tiene como entrada una descripción combinatoria de la gráfica *G* y produce como salida un dibujo de *G* de acuerdo a un estándar de gráfica proporcionado. El dibujo es descrito en términos de gráficos primitivos tales como el dibujo de una línea, o el llenado de un círculo, los cuales pueden ser interpretados.

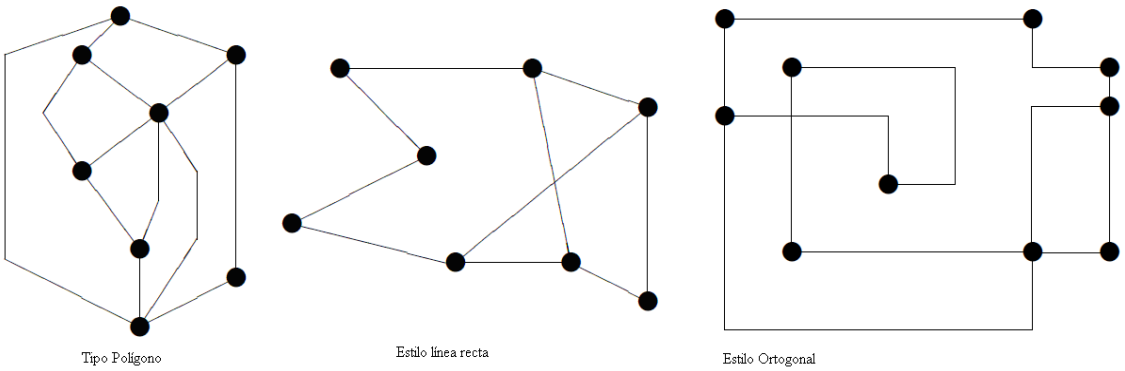


Figura 18 Dibujos de tipo poligonal, estilo lineal y ortogonal

En un estándar gráfico, una gráfica tiene un número infinito de maneras para dibujarla. Sin embargo en casi todas las presentaciones de las aplicaciones, la utilidad de un dibujo depende de su legibilidad, esto es, la capacidad de expresar el significado del diagrama de manera rápida y clara. Los aspectos de legibilidad se expresan en términos de estética, los cuales pueden ser formulados como objetivos de optimización para los algoritmos de trazado. En general, la estética depende del estándar gráfico adoptado y de la clase particular de gráficas de interés. Un aspecto fundamental de la estética es la minimización de cruces entre aristas, En las gráficas de tipo polígono es deseable evitar dobleces en los lados. En los dibujos de retícula, el área del rectángulo más pequeño que cubra todo el gráfico debe ser mínima. En todos los estándares gráficos, es deseable la visualización de simetría. Se debe remarcar que la estética es un concepto subjetivo y podría verse sujeta a una manufactura hecha a la medida para cubrir preferencias personales, tradiciones o cultura. En la figura 19 se muestran dos maneras diferentes de dibujar la gráfica de tipo cubo.

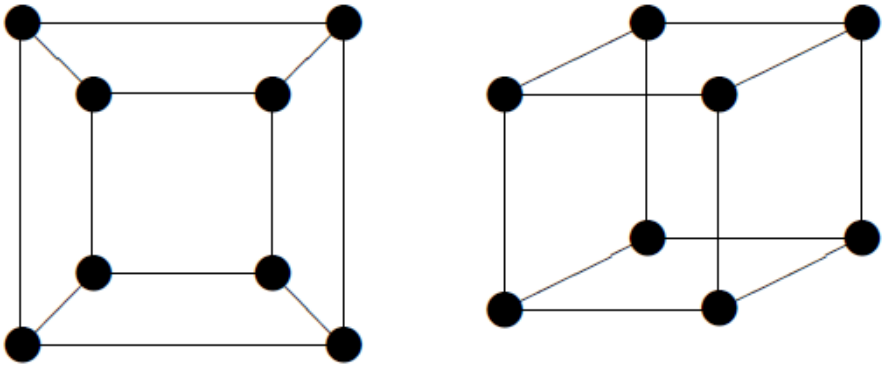


Figura 19 Diferentes estilos de dibujo.

Los trabajos de investigación sobre algoritmos de dibujo de gráficas se extienden sobre un amplio espectro en las ciencias de la informática, desde *VLSI* (Modelado de Circuitos de Vectores Lineares) hasta diseño de base de datos [104].

Las investigaciones sobre dibujos de retícula ortogonales fueron motivadas al principio por los problemas en el trazado de circuitos impresos. Para este estándar de gráficas, la minimización del número de dobleces en las aristas es muy importante tanto para la legibilidad como para su uso en aplicaciones *VLSI*. Cualquier gráfica plana de grado mayor a 4 admite una retícula ortogonal con área de O (n^2), adicionalmente existen gráficas de grado mayor a 4 que necesitan áreas de orden cuadrático, el resultado de las investigaciones se puede ver en [105, 106].

2.6.3 ESPECIFICACIÓN FORMAL EN NOTACIÓN Z.

Z es una especificación formal que utiliza notaciones matemáticas para describir de una manera precisa las propiedades que debe tener un sistema de información sin tener en cuenta demasiado el modo de implementar dichas propiedades [107]. Es decir que se describe que debe tener el sistema sin decir como implementarlo.

Una especificación formal puede servir como un único punto de referencia confiable para aquellos que quieran investigar necesidades de los clientes, para aquellos que quieran implementar programas que satisfagan dichas necesidades para los que prueban los resultados y para aquellos que escriben los manuales de funcionamiento de los sistemas. Esto debido a que una especificación formal es independiente del código, una especificación formal puede ser implementada en código de manera más rápida.

Una manera de lograr las metas mencionadas es la de utilizar notación matemática. Los tipos de datos matemáticos no están orientados a la representación en el código, pero obedecen a una rica colección de leyes matemáticas las cuales hacen posible el razonamiento de manera efectiva acerca de cómo se debe comportar el sistema especificado.

La notación *Z* es un lenguaje formal para crear especificaciones, el cual es usado para modelar sistemas informáticos, se enfoca en la especificación clara de los programas y en la formulación de pruebas relacionadas con el comportamiento del programa.

Una característica importante de la notación *Z* es la manera de descomponer los programas en pequeñas piezas llamadas *Esquemas*. Al descomponer la especificación en *esquemas* el programa se puede presentar pieza por pieza. Cada pieza puede ser ligada a un comentario, el cual explica de manera informal el significado del contenido formal.

Los *esquemas* en *Z* son utilizados para describir tanto aspectos dinámicos como estáticos de un sistema, tales como los estados en los que se encuentra un sistema, la identificación de las condiciones invariantes durante el movimiento entre estados, las operaciones que se ejecutan, la relación entre las entradas y salidas.

Este lenguaje es una aproximación demasiado formal de la especificación, se considera que el punto notable del mismo radica en la precisión para definir requerimientos (en sistemas nuevos y

con notación matemática), sin embargo su uso trae asociado el aprendizaje de una nueva sintaxis, la que va asociada al requerimiento formal, es por esta última característica que no se considera compatible al propósito de la presente investigación, pues aquí lo que se pretende como salida es algo legible para un analista en términos de abstracción de la realidad y no que el usuario deba aprender o interpretar un nuevo lenguaje en notación matemática para poder conocer los resultados de la ingeniería inversa. Sólo para efectos de referencia teórica se menciona la existencia de esta notación.

CAPITULO III. DISEÑO Y CONSTRUCCIÓN DE LA SOLUCIÓN.

Este capítulo es la propuesta de solución al problema descrito en el capítulo I, se expone aquí el diseño del autor de manera conceptual y descriptiva en un alto nivel, ¿cómo es posible mejorar la manera actual de hacer las cosas en el entorno descrito para el problema identificado? Se enuncia y detalla el procedimiento que es el título de la tesis y se perfila el entregable principal: un producto de software que pretende mejorar el estado actual de hacer las cosas, este producto tiene un diseño inicial que se basa en teorías de ingeniería de software, de compiladores y de tecnologías de código abierto. Se incluye además la propuesta opcional de inclusión de algoritmos básicos de extracción de conocimiento y reglas de negocio a partir de los datos que extrae el analizador sintáctico y de igual manera se detalla el prototipo inicial de la *GUI*.

3.1 PRINCIPALES CARACTERÍSTICAS DE LA SOLUCIÓN A DISEÑAR.

El diseño de la propuesta que a continuación se desarrolla se basó al nivel global en el problema a resolver identificado en el capítulo 1. De igual manera, el mismo diseño trata de limitar el alcance del proyecto a un nivel de cierto detalle de implementación. Se ha presentado en el capítulo anterior el marco teórico sobre el que se puede basar una solución aplicativa. La colección de todas esas investigaciones se considera referencia documentada útil, sin embargo el entregable a diseñar no incluirá la aplicación de todas estas investigaciones previamente descritas, lo cual es algo fuera de alcance debido a las restricciones en recursos que se tienen. Sobre estas premisas se enuncian en primera instancia algunas de las restricciones identificadas a priori. Después se bosqueja una aproximación del entregable o prototipo del resultado que se pretende producir. Y por último se describen los principales módulos que componen dicho entregable.

Para la implementación del diseño se tuvo dependencia de un factor muy importante: la disponibilidad tanto de tiempo como de recursos humanos para ejecutar la tarea. La viabilidad del proyecto siempre fue dependiente de la compatibilidad entre de la cantidad de tiempo valorado como necesario para entregar una solución aplicativa aceptable y la cantidad de detalles que se propusieron como viables en el presente diseño.

3.1.1 RESTRICCIONES INICIALES DE LA PROPUESTA.

Antes de iniciar el diseño, se comentan algunas restricciones consideradas como punto de referencia o guía. La naturaleza del procedimiento a plantear no se enfoca a la reestructura de un programa o partición del mismo en unidades funcionales mejor estructuradas y más pequeñas de compilación, que como ya se ha revisado en otros trabajos de investigación, se proyectan como objetos ante una transformación del sistema heredado a un paradigma orientado a objetos. El principal nicho de usuarios al que se dirige esta investigación pertenece al mercado de organizaciones donde el tema principal es el mantenimiento de un sistema heredado o su evolución.

Con base a lo anterior, el uso de teorías de compiladores no implica que se hará una revisión exhaustiva sobre la exactitud léxica, sintáctica y semántica del programa a analizar, ni tampoco sobre la pureza o calidad en la estructura del código, tampoco se pretende sugerir optimizaciones en el control de flujo del programa o en el uso de sus variables. Si durante la implementación de la solución se detectan situaciones muy obvias como uso de variables no inicializadas o bien secciones de código que nunca se ejecutan, se deja abierta la opción de crear una sección dentro del resultado final para reportar el incidente detectado, la implementación de esta parte opcional depende de la extensión limitada en tiempo y recursos para implementar el diseño propuesto en este capítulo.

Podemos derivar la siguiente asunción: Dado que los programas objeto de análisis son programas que han probado su funcionalidad en ambientes de producción, son programas que compilan sin error y que si bien pudieran tener errores asociados a su dominio de información (su naturaleza funcional dentro de la organización), dichos errores no son atribuibles a errores de compilación, por lo que el procedimiento a desarrollar y su herramienta auxiliar no tendrán como objetivo primordial detectar errores de compilación, sino la extracción de conocimiento de carácter funcional.

La forma tradicional de análisis manual implica una cierta familiaridad con los conceptos de dominio, es decir, normalmente un analista de aplicativos ya posee un cierto conocimiento sobre los conceptos que representan las diferentes entidades de información que intervienen en un programa sujeto a análisis, de igual forma los nombres de las variables tienen muchas veces un significado abstracto que ayuda a la inteligencia humana a predecir su significado funcional. Se propone que dicho conocimiento, que pertenece únicamente a la abstracción del cerebro humano, pueda ser mezclado con los resultados de una parte automática.

3.2 PROCEDIMIENTO PROPUESTO DE ANÁLISIS.

En el capítulo dos, específicamente en el punto 2.2.2 se presentó una serie de consideraciones y “buenas prácticas” que normalmente se siguen cuando se analiza funcionalmente una aplicación en la plataforma mainframe, en el lenguaje *COBOL*. El trabajo es sobre todo manual y ha sido

observado de manera personal por el autor en casi 10 años de experiencia profesional en el área. Cada analista tiene su manera de documentar los resultados del análisis, sin embargo en síntesis, esta actividad consiste en coleccionar piezas de información extraídas del código fuente y además en ordenar y clasificar estos datos para obtener un significado a manera de resumen. El procedimiento que se propone para cumplir con los objetivos de colaborar a la solución de problemas asociados al análisis manual dio inició con la documentación de estas actividades a desarrollar de forma manual presentadas en el punto 2.2.2. El aporte aplicativo consiste en auxiliar dichas actividades con un artefacto de software que realizará muchas de las tareas manuales de forma automática, con menor riesgo y en menor tiempo. El procedimiento de análisis que se propone se describe a continuación. Este procedimiento es la guía sobre la que se basa a su vez el diseño e implementación de una herramienta automática, que como desde ahora se anticipa, es la parte medular del procedimiento.

3.2.1 PROCEDIMIENTO DE ANÁLISIS PARA SISTEMAS HEREDADOS EN LA PLATAFORMA MAINFRAME.

1. Todas las actividades asociadas al análisis manual de aplicativos en la plataforma mainframe, específicamente para aplicativos codificados en lenguaje *COBOL* y que también han sido observadas de manera frecuente durante la experiencia profesional del autor, tienen probada eficacia, los resultados obtenidos son válidos desde el punto de vista práctico, por lo que el primer punto del procedimiento consiste en tener como referencia documentada el punto 2.2.2 del capítulo 2 de esta investigación, los siguientes pasos basan su propósito en el auxilio de tales actividades y consideraciones. Se pretende reducir tiempo a invertir y riesgo asociado a las prácticas de naturaleza manual mencionadas, por medio del diseño e implementación de una herramienta semiautomática, un artefacto de software. Este artefacto de software es el núcleo del procedimiento propuesto.

2. Se propone que gran parte de la información útil para el análisis se obtenga de un compilador, para lo cual se deberá diseñar y construir un analizador léxico y sintáctico para el dialecto de *COBOL* que se utiliza en una plataforma mainframe, se trabajará en el caso particular del dialecto de *IBM*® que se ejecuta en los sistemas operativos *MVS*/*ESA* y *Z*/*OS*. Los componentes a implementar serán capaces de analizar la mayoría de los programas *COBOL* en funcionamiento en los mencionados ambientes que tienen código heredado. El artefacto tendrá limitantes técnicas, antes de iniciar la implementación se anticipan, por ejemplo las siguientes: No se incluirá el uso de caracteres nacionales (mapa de caracteres especial para la plataforma), así como *COBOL* orientado a objetos, ni tampoco programas anidados. Queda también fuera de alcance la validación sintáctica formal de lenguajes incrustados, tales como *XML*, *SQL* o *CICS*. Estas limitaciones obedecen a los recursos disponibles (tiempo y ejecutores).

3. Se deben implementar también algunas validaciones de carácter semántico, las cuales ayudan al entendimiento funcional de las variables del componente en análisis. Esta parte de la implementación servirá para determinar el flujo de algunos datos y hacer deducciones sobre las variables que son candidatas a ser variables de dominio.

4. Una tarea auxiliar en el proceso de extracción de conocimiento consiste en implementar un analizador simple de líneas de comentario, mediante un algoritmo básico de reconocimiento de palabras en idioma español o inglés, se pretende identificar líneas de comentario que son candidatas a describir funcionalidad dentro del código, para colocar dicha descripción en algunas secciones del reporte de salida.

5. Desde el punto de vista funcional, el analizador sintáctico, es el extractor de gran parte de la información contenida en el programa. La información extraída debe ser procesada y formateada para expresar conocimiento útil. El proceso complementario para la explotación de la información extraída se basará en el diseño de una base de datos de conocimiento, en la cual se almacenaran de manera estructurada componentes predefinidos de las producciones de la gramática del lenguaje. Se debe revisar la gramática para seleccionar las producciones que aportan conocimiento funcional, entre otras: la creación y modificación de variables, la afectación a las entidades de información de entrada y salida, estructuras de control, elementos externos usados y cálculos aritméticos para producir el valor de una variable, más complejos que una adición o un decremento simple, las variables identificadas por medio de esta técnica son candidatas a ser variables de dominio.

6. El siguiente paso del procedimiento consiste en incluir dentro de la base de datos de conocimiento, una estructura de datos para almacenar el control de flujo del programa en análisis y también para almacenar el flujo de datos de algunas variables, las cuales deben ser seleccionadas siguiendo criterios predefinidos. La alimentación de las tablas de flujo de control y de datos se hará durante el proceso de ejecución del compilador. El flujo de control del programa es una gráfica que representa de manera secuencial cómo se ejecutan las instrucciones y también dónde hay bifurcaciones y ciclos, lo anterior es factible ya que *COBOL* es un lenguaje de tipo imperativo. El flujo de datos es una estructura que ilustra en que parte del programa es creado, utilizado, actualizado y destruido el valor de una variable. No se propone hacer un análisis exacto del flujo de datos, sino solamente lo que quede registrado en una sola pasada del compilador y para algunas variables que cumplen con criterios predefinidos. Como resultado de esto, serán almacenados solamente algunos valores del conjunto de todos los valores posibles que pudiera tomar una variable seleccionada en tiempo de ejecución.

7. El resultado final del análisis se propone como un reporte, el cual es una aproximación a la especificación de diseño funcional del programa analizado. El reporte final de análisis debe tomar como fuente de información la base de datos de conocimiento. Las secciones a diseñar contienen información que se considera suficiente para dar una idea del propósito del programa. La primera sección es el diagrama de entidades de información de entrada y salida. Enseguida se propone el reporte en forma tabular de los componentes externos usados, esta parte es la sección de referencias. La siguiente sección es una representación breve del flujo del programa, en este caso se presentarán los principales saltos en el control de flujo, la representación secuencial de cómo se ejecutan los bloques de instrucciones y de manera opcional, las variables utilizadas o modificadas dentro de los bloques de código. Se proponen tres secciones adicionales de carácter opcional, la implementación de estas depende de restricciones en tiempo, estas secciones podrían implicar el uso de algoritmos más avanzados, es por esto que se plantean como opcionales. Dichas secciones son: “Validaciones funcionales implementadas”; “Reglas funcionales identificadas” y “Otras Consideraciones” tales como operaciones detectadas con variables de dominio o sugerencias que tienen que ver con la calidad de código o instrucciones detectadas como riesgosas.

8. Varias secciones del reporte de salida se deben diseñar con la posibilidad de ser modificadas, con esto, las conclusiones de la parte automática del procedimiento podrán ser sobre-escritas. La especificación resultante podrá ser modificada para colocar resultados del análisis manual, que como ya se había mencionado tiene un nivel de abstracción mayor que los resultados obtenidos de manera automática.

9. La combinación del resultado obtenido de manera automática con el que produce la aportación humana, se anticipa como una especificación que tendría como principal característica la de haber sido generada de una manera ágil, confiable y sobre todo entendible, con esto se cumple el objetivo primordial: identificar de manera esencial y confiable la funcionalidad de un componente en un sistema heredado disminuyendo el riesgo del "error humano" asociado a tareas repetitivas y en gran cantidad.

10. El tema que queda abierto es un método para incluir el conocimiento obtenido para una unidad de compilación en un repositorio global que abarque un aplicativo de un sistema información de varios o todos los aplicativos que constituyen los sistemas heredados de una organización. Una forma de hacerlo es crear una base de datos que tenga las particularidades de cada unidad de compilación, pero que en un segundo nivel de abstracción separe las entidades o elementos corporativos, de uso común para varios aplicativos, al mismo tiempo se deberían incluir envolturas lógicas en la base de datos para separar los componentes por aplicativo. El procedimiento sería hacer trabajo unitario e ir agregando el resultado a un repositorio global (implementando funciones de agregación global), lo que al final facilitaría mucho el mantenimiento y evolución de los sistemas heredados.

11. La plataforma de desarrollo es una computadora personal, el lenguaje de implementación será Java, por lo cual el código fuente *COBOL* a analizar (y los componentes externos asociados) deberán ser transmitidos de alguna manera entre plataformas, la automatización de este tipo de detalles en la implementación queda fuera de alcance, se utilizarán como ejemplos varios programas ya alojados manualmente en la computadora personal donde se desarrolle la herramienta, dichos programas tienen como único propósito servir como ejemplo para la demostración y evaluación del procedimiento en funcionamiento.

12. La interfaz gráfica de usuario será construida como un sitio Web, el cual consta de pantallas muy simples que sirven para solicitar el análisis de un programa, indicar donde se encuentran los componentes externos y para ver o editar los resultados. La elección de esta arquitectura de implementación se debe a que se pretende facilitar el uso de la misma, sin que se tenga que instalar alguna aplicación en cada cliente que utilice la herramienta. De igual forma el desplegado de resultados no necesitará más que de un navegador Web en cada cliente solicitante, pues serán páginas *HTML*. La arquitectura de desarrollo es toda de código abierto, por lo que no se necesitará adquirir licencias adicionales ni para ambiente de ejecución, ni para el lenguaje de desarrollo, ni para la visualización de los resultados.

13. El procedimiento descrito anteriormente será aplicado a varios programas de ejemplo, para demostrar que el resultado es un reporte obtenido en tiempo reducido, que ayuda a entender de manera aproximada lo que el programa hace funcionalmente y que reduce además el riesgo de realizar trabajo manual en tareas que pueden ser ejecutadas de manera automática.

3.2.2 DESCRIPCIÓN INICIAL DEL ARTEFACTO DE SOFTWARE.

En el software a desarrollar, se plantea como principal propuesta la extracción de la estructura del flujo de ejecución incluida en el código fuente, así como las referencias de entidades externas con las cuales se comunica el programa, este es el principal punto de referencia cuando de hace el análisis manual de código. El proceso inicial a diseñar en la herramienta automática/artefacto de software, es pues la extracción de los componentes léxicos, sintácticos y semánticos del programa. En este punto es donde se hará la aplicación de técnicas de la teoría de compiladores, de manera paralela se deben identificar las entidades de información y los puntos dentro del programa donde se ejecutan operaciones *CRUD*. El resultado del análisis es un reporte, el diseño del reporte debe contener varias secciones, la primera corresponde al flujo de control dentro del código, el conocimiento reportado en esta sección debe tener un formato visual o amigable. Con esto, el flujo imperativo y secuencial de procedimientos internos dentro del programa, podrá ser entendido con mayor facilidad. El reporte del análisis debe tener también una sección de información útil al analista sobre las entidades de información de entrada y salida, sobre algunas variables de dominio detectadas y sobre componentes de arquitectura auxiliares, tales como el llamado a subrutinas externas, inclusión de fragmentos externos de código (*COPY*). Esta estructura primaria de análisis puede bastar para entender la funcionalidad del programa, sin embargo el reporte del resultado debe poder ser enriquecido con la introducción de conocimiento de forma manual, con una funcionalidad que facilite el complemento manual de la primera aproximación automática del análisis.

Para resolver el diseño de la herramienta, se listan las siguientes características que la herramienta semiautomática debe obtener dentro del código de un componente en análisis, estas características, como se expuso en el marco teórico, son las mínimas a considerar para obtener conocimiento útil desde el código fuente.

- Número de líneas de código.
- Cantidad de párrafos, nombre y tamaño.
- Texto íntegro de los comentarios, sería útil identificar texto que pertenece a algún idioma (español, inglés) para desechar código comentado y poder tomar en cuenta cualquier texto que pudiera documentar funcionalidad dentro del código.
- Cantidad e identificación de llamados a módulos externos (rutinas).
- Cantidad e identificación de entidades externas de información (Tablas de bases de datos, archivos). Así como el tipo de entidad (entrada, salida, entrada/salida).
- Estructura de desplegado de datos que corresponde a cada una de las entidades de información utilizadas.
- Nombres y cantidad de variables.
- Identificación del uso de variables no inicializadas.
- Identificación y expansión de estructuras de datos utilizadas (*COPIES*, grupos 01 de variables)
- Instrucciones condicionantes simples y anidadas (*IF* por ejemplo).
- Instrucciones de finalización del programa distribuidas a lo largo del código. (Finalización de ejecución)
- Instrucciones *CRUD* (insertar, leer, actualizar, borrar) sobre las entidades de información externa.
- Comunicación con sistemas externos (*CICS*, *DB2*, *JCL*).
- Identificación de mensajes de error enviados por el programa.

- Identificación de operaciones aritméticas de riesgo o críticas (divisiones, incremento de índices de arreglos, cálculos).
- Ciclos controlados por instrucciones *GO TO*.
- Ciclos controlados por instrucciones *PERFORM*.
- Flujo general del programa. Principales flujos de ejecución (párrafos, secciones) hasta una cierta profundidad lógica.

Cuando la información anterior esté disponible y clasificada en algún repositorio o almacén de datos se propone aplicar alguna técnica de extracción de conocimiento para deducir algunas de las reglas de negocio implementadas en el código, esta parte estará limitada por el tiempo disponible para concluir la investigación y es uno de los puntos donde se anticipa cierta relajación y fuente de temas abiertos.

Por último, el resultado final es la inclusión del conocimiento en la base de datos a un reporte que se considera una aproximación de la especificación técnico-funcional del componente analizado, se anexa (**Anexo B**) un prototipo con una primera aproximación de lo que se espera como resultado final del análisis de un componente.

Las especificaciones de arquitectura que se proponen para el artefacto de software a implementar son las siguientes:

- Lenguaje de implementación: Java version "1.5.0_19"
- Ambiente de ejecución: Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_19-b02)
- Servidor Web: Apache Tomcat 5.5
- Base de datos: *MySQL* server 5.0
- Ambiente de soporte de desarrollo (Framework): Struts 2.0.14
- Lenguaje auxiliar de la especificación: Basado en *UML*.
- Para efectos de asignar una identidad, la herramienta automática a construir se denominará *TesisAVE* – Trabajo de Tesis por Antonio Vega Eligio.

3.3 DISEÑO DEL MODELO DE NEGOCIO.

La primera parte del diseño consiste en el modelado del marco de referencia del proyecto al nivel organización. Se tienen dos organizaciones, la primera es quien patrocina el proyecto, cuya misión es la de ser una institución de estado que es rectora de la educación tecnológica pública en México, líder en la generación, aplicación, difusión y transferencia del conocimiento científico y tecnológico, que fue creada para contribuir al desarrollo económico, social y político de la nación. Por otro lado se tiene a una hipotética organización usuaria del resultado de la investigación, la misión y visión de la misma es de naturaleza variable, pero la principal característica es que debe tener entre sus activos sistemas informáticos heredados en la plataforma mainframe, en particular programas codificados en *COBOL*, dentro de esta organización debería haber un portafolio de proyectos orientados al mantenimiento y evolución de sus sistemas heredados y al hacer uso del

procedimiento propuesto, la organización se beneficiaría ahorrando en costos y reduciendo el riesgo asociado al trabajo manual.

En varios diagramas del diseño se ha utilizado *UML*™ [108] para elaborar los mismos, en algunos otros diagramas del diseño no se usa *UML*™, por considerar más ilustrativo el uso de diagramas de flujo tradicionales.

El diagrama de involucrados en la implementación del proyecto se ilustra en la figura 20. En él se muestra a los principales actores que participan en el proyecto (en la organización patrocinadora) y también de su interacción con el entregable principal del mismo (en la organización usuaria). Esta propuesta de modelo de negocio es sencilla y pretende únicamente ilustrar la utilidad del resultado obtenido en un contexto de negocio.

En las figuras 21 y 22 se presenta el flujo de la información al nivel de la organización usuaria. Como parte de la ejecución de proyectos de carácter informático en la organización usuaria, se derivan tareas de análisis de código en sistemas heredados, el uso de la herramienta que tiene como entrada el código fuente ayuda a perfilar el modelo funcional del sistema actual. Una proyección del uso reiterado del procedimiento desemboca en un repositorio de conocimiento de todos los componentes, mismo que puede ser visto como una extensión del proyecto.

Por último, se elaboró diagrama de comportamiento con los casos de uso para el entregable, el mismo se muestra en la figura 23. En este diagrama se perfilan las principales funciones que debe realizar el procedimiento implementado. Esto es un diagrama de alto nivel del entregable final y de la interacción del mismo con sus usuarios. Se tienen proyectados cuatro paquetes que realicen las funciones del receptor de solicitudes, el paquete que realiza las funciones del compilador, el que realiza la estructuración y reporte del conocimiento y por último el que se encarga de administrar las peticiones de modificación de resultados. El entregable se perfila de forma global como una herramienta útil a un analista de sistemas heredados en *COBOL* para la plataforma mainframe.

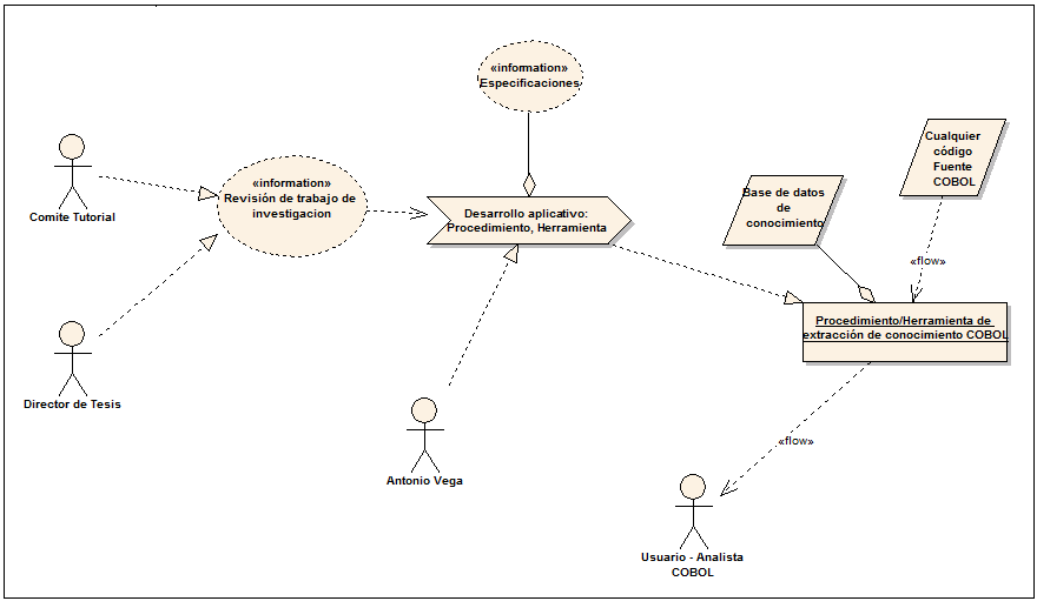


Figura 20 Diagrama de actores involucrados en el proyecto. Patrocinadores y usuarios.

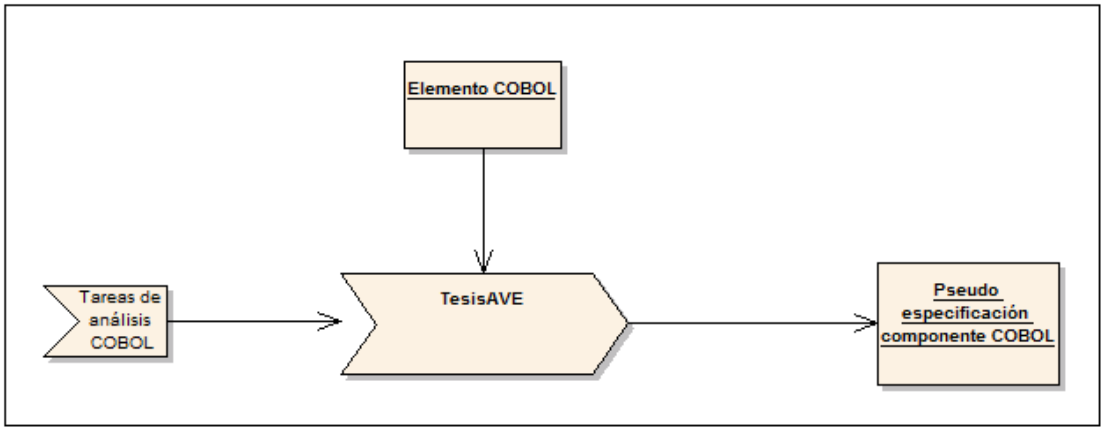


Figura 21 Flujo de negocio.

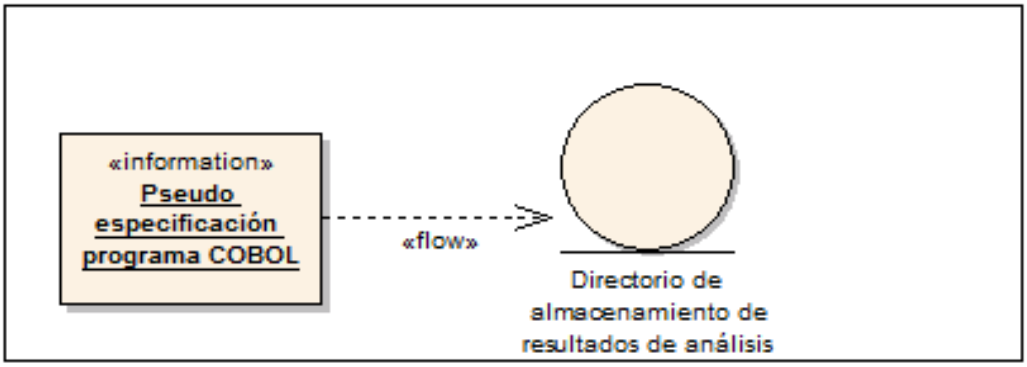


Figura 22 Objetos de negocio.

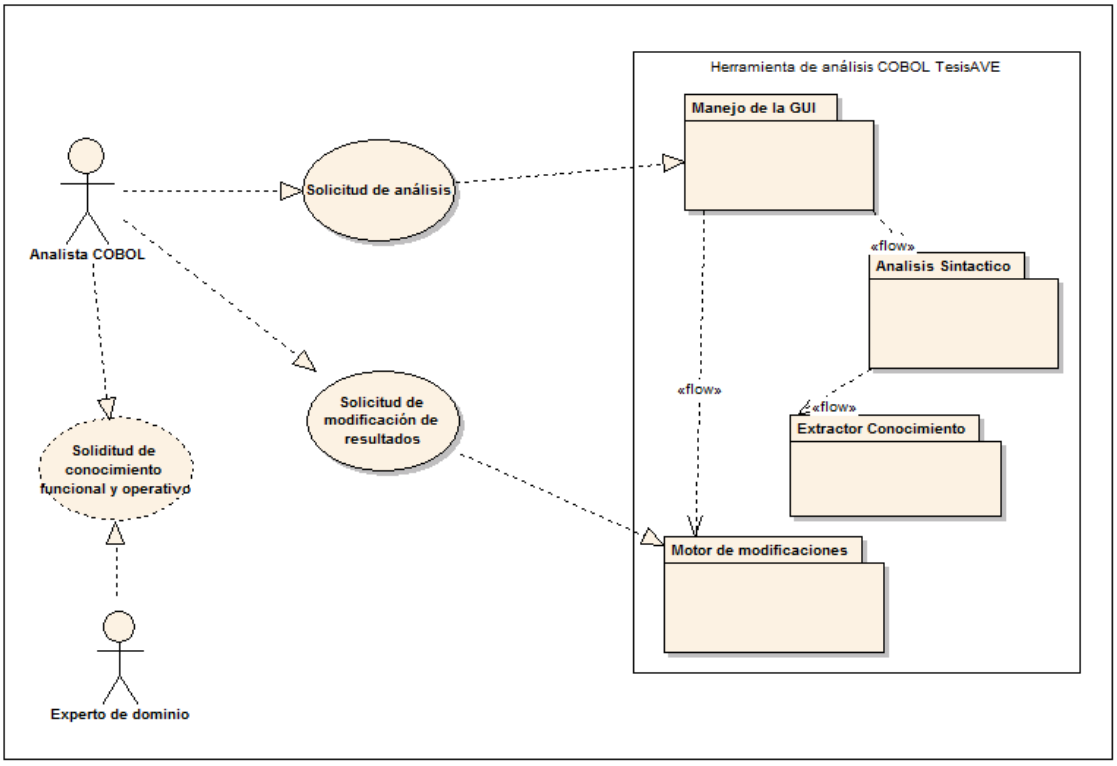


Figura 23 Diagrama de comportamiento del producto entregable.

3.4 ANALIZADOR LÉXICO Y SINTÁCTICO DE COBOL.

La sintaxis del dialecto de *COBOL* frecuentemente utilizado en la plataforma mainframe [34] es la base para construir el analizador léxico y sintáctico, también se propone desarrollar algunas partes del analizador semántico para obtener información útil durante el análisis de un programa, principalmente para los tipos de variables, el control de índices en arreglos y la asignación correcta de valor a las variables.

Como en el artefacto a implementar no se tiene como objetivo principal detectar unidades de compilación sintácticamente válidas, lo cual complica su construcción, sino más bien obtener información que sea útil para un análisis de funcionalidad. Muchas partes que estarían implementadas en un compilador formal no serán rigurosamente implementadas aquí, con esto se quiere decir que, si por ejemplo, una declaración no tiene que ver con la funcionalidad sino más bien con el ambiente de ejecución, esta parte simplemente será ignorada.

Entre las principales funciones del analizador sintáctico a construir tienen importancia especial, las siguientes:

- Traducir los elementos llamados *COPYS*, que son archivos independientes físicamente que contienen código *COBOL* o declaración de variables.
- Identificar los comentarios.
- Eliminar espacios duplicados.
- Obtener elementos/símbolos entre 2 espacios (*tokens*).
- Identificar las cadenas de caracteres que continúan en más de una línea física.
- Recolectar nombres de archivos usados.
- Tratar de agrupar datos al nivel de *COPY* para entidades externas de información y en caso de que no se maneje así su formato, ubicar la estructura interna de datos correspondiente al nivel 01 o la de más alto nivel.
- Calcular para las variables en *Working Storage* su longitud y tipo de registro.
- Identificar secciones.
- Identificar párrafos.
- Identificar frases/oraciones.
- Identificar instrucciones.
- Identificar puntos de finalización del programa.

Las partes del compilador a construir, tendrán una funcionalidad modular en la medida de lo posible apegándose a la teoría tradicional de diseño de compiladores [52], como se muestra en la figura 24.

- El analizador léxico toma como entrada el código fuente del programa (una cadena de caracteres de longitud finita) e identifica uno a uno los símbolos (*tokens*) de la micro estructura del lenguaje: Caracteres válidos, palabras reservadas, nombres de: divisiones, secciones, variables, procedimientos, archivos, párrafos, funciones, módulos ejecutables

externos, caracteres de formato, comentarios, código externo incrustado (*CICS*, *DB2*, Sistema Operativo, *IMS*, etc., como primera aproximación el código incrustado que pertenece a otros lenguajes no será analizado de manera rigurosa). La identificación de componentes léxicos se debe implementar por medio de expresiones regulares, autómatas finitos, catálogos de palabras reservadas o cualquier otra combinación de técnicas. Una vez identificado un *token*, se envía al analizador sintáctico para que evalúe si forma parte de una regla de producción válida de la gramática.

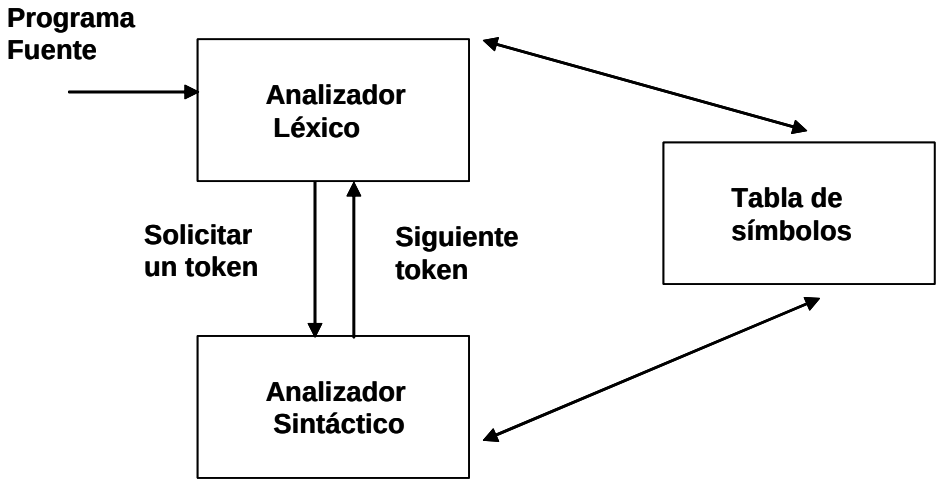


Figura 24 Esquema del analizador sintáctico.

- Durante el análisis de la sintaxis del lenguaje, la técnica más utilizada debe ser el análisis recursivo descendente con retroceso limitado por ser el más sencillo. Es decir, la especificación de las instrucciones debe ser compatible con la forma canónica necesaria para un análisis de este tipo, que es la siguiente:

  $X \rightarrow Y1 \mid Y2 \mid \ldots \mid Ym \mid Z1Z2\ldots Zn; \quad m,n > 0$

  La forma anterior quiere decir que del lado derecho de una regla de producción, deben venir un símbolo terminal o un no terminal seguidos de una concatenación de terminales y no terminales. Con base a esto determinar si una secuencia de derivación de componentes del tipo gramática libre de contexto, deriva en una instrucción válida en la forma Extended Backus–Naur Form (*EBNF*)[52].

```
<Simbolo_Inicial> ::= <Cadena_NoTerminales_Terminales> |
                      <Cadena_NoTerminales_Terminales>
<Simbolo_No_Terminal> ::= <Cadena_NoTerminales_Terminales> |
                          <Cadena_NoTerminales_Terminales>
...
<Simbolo_No_Terminal> ::= <Terminales>
```

  Un ejemplo de una instrucción a derivar en el caso de *COBOL* se observa en la figura 25.

La técnica mencionada debe ser combinada con una arquitectura que facilite la implementación de las reglas de producción en forma de parámetros, con esto se pretende crear una herramienta que pueda tener extensiones para aceptar otros lenguajes que forman parte del código incrustado,

por ejemplo *CICS* y *SQL*, estos tramos de código incrustado no serán analizados como parte de este trabajo de investigación. El diseño propuesto es el de dos tablas de una base de datos, una con los elementos de la micro estructura de un lenguaje *X* y la otra con las producciones sintácticas del mismo lenguaje *X*, entonces el código debe recuperar el axioma del lenguaje y basar el análisis en la validación de las reglas contenidas en los parámetros, mientras se recuperan elementos del código fuente (*tokens*).

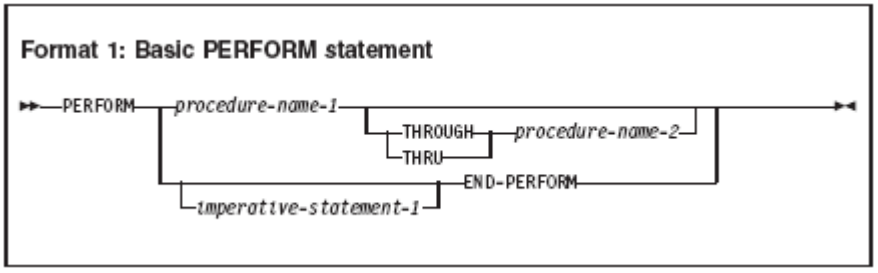


Figura 25 Instrucción PERFORM de COBOL.

La arquitectura funcional propuesta consta de una tabla de palabras reservadas del lenguaje y de una tabla de reglas de producción sintáctica. La primera tabla contiene información del lenguaje al que pertenece la palabra, la identificación asignada a la misma y una clasificación propuesta, que ayudará a identificar dicho símbolo en diferentes niveles de abstracción durante el análisis léxico, sintáctico y posiblemente semántico. El algoritmo principal del extractor de conocimiento se basará en la identificación de las reglas de producción que contienen conocimiento funcional.

El analizador léxico se basa en la lectura de líneas del código, el método inicial es el que atiende la solicitud de entregar un símbolo de la micro estructura del lenguaje, el algoritmo propuesto es el de cargar un buffer en memoria con los elementos contenidos en un determinado número de líneas del código fuente. La carga de dichos símbolos se hace tomando en cuenta los caracteres separadores, posteriormente se tiene un algoritmo que identifica y clasifica el símbolo obtenido de la línea leída, es decir si se trata de un signo de puntuación, una palabra clave de la definición de una función o instrucción, etcétera. El método que identifica los símbolos tiene como uno de sus criterios la consulta a la base de datos, específicamente al catálogo de palabras reservadas del lenguaje. En la figura 26 se muestra el flujo principal del diseño del analizador léxico.

El motor de funcionamiento del analizador sintáctico es un método que identifica los símbolos terminales de una instrucción a través de la consulta de las reglas de producción en una base de datos. La base de datos que contiene las reglas de producción se diseñó de tal manera que puedan ser introducidas en la forma canónica para construir un analizador *LL(1)*, de igual manera existe una catálogo maestro de reglas que determina cuál regla contiene conocimiento funcional y cuál regla puede ser ambigua, en cuyo caso esta tabla, la tabla de decisión proporciona información suficiente para decidir cuál regla es la más conveniente para validar, con las limitaciones *LL(1)*, las reglas de producción que no son predecibles con un símbolo por adelantado caerán en el caso del análisis conocido como técnica de fuerza bruta.

La funcionalidad del analizador sintáctico se basa en un algoritmo de alto nivel que valida una a una las cuatro divisiones de un programa *COBOL*. Para cada una de estas macro validaciones, se administra la petición de validación de reglas de producción y la recuperación de errores, ya sea por regla o por símbolo de referencia. La validación de la sintaxis se diseñó para detectar específicamente reglas de producción que contienen conocimiento importante para el análisis

funcional, tal funcionalidad se incluyó como parámetro en la base de datos. Cada que el analizador detecta una regla con "conocimiento" funcional, la información es estructurada y almacenada en una base de datos de conocimiento para su posterior explotación. La figura 27 contiene el flujo de información del analizador sintáctico.

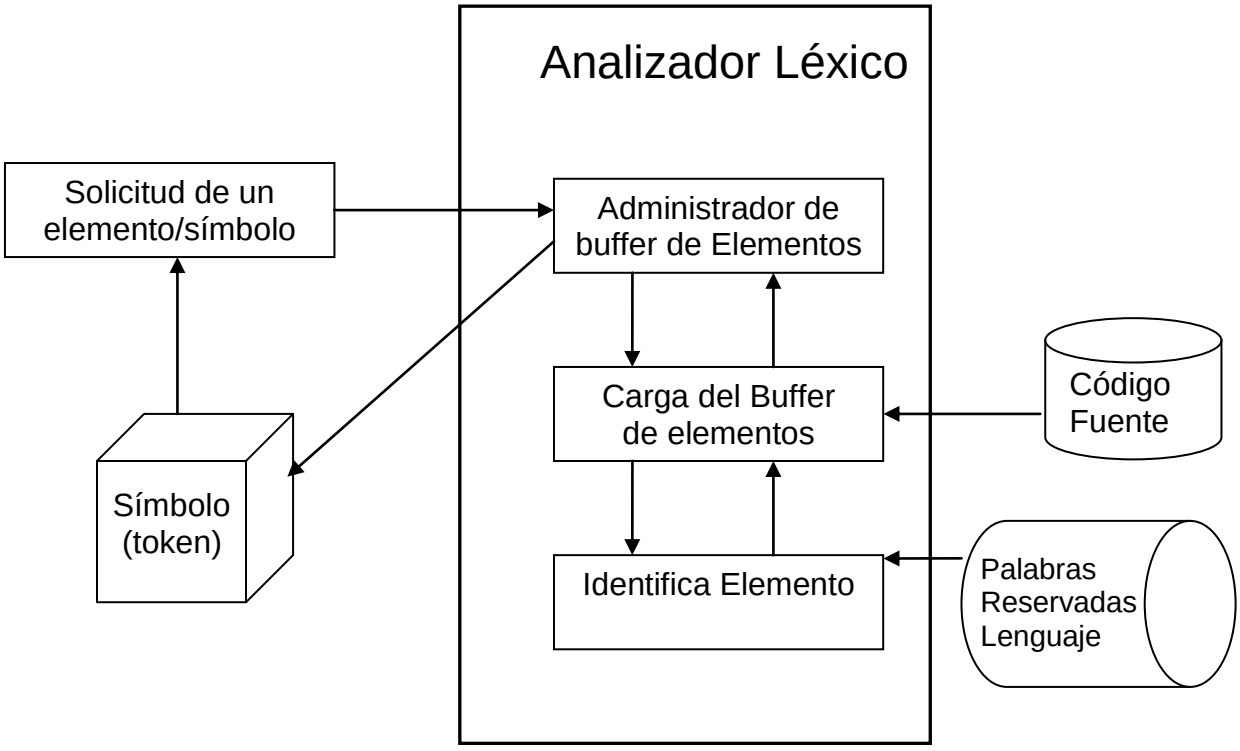


Figura 26 Analizador léxico.

El diseño del analizador sintáctico se basa en gran parte en el reacomodo de las reglas de producción de la gramática, para que con la lectura adelantada de un símbolo sea posible predecir cuál regla de producción debe ser aplicada. La técnica de análisis aplicada sobre la cadena de entrada es de izquierda a derecha tomando el símbolo más a la izquierda de las reglas de producción como referencia, adicionalmente se tiene la lectura por adelantado de un símbolo a la entrada, este es el análisis menos costoso en cuanto a calculo computacional. El lexema de la gramática no se valida en su totalidad, sino que se ha hecho una división emulando a las 4 que componen a un programa en *COBOL*. El algoritmo consiste en un llamado al procedimiento genérico de validación de reglas de producción. Antes de hacer el llamado al mencionado motor genérico de validación de reglas se hace lo siguiente:

Se decide cuál es la producción que se va a validar, tomando como referencia un símbolo leído de la cadena de entrada. Como las reglas de producción se han almacenado de manera que sea posible conocer cuál es el primer símbolo terminal de lado derecho, esto equivale al conjunto *PRIMERO* (regla *N*) y además se ha asegurado que no hay cadenas vacías del lado derecho de las producciones ni tampoco conflicto por símbolos comunes en los conjuntos *PRIMERO*, en cuyo caso se debería solicitar leer más símbolos a utilizar como pre análisis, esto último se evitará en la medida de lo posible, sin embargo cuando las producciones de la gramática no puedan ser adaptadas al análisis *LL(1)* el algoritmo debe tener la flexibilidad de leer hasta 3 símbolos por adelantado para poder predecir la regla de producción a validar, en su defecto la técnica de fuerza bruta aplica.

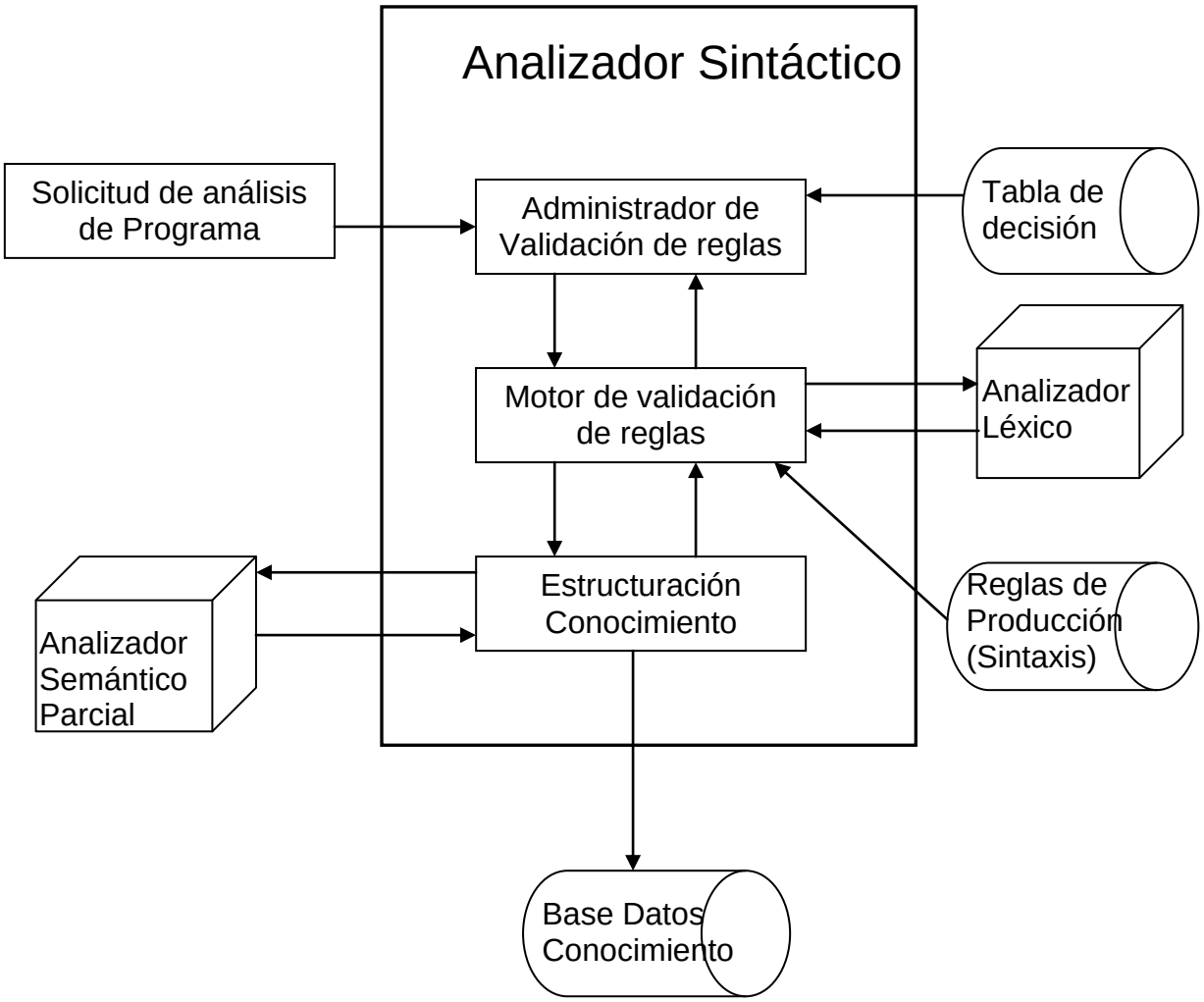


Figura 27 Analizador sintáctico.

En caso de que los símbolos de pre-análisis no coincidan con alguna regla se envía un mensaje de error de sintaxis y el símbolo se ignora, para repetir el proceso con un nuevo símbolo leído. Cada que un símbolo terminal coincide con el símbolo que se ha leído de la cadena de entrada se lee el siguiente elemento de la cadena de entrada y también el siguiente símbolo esperado en la regla de producción. Cuando un símbolo de una regla es un no terminal, entonces aplica un llamado recursivo al algoritmo/procedimiento validador de una regla.

Siempre se debe procurar en la medida de lo posible que la estructura de las reglas de producción tenga la propiedad *LL(1)*: Los conjuntos *PRIMERO* de dos reglas de producción deben ser disjuntos.

En general las reglas de producción de la gramática cumplen con el requisito para ser analizadas como *LL(1)* si dadas dos producciones con el mismo símbolo no terminal del lado derecho, por ejemplo *A*:
$A \rightarrow \alpha$ Y $A \rightarrow \beta$

Se debe de cumplir que $PRIMERO+(\alpha) \cap PRIMERO+(\beta) = \varnothing$

Donde *PRIMERO*+ (α) = *PRIMERO* (α) ∪ *SIGUIENTE* (α), si ε ∈ *PRIMERO* (α)
= *PRIMERO* (α) en cualquier otro caso.

PRIMERO (α) = Es el conjunto de símbolos de la gramática que aparecen como el primer símbolo de una cadena que deriva de α. Esto es **x** ∈ *PRIMERO* (α) si y solo si α ➔* **x**γ para alguna γ.
SIGUIENTE (α) = Es el conjunto de todas las palabras de una gramática que pueden aparecer inmediatamente después de una cadena α.

Si se toma en cuenta lo anterior, un algoritmo simple que reconoce un nodo del lado izquierdo del árbol de expansión de la regla es:

Consideremos A ➔ β1|β2|β3
Con $PRIMERO^{+}$ (β_1) ∩ $PRIMERO^{+}$ (β_2) ∩ $PRIMERO^{+}$ (β_3) = ∅

/* Buscar una A en la cadena de entrada – el código fuente */
si (*palabra_actual* ∈ *PRIMERO* (β_1))
encontrar una β_1 y devolver verdadero
caso contrario si (*palabra_actual* ∈ *PRIMERO* (β_2))
encontrar una β_2 y devolver verdadero
caso contrario si (*palabra_actual* ∈ *PRIMERO* (β_3))
encontrar una β_3 y devolver verdadero
caso contrario
reportar un error y devolver falso

El diseño de la validación e identificación de todas las reglas de producción se basa en la siguiente estrategia:

- Se colocan las reglas de producción de la gramática en una base de datos con un formato que tiene una estructura que puede ser validada por un algoritmo genérico.
- Se utiliza un algoritmo tipo motor genérico para analizar de manera común y recursiva las reglas de producción almacenadas en la base de datos como parámetros.
- La tabla de parámetros (entidad = regla_produccion) debe tener un renglón por cada símbolo no terminal de las reglas de producción, cada no terminal tendrá tantas columnas/atributos como símbolos terminales/no-terminales tenga el lado derecho de la regla de producción que lo deriva.
- El algoritmo de análisis será controlado por la tabla de parámetros y la solicitud de lectura de la cadena de entrada.
- Adicionalmente se han agregado atributos a la tabla mencionada, los cuales forman parte del diseño del extractor de conocimiento, en dichos atributos estará indicado qué tipo de conocimiento funcional almacena la regla y también en qué entidad de la base de datos de conocimiento funcional deberá ser almacenada la información extraída de la regla.
- Se ha agregado una entidad predictiva a la base de datos para controlar casos especiales donde las producciones de la gramática no permiten un análisis *LL(1)*, en ese caso los campos y renglones en la tabla de producciones simulan la lectura por adelantado de más símbolos de entrada, cuando se necesitan más símbolos de entrada para elegir la regla de producción a igualar con la cadena de entrada o bien concluir que se trata de un error de sintaxis.

El algoritmo del motor genérico de análisis, basado en la tabla de parámetros es el siguiente:

token ← siguiente_elementoléxico() – analizador léxico
push *FDA* en la Pila (**F**in **D**e **A**rchivo/cadena de entrada)
push el símbolo inicial, ***S*** en la Pila, el lexema, tomado de la tabla de parámetros.
CDP ← Elemento superior, **C**abecera **D**e la **P**ila
CICLO infinito
    Si *CDP = FDA* Y *token = FDA* entonces
        Terminar ciclo y reportar proceso exitoso, regla aceptada.
    Caso contrario - Si *CDP* es un símbolo terminal, entonces
        Si *CDP* es igual a *token*, entonces
            Pop Pila. *CDP*, símbolo terminal, elemento reconocido
            token ← siguiente_elementoléxico() – analizador léxico
        Caso contrario
            Reportar error al analizar *CDP*
    Caso Contrario //*CDP* es un no-terminal
        Si TABLA[*CDP, token*] es A → $B_1B_2…B_k$ entonces
            Pop Pila //Desechar símbolo A
            Push B_k, B_{k-1}, … . B_1 //En orden inverso, se apilan los símbolos del NT.
        Caso Contrario
            Reportar error al expandir *CDP*
    CDP ← Elemento superior, **C**abecera **D**e la **P**ila.
Fin CICLO infinito

Este algoritmo es de carácter genérico, por lo que los detalles de la construcción implementada no están incluidos, sin embargo un bosquejo de tales detalles se describe a continuación:

- El símbolo inicial también estará almacenado en la tabla de reglas de producción.
- La pila no será construida explícitamente sino que el programa a través del llamado al procedimiento genérico y recursivo estará creando una pila de llamados con la respuesta buscada, es decir si la cadena de entrada iguala las derivaciones de las reglas de producción de los no símbolos no terminales, entonces se indica mediante la activación de una bandera que la regla de producción ha sido aceptada.
- Se debe agregar además un proceso de manejo de errores, el cual decide qué se puede ignorar y en qué momento se debe detener la ejecución del análisis.
- Cada que una regla sea aceptada, la misma definición de la regla en la tabla contiene información sobre el conocimiento funcional que puede ser obtenido de dicha regla, así como el destino de dicha información funcional.
- Cuando una regla de producción o un conjunto de ellas no pueden ser reacomodadas para cumplir el requisito de una gramática *LL(1)* se debe agregar un paso previo de decisión, este paso se basa en una tabla que contiene las reglas que son ambiguas en sus primeros símbolos. La decisión sobre cuál regla se debe igualar con la cadena de entrada se basa en tres atributos de la tabla mencionada: una secuencia de intento de validación, los símbolos que son ambiguos y la propuesta de un siguiente intento de regla ambigua a validar (cuya ausencia indica un error de sintaxis). Este procedimiento se propone únicamente donde inclusive hasta 4 símbolos (o más) leídos por adelantado no son capaces de determinar sin ambigüedad cual regla debe ser igualada con la cadena de entrada. Un ejemplo representativo de esta situación es la instrucción *DIVIDE* en el dialecto *COBOL* que es objeto de esta implementación, para

esta instrucción existen 5 variantes y la lectura de hasta 4 símbolos de la cadena de entrada no es suficiente para determinar de cuál variante se trata con plena certeza.

- El paso descrito previamente es una optimización en el rendimiento del analizador, pero la ausencia de tal algoritmo predictivo (ante un escenario de recorte del entregable) debe tener un procedimiento de defecto y ese procedimiento es el algoritmo de fuerza bruta. Si un primer símbolo relativo leído de la cadena de entrada es común en varias reglas de producción candidatas de la gramática, la ausencia de un algoritmo predictivo de *N* símbolos debe hacer que el algoritmo de defecto intente el amarre contra todas las reglas candidatas con el respectivo retroceso en la lectura ante cada intento fallido.

Existen dos tablas núcleo del analizador, una primera se utiliza para decidir qué regla igualar, con base a símbolos leídos de la cadena de entrada, la segunda es el detalle completo de una regla de producción que tiene un no terminal como lado derecho. El diseño de las mismas se puede observar en las figuras 28 y 29.

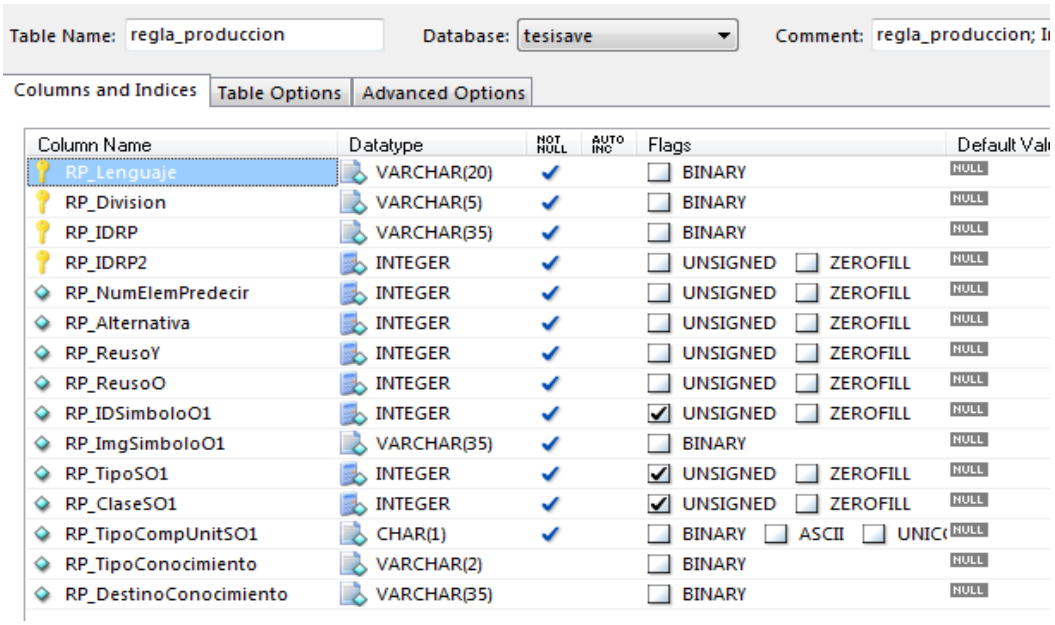


Figura 28 Diseño de la tabla regla_produccion.

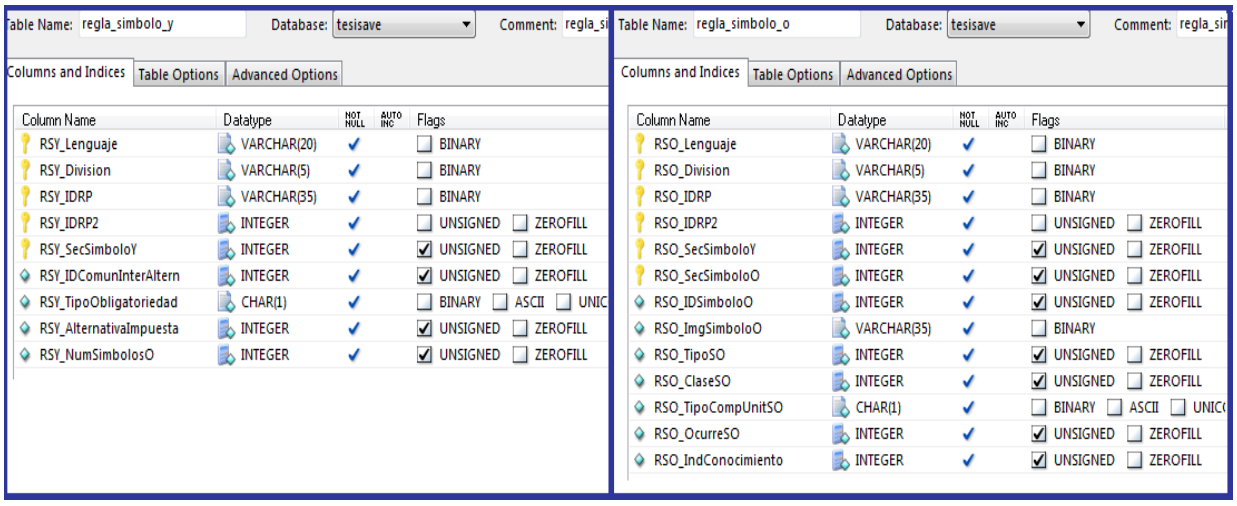


Figura 29 Diseño de las tablas regla_simbolo_y y regla_simbolo_o.

Cuando sea necesario se deben desarrollar validaciones de tipo semántico, como por ejemplo del tipo de variables involucradas en una operación aritmética o sobre el tamaño en el movimiento de valores entre variables y sobre todo en estructuras de datos de las llamadas variables de grupo de *COBOL*, que implican múltiples movimientos en cascada para variables padre. Las validaciones de tipo semántico se evitarán en la medida de lo posible, ya que la validación rigurosa queda fuera de alcance del objetivo del presente trabajo.

En la figura 30 se muestra el lexema para la *PROCEDURE_ DIVISION* de un programa *COBOL*.

Resultset 1

RSO_Lenguaje	RSO_Division	RSO_IDRP	RSO_IDRP2	RSO_SecSimboloY	RSY_TipoObligatoriedad	RSO_SecSimboloO	RSO_IDSimboloO	RSO_ImgSimboloO
COBOL	PD	PROCEDURE_DIVISION	1	1	O	1	402	PROCEDURE
COBOL	PD	PROCEDURE_DIVISION	1	2	O	1	206	DIVISION
COBOL	PD	PROCEDURE_DIVISION	1	3	P	1	0	USING_CLAUSEPD
COBOL	PD	PROCEDURE_DIVISION	1	4	P	1	0	RETURNING_CLAUSE
COBOL	PD	PROCEDURE_DIVISION	1	5	O	1	3	.
COBOL	PD	PROCEDURE_DIVISION	1	6	P	1	0	DECLARATIVES_CLAUSE
COBOL	PD	PROCEDURE_DIVISION	1	7	O	1	0	SECTPARA_CLAUSE

Figura 30 Definición del lexema: PROCEDURE DIVISION.

El diseño del proceso propuesto para detectar líneas de comentarios que son candidatas a describir funcionalidad dentro del código consiste en tres partes principales, que son ilustradas en la figura 31.

Primero. Las primeras líneas de comentario encontradas durante el análisis sintáctico deberán ser consideradas, porque la mayoría de los programas *COBOL* tienen ahí descripciones acerca de la funcionalidad del programa. Las líneas de comentarios que aparecen antes de la definición de un párrafo o sección en la *PROCEDURE DIVISION* del programa también son candidatas a describir la funcionalidad de bloque de código correspondiente. Las líneas de comentario que preceden a la definición de una entidad de información de entrada o de salida, son candidatas a contener información de dominio.

Segundo. Una línea de comentario preseleccionada como candidata a tener conocimiento funcional se somete a un proceso de análisis para estrechar la probabilidad de que contenga texto no útil. El proceso de análisis consiste en asignar una calificación que indica una probabilidad de que el texto en la línea de comentario tenga contenido en lenguaje natural, esto estaría limitado a inglés y español. Se define también un umbral basado en el número de elementos encontrados en la línea para determinar que hay una buena probabilidad de que el texto es útil. Otras discriminaciones aplican, por ejemplo si la línea solo contiene espacios o asteriscos deberá ser descartada. El análisis se basa en tres algoritmos que asignan una calificación parcial.

El primero consiste en dividir la cadena de caracteres de una línea de comentarios en grupos de 3 letras, cada grupo obtenido se busca en una tabla que contiene las secuencias de caracteres más usadas para un idioma, por ejemplo español. Se pondera el número total de grupos en la línea, el número de coincidencias, su probabilidad estadística de aparición en el idioma en tratamiento y se asigna la primera calificación parcial.

El segundo algoritmo consiste en buscar cada palabra en la línea de comentarios en un diccionario de las palabras más usadas para el idioma en tratamiento, de la misma manera el número total de palabras en el texto contra el número de coincidencias determina una segunda calificación parcial.

Un tercer algoritmo proporcionará una calificación negativa, se definirá un conjunto de expresiones regulares que se igualan al código *COBOL* con los verbos más usados, cada coincidencia resta puntos a la calificación porque se incrementa la posibilidad de que la línea sea código *COBOL* comentado. Una vez obtenida esta tercera calificación parcial se hace una suma aritmética de las tres y se almacena.

El mismo procedimiento aplica para el segundo idioma, inglés, los pesos asignados a cada coincidencia en los algoritmos se definen dinámicos para que puedan ser ajustados durante las pruebas en caso de producir resultados con mucho margen de error. Si cualquiera de las dos calificaciones supera el umbral definido para idioma español o inglés, esto quiere decir que el texto ha sido identificado como texto en inglés o español y por lo tanto se marca como candidato a descripción de funcionalidad.

Tercero. Esta implementación es un ejemplo sencillo de cómo seleccionar información dentro del programa que pueda ser útil cuando, es importante señalar que al ser líneas de comentario la fidelidad de la descripción obtenida contra la funcionalidad real en tiempo de ejecución del programa puede ser distinta, es por eso que se debe tener la funcionalidad de editar el resultado.

Es muy probable que una línea de comentario que no es código comentado sea una descripción funcional del programa, sin embargo es probable también que muchas veces estos comentarios contengan información inexacta o desactualizada, la parte automática no será implementada con algoritmos que determinen que dichos comentarios tienen información exacta. Esta es una de las partes que se podrán modificar manualmente y cuya implementación depende de las limitantes en recursos del proyecto.

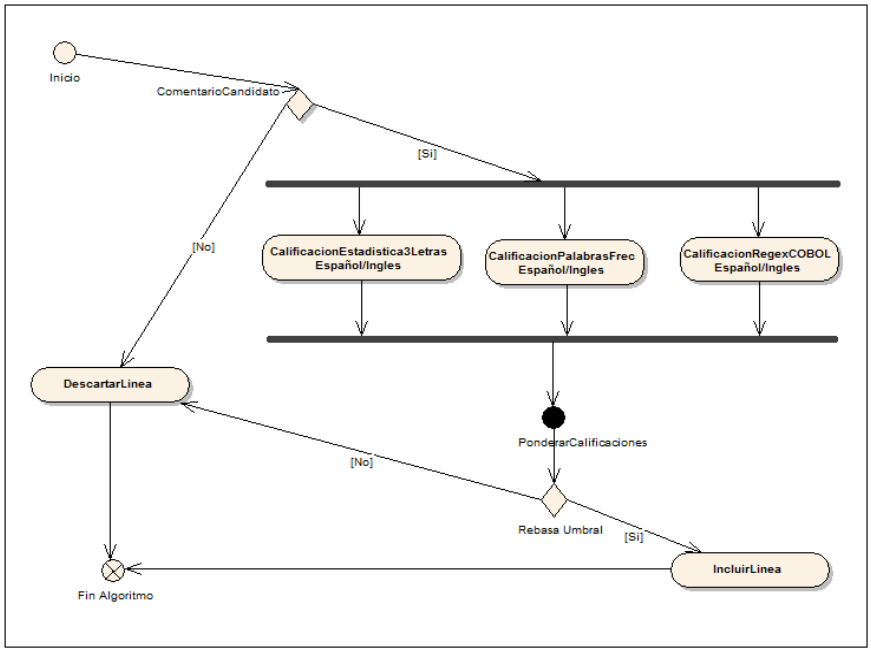


Figura 31 Diagrama de actividad para el procedimiento que selecciona líneas de comentarios candidatas.

3.5 ANALIZADOR DE CONTROL DE FLUJO DEL PROGRAMA Y DE FLUJO DE DATOS.

La extracción del control de flujo de un programa forma parte importante de los resultados de la herramienta automática a construir. El flujo de ejecución de un programa aporta conocimiento para entender su funcionalidad. Una vez implementado el analizador sintáctico de *COBOL*, se deben identificar los elementos que forman parte del flujo del programa y almacenarlos en una base de datos para reportar posteriormente dicho conocimiento.

Cuando se hace el análisis sintáctico de las instrucciones, como parte de las propiedades definidas para las mismas, se hace una definición previa en la base de datos acerca de los símbolos en la producción que determinan saltos en el flujo de ejecución. El resultado con el conocimiento a recolectar está limitado al análisis estático del compilador, por lo que el gráfico de flujo a obtener es una aproximación estática. Al ser *COBOL* un lenguaje imperativo el resultado obtenido utilizando esta técnica debe ser bastante confiable.

El conocimiento extraído del programa relativo al flujo de ejecución de las instrucciones es una gráfica, donde los nodos representan una o más instrucciones que se ejecutan de manera secuencial y sin condicionamiento y por otro lado las aristas de la gráfica representan los saltos en el control de ejecución, como por ejemplo instrucciones que determinan saltos condicionados o no condicionados. El diseño se basa entonces en la construcción de la gráfica de control de flujo a partir de elementos del lenguaje extraídos durante el análisis sintáctico, a continuación se enuncian varias consideraciones para construir la gráfica.

La primera parte de la gráfica de control de flujo a identificar es el punto de inicio del programa, así como los puntos en los que un programa finaliza. Un bloque básico es una secuencia de instrucciones consecutivas en las que el flujo de control entra al principio y sale al final sin detenerse y sin la posibilidad de saltar fuera del bloque, excepto al final.

Se define una cabecera que identifica a un bloque de código, como la primera instrucción de un bloque básico. De esta manera, se tiene de inicio a la primera instrucción de un bloque como la etiqueta de cabecera, sin embargo también una etiqueta destino de un salto *GOTO* es una cabecera de bloque y cualquier instrucción que va después de un salto *GOTO* es una cabecera.

Cada bloque básico consiste en la cabecera y en todas las demás instrucciones, hasta la siguiente cabecera, pero sin incluir a esta última. Como el código de entrada es *COBOL*, todos los procesos son ejecutados en la *PROCEDURE DIVISION*, entonces el diagrama de control de flujo debe contener instrucciones de esta división del programa.

Si es un diagrama detallado, debería contener cada instrucción como un nodo. Sin embargo una versión recortada del diagrama de control de flujo tendría como nodos bloques de instrucciones. Todas las instrucciones sucesivas de categorías tales como *MOVE*, *ADD*, *ACCEPT*, etc. que pueden incluir ya sea un salto o una ramificación del flujo se identifican como el código

secuencial. Otras instrucciones como *IF-ELSE*, *EVALUATE*, *PERFORM* y *GOTOs* pueden ser identificadas como bloques diferentes y también como instrucciones en las cuales el código continuo termina.

La herramienta debe elaborar y almacenar la grafica de control de flujo con todo el nivel de profundidad y detalle posible, se ha elegido el método tabular, para lo cual se han diseñado las dos tablas recomendadas para almacenar una gráfica dirigida, la de nodos de la gráfica y la de aristas. El diseño de las mismas se puede ver en las figuras 32 y 33.

El reporte de la gráfica de control de flujo se propone en forma de texto con sangrías, de acuerdo al nivel de profundidad lógica de los llamados a bloques de código dentro del programa, en esta presentación cada nodo es el nombre o título de un bloque de código, se colocan dentro del reporte también las instrucciones que provocan saltos en el flujo del programa. A continuación se ilustra un ejemplo de cómo debería quedar un reporte del flujo de ejecución. Si bien puede parecer sencillo el diseño, un reporte de este tipo extraído de manera automática es de gran ayuda a un analista que se encuentra en proceso de analizar un componente *COBOL* que contiene miles de líneas.

Se presenta también un prototipo de la representación del control de flujo de un programa en el reporte de salida. Este formato puede parecer poco amigable, sin embargo un analista *COBOL* encuentra de gran utilidad este tipo de resúmenes que indican la secuencia de ejecución de bloques de código.

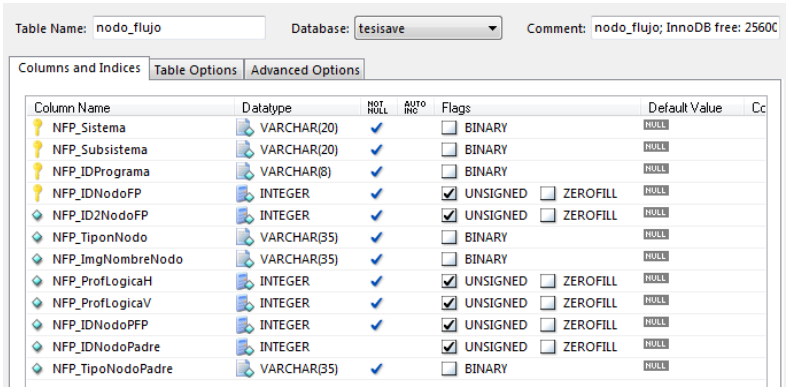


Figura 32 Definición de la tabla de nodos de la gráfica de nodo de flujo.

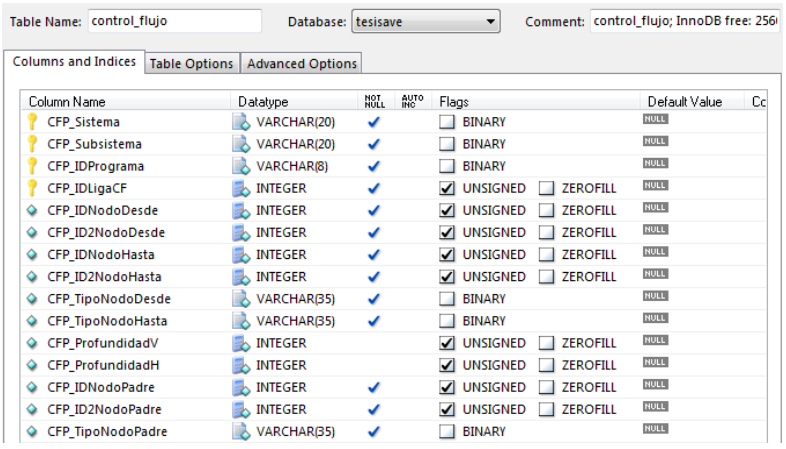


Figura 33 Definición de la tabla de aristas de la gráfica de control de flujo.

Prototipo de reporte de control de flujo.

0000-PARRAFO-INICIAL
  Bloque1
  IF *expresion1*
    Bloque2
    Ejecuta 9000-PARRAFO-COMUN1
    Ejecuta 9900-PARRAFO-COMUN2
  FIN-SI
  IF *expresion2*
    Siguiente Oración
  ELSE
    Bloque3
    Ejecuta 9000-PARRAFO-COMUN1
  FIN-SI
  Ejecuta 1110-PARRAFO-LECTURA
  IF *expresion4*
    IF *expresion5*
      Bloque4
    Caso Contrario
      Bloque5
    FIN-SI
    Ejecuta 9900-PARRAFO-COMUN2
  FIN-SI
  Ejecuta 0800-PROCECIMIENTO HASTA *Fin-Archivo*
    Bloque6
    IF *expresion6*
      Bloque7
    ELSE
      Bloque8
    FIN-SI

El análisis de flujo de datos de datos es una técnica para obtener información del conjunto de valores posibles, calculados en varios puntos de un programa. La gráfica de control de flujo del programa se utiliza para determinar aquellas partes del mismo en las cuales un valor particular asignado a una variable puede ser propagado. Una manera simple de hacer un análisis de flujo de datos es el de definir ecuaciones de flujo de datos para cada nodo de la grafica de control de flujo y resolverlas calculando repetidamente la salida a partir de la entrada en cada nodo hasta que el sistema completo sea estable, esto es que se alcance un punto invariante o fijo.

La herramienta a construir no realizará un análisis formal del flujo de datos, para fines prácticos sólo se hará una aproximación del valor final de cada variable después de una pasada en el análisis estático. El diseño de esta parte es una tabla en la base de datos de conocimiento, la cual proporciona información sobre advertencias que se sugiere reportar, tales como variables no inicializadas y el conjunto de valores que pueden almacenar ciertas variables asociadas a las entidades externas de información que utiliza el programa en análisis. El diseño de la tabla de almacenamiento de variables considera principalmente las variables de tipo estructura de *COBOL*. Los principales atributos de la tabla de flujo de datos en la herramienta a desarrollar son el nombre de la variable, el bloque de código donde sufre modificaciones y un identificador de flujo

que es de utilidad cuando el valor depende del flujo de ejecución, se almacenan valores dependientes del flujo de ejecución relativos a un bloque de código. El diseño de la tabla de datos se muestra en la figura 34.

```
--
-- Definition of table flujo_dato
--
DROP TABLE IF EXISTS flujo_dato;
CREATE TABLE flujo_dato (
  FDP_Sistema varchar(20) NOT NULL,
  FDP_Subsistema varchar(20) NOT NULL,
  FDP_IDPrograma varchar(08) NOT NULL,
  FDP_IDVariable integer unsigned NOT NULL,
  FDP_IDFlujoDato integer unsigned NOT NULL,
  FDP_Accion char(01) character set ucs2,  --Generador, Modificador_Terminador, Usuario
  FDP_IDBloque integer unsigned,
  FDP_TipoValor char(01) character set ucs2 NOT NULL, --Valor asignado en el bloque de referencia, en variable esta el final de la pasada estática
  FDP_Valor_X varchar(100),
  FDP_Valor_N float,
  PRIMARY KEY  (FDP_Sistema,FDP_Subsistema,FDP_IDPrograma,FDP_IDVariable,FDP_IDFlujoDato),
  KEY Index_2 (FDP_IDVariable,FDP_Accion)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='flujo_dato';
```

Figura 34 Definición de la tabla flujo de dato.

3.6 EXTRACCIÓN BÁSICA DE REGLAS DE NEGOCIO.

Los algoritmos para extraer las reglas de negocio a partir de la información que se obtiene del compilador pueden ser demasiado complejos y quedan fuera del alcance de este trabajo. Ante la limitante de tiempo y recursos a invertir, se incluirán tres algoritmos sencillos como ejemplo aplicativo, mismos que ilustran como se pueden inferir reglas de negocio de manera práctica y que el resultado de los mismos proporciona conocimiento útil sobre la funcionalidad de un programa.

El primer algoritmo para extraer componentes de una regla de negocio consiste en identificar cuando una estructura de datos asociada a una entidad de información externa sufre actualización en sus valores. Si al menos 3 componentes de una estructura de datos de este tipo son actualizadas en un bloque de código, este bloque se marca como candidato a implementar una regla de negocio, parte de la información a reportar incluye al nombre de la entidad de información que es manipulada.

El segundo proceso de identificación consiste en reportar los programas externos que son invocados dentro de un bloque de código, cada que sea identificado el llamado a un módulo externo, el bloque de código que contiene el llamado se reporta como candidato a regla de negocio.

Finalmente, dentro de bloques de código que tienen los niveles de ejecución lógica más altos en profundidad, se deben identificar instrucciones de bifurcación que lleven a:

- Llamado de módulos externos.
- Bloques de código que modifican valores de una estructura de datos que pertenece a una entidad de información externa.

- Operaciones *CRUD* sobre una entidad de información externa, archivos o tablas de base de datos.

Estas instrucciones de bifurcación y los nombres de los bloques de código que se ejecutan como resultado de las bifurcaciones son también candidatos a reglas de negocio. En la figura 35 se muestra el diagrama de comportamiento que ilustra de manera general el flujo de utilización de los tres algoritmos que se implementaran de manera paralela al análisis sintáctico.

Parte del diseño consiste también en reportar operaciones con más de tres modificaciones simples sobre una misma variable, esto forma parte de la inclusión de conocimiento experimental. Normalmente operaciones más complejas que un simple incremento sobre una misma variable perfilan variables de dominio, por ejemplo calculo de tasas de interés, facturación de conceptos, calculo de impuestos, etcétera.

La solución propuesta para cumplir el objetivo enunciado en este punto consiste en colocar tres catálogos de información (extraída del compilador) en forma de una marca que informe el destino del conocimiento a reportar durante el análisis. Estos catálogos de destino son: el flujo de control, la tabla de variables con la bitácora de modificaciones de valores y las tablas de entidades de información externa (bases de datos, archivos y módulos externos). Como la información es obtenida durante el proceso de compilación, la adición de atributos a las tablas de definición de reglas (sintaxis) simplifica y optimiza el algoritmo.

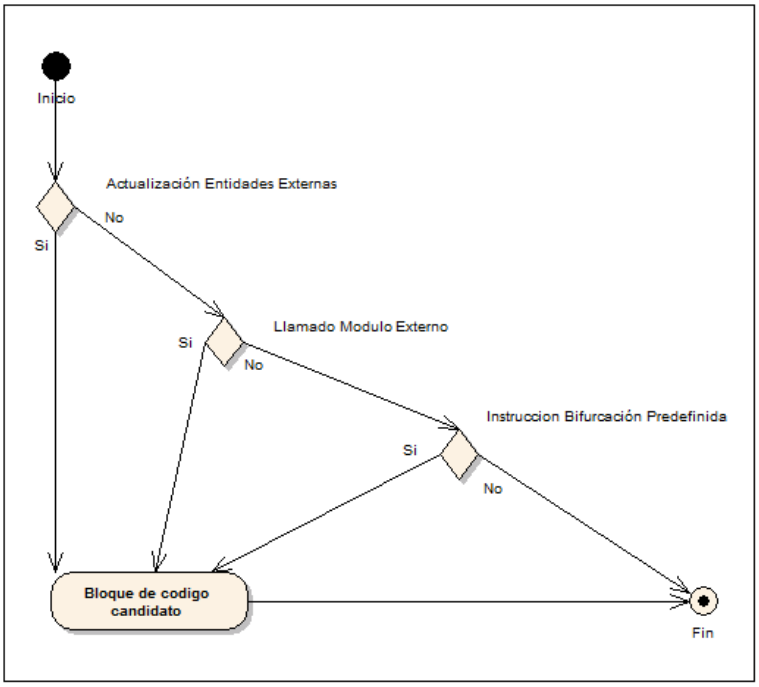


Figura 35 Diagrama de actividad para la identificación de reglas de negocio candidatas.

Cuando el algoritmo propuesto detecta un tramo de instrucciones dentro del código que son candidatas a implementar una regla de negocio, se propone una seudo-traducción antes de presentar el conocimiento encontrado. Como los verbos de las instrucciones en *COBOL* son palabras en inglés se debe implementar una tabla de la base de datos que servirá de guía para hacer una traducir los verbos que forman parte de la candidata a regla de negocio, por ejemplo un

verbo "*MOVE*", puede ser transformado por la palabra "*Mover*", con esto se pretende dar una presentación amigable del conocimiento, en la figura 36 se muestra el diagrama de actividad.

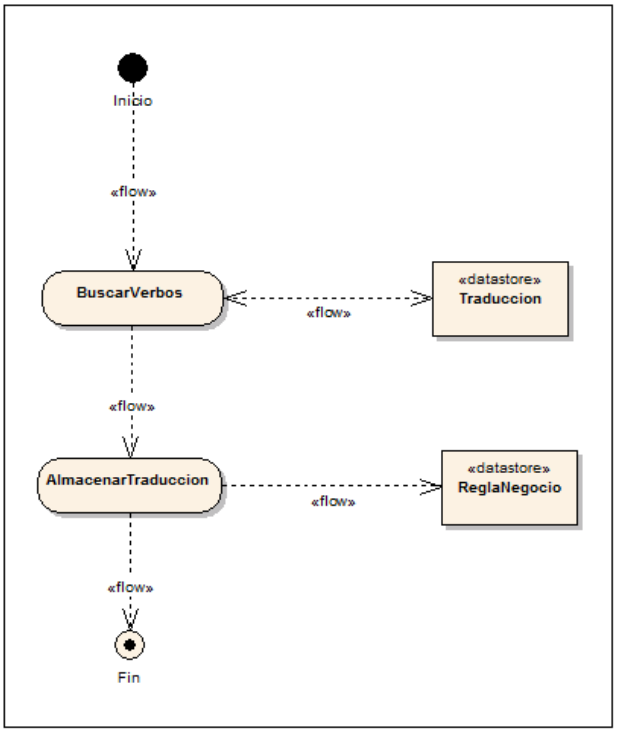


Figura 36 Diagrama de actividad para traducir verbos COBOL contenidos en instrucciones candidatas a reglas de negocio.

3.7 ESQUEMA DE LA BASE DE DATOS Y DIAGRAMA DE CLASES DEL SOFTWARE A IMPLEMENTAR.

Se presenta en la figura 37 el diagrama de Clases del software implementado (analizador sintáctico de *COBOL* y el normalizador de conocimiento), el cual se basa en tres clases principales: *AnalisisLexico*, *AnalisisSintactico* y *Conocimiento*. Estas Clases hacen uso de diversos objetos para acceder a la base de datos (obtener parámetros y agregar conocimiento), al código fuente (leer líneas del programa) a los archivos de trabajo (errores y depuración tipo trazado del análisis) y también para solicitar algoritmos diversos como validar expresiones regulares, manejo de colecciones de objetos, resultado parcial de emparejamiento, reinicio por reglas ambiguas, etcétera. En el **Anexo C** se presenta un acercamiento del diagrama de clases.

La base de datos de parámetros-conocimiento diseñada se muestra en la figura 38. Los *Parámetros* definen las reglas de producción de la gramática del dialecto *COBOL* analizado, estos parámetros son los que dirigen el análisis sintáctico y determinan dónde hay conocimiento funcional. En las entidades de *Conocimiento* se almacenan de forma normalizada todos los símbolos que contienen descripción de funcionalidad del programa en análisis para que sean reportados posteriormente de manera amigable.

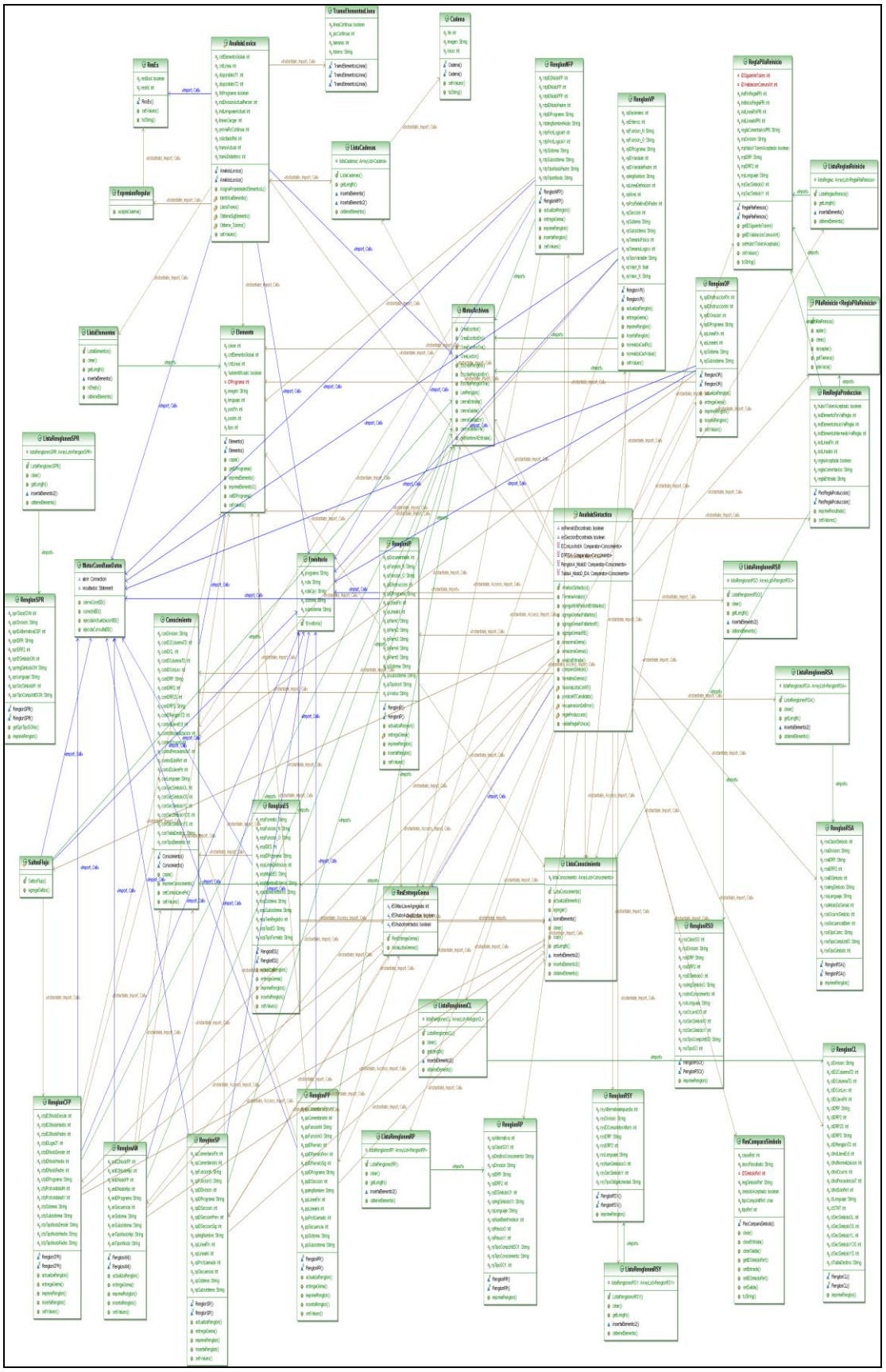


Figura 37 Diagrama de clases del artefacto de software a construir.

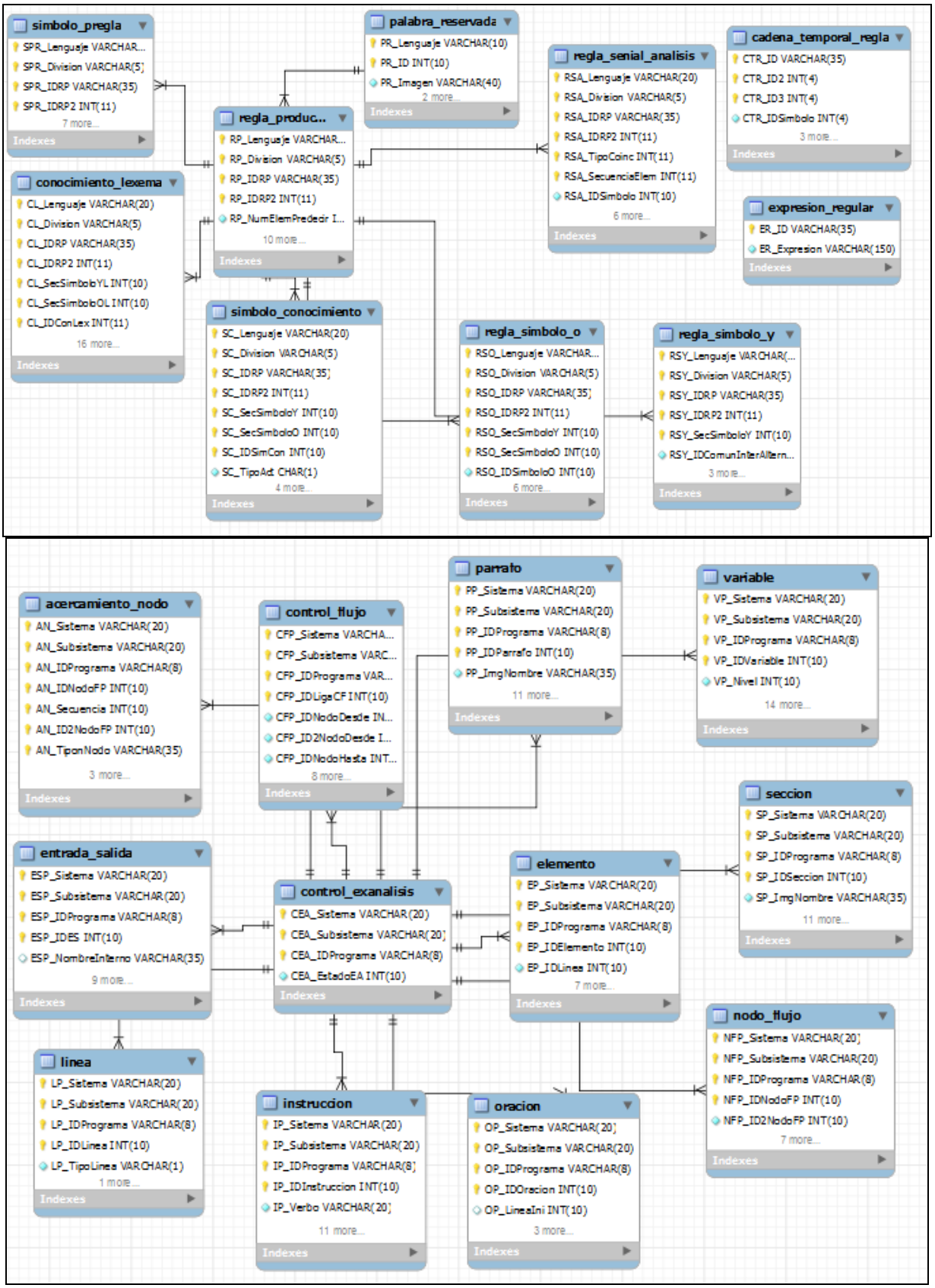


Figura 38 Esquema de la base de datos de parámetros y de conocimiento de la herramienta automática.

3.8 REPORTE DE LOS RESULTADOS.

La manera de explotar el conocimiento obtenido (y almacenado) por la herramienta automática, es elaborar una salida con la ayuda de imágenes prediseñadas o texto formateado para expresar de manera gráfica el elemento analizado con sus entidades de información externa de entrada y salida y además una cierta representación resumida de los principales procedimientos y como es el camino de ejecución de algunos flujos lógicos de ejecución hasta la salida del programa.

Una de las partes más importantes del diseño es la interfaz gráfica de usuario. El objetivo del diseño de la interfaz de usuario es el de hacer la interacción con el usuario lo más simple y eficiente posible, en términos de cumplimiento de los objetivos sobre los que fue basado el producto final a obtener.

Con base en el conocimiento almacenado por el autómata en la base de datos (fases previas de la investigación), se debe elaborar una salida con la ayuda de imágenes prediseñadas o texto formateado para ilustrar de manera gráfica o esquematizada información sobre el elemento analizado. El resultado del análisis se perfila como una aproximación a la especificación original del programa, los principales atributos a mostrar son las entidades de información externa de entrada y salida, información sobre componentes externos importados, el flujo de control interno del componente, las validaciones implementadas e información adicional, tal como algunas reglas funcionales implementadas, sugerencias sobre estandarización de código y comportamiento del flujo de datos. Es importante hacer notar que se implementarán ejemplos de cómo se puede ser explotada la información obtenida por el compilador, dando importancia al diagrama de entidades y al control de flujo lo cual de manera razonable proporciona conocimiento al analista sobre la principal funcionalidad de un componente. El razonamiento anterior cubre el objetivo de la investigación: de manera semiautomática se obtiene conocimiento útil a un analista *COBOL*, el cual tiene riesgo mínimo de contener errores, además el analista tendrá la capacidad de modificar los resultados de la parte semiautomática. La agregación de más características a los resultados está limitada en recursos y tiempo, básicamente a la restricción en fecha para que el presente trabajo de investigación sea entregado para su revisión y evaluación.

La primera interfaz de comunicación con el usuario de la herramienta es la forma de solicitud de análisis de un elemento *COBOL*. El diseño del formulario de solicitud se ilustra en la figura 39. Los componentes de la pantalla de comunicación con el usuario son sencillos, únicamente se solicita el nombre del programa a analizar y también se deja preparada la herramienta para aceptar la ruta de los componentes adicionales tales como la declaración de los formatos de las tablas de base de datos utilizadas y también de los componentes llamados *COPYBOOKs* (código *COBOL* almacenado en un archivo separado, que no compila de manera independiente) y que pueden ser incluidos por medio de una instrucción *COBOL* en el programa. La herramienta tiene por defecto un directorio de trabajo y ahí deberán ser almacenados el código a analizar y los elementos externos asociados. Se incluye un botón de limpieza de los campos de entrada y el botón que solicita al servidor el inicio del análisis del programa ingresado en el correspondiente campo de entrada.

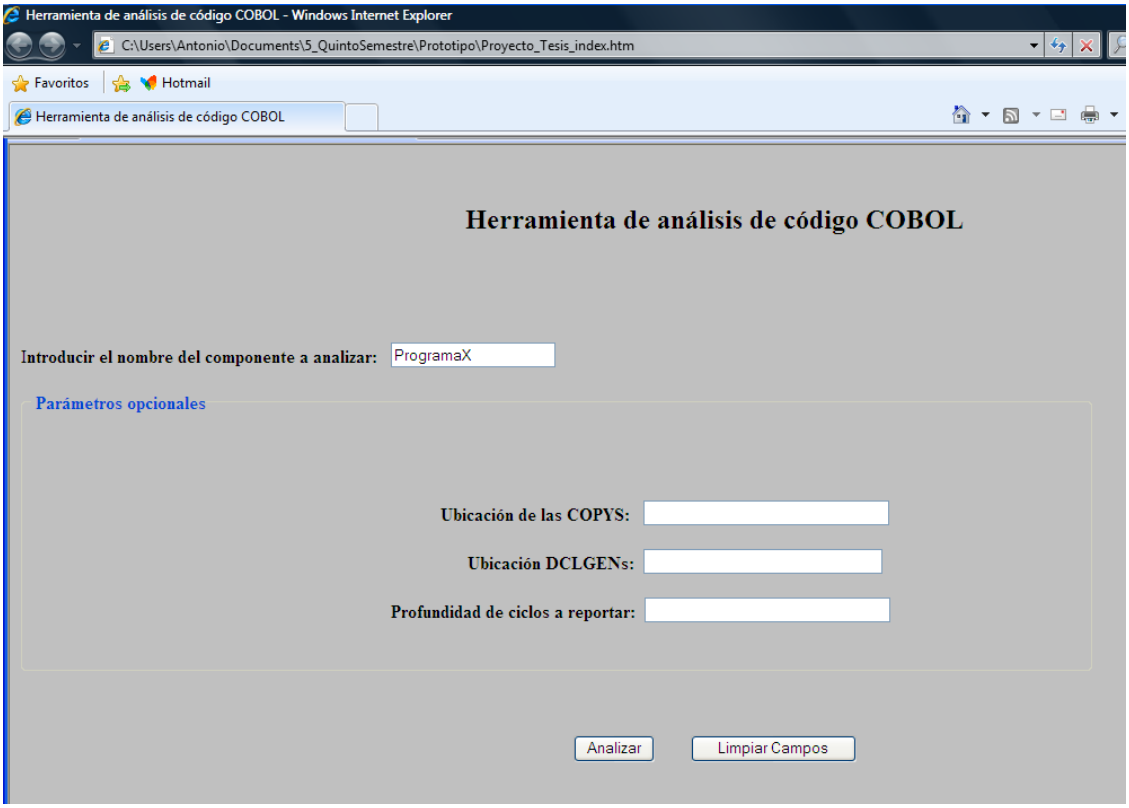


Figura 39 Pantalla de solicitud de análisis.

Una vez que se lanza la solicitud de análisis, el código Java se ejecuta y analiza el programa, como se comentó al inicio del presente punto, el resultado es un documento en forma de página Web con la estructura de una especificación de diseño del componente analizado. El reporte de análisis mencionado anteriormente, la página Web, se divide en varias secciones, en cada una de las cuales se colocan hiperligas que permitirán al analista, el usuario de la herramienta, modificar las conclusiones del autómata de manera manual.

La primera sección del resultado es el diagrama de entidades de información externa, de entrada o de salida, ya sea en forma tabular o por medio de una gráfica. Este diagrama parece no expresar mucho conocimiento, pero es de gran utilidad visualizar las entidades de información con las que interactúa el programa. En la figura 40 se muestra el diseño de esta sección del reporte de resultados, cuyo nombre es "*DIAGRAMA*".

El ejemplo de diseño de la tabla de la base de datos que debe servir como fuente de conocimiento para elaborar de manera dinámica la sección "*DIAGRAMA*" se muestra a continuación.

```
Elemento     Entidad_Ext     Tipo
----------   -----------     ---------------
Programa X    Archivo1       Archivo
Programa X    Tabla1         Tabla BaseDatos
Programa X    ModuloExt1     ModuloExterno
```

El diagrama que resulte de la tabla de información de entidades externas al programa se propone como una clase Java de tipo *Applet* incrustada en el código *HTML* que representa el reporte de

salida o bien una clase Java directamente podría entregar el código *HTML* que haga referencia a las partes del dibujo (flechas, cilindros, cuadros, etc.) y ensamblarlos cuando se presente la página del reporte en el navegador.

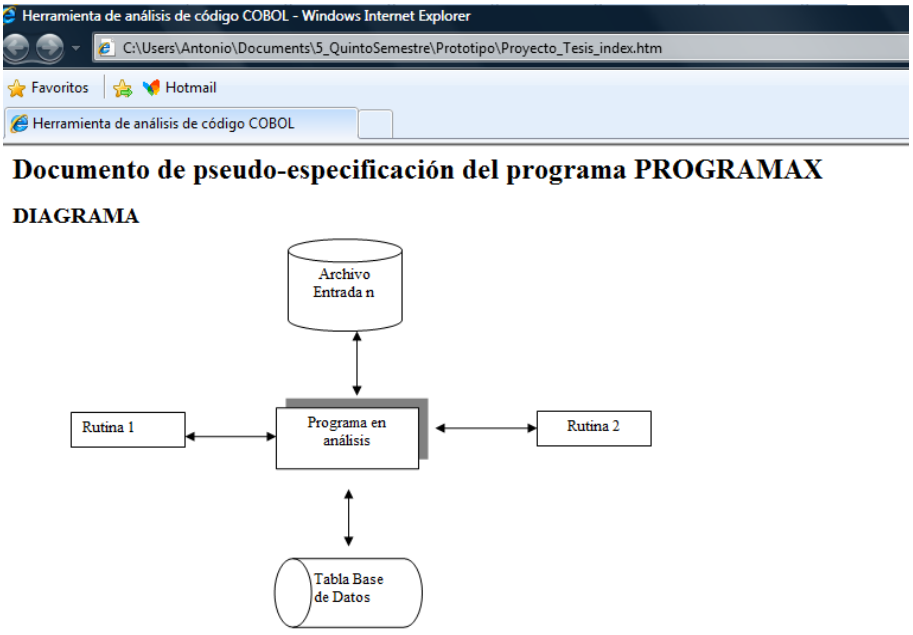


Figura 40 Resultado de análisis. Diagrama de entidades externas.

La siguiente sección del reporte de salida contiene información importante para el analista, esta información se presenta en forma tabular. El contenido principal de esta sección se refiere a los formatos de registro de las entidades externas de entrada/salida de las cuales el componente obtiene o envía información, de igual manera se propone reportar los elementos externos que contienen código que no compila de manera independiente y también los nombres de módulos compilados de manera externa. En la figura 41 se puede observar el diseño de esta sección a la cual se ha llamado: "*REFERENCIAS ASOCIADAS*".

Las secciones tabulares o de forma de lista del reporte pueden ser tomadas literalmente de una tabla con una estructura análoga. Por ejemplo la subsección "*FUNCIONES*" en el prototipo del reporte de salida, para esta sección la tabla fuente en la base de datos se propone como algo parecido al siguiente ejemplo:

```
Nombre    DescripcionO           DescripcionN
--------- ------------           ----------------------
Función 1 Funcionalidad Original Funcionalidad Modificada (a través de pantalla de mantenimiento)
```

El programa que construye el reporte tomaría "*DescripcionN*" como fuente primaria, la alimentación de este campo se reserva para la pantalla de mantenimiento manual de conocimiento. En caso de que el campo que contiene el conocimiento "manual" no esté alimentado o sea nulo, se propone tomar "*DescripcionO*" que es lo que originalmente alimenta la herramienta automática. Con lo anterior se cumple la funcionalidad de dar prioridad en el reporte a cualquier conocimiento que haya sido ingresado de manera manual por medio de las pantallas de mantenimiento.

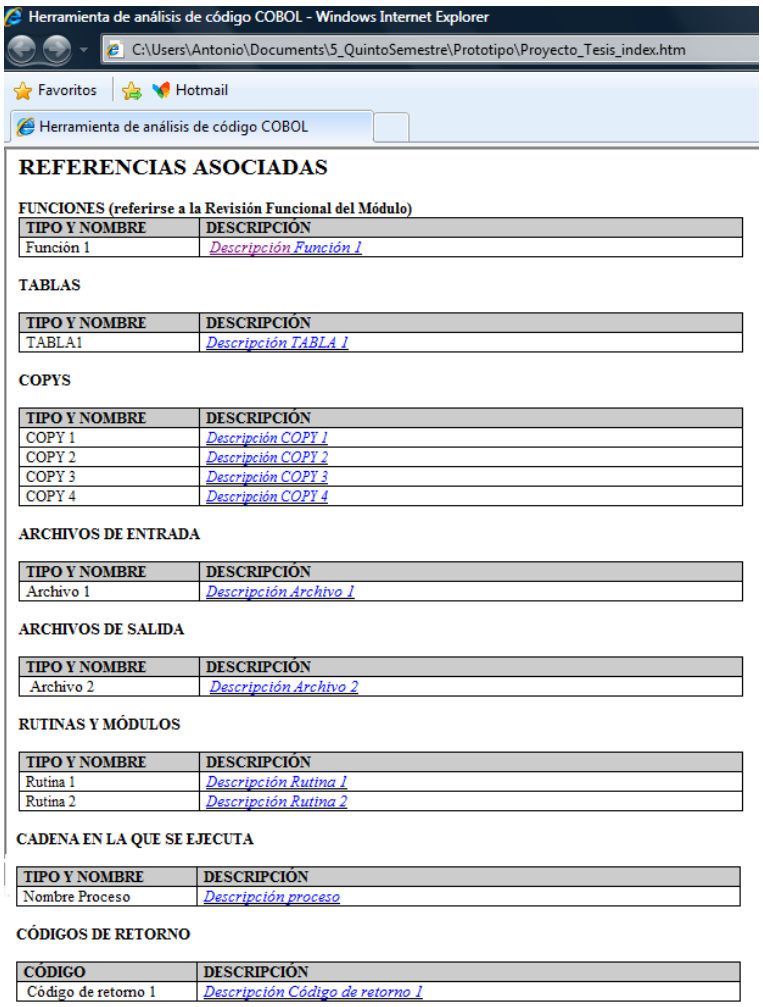


Figura 41 Resultado de análisis. Tabla de referencias asociadas.

La siguiente sección del reporte es una representación resumida de los principales procedimientos internos detectados en el componente y cómo es el camino de ejecución de algunas ramas del árbol de ejecución hasta la salida o entrega de resultados del programa.

Esta sección del reporte, titulada "*DESCRIPCION*" en el prototipo tiene como fuente de información una gráfica definida con nodos y aristas. La gráfica corresponde al flujo de un programa imperativo, la cual representa un diagrama estático de la secuencia de ejecución de procedimientos. La grafica no contendría dibujos propiamente sino texto con el nombre de los nodos. La secuencia del flujo de llamados entre bloques de código se representa con tabuladores y saltos de línea, una estructura parecida a los niveles de definición de datos en *XML*.

Se muestra enseguida un bosquejo de definición de las tablas de la base de datos que contiene el nombre de los bloques de código, los nodos y de cómo se liga la secuencia de ejecución entre ellos, las aristas de la gráfica. Como la información que contendrán las dos tablas depende de lo que pueda alimentar el compilador, la gráfica corresponde a un análisis estático del flujo imperativo de procedimientos:

- ***Definition of table bloque --Estos son los nodos***

```
DROP TABLE IF EXISTS nodo_flujo;
CREATE TABLE nodo_flujo (
  NFP_Sistema varchar(20) NOT NULL,
  NFP_Subsistema varchar(20) NOT NULL,
  NFP_IDPrograma varchar(08) NOT NULL,
  NFP_IDNodoFP integer unsigned NOT NULL,
          -- Este es equivalente a ID2-TipoNodo
  NFP_ID2NodoFP integer unsigned NOT NULL,
          -- Índice primario en su tabla contenedora,
          -- TODAS tienen, instrucción es la de default
  NFP_TiponNodo varchar(35) NOT NULL,
          -- Tabla contenedora del nodo Subbloque (B)
          -- para trozos de codigo IF, EVALUATE, GOTO,
          --(S)ection, (P)arrafo
  NFP_ImgNombreNodo varchar(35) NOT NULL,
  NFP_ProfLogicaH integer unsigned NOT NULL,
          -- Indentado en la pila del analizador estático,
          -- cada que entra a un terminal se incrementa el contador,
          -- es como la sangría (puede ser el mismo)
  NFP_ProfLogicaV integer unsigned NOT NULL,
          -- Contador secuencial de PRODUCCIONES de un mismo
          -- nivel de profundidad H (sangría)
  NFP_IDNodoPFP integer unsigned NOT NULL,
          -- ID padre en esta Entidad, es equivalente a
          -- IDNodoPadre-TipoNodoPadre.
  NFP_IDNodoPadre integer unsigned,
          -- Índice primario en su tabla contenedora,
          -- TODAS tienen, instrucción es la de default
  NFP_TipoNodoPadre varchar(35) NOT NULL,
          -- Tabla contenedora del nodo padre
  PRIMARY KEY  (NFP_Sistema,NFP_Subsistema,NFP_IDPrograma,NFP_IDNodoFP),
  KEY Index_2 (NFP_ID2NodoFP,NFP_TiponNodo)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='nodo_flujo';
```

- ***Definition of table control_flujo --Estas son las aristas***

```
DROP TABLE IF EXISTS control_flujo;
CREATE TABLE control_flujo (
  CFP_Sistema varchar(20) NOT NULL,
  CFP_Subsistema varchar(20) NOT NULL,
  CFP_IDPrograma varchar(08) NOT NULL,
  CFP_IDLigaCF integer unsigned NOT NULL,  --ID Único de la liga
  CFP_IDNodoDesde integer unsigned NOT NULL,
          -- Hermana desde ID tabla nodo
  CFP_ID2NodoDesde integer unsigned NOT NULL,
          -- Hermana desde ID tabla contenedora
  CFP_IDNodoHasta integer unsigned NOT NULL,
          -- Hermana hasta  ID tabla nodo
  CFP_ID2NodoHasta integer unsigned NOT NULL,
          -- Hermana hasta  ID tabla contenedora
  CFP_TipoNodoDesde varchar(35) NOT NULL,     --Tabla contenedora
  CFP_TipoNodoHasta varchar(35) NOT NULL,     --Tabla contenedora
  CFP_ProfundidadV integer unsigned,
  CFP_ProfundidadH integer unsigned,
```

```
  CFP_IDNodoPadre integer unsigned NOT NULL,  --Padre
  CFP_ID2NodoPadre integer unsigned NOT NULL,
  CFP_TipoNodoPadre  varchar(35) NOT NULL,    --Tabla Contenedora
  PRIMARY KEY  (CFP_Sistema,CFP_Subsistema,CFP_IDPrograma,CFP_IDLigaCF),
  KEY Index_2 (CFP_IDNodoHasta),
  KEY Index_3 (CFP_IDNodoDesde,CFP_IDNodoHasta),
  KEY Index_4 (CFP_ID2NodoHasta),
  KEY Index_5 (CFP_ID2NodoDesde,CFP_ID2NodoHasta)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='control_flujo';
```

El siguiente diagrama de flujo de ejemplo, tendría como fuente los datos poblados en las dos tablas mostradas (*Nodo* es el nombre de una función/procedimiento dentro del código para mayor claridad de lo que se está representando):

```
Inicio (etiqueta ilustrativa, no es parte de la gráfica)
     Nodo1
        Nodo2   Nodo3
     Nodo4
        NodoComun1
     Nodo5
        Nodo6   Nodo7
                  Nodo8   Nodo9
                             NodoComun1
     Nodo10
       NodoComun1
Fin (etiqueta ilustrativa, no es parte de la gráfica)

   NodosComunes (más de un hijo/padre, lista al fondo del esquema principal)
     NodoComun1
```

Tabla de nodos

IDPrograma	IDBloque	NombreBloque
ProgramaX	1	Nodo1
ProgramaX	2	Nodo2
ProgramaX	3	Nodo3
ProgramaX	4	Nodo4
ProgramaX	5	Nodo5
ProgramaX	6	Nodo6
ProgramaX	7	Nodo7
ProgramaX	8	Nodo8
ProgramaX	9	Nodo9
ProgramaX	10	Nodo10
ProgramaX	11	NodoComun1

Tabla de aristas

IDPrograma	IDArista	SecuenciaParalela	BloquePadre	BloqueHijo
ProgramaX	1	1	-	1
ProgramaX	2	1	1	2
ProgramaX	3	2	1	3
ProgramaX	4	1	2	4
ProgramaX	5	2	3	4

```
ProgramaX   6       1           4           11
ProgramaX   7       1           11          5
ProgramaX   8       1           5           6
ProgramaX   9       2           5           7
ProgramaX   10      1           6           10
ProgramaX   11      1           7           8
ProgramaX   12      2           7           9
ProgramaX   13      1           9           11
ProgramaX   14      1           8           10
ProgramaX   15      1           11          10
ProgramaX   16      1           10          11
```

Un ejemplo de cómo sería la presentación del flujo de ejecución de procedimientos en el reporte final se muestra en la figura 42.

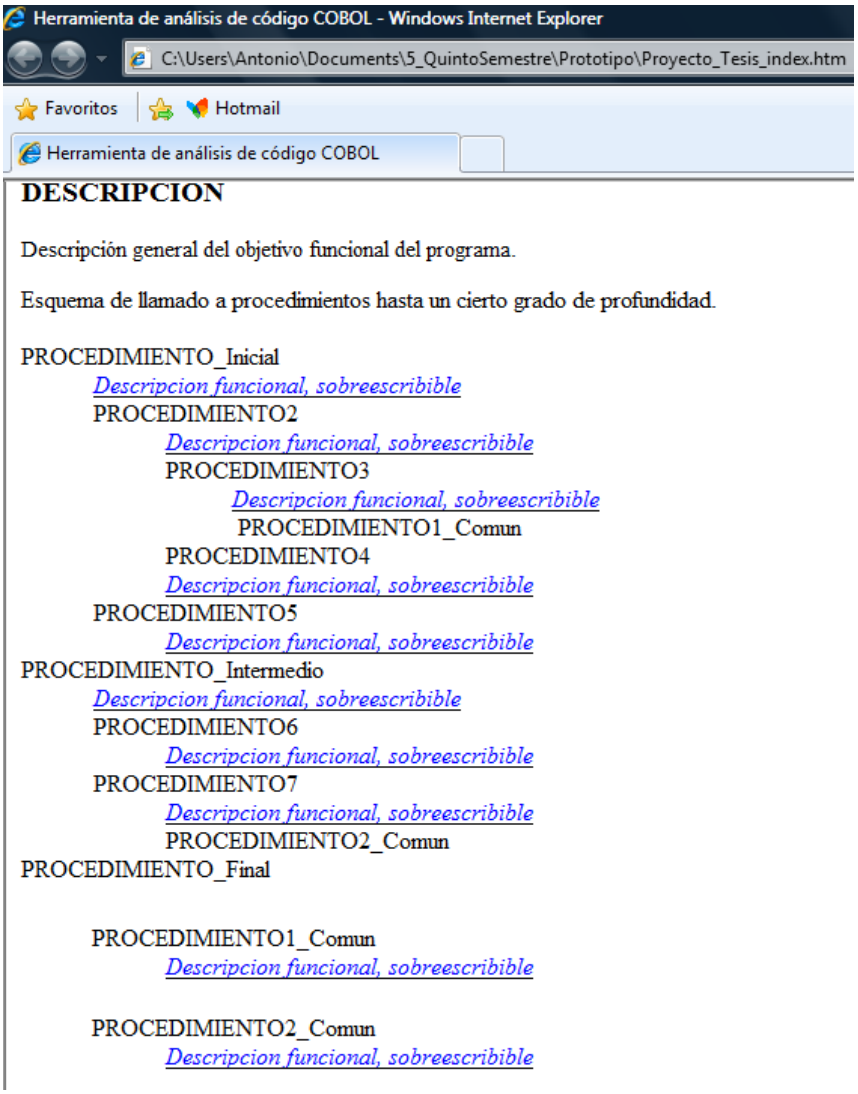


Figura 42 Resultado de análisis. Diagrama de procedimientos.

Por último, se muestra en la figura 43 una propuesta de presentación para reportar el conocimiento obtenido al aplicar los algoritmos básicos de extracción de reglas de negocios (sección 3.6 del presente del presente capitulo). Es posible que esta parte no sea desarrollada con riguroso detalle, como ya se comentó anteriormente.

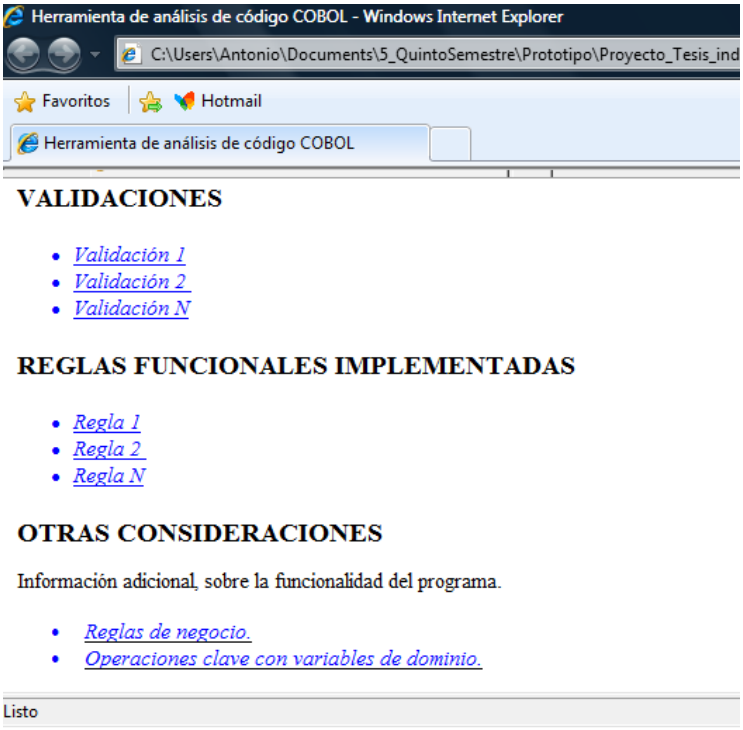


Figura 43 Resultado de análisis. Secciones adicionales propuestas.

La inclusión de conocimiento obtenido manualmente será posible por medio de una pantalla genérica de modificación. La manera de incluir esta funcionalidad consiste en agregar hiperligas en las secciones del reporte de salida que sean susceptibles a sufrir modificación en su contenido. Al seleccionar la liga, se debe presentar una pantalla genérica de modificación de contenido, los parámetros que deben pasarse en el llamado entre pantallas serán los identificadores de la sección del reporte a modificar, con esto el programa administrador de la modificación pueden consultar la base de datos con el contenido de la sección del reporte y presentarlo, de igual manera se debe desplegar el contenido que tiene la posibilidad de ser modificado y en su caso ejecutar la modificación o bien cancelar la solicitud. La figura 44 muestra el prototipo de la pantalla de modificación del reporte final.

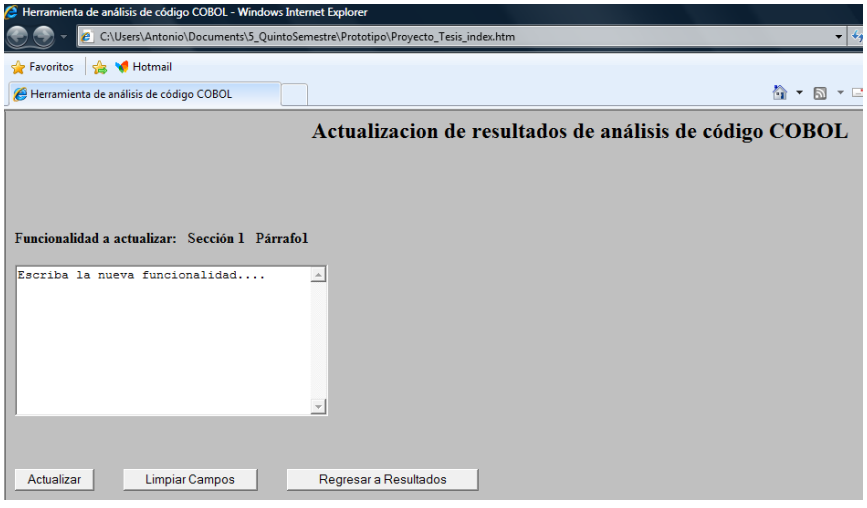


Figura 44 Modificación de resultado. Pantalla genérica de modificación.

3.9 CONSIDERACIONES PARA LA INTERPRETACIÓN DEL RESULTADO OBTENIDO.

Para la interpretación de resultados obtenidos por parte de la herramienta automática (la seudo-especificación) se deben agregar comentarios o documentación que ayuden a tal propósito. Se propone que estas consideraciones se incluyan como parte de la ayuda del sistema construido o en su defecto deberán ser documentadas como parte de la evaluación de los resultados obtenidos de la propuesta.

Se debe indicar en esta parte del entregable en qué consiste cada sección de la seudo-especificación de salida elaborada por la herramienta automática. Se documentará un apartado con advertencias cuando el código analizado contenga potenciales problemas asociados a los errores más comunes de la plataforma y que tengan que ver con la sintaxis, semántica y ejecución del programa.

Una de las características a agregar es la sobre escritura de los resultados, por ejemplo ciertos puntos de la especificación en donde se presenta la aproximación funcional elaborada por el artefacto. Para estas secciones, se tendrá la facilidad de permitir modificación manual agregando texto de formato libre, un ejemplo de sobre escritura es la inclusión de conocimiento de mayor nivel de abstracción funcional por parte del analista o por fragmentos literales de código que describan mejor la funcionalidad del componente. Se deben documentar, al finalizar la implementación, las indicaciones para utilizar la funcionalidad de modificación.

De igual manera se deben documentar advertencias sobre las secciones donde el conocimiento expresado en el reporte de salida se obtuvo de manera completamente automática.

En la parte de presentación de comentarios, si se trata de los que recolectó el algoritmo automático se debe advertir al analista, el usuario, que el texto corresponde a comentarios encontrados en el código presentados de manera fidedigna y que la funcionalidad o conocimiento descritos pueden no ser exactos.

3.10 COMENTARIOS FINALES DEL DISEÑO AL FINALIZAR LA IMPLEMENTACIÓN.

El propósito del presente capítulo ha sido el de describir la propuesta que contribuye a la solución del problema descrito en el capítulo 1 y por lo tanto se apega a este objetivo de alto nivel. El componente principal de la propuesta requiere un diseño previo, es necesario hacer un bosquejo formal acerca de cómo se planea construir el artefacto de software. Este diseño se ha basado en tomar como referencia algunas de las investigaciones relacionadas presentadas en el marco

teórico, el capítulo 2. Se han descrito a un nivel alto los principales componentes ó módulos del software que se va a producir. El diseño detallado no se presenta en este documento, el mismo tiene que ver con especificaciones y documentación a detalle de la implementación de los algoritmos en el código, lo que no está relacionado con el tema esencial del proyecto. Se considera pues que el diseño de alto nivel cumple con la función de describir la forma en que se construyó el artefacto de software.

El segundo comentario pertinente es que: el diseño presentado en este capítulo da el perfil funcional de varios módulos a implementar, al momento de terminar esta propuesta en concepción no se había iniciado la implementación en forma, sin embargo se anticipó que varios de los módulos con diseño de alto nivel tal vez no se lograrían materializar. Ante la detección de desviaciones en el proyecto de implementación la única medida de control disponible, por diversos factores que se confirmaron en la etapa final del proyecto, fue el recorte en los entregables, este siempre fue un riesgo latente del proyecto aplicativo. En efecto, tal riesgo se materializó y hubo recortes, varias partes del diseño no se implementaron al final, a cambio se eligieron partes a entregar que cubren un resultado que satisface en gran parte los objetivos fijados al inicio del proyecto. El paso inmediato a seguir era el de eliminar del diseño las partes que no se cubrieron. Esto último no se hizo por una razón: el diseño propuesto puede ser terminado en una investigación posterior y se considera que la propuesta inicial implementada al 100% es viable y contribuye con mejor eficacia a la solución del problema detectado. El diseño propuesto es en sí un entregable del proyecto y se considera que aporta en el ámbito de especificación de alto nivel.

CAPÍTULO IV. EVALUACIÓN E IMPACTO DE LA PROPUESTA.

El presente capítulo es la evaluación de la propuesta. El producto desarrollado es sintetizado en cuanto a alcance final, se describen sus principales características funcionales, la manera de utilizarlo y las pruebas ejecutadas sobre el mismo para emitir una evaluación inicial. Se documentan en este capítulo también las lecciones aprendidas en materia de administración del proyecto informático asociado a la implementación del diseño.

4.1 ALCANCE Y LIMITACIONES DE LA PROPUESTA.

Durante la planeación de tareas a ejecutar del proyecto descrito en los tres capítulos anteriores, se tenía riesgo latente de hacer varios ajustes en el entregable final del mismo, en realidad la implementación del diseño del artefacto de software asociado inició con mucha incertidumbre sobre su alcance final, pero en contraste se contaba con un objetivo bien definido y también se tenía claro qué parte del problema es el que se quería resolver, o por lo menos contribuir a la solución de una manera significativa por medio de un diseño inicial de alto nivel bastante descriptivo. La hipótesis propuesta fue bastante “elemental”: basados en la aplicación de teorías de análisis sintáctico principalmente, se puede proponer un procedimiento aplicativo que contribuye a atenuar varios de los problemas asociados al análisis de código manual, el problema se cerró a una plataforma específica donde el autor tiene la mayor parte de experiencia profesional: la arquitectura de aplicaciones empresariales en los equipos mainframe. Esto constituye el primer capítulo. La síntesis se presentó a continuación como el marco teórico, la colección y compendio de una serie de investigaciones que ya han sido desarrolladas versando sobre el tema al nivel general y enfocados a otros ambientes de ejecución u objetivos, aquí se pudo constatar que la investigación está justificada, se pretende contribuir con conocimiento aplicativo en el caso específico de la plataforma mainframe, cuando los desarrollos actuales en este contexto son productos con licencia y su arquitectura no es de código abierto. Enseguida se enunció el procedimiento aplicativo para comprobar la hipótesis: Tomando como referencia el modo de proceder durante el análisis de aplicativos en forma de “buenas prácticas”, que han sido colectadas y observadas durante la experiencia profesional del autor, se diseñó la prueba de la hipótesis. La parte principal de dicho diseño fue un artefacto de software basado en teoría de

compiladores, que es la principal contribución de esta investigación aplicativa. La materialización del artefacto constituye de hecho el apartado más riesgoso del proyecto. Una meta paralela fue también incluir el uso de técnicas de conceptualización e implementación de un proyecto informático con plataforma de implementación de código abierto y de tecnologías Web. Se ha concluido la investigación y es momento de enunciar la siguiente tesis: aún con limitaciones en la implementación del entregable y con base en los resultados obtenidos en las pruebas ejecutadas usando el procedimiento propuesto, se concluye que en efecto, aplicando teorías de compiladores, aún un entregable limitado perfila la funcionalidad de un componente de manera precisa y ayuda al analista de aplicativos a obtener la especificación de los componentes a un nivel de mayor abstracción, en una suerte de ingeniería inversa a partir del código fuente. Habiendo sintetizado la conclusión de una manera global, se procede a detallarla en los siguientes párrafos. De igual manera, en las siguientes líneas se enuncia qué partes del diseño descriptivo original desarrollado en el capítulo 3 no fueron implementadas y la causa de tal carencia, que es el tema de la administración del proyecto. El propósito del presente capítulo es pues, el de proporcionar un panorama general de evaluación para con esto terminar la investigación.

4.1.1 APORTACIÓN GENERAL DEL PROYECTO APLICATIVO.

El procedimiento de análisis que se ha desarrollado puede ser considerado un intento acertado para utilizar de manera auxiliar lenguajes de programación ajenos a la plataforma mainframe; así como del uso de herramientas de código abierto para analizar aplicativos que muy probablemente no van a ser descontinuados, por lo menos en un mediano plazo, sino que por el contrario tienen tendencia a crecer y evolucionar.

Se ha presentado un compendio de pasos a seguir o procedimiento que es en realidad una colección de buenas prácticas que han sido observadas por los analistas durante el desarrollo de proyectos informáticos en experiencias pasadas, es conocimiento práctico y empírico en gran parte que se ha presentado a manera de referencia en este trabajo por parte del autor, lo que se puede considerar documentación bastante útil para la plataforma de estudio.

Sin embargo la propuesta de esta investigación mejora el procedimiento actual (manual en muchas organizaciones) al añadir la conceptualización, diseño y construcción --por lo menos de un prototipo funcional-- de una herramienta semiautomática que extrae conocimiento del código y que lo presenta de manera resumida a un hipotético usuario analista de sistemas en aplicativos mainframe, así mismo se propone que el analista pueda introducir ideas más abstractas de funcionalidad a los resultados que bosqueja la herramienta auxiliar. Varios de los elementos en la macro estructura del lenguaje contienen de hecho toda la funcionalidad de negocio, solo que expresada de manera no tan obvia, el uso de las teorías y técnicas de compilación para producir algo práctico y funcional es también una aportación que se debe resaltar.

Ya se ha descrito cuál es la situación y las causas de porqué la mayoría de las herramientas de análisis en esta plataforma tienen licencia y el enfoque del mercado es generalmente para empresas grandes, las que son conocidas como triple A. De aquí se desprende una aportación adicional, dado que la cultura del "código abierto" es poco frecuente en estos ámbitos, la herramienta propuesta se basa al 100% en este tipo de tecnologías.

Se encontraron dos trabajos de investigación previos de código abierto relativos a la escritura de un analizador sintáctico que pudiera aplicarse a *COBOL*. La primera es la herramienta llamada *YACC* (Yet Another Compiler-Compiler) [109], esta herramienta toma como entrada las reglas de producción de la gramática del lenguaje en notación *BNF* y genera el código del analizador sintáctico en lenguaje C, esta herramienta fue desarrollada por Stephen C. Johnson en *AT&T* para el sistema operativo Unix en 1979. De igual manera se encontró *JavaCC* (Java Compiler Compiler) [110] que toma una entrada similar a *YACC* y genera el código del analizador sintáctico en lenguaje Java, a diferencia de *YACC*, *JavaCC* genera analizadores descendentes (top-down), lo que lo limita a la clase de gramáticas *LL(K)* (en particular, la recursión desde izquierda no se puede usar). El constructor de árboles que lo acompaña, la clase "*JJTree*", construye árboles de abajo hacia arriba (bottom-up). *JavaCC* está licenciado bajo una licencia *BSD* (Berkeley Software Distribution) software libre para fines de investigación [111].

Se consideró la limitante de que el participante único del proyecto no es un experto en codificación de lenguaje C, por lo cual se eligió tomar como referencia la herramienta *JavaCC*, dado que se supuso que Java sería mucho más amigable para aprender que *C*. Se ejecutaron pruebas sobre el código resultante para la gramática *EBNF COBOL* 85 experimental creada en 2002 por Ralf Lämmel y Chris Verhoef [112], se hicieron varias pruebas con código *COBOL* de ejemplo que es ejecutado actualmente en ambientes de producción y el resultado no fue aceptable. El compilador generado tenía poca robustez para recuperarse de los errores, se probó para 3 programas únicamente y se observó que detenía el análisis ante varios intentos de recuperación no exitosos en tiempo de ejecución (en las primeras 10 de 600 líneas de código *COBOL* o antes el analizador se detenía). Se evaluó la posibilidad de usar el código generado como referencia, pero al final se decidió descartarlo por considerarlo no viable. El tiempo invertido para hacerlo funcionar fue significativo (unas 80 horas) y una vez funcionando, los resultados no fueron alentadores. De haberlo sido, el alcance del proyecto hubiese estado más enfocado a la explotación de los datos generados por el compilador, tema que se dejó muy limitado en el entregable final. Como es código generado por un autómata es poco legible para los humanos, no se pudo encontrar una manera para incluir la estructuración del conocimiento en el código y lo más desalentador, o bien el programa en análisis tenía que ser "perfecto" y contener todas las reglas de producción implementadas en el compilador al pie de la letra o bien se modificaba la definición de la gramática para adaptarla al dialecto usado o bien se adaptaba el código fuente para hacer la recuperación de errores más relajada, al estar todas las instrucciones (reglas de producción) implementadas como código duro por el autómata, cualquier inclusión de código para almacenar conocimiento o relajar la recuperación de errores debería ser al mismo nivel, una tarea no viable en el "pajar" que representaban 24, 500 líneas de código generadas.

Después de la prueba no exitosa con *JavaCC*, la siguiente tarea del proyecto fue diseñar y construir los analizadores léxico y sintáctico de *COBOL* desde cero (para el dialecto mainframe de *IBM*® [34], uno de los más representativos de la plataforma mainframe). Esto fue lo más demandante de recursos y su materialización se considera también una aportación importante. El resultado fue un artefacto que es la fuente de datos funcionales extraídos del código, las posibilidades de explotación de los mismos son numerosas. Este tema es uno de los retos que se debe estudiar más a fondo en investigaciones subsecuentes. Por el momento, solamente algunos ejemplos de uso práctico se implementaron y aun con este recorte de alcance ante la premisa de finalizar la investigación, el entregable de esta iteración es aceptable.

Con referencia a la arquitectura de implementación elegida, además de lo mencionado en los tres párrafos anteriores relacionados con la implementación en lenguaje Java, se detectaron otras

justificantes/restricciones, además del conocimiento limitado del autor. Por ejemplo elegir lenguaje ensamblador para la implementación era casarse con un procesador; en Microsoft® se tiene el tema de las licencias en las herramientas de desarrollo; elegir desarrollar el entregable en plataforma mainframe implicaba tener un mainframe disponible para el desarrollo, lo cual no era viable. La plataforma elegida hace posible usar el entregable en un entorno accesible: una computadora personal (o un servidor de manera opcional), sin necesidad de instalar software con licencia pues los productos: Java/*MySQL*/Frameworks de código abierto/Apache Tomcat son todos descargables de manera gratuita y no crean dependencia tecnológica entre sí. Con esto se entrega un prototipo que es ejecutable sin muchas limitantes, el cual puede ser analizado para ser optimizado/escalado o bien explotado al nivel de usuario por analistas que tengan un cierto nivel de experiencia en análisis de aplicativos en *COBOL*.

El diseño del analizador se enfocó a la explotación de conocimiento a la par que se ejecuta el análisis sintáctico de los programas, la arquitectura tuvo que ser modificada 2 veces para recortar el número de producciones gramaticales a reconocer. Los recursos para implementar esta parte de la investigación, que se conceptualizó como auxiliar fueron mal evaluados, una sola persona trabajando en tiempo parcial invirtió casi 800 horas entre diseño de la arquitectura, codificación y pruebas unitarias, el diseño de este analizador adaptado para normalizar el conocimiento extraído del código es en sí un resultado sobre el cual se pueden fundamentar muchas mejoras o extensiones futuras. Si se tiene como base esta herramienta implementada y optimizada al 100% se pueden derivar desde aplicativos de reingeniería, analizadores cruzados entre aplicativos y escalamiento no solo a programas individuales sino a grupos de elementos (aplicativos completos). En conclusión, se tiene una buena base para derivar muchas mejoras en materia de colección automática de conocimiento aplicativo.

La evaluación de esta herramienta en el aspecto cognitivo es muy prometedora, se hicieron pruebas con varios ejemplos de código real y el analizador fue capaz de extraer piezas atómicas de conocimiento de manera adecuada, cuando se habla de programas *COBOL* que tienen alrededor de 10 mil líneas de código esta sola herramienta extractora de de indudable utilidad para cualquier analista que tenga cierta experiencia en el análisis de aplicativos. En primer lugar porque evita el recorrido visual de las líneas de código para extraer piezas de información claves para entender el funcionamiento de un componente (las piezas de información más importantes son el flujo imperativo del código y el de datos sobre las interfaces de entrada y salida). El analizador sintáctico implementado reconoce y es capaz de extraer los verbos y elementos sintácticos más utilizados en los ambientes de producción para este lenguaje, con esto nos referimos a variables e instrucciones. En el diseño implementado, existe un punto identificable en el código desde donde se puede hacer el envío de los elementos léxicos del lenguaje identificados como piezas de conocimiento funcional a una base de datos que las almacena para su posterior explotación.

El rendimiento de la herramienta no está optimizado para ser un parte de un compilador productivo, programas de relativa complejidad son analizados entre 2 y 4 minutos aproximadamente. Esto tiene que ver con la mezcla de las dos técnicas usadas en el analizador: la de fuerza bruta y la de predicciones *LL(1)* parametrizables en una base de datos. El tiempo de análisis se puede reducir con algoritmos adicionales, sin embargo, esta optimización en el rendimiento quedó fuera de alcance. No fue objetivo de este trabajo comparar el tiempo de análisis automático de esta herramienta contra el tiempo que lleva compilar un programa del mismo tamaño en la plataforma nativa (hasta décimas de segundo) ya que estamos hablando de un compilador implementado en lenguaje ensamblador, completamente optimizado con

operaciones mínimas de acceso a disco y ejecutándose en una computadora que tiene un *CPU* de velocidad muy superior al de una computadora personal, que es donde se implementó el analizador. Otro punto que incrementa la lentitud es que el lenguaje de implementación elegido fue Java. Java es lento comparado con otras tecnologías ya que no es un lenguaje que interactúe directamente con el procesador, sino que se ejecuta de manera triangulada por medio de la máquina virtual. Sin embargo se gana con este lenguaje al pertenecer el mismo una plataforma de código abierto y que es compatible con entornos Web, donde no se necesitan licencias para expresar de manera gráfica el conocimiento en una forma aceptable, como quedó demostrado con la presentación final de los resultados. Otra ventaja del lenguaje de implementación Java es que los componentes construidos pueden ser encapsulados e incluidos fácilmente en otros desarrollos que se pueden basar en este resultado como punto de partida en investigaciones posteriores.

4.1.2 ALCANCE Y LIMITACIONES TÉCNICO-FUNCIONALES DEL ENTREGABLE DEL PROYECTO.

Para construir el motor de conocimiento de análisis, se propuso en el diseño inicial identificar todos los componentes gramaticales de la macro (sintaxis) y micro (elementos léxicos) estructura del lenguaje. Esto al final no fue posible, las causas de explican en el apartado de la administración del proyecto, se describen a continuación los principales puntos a destacar en tema de alcance y recorte en el entregable.

El analizador léxico reconoce la mayoría de los elementos de la micro estructura del lenguaje. No reconoce ni clasifica aún elementos basado en las columnas donde aparecen, lo cual tiene reglas especiales en el dialecto *COBOL* utilizado, tampoco reconoce palabras de doble byte ni declarativas para el compilador. Tiene implementados 20 algoritmos de expresión regular que reconocen una gran cantidad de elementos por coincidencia genérica simulando a los del tipo autómata de estado finito con único estado final de aceptación de la cadena de entrada. Reconoce 587 palabras reservadas del lenguaje. Tiene implementado un algoritmo de memoria intermedia que es capaz de almacenar símbolos (*tokens*) contenidos en 20 líneas de código hacia adelante con referencia al elemento actual en análisis, lo cual lo hace versátil para implementar la técnica *LL(4)* mínima requerida para la completa predicción de reglas de producción (funcionalidad no implementada en este entregable). Se propuso en el diseño inicial de manera opcional codificar cambios de modo al detectar código incrustado de otros lenguajes como *SQL* o las macros de comunicación con *CICS* y que, en el cambio de modo estos elementos "externos" fuesen reconocidos, esta funcionalidad no fue implementada y se deja como tema abierto.

El analizador sintáctico está codificado para reconocer 38 verbos de instrucciones, se hizo prueba exhaustiva de los verbos *MOVE*, *PERFORM* (anidados), *IF* (anidados), *CALL* y *OPEN* debido a que su información fue incluida en el reporte de conocimiento. El programa es robusto ya que se recupera de los errores ante una regla mal implementada en el código o que tenga sintaxis errónea desde la cadena de entrada (el código fuente) o que simplemente no esté implementada en el código del analizador, como se mencionó en el diseño, la prioridad no es validar de manera rigurosa el componente para su transformación en ejecutable de lenguaje de máquina, sino extraer conocimiento asumiendo que es un componente productivo que ya compila sin errores. Además de los verbos implementados en las reglas de producción correspondientes (lexemas), se codificaron todos los símbolos no terminales asociados al "contexto" de las mismas, el número de reglas de producción que se codificaron en total es de 320 (símbolos no terminales derivados del

lexema parcial, la instrucción). La construcción se dejó muy avanzada en términos de logro del objetivo puntual de este componente (reconocer *todas* las producciones gramaticales del dialecto), pero el tiempo disponible de implementación se quedó corto y la consecuencia directa es que el analizador no está optimizado en velocidad de ejecución ni tampoco fue probado exhaustivamente para depurar los defectos. Los 38 verbos codificados no fueron probados de manera exhaustiva, se llegaron a detectar de manera fortuita producciones algo bizarras, de las del tipo que permite *COBOL*, que no son aceptadas, corregir estos defectos es un refinamiento que requiere invertir más tiempo y se tuvo que dejar como tema abierto. Para ilustrar este último punto, se menciona que existen en el dialecto variables de tipo estructura que pueden ser referenciadas por el nombre de la estructura, por el índice si pertenecen a un arreglo de variables y por una funcionalidad de sub cadena si la variable es alfanumérica, la combinación de todo esto en un programa real no fue reconocida, la causa es que se decidió que la referencia por nombre de estructura no es muy "común" y por lo tanto no fue implementada como regla de producción en el código del analizador, se menciona entonces, que la siguiente instrucción no es reconocida, la cual estaba incluida en un programa de prueba de 28 mil líneas de código:

COMPUTE WS-REC-LEN = LENGTH OF FFF-CAP (1, 2, 3) +
LENGTH OF FFF-TAX-SUR-RECORD (X IN Y: Z).

En compensación, el analizador se recupera al leer el punto que marca el fin de una oración en la gramática del dialecto o una palabra clave que indica el inicio de una nueva instrucción, con lo cual ignora esta producción y queda listo para seguir intentando emparejamientos del siguiente símbolo en la cadena de entrada contra las reglas de producción de la gramática.

El tema de la explotación de los datos generados por el analizador sintáctico para convertirlos en información útil es un terreno que quedo también abierto. Es en esta funcionalidad donde se hizo el recorte más significativo contra el diseño inicial proyectado al nivel funcional. Se eligieron las producciones que dan una aproximación cercana a lo que se esperaba. El diseño inicial del normalizador de conocimiento se recortó a la parte ilustrativa más importante: la estructura del flujo imperativo del programa en análisis. Desde el punto de vista de la experiencia profesional del autor, lo que podría perfilar un entregable muy ilustrativo incluyó lo siguiente:

- La definición de los archivos de entrada.
- La definición de variables y la identificación de su tipo.
- Los verbos que definen la estructura de flujo del programa en gran parte de los casos en programas de producción: *IF*, *PERFORM* y *GOTO*
- Los verbos que ilustran funcionalidad y que aparecen de manera frecuente: *OPEN*, *CALL*, *MOVE*.

En resumen, la funcionalidad de normalización de conocimiento se codificó para soportar los componentes de la estructura de flujo imperativa del programa, las entradas y salidas de tipo archivo de acceso secuencial, las variables declaradas y 5 instrucciones, las cuales fueron probadas exhaustivamente en el analizador sintáctico. Aún con los recortes, solo esta parte es una aportación importante. En un caso de prueba se generó la estructura visual del flujo de un programa en 5 minutos, que comparados contra las 4 horas que toma aproximadamente generar de forma manual el bosquejo para un programa de tamaño similar (por experiencia propia del autor).

Para la interfaz de usuario se diseñaron 4 pantallas de interacción, el producto final tiene como principal característica la de presentar de manera gráfica el conocimiento colectado durante el análisis automático, esto está alineado con el fin principal de la investigación: ayudar a un analista a perfilar la funcionalidad de un componente. La principal característica de la interfaz gráfica de usuario desarrollada es que contiene un algoritmo ilustrativo que presenta un dibujo de las entradas y salidas del programa basado en la información de la base de datos de conocimiento. De igual manera presenta el flujo imperativo de ejecución con un nivel de profundidad lógica alimentado por parámetro, no se implementó el desplegado de los saltos incondicionales (*GO TO*) que sí son reconocidos en el análisis sintáctico. La interfaz hace uso combinado de 4 Frameworks de desarrollo Web de arquitecturas empresariales: *STRUTS2*, *Hibernate*, *Spring* y *ExtJS* (*AJAX*) esta parte tuvo que ser investigada "desde cero" ya que no se poseían conocimientos previos. El tiempo no alcanzó para implementar más algoritmos de presentación gráfica del conocimiento, sin embargo el resultado logrado en este ámbito fue bueno.

Cuando el analizador léxico de *COBOL* estaba siendo probado de manera unitaria, el tiempo de implementación del proyecto ya era significativo, la propuesta inicial era incluir analizadores léxicos (y sintácticos) de lenguajes incrustados, en específico para los dos principales en la plataforma destino: *SQL* para *DB2* y *CICS* el lenguaje de comunicación con el monitor de transacciones. La desviación en tiempo impuso que esto debía quedar fuera de alcance, hay secciones identificadas dentro del código donde estas funciones pueden ser agregadas, se debe seguir la misma filosofía que para el lenguaje principal de análisis: reconocer las reglas de producción y decidir con la ayuda de parámetros cuales componentes aportan conocimiento (Respuestas del usuario, Tablas, Campos y Acciones sobre la base de datos) para almacenarlos y explotarlos.

Los algoritmos de reconocimiento de comentarios con información acerca de la funcionalidad del componente (reconocedores muy básicos de lenguaje natural presente en los comentarios del código) tampoco fueron implementados, se hizo una propuesta en el diseño sobre este punto, pero no fue construida en el prototipo final.

Para recapitular, se listan a manera de resumen los puntos más importantes a considerar para el artefacto de software desarrollado, en cuanto a especificación, alcance y limitaciones funcionales y técnicas se refiere.

- Está desarrollado en plataforma de computadora personal compatible con software *Microsoft*® (*Windows*).

- No de desarrolló un modulo de conectividad con la plataforma mainframe, por lo que el análisis de código se basa en la premisa de que el código fuente se tiene que copiar manualmente desde la plataforma fuente (el mainframe) hacia la plataforma analizadora (la computadora personal).

- Los elementos externos con código *COBOL* (conocidos como *COPY* o *COPYBOOK*) no fueron incluidos en el prototipo final, ni tampoco los dos principales lenguajes incrustados: *SQL* y los scripts de *CICS*.

- El análisis sintáctico de *COBOL* orientado a objetos y los programas anidados no están implementados en el prototipo final.

- Si bien se estructuró la base de datos de conocimiento para manejar grupos de componentes (por aplicativos), la herramienta auxiliar no analiza aplicativos completos, sino elementos unitarios de compilación.

- El analizador léxico desarrollado no reconoce los siguientes elementos de la micro estructura del dialecto utilizado: Palabras de doble byte definidas por el usuario, clases de palabras relacionadas a *COBOL* orientado a objetos, palabras de manejo de librerías externas, nombres de funciones, nombres de variables con calificadores.

- El lenguaje *COBOL* tiene una gramática que no es *LL(1)*, la sintaxis es muy parecida al idioma inglés. El analizador sintáctico reconoce las formas básicas de 38 producciones gramaticales que describen las instrucciones *COBOL*, lo cual incluye en cascada toda la serie de análisis recursivo iniciando desde el lexema hasta el último símbolo terminal de la producción. Se debe mencionar sin embargo, que existen variantes de estas producciones que son excesivamente complejas, un ejemplo es la instrucción *COMPUTE* con manejo de excepciones. En general la mayoría de programadores *COBOL* no incluyen este tipo de variantes en su código, por lo que cuando se decidió recortar el número de procedimientos a codificar, este tipo de variantes fueron elegidas.

- El tiempo a invertir para probar y refinar el artefacto de manera extensa y con resultados estadísticos robustos está fuera de alcance. Sin embargo, se probaron 3 programas de producción reales y los resultados se muestran más adelante, de esta pequeña muestra se puede extrapolar que el analizador tiene un buen porcentaje de cobertura cuando se use con programas de producción.

- No se incluyo la totalidad del diseño descrito en el capitulo tres para la explotación del conocimiento que proporciona el analizador sintáctico, se concluyó con un prototipo que describe únicamente la descripción de las entidades de entrada y salida en su forma de archivos, la estructura del flujo imperativo del programa, la lista de variables de grupo de las entidades de entrada y un ejemplo de instrucciones formadas con malas prácticas de programación: Instrucción *IF* sin su terminador *END-IF*.

- La interfaz de usuario para invocar el analizador es amigable, además se incluyó la posibilidad de incluir comentarios al resultado automático de forma manual por parte del usuario en la funcionalidad de los nodos del flujo imperativo de ejecución.

- No se incluyeron algoritmos heurísticos de análisis de información para deducir reglas de negocio o funcionalidad más abstracta, ni tampoco el análisis de los comentarios dentro del componente para identificar lenguaje natural incrustado, candidato a describir funcionalidad, el tiempo fue la limitante, pero el diseño está propuesto en el capítulo 3 con algoritmos no tan complejos, aún así este tema queda como abierto.

- El analizador léxico construido es bastante robusto, tiene facilidad de leer $n=4$ símbolos por lo menos de la cadena de entrada por adelantado y en retroceso. Se hicieron pruebas de progresión con programas de producción razonablemente grandes, avanzando y retrocediendo la solicitud de elementos-símbolos terminales del lenguaje (*tokens*) y se obtuvieron resultados

aceptables, se utilizó la especificación de *COBOL* "*Enterprise COBOL for z/OS*" y se extrajeron 509 palabras reservadas mismas que son reconocidas en su totalidad.

- Se conceptualizó el artefacto de software como herramienta que usa tecnologías de código abierto. A continuación se listan los productos usados para la implementación del analizador semiautomático (difieren las versiones contra las propuestas en el diseño inicial, las que se mencionan aquí son las versiones mínimas requeridas definitivas):

  - JavaTM Platform, Standard Edition 6 Development Kit *JDKTM* 6. Compilador gratuito Java versión 1.6 Software libre de Sun, subsidiaria de Oracle Corporation [113].

  - Java jre6 Version 6 Actualización 22 (build 1.6.0_22-b04). Máquina virtual de ejecución de código java para Windows, software libre de Sun, subsidiaria de Oracle Corporation [114].

  - Apache-Tomcat-6.0.29 Contenedor Web de código abierto para páginas *HTML*, *JSPs* y servlets de Apache Org [115].

  - *MySQL* Server 5.0 Base de datos relacional de código abierto de Oracle Corporation [116].

  - Netbeans 6.0.1 Ambiente integrado de desarrollo para lenguaje Java de licencia pública de código abierto de Oracle Corporation [117].

Únicamente para la presentación del reporte de usuario final *GUI* (Graphic User Interface), se utilizaron las siguientes herramientas:

- Ext JS Libreria javascript de la compañía Sencha que usa técnicas *Ajax*, *DHTML* y *DOM* [118].

- Apache *STRUTS2* Framework de desarrollo (*APIs*) de aplicaciones Web de Apache Org [119].

- Spring Framework de desarrollo (*APIs*) de aplicaciones en capas para *Java/J2EE* de Spring Source [120].

- Hibernate Framework de desarrollo (*APIs*) de mapeo objeto-relacional (*ORM*) para la plataforma Java de JBoss subsidiaria de la compañía Red Hat [121].

4.2 ADMINISTRACIÓN DEL PROYECTO.

El proyecto informático asociado a la creación del software entregable tiene varias particularidades que se considera pertinente documentar. Se pueden identificar varias características inherentes a los proyectos tradicionales. Se fundamentó en la definición de una serie de objetivos a cumplir, a petición de un patrocinador, los objetivos se debían satisfacer en una determinada fecha, con cierta cantidad de recursos asignados y con requisitos de finalización

mínimos a satisfacer. En consecuencia se elaboró un primer bosquejo de la manera en que se lograría el cumplimiento de los objetivos, se definió qué es lo que se planeaba hacer para cumplir con los objetivos fijados, primero al nivel general y conforme el tiempo de implementación avanzó poco a poco se fueron cerrando los detalles, se dio un avance de manera gradual en los entregables al detalle hasta cumplir con los objetivos en un cierto grado y ante las restricciones de tiempo se determinó que el proyecto debía terminar sin haberlos cumplido al 100%.

Las desviaciones significativas entre lo planeado y lo logrado al finalizar un proyecto informático, aparecen no pocas veces en la vida real y normalmente surgen cuando el proyecto se basa en asunciones bastante alejadas de la realidad o cuando se tienen severas limitaciones en cualquier factor crítico (costo, tiempo, recursos humanos) y aún así el proyecto da inicio.

En el caso específico del presente trabajo debemos considerar una serie de antecedentes que nos servirán para tener una mejor perspectiva sobre su evaluación, sobre los temas que quedaron implementados y de cómo aportan estos en materia de solución al problema identificado (el objetivo primordial). De igual manera se considera que la documentación de las causas por las que no todos los puntos previstos en el plan original fueron implementados forma parte de las lecciones aprendidas.

En primer lugar, como ya se expuso, el problema central de la investigación aplicativa tiene un contexto amplio, una parte medular de la propuesta de solución consistió en la elaboración del prototipo de software y la implementación del mismo se podría considerar como uno de sus sub-proyectos, por así decirlo y este sub-proyecto es el que más recursos demandó. El objetivo inicial fue el de implementar el artefacto de software para extraer información funcional de un programa *COBOL* a partir de su código fuente. Se elaboró el perfil de los entregables al nivel descriptivo general, módulos internos contenidos, interacción con el usuario, comportamiento funcional y la arquitectura propuesta en su implementación. Enseguida se evaluaron los recursos disponibles para cumplir los objetivos de alto nivel, el único recurso humano y sus habilidades en los diferentes roles a jugar en el proyecto así como el tiempo disponible a invertir. El tema económico es no aplicable en este proyecto por lo cual no se evaluó algo en este apartado. Se definieron tareas a cumplir y fechas de entrega.

Las prácticas tradicionales de administración de proyectos quedaron en esta instancia muchas veces fuera de contexto debido a varias restricciones inherentes a la naturaleza de la investigación. En la planeación del proyecto se anticipó y estimó un tiempo de un año de diseño e implementación del artefacto de software. Principalmente se baso la estimación en lo siguiente:

- Falta de experiencia en la plataforma de desarrollo y el lenguaje de implementación (Web-Java) del único participante.
- Falta de experiencia en la escritura de compiladores del participante.
- Acceso limitado a la plataforma del lenguaje fuente del análisis (*COBOL*-mainframe) para obtener código de prueba.
- El participante del proyecto haciendo el papel de todos los involucrados: Administrador del proyecto, usuario, arquitecto, diseñador, consultor-investigador de la arquitectura de implementación, codificador, ejecutor de pruebas, auditor parcial del proyecto.
- Tiempo limitado para finalizar la investigación, tanto para dedicarlo a la ejecución de las tareas durante la implementación como en el plazo global para concluir todo el proyecto.

Tradicionalmente cuando un administrador de proyectos anticipa este tipo de restricciones tiene varias opciones como medidas de control ante desviaciones materializadas, entre otras la de solicitar a recursos adicionales sean estos humanos (especialistas) o de presupuesto, inclusive como se observa en el mundo real se pueden justificar y obtener extensiones de tiempo, aceptación de entregas parciales, etcétera. Un proyecto tradicional, además de este tipo de alternativas, cuenta con un rango menor de incertidumbre, ya que son proyectos que se desarrollan en un entorno con experiencias similares previas es decir, que en una organización tradicional, normalmente las "buenas prácticas" de administración de proyectos tienen una cierta base de apoyo, conocimiento y referencia previos (una base de datos de conocimiento e implementaciones previas).

Se sabe que la planeación de un proyecto se basa en estimaciones y se calcula con holgura en tiempos para un cierto margen de riesgos ante la incertidumbre de no saber exactamente como se desviará o ajustará al plan la implementación durante su ejecución. Se ha leído en algunos textos o escuchado en el ámbito laboral que la administración de proyectos es un "arte", probablemente así sea en muchos ámbitos, sin embargo tal vez esta astucia o "maña" para calcular tiempos y recursos solo se manifiesta como algo confiable cuando ya hay algo previo sobre lo que el administrador puede tomar como referencia y también si se cuenta con especialistas de respaldo que lo aconsejen. En este caso el participante del proyecto, jugador de todos los roles inició la planeación de las actividades sin tener experiencia propia sobre la implementación de un proyecto similar, con limitantes en varios factores críticos, sin una base de datos de conocimiento previo en proyectos similares y sin un equipo de especialistas consultores. Durante la descripción del problema detectado en la plataforma mainframe, de una manera involuntariamente profética, se mencionó que un proyecto solo puede planearse con ciertos pronósticos de éxito cuando se conoce a un nivel aceptable el ámbito funcional sobre el que se trabajará, la solución propuesta en esta investigación irremediablemente condujo a una de estas situaciones, tal vez descritas con un cierto nivel de crítica durante la descripción del problema a atacar. Para aclarar un poco lo que sucedió: los riesgos considerados para valorar el tiempo de implementación en un año se materializaron y el tiempo de holgura para controlarlos se quedo corto, fue mal calculado. El principal impacto negativo fue la falta de experiencia y lo complejo de la sintaxis del lenguaje para ser validada. Hubo momentos en que el caso funcional real rebasó la previsión del diseño y solucionar estos errores fue altamente costoso en tiempo, el segundo factor negativo fue el tiempo disponible a dedicar siempre limitado y en tercer lugar la falta de experiencia en la plataforma de implementación (Web). El reto fue de alguna manera interesante, tal vez no se arriesgó dinero, pero tampoco se puede hablar de desenlaces muy deseables en el caso de no lograrse algo de valor. El resultado no se puede ni con mucho llamar fracaso, es de hecho aceptable.

El objetivo de la investigación fue cuidado en todo momento, la implementación de la arquitectura técnica requirió tiempo considerable es cierto, pero en todo momento el diseño se enfocó a la explotación de conocimiento: Los analizadores léxico y sintáctico se diseñaron para hacer la recuperación de errores y la selección de componentes léxicos del lenguaje con conocimiento algo manejable dentro del código. Tal vez suena un poco trivial o un poco vacía la frase, pero "elaborar un analizador sintáctico para *COBOL*" implica bastantes recursos y experiencia en la codificación de compiladores, cosa que se carecía, si bien los conceptos investigados como parte del marco teórico fueron de gran utilidad, trasladar dichas teorías a un lenguaje poco amigable para la escritura de compiladores tuvo sus retos de implementación.

Para este tipo de proyectos de investigación aplicada es interesante "alegar" o vaticinar qué hubiera pasado si el proyecto se hubiese implementado de otra forma. La principal sugerencia es

que este proyecto debió ser tomado por más de una persona y con diferentes especialidades, estos especialistas podrían colaborar en entregas iteradas para lograr los objetivos completos de la investigación, entre los principales se pueden listar:

- Un especialista del requerimiento funcional que perfilase de manera focalizada el entregable, el tendría el rol de administrador del proyecto y de usuario final, el perfil de este participante es el de un arquitecto o analista experimentado en la plataforma mainframe, quien hubiese analizado de forma manual mucho código en *COBOL* y para proyectos complejos, el podría definir de manera práctica y concisa los entregables, esta parte se cumplió en parte, si bien el autor de esta investigación cubre el perfil, él mismo tuvo que tomar otros roles, desde codificador en un lenguaje ajeno a su experiencia profesional hasta ejecutor de pruebas, el lema divide y vencerás en este caso tuvo poca aplicabilidad.

- Un investigador/arquitecto funcional que aportase teorías en las diferentes partes que componen el software. La restricción del tiempo hizo que varios requerimientos se tuvieran que dejar como fuera de alcance. Ya no se profundizó más sobre el estado del arte, ni se trabajó en la conceptualización práctica de varios puntos ya investigados (la implementación técnica). Al concluir el proyecto, si se revisan los 3 primeros capítulos la investigación cumple el objetivo de presentar un problema detectado en la industria, proponer una propuesta que ayuda a la solución del problema y también teorías que pueden ser integradas en la solución propuesta. El tema que quedó limitado es la implementación práctica del diseño realizado.

- Un especialista en escritura de compiladores, ya que el tema de la construcción del compilador fue el principal reto y no debió serlo porque ese no era el objetivo, la ganancia fue más académica en este punto ya que se aprendió mucho al escribir un compilador adaptado a las necesidades de la investigación. Esto debido a que el lenguaje *COBOL* no es manejable para implementar un analizador sintáctico con las técnicas tradicionales.

- Un especialista constructor de interfaces con el usuario que se encargase de la presentación amigable del conocimiento. Esta es otra de las áreas donde el autor no es un especialista, se investigaron varias técnicas de presentación visual del conocimiento pero la realidad es que esto definitivamente se dejó para última instancia y el conocimiento fue presentado de manera limitada en el entregable final.

Con los 4 especialistas incluidos en el proyecto, el entregable propuesto en el diseño debió estar optimizado, todos los componentes mencionados en el capítulo tres incluidos y la presentación final para el usuario debió tener incluidos mejores contenidos visuales, lo cual ayuda mucho a la hora de analizar algo. La aritmética nos dice que el tiempo de implementación se debió reducir en ¼ lo cual no es del todo acertado, pero lo que sí es acertado es que aún con la misma cantidad de tiempo invertido en la implementación, el entregable hubiese tenido mejor calidad, sin ánimo de denostar el resultado actual.

El propósito esencial del proyecto no era desarrollar un compilador de *COBOL*, esta herramienta se planeó en sí como un paso auxiliar para lograr un fin más ambicioso y global que es la explotación de la información producida por el mismo. La parte de la construcción del analizador sintáctico fue valorada en un año de trabajo, la misma tuvo desviación, esto es parte de las lecciones aprendidas del proyecto y causó desviación en los demás entregables planeados. Este

entregable nos recuerda tristemente que en materia aplicativa: Hay una gran zanja a librar entre el diseño propuesto y la materialización del mismo (diseño detallado y construcción) y no hablemos de la abstracción global más elevada que es lo que un usuario patrocinador solicita en sus requerimientos de negocio (en este caso el autor se encuentra en el grupo de usuarios finales, por lo que nunca hubo ambigüedades en este proyecto sobre lo que se esperaba materializar).

La investigación de manera global tuvo la siguiente valoración *inicial* en horas:

- Investigación del marco teórico. 100 Hrs.
- Presentación del trabajo escrito final. 100 Hrs.
  (Incluye la conceptualización del procedimiento propuesto)
- Diseño general de la solución 80 Hrs.
- Diseño del analizador léxico 80 Hrs.
- Construcción y pruebas del analizador léxico 400 Hrs.
- Diseño del analizador sintáctico 100 Hrs.
- Construcción y pruebas del analizador sintáctico 600 Hrs.
- Diseño del modulo de almacenamiento de conocimiento 60 Hrs.
- Construcción y pruebas del módulo de almacenamiento de conocimiento 100 Hrs
- Diseño de la interfaz gráfica de usuario 80 Hrs.
- Construcción y pruebas de la GUI 120 Hrs.

- Total 1820 Hrs.

Eran dos años y medio aproximadamente invirtiendo en promedio dos horas cada día. El tiempo sin embargo se extendió seis meses más, dedicando en varias ocurrencias más de las dos horas diarias planeadas, hasta alcanzar las 2800 horas. La causa principal fue la implementación del software afectado por diversas eventualidades y riesgos materializados. Se muestran estadísticas y gráficas en las figuras 45-47, la figura 45 contiene estadísticas de todo el proyecto y la de la figura 46 únicamente del sub-proyecto de creación del software.

Fecha Plan	Fecha Real	Hito	Días P	Días R	Meses P	Meses R	Horas P	Horas R
01/08/2008	01/08/2008	Inicio Introduccion						
29/08/2008	29/08/2008	Fin Introduccion	28	28	0.92	0.92	84	84
08/09/2008	08/09/2008	Inicio Definicion Problema						
28/11/2008	28/11/2008	Fin Definicion Problema	80	80	2.63	2.63	240	240
26/01/2009	26/01/2009	Inicio Marco teórico						
24/07/2009	24/07/2009	Fin Marco teórico	178	178	5.86	5.86	534	534
03/08/2009	03/08/2009	Inicio Diseño						
26/03/2010	26/03/2010	Fin Diseño	233	233	7.66	7.66	699	699
01/04/2010	01/04/2010	Inicio Construccion del diseño						
30/10/2010	30/03/2011	Fin Construccion del diseño	209	359	6.88	11.81	627	1077
05/11/2010	01/04/2011	Inicio Evaluación Proyecto						
10/12/2010	30/05/2011	Fin Evaluación Proyecto	35	59	1.15	1.94	105	177
			763	937	25.10	30.82	2289	2811
					2.09			
					Años			

Figura 45 Comparativo tiempos de implementación del proyecto global. Planeado Vs Real.

Implementación del Software		Tiempos Reales				
Funcionalidad	Fecha Inicio	Fecha Fin	Dias	Meses	Años	Horas
Análisis Lexico	06/08/2009	21/12/2009	137	4.51	0.38	411.00
Análisis Sintáctico	21/09/2009	22/02/2011	519	17.07	1.42	1557.00
Normalizador Conocimiento	11/01/2011	12/03/2011	60	1.97	0.16	180.00
GUI	22/01/2011	25/03/2011	62	2.04	0.17	186.00
Total			778	25.59	2.13	2334.00
Total Lineas	TotalHoras	Meses Tiempo Completo			Horas Laborales	
13410	2334.00	14.5875				

Figura 46 Métricas finales implementación del software del proyecto.

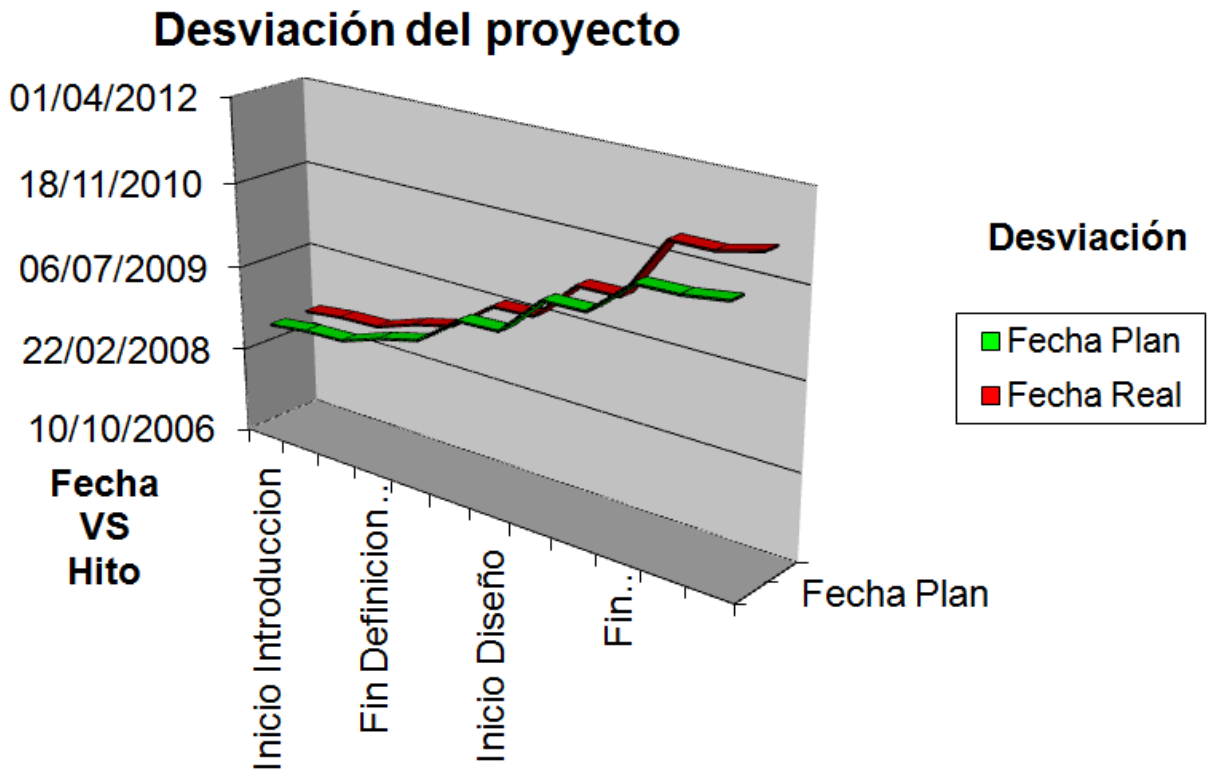


Figura 47 Gráfica de desviación del proyecto.

4.3 ESPECIFICACIONES DEL ENTREGABLE DE SOFTWARE.

El entregable de software consta de tres carpetas, una llamada "*proyectoNetbeans*", otra llamada "*ejecutable*" y la última llamada "*código*". En este apartado se describe el modo de proceder para

ejecutar el artefacto desde cualquiera de las tres carpetas. Posteriormente se describen las características funcionales implementadas en el prototipo.

4.3.1 INSTRUCCIONES PARA EJECUTAR EL SOFTWARE.

El componente de software desarrollado es un proyecto que se ejecuta en un ambiente Web, por lo tanto su ejecución depende de un ambiente de ejecución que es el servidor Web. Los productos de software que debe tener la computadora donde se planea ejecutar la herramienta son los siguientes:

- Java jre6 Versión 6 Actualización 22 (build 1.6.0_22-b04). Máquina virtual de Java de Sun Microsystems.
- Apache-tomcat-6.0.29. Contenedor Web de JSPs y Servlets de Apache Org.
- MySQL Server 5.0. Base de datos relacional de código abierto de Oracle Corporation.
- MySQL Tools for 5.0. Herramientas de acceso a la base de datos MySQL 5.0 de Oracle Corporation.

Estos cuatro productos de uso libre deben estar previamente configurados para hacer la computadora de ejecución un servidor Web y de base de datos.

4.3.1.1 Carga de la base de datos de parámetros.

Parte del entregable es un respaldo de base de datos de *MySQL*, la carpeta llamada "*parametros*" tiene dentro de la misma un archivo llamado "*tesisave.sql*" que es la base de datos de parámetros del analizador, ver la figura 48. Se describen a continuación la serie de pasos que se deben seguir para instalar y cargar la base de datos en la computadora de ejecución.

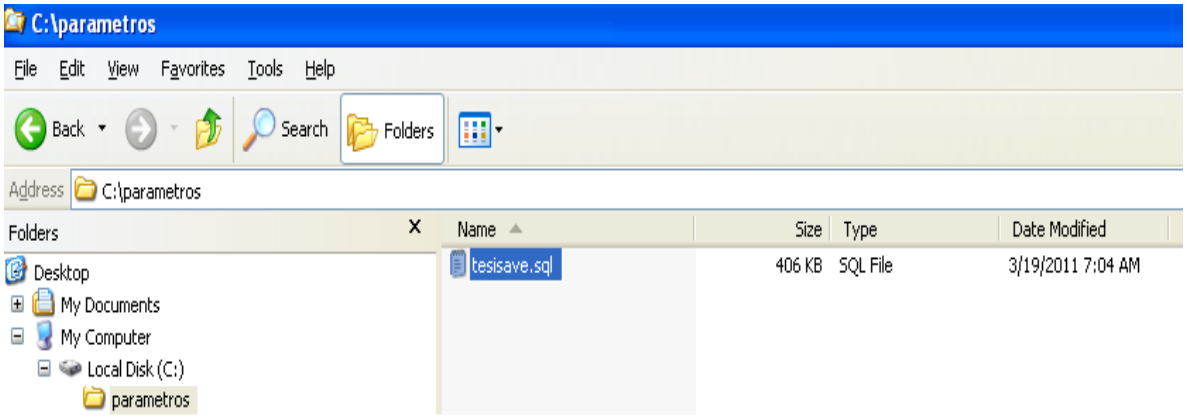


Figura 48 Base de datos de parámetros.

a. En primer lugar se debe crear en *MySQL* una base de datos llamada "*tesisave*". Abrir la consola del *administrador de base de datos* (con el usuario "*root*") y crearla. En la ventana de inicio del administrador, en el panel "*Schemata*" en el área de lista de los esquemas dar clic con el botón derecho del ratón y seleccionar la opción "*Create New Schema*", ver la figura 49.

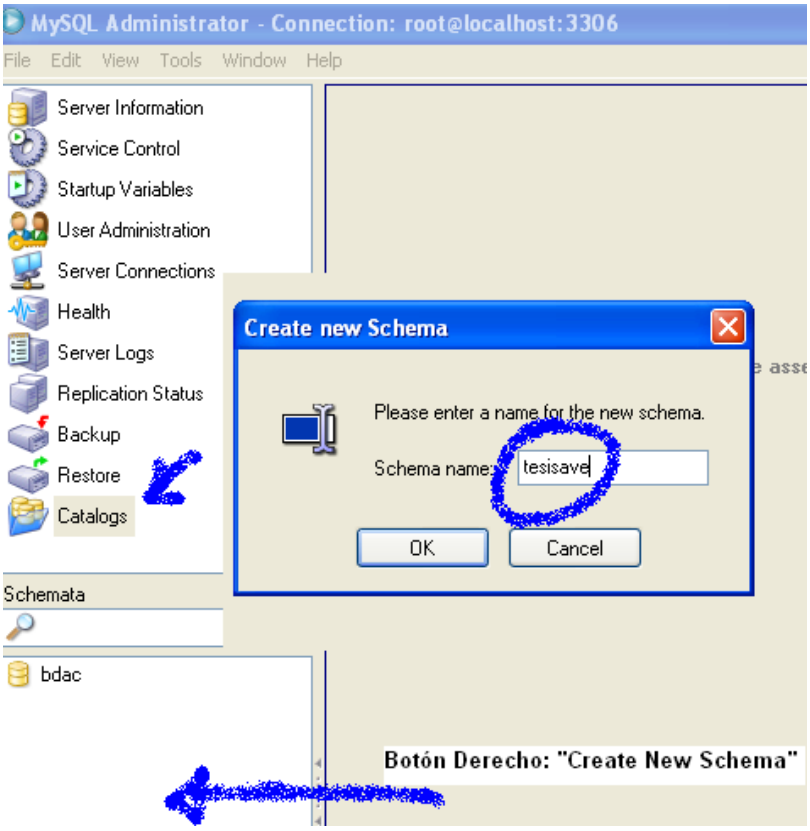


Figura 49 Creación de la base de datos de parámetros en MySQL.

b. Después ir a la ventana "*Restore*" y dar clic en el botón "*Open Backup File*", como en la figura 50.

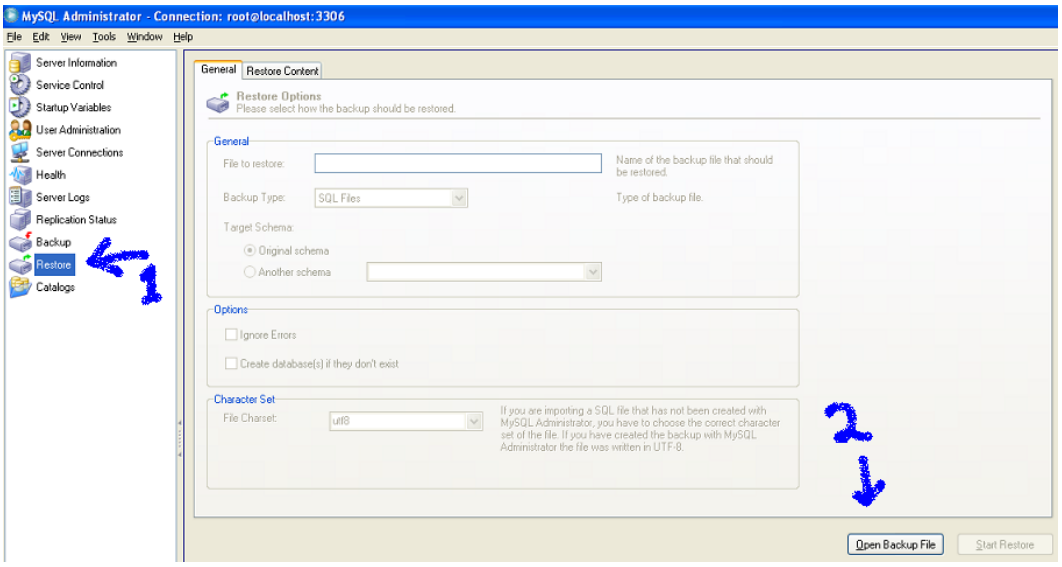


Figura 50 Carga de la base de datos de parámetros.

c. En la ventana "*Open*" que se abre, localizar el respaldo de la base de datos donde se tenga guardado y dar clic en el botón "*Open*" (figura 51).

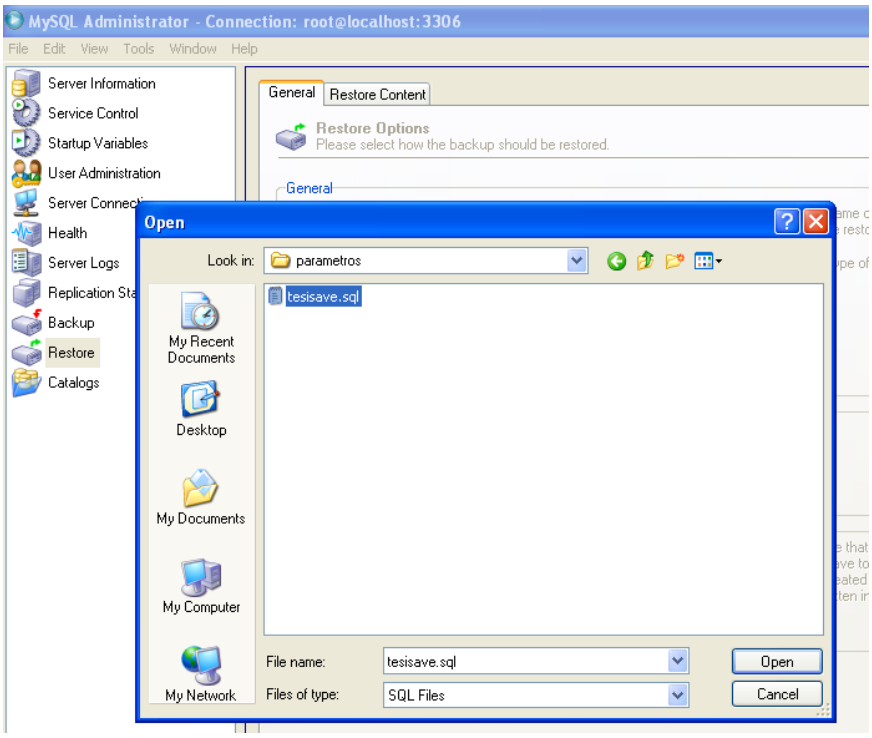


Figura 51 Apertura del respaldo de la base de datos de parámetros.

d. Dar clic en el botón "*Start Restore*" y esperar a que se carguen los datos en el archivo (figura 52).

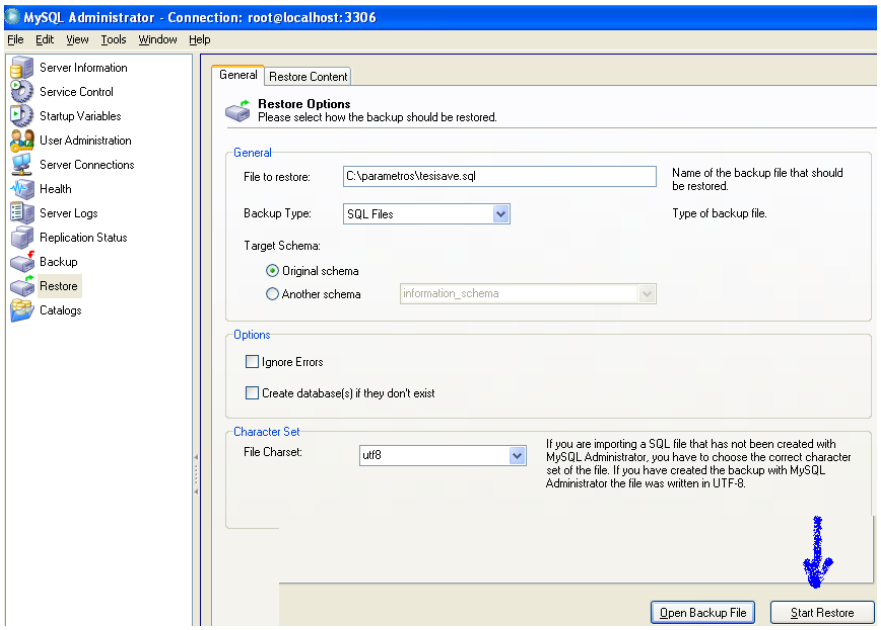


Figura 52 Restauración de la base de datos de parámetros.

e. En la aplicación del *administrador de la base de datos* (utilizar el usuario "*root*"), ir al módulo de administración de usuarios y crear el siguiente usuario para conectarse a la base de datos de

parámetros (y de conocimiento), Usuario: "*tesisave*"; Contraseña: "*antonio*". Asignar todos los privilegios *CRUD* sobre la base de datos recién creada/cargada ("*tesisave*") al usuario "*tesisave*", ver las figuras 53 y 54.

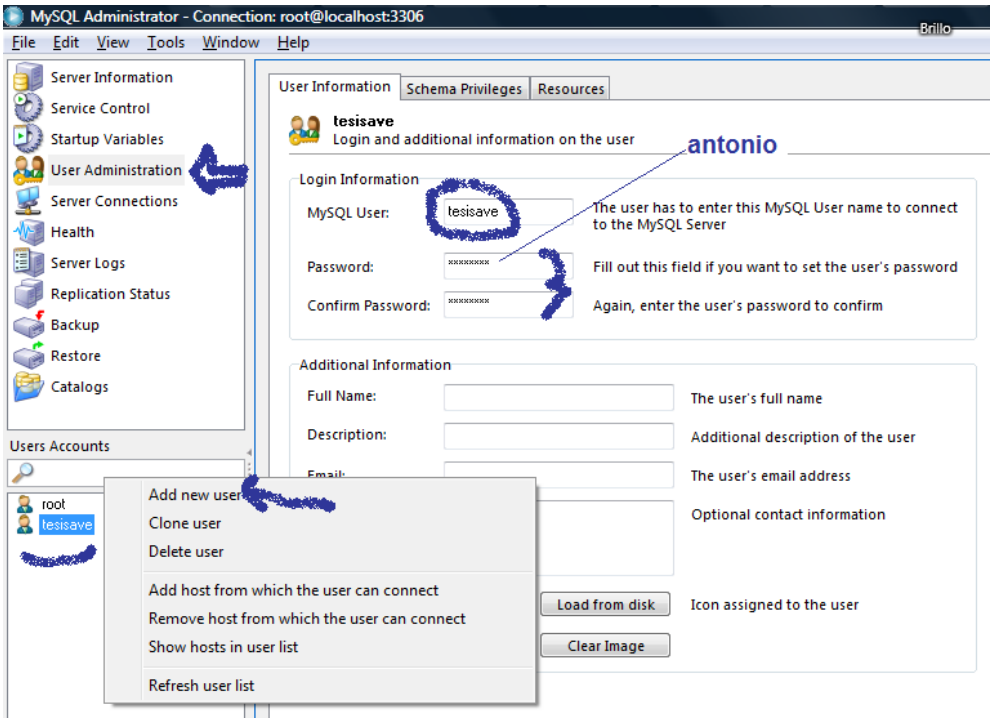


Figura 53 Creación del usuario para acceder a la base de datos de parámetros.

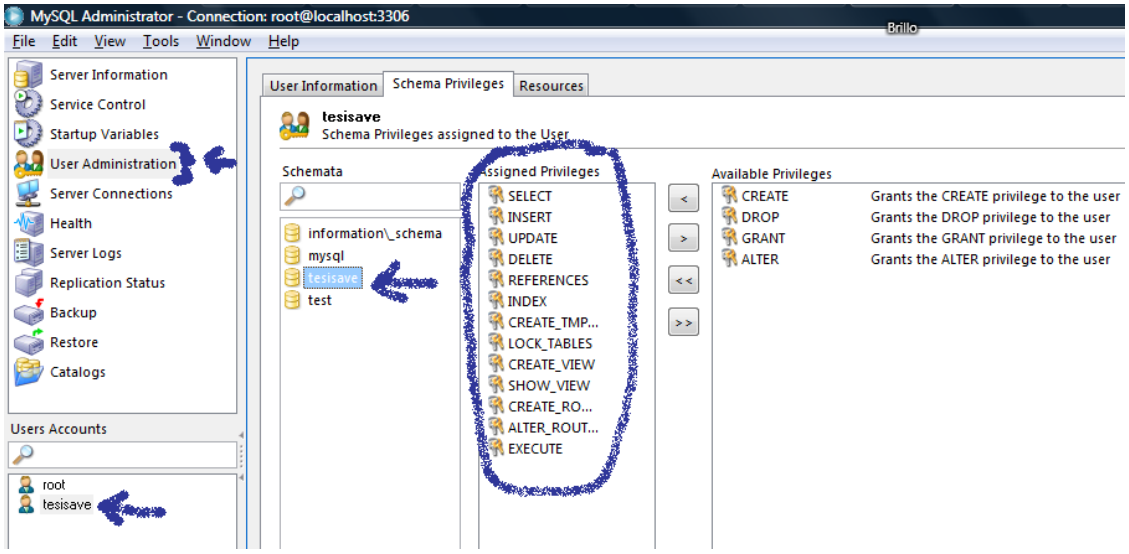


Figura 54 Asignación de permisos al usuario de la base de datos de parámetros.

f. La base de datos se carga y se puede verificar que todo ha salido bien con la siguiente instrucción en el *navegador de consultas*: Teclear la consulta "*SELECT COUNT(*) FROM regla_simbolo*" y el resultado debe ser 1485 (figura 55).

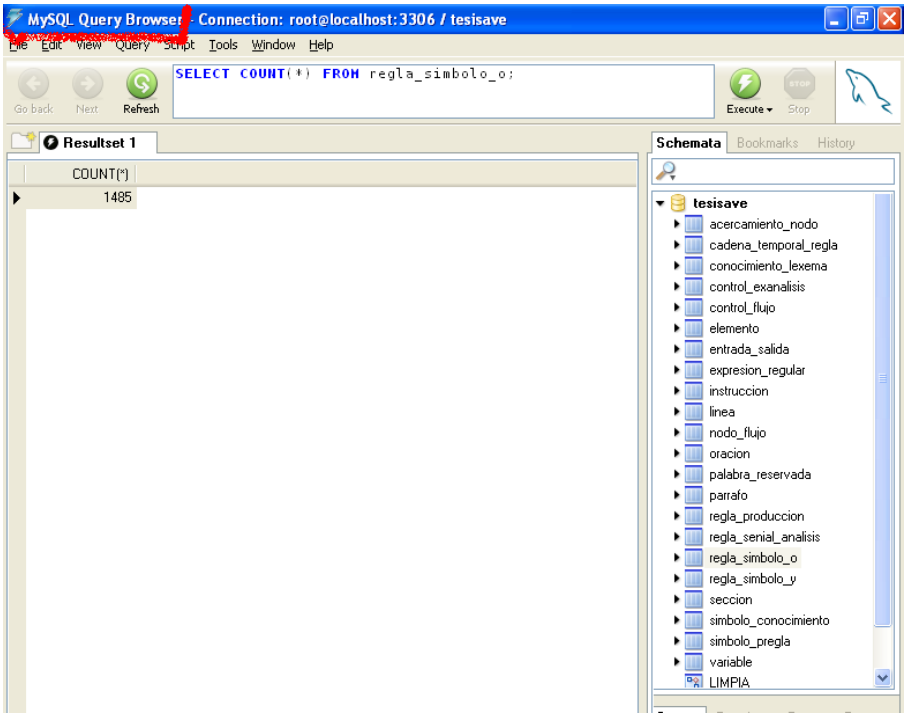


Figura 55 Consulta de verificación de la carga de la base de datos de parámetros.

4.3.1.2 Desplegar y ejecutar el proyecto en el servidor Web.

a. Arrancar el servidor Web y contenedor de servlets: *Apache Tomcat* con el comando siguiente o con un comando equivalente de acuerdo a la computadora servidor donde se quiera ejecutar el proyecto Web, el resultado se muestra en la figura 56.

Directorio_Instalación/bin/Startup_script
C:\Program Files\apache-tomcat-6.0.29\bin\startup.bat

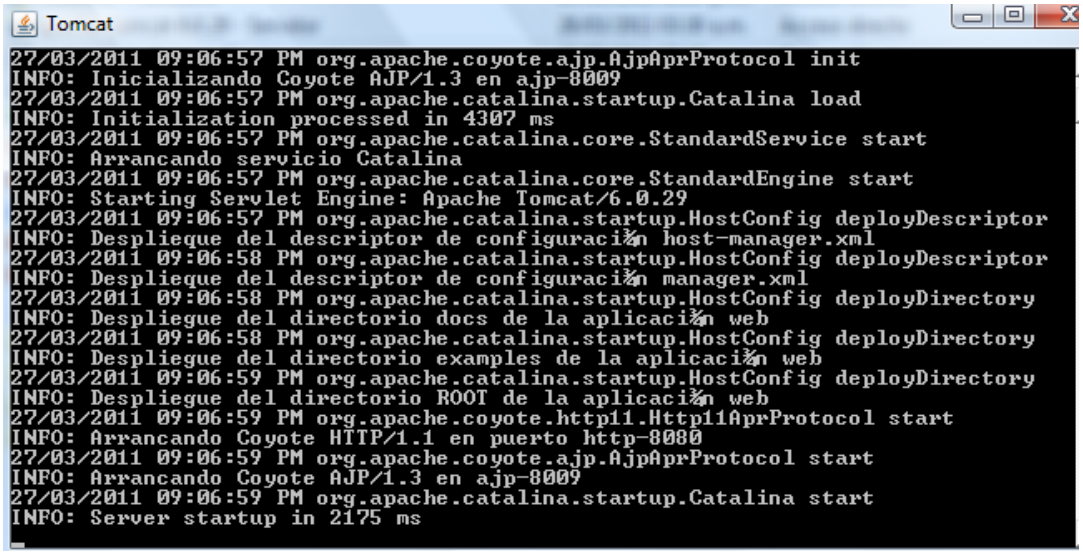


Figura 56 Arranque del servidor Web Apache-Tomcat.

b. Una vez que el servidor Web está en ejecución, utilizar un navegador Web (por ejemplo Internet Explorer), para ir a la página de administración del servidor (usar el usuario administrador definido para su instalación): http://localhost:8080/manager/html, ver figura 57.

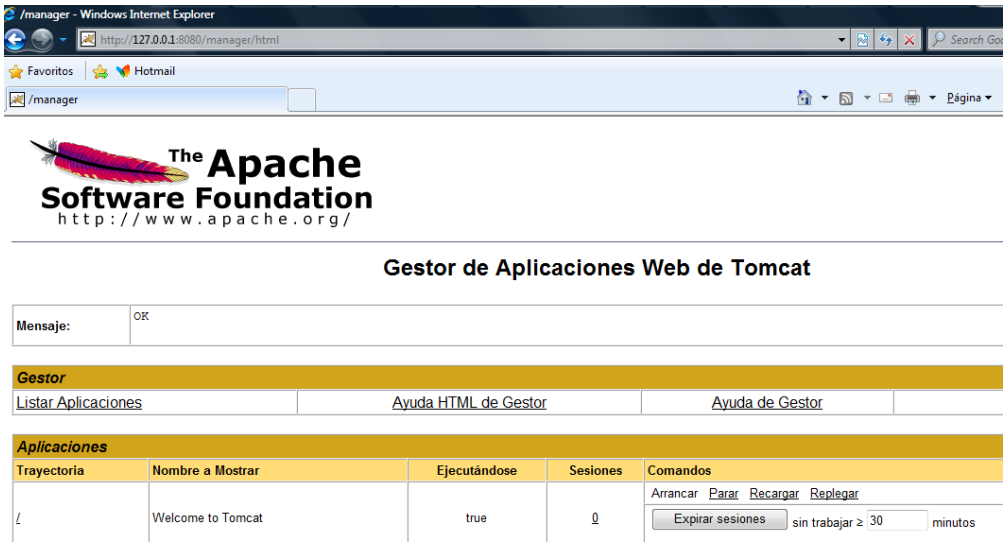


Figura 57 Página de administración del servidor Web Apache-Tomcat.

b. Cargar el archivo "*TesisAVE.war*" proporcionado en el disco entregado (figura 58). Usar la sección "*Desplegar → Archivo WAR a desplegar*" de la página de administración del servidor para localizar la ubicación del archivo y dar clic en el botón "*Desplegar*", ver figuras 59 y 60.

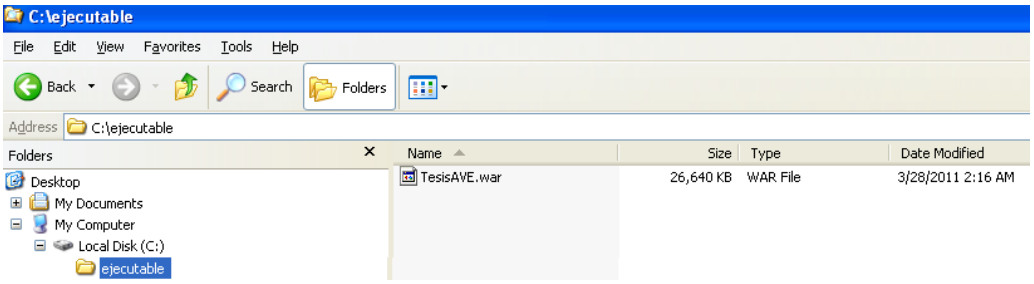


Figura 58 Archivo war del proyecto.

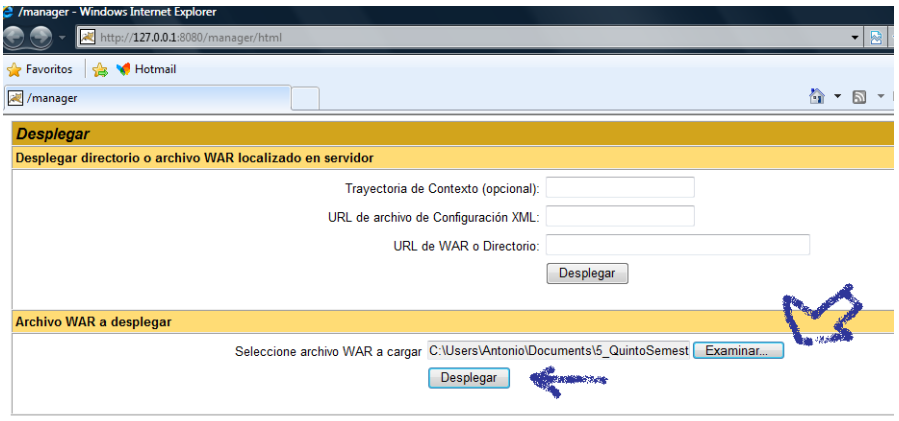


Figura 59 Sección de despliegue de archivo war en la página de administración del servidor Web.

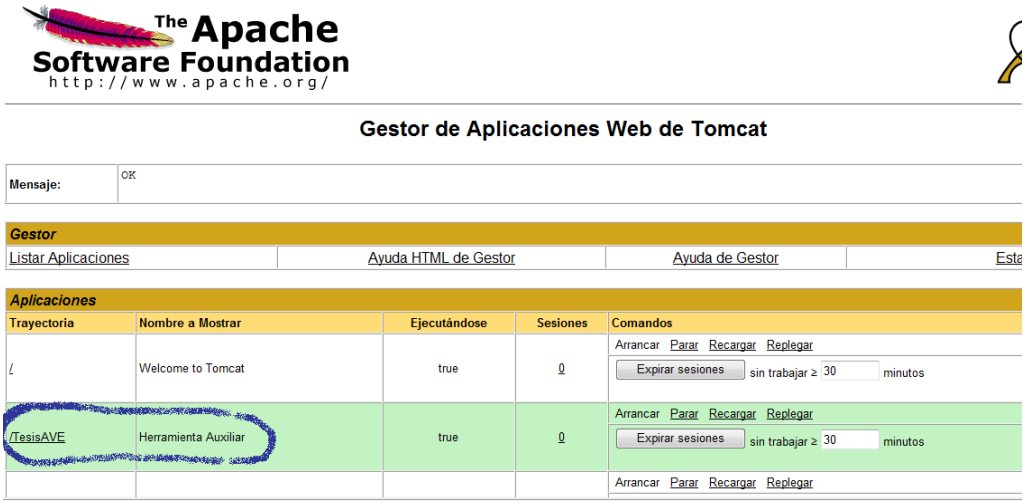


Figura 60 Proyecto TesisAVE desplegado en el servidor Web Apache-Tomcat.

d. Usar una nueva pantalla del navegador Web, para ir a la página inicial de la aplicación: La URL es: http://localhost:8080/TesisAVE/. Ver las figuras 61 y 62.

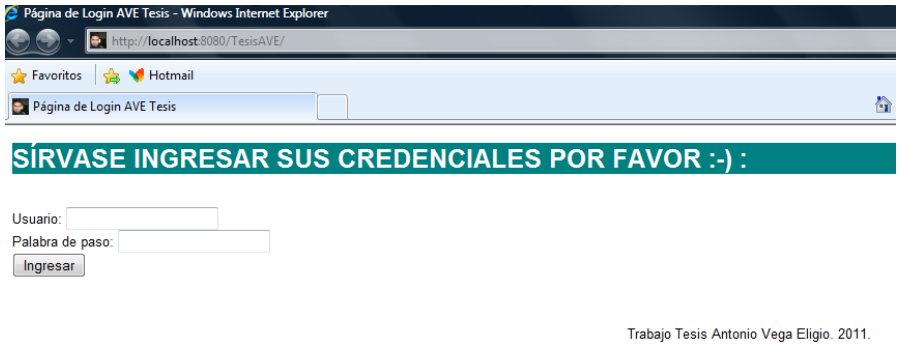


Figura 61 Página de firma de la aplicación TesisAVE.

e. El usuario y palabra de paso de la aplicación son: *antonio*/*antonio* (figura 61).

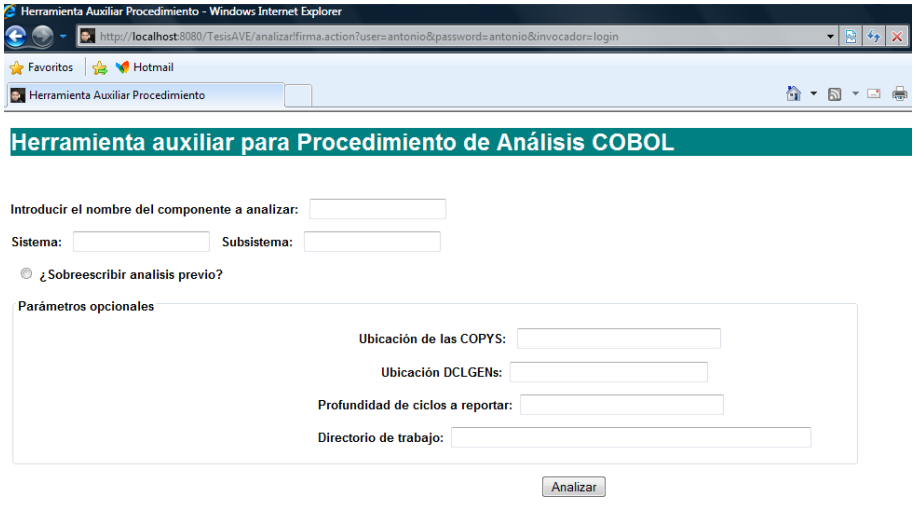


Figura 62 Página de inicio de la aplicación TesisAVE.

f. Se incluyó código *COBOL* de ejemplo de un programa en el directorio de trabajo del proyecto: "*/TesisAVE/trabajo*". En el campo "*Introducir el nombre del componente a analizar*" poner "*HA3CMAME*" y dar clic en el botón: "*Analizar*", ver figuras 62 y 63.

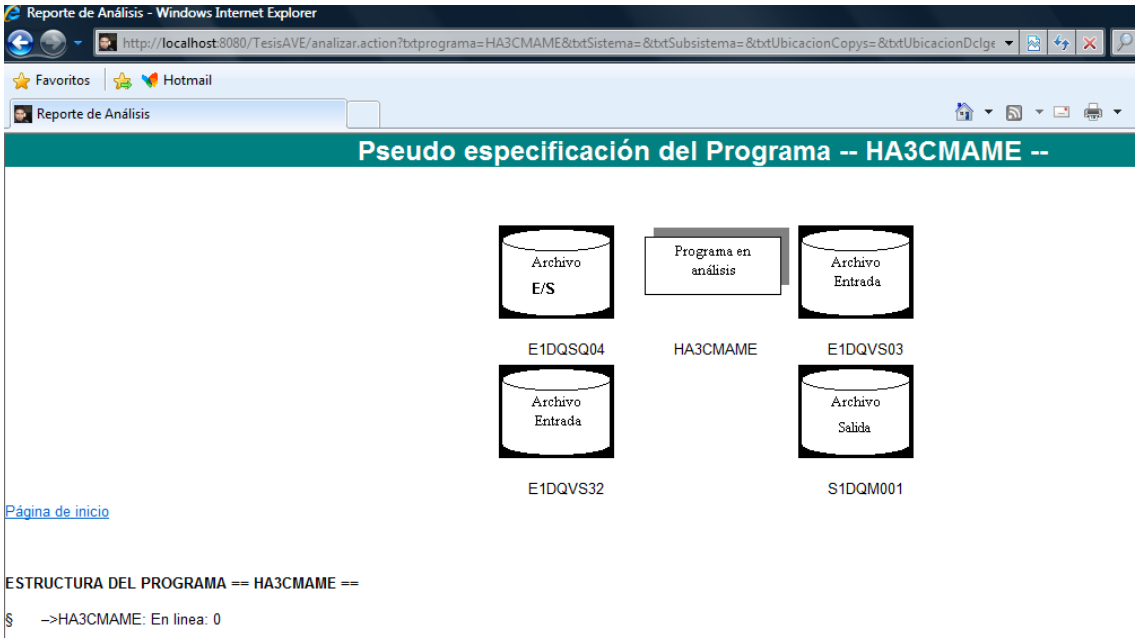


Figura 63 Página de reporte de resultados de la aplicación TesisAVE.

4.3.1.3 Compilar el código fuente con Netbeans.

Se incluyó el proyecto *Netbeans* en la carpeta llamada "*proyectoNetbeans*", dentro de la misma hay una subcarpeta: "*TesisAVE*", este es el proyecto *Netbeans* (figura 64).

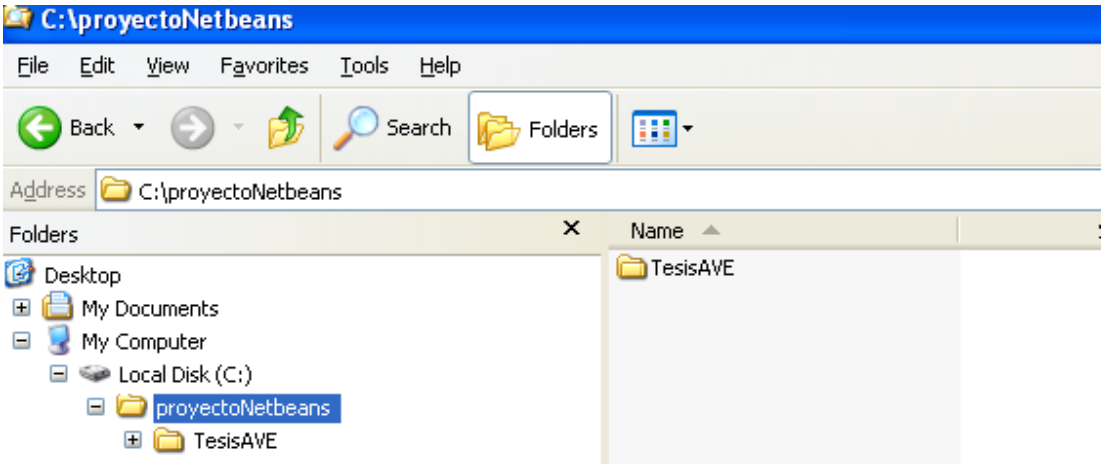


Figura 64 Carpeta del proyecto NetBeans TesisAVE.

Se requiere de tener instalado software de desarrollo para Java en la computadora donde se vaya a compilar el código. Los productos y versiones del software de desarrollo con las que se creó el entregable son las siguientes:

Versión: NetBeans IDE 6.9.1 (Build 201007282301).
Compilador Java: 1.6.0_22;
Máquina Virtual Java: Java HotSpot(TM) Client VM 17.1-b03;
Sistema: Windows XP version 5.1 running on x86; Cp1252

Son necesarias por lo menos las mismas versiones (o superiores) instaladas en la computadora de compilación. Los requisitos mencionados en el punto "4.3.1.2 Desplegar y ejecutar el proyecto en el servidor Web." también aplican cuando se va a recompilar el código.

a. Localizar la carpeta que contiene el proyecto *Netbeans* y abrirlo con el comando "*Open Project*" del menú archivo en la ventana principal de la aplicación *Netbeans*, ver la figura 65.

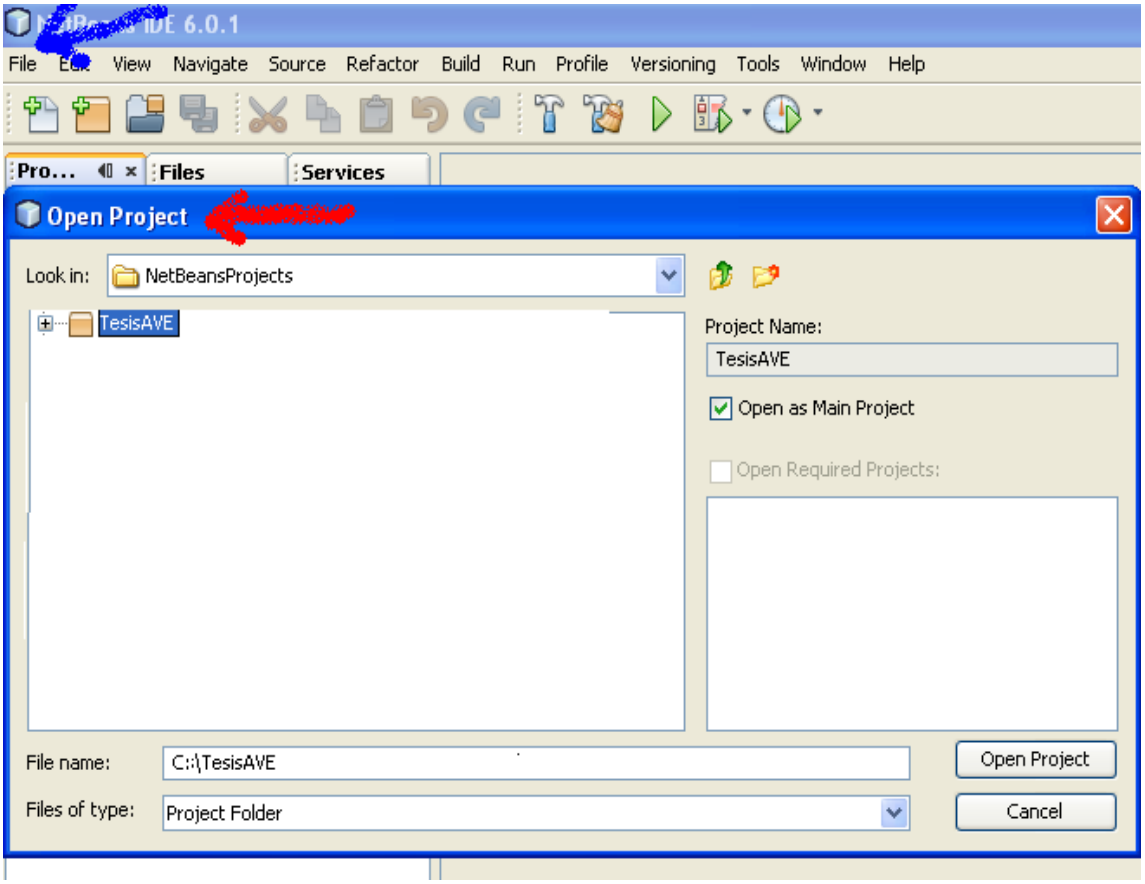


Figura 65 Apertura del proyecto NetBeans TesisAVE.

b. Una vez que el proyecto está abierto, asegurarse que el proyecto es de tipo Web y que todas las librerías que tiene la carpeta "*lib*" del proyecto *Netbeans* se han importado correctamente. En el caso de que el proyecto no se abra como proyecto Web, se debe crear un proyecto Web nuevo vacío con el mismo nombre ("*TesisAVE*") y todas las carpetas, excepto la de las librerías se deben copiar manualmente desde el disco entregado hasta las carpetas de trabajo de *NetBeans* del proyecto nuevo. Para las librerías, cuando estas no se importen en absoluto o se haga creación manual de un proyecto vacío nuevo, se deben importar de forma manual en *NetBeans*, desde la carpeta donde se importó el proyecto: Dar clic con botón derecho del ratón sobre la carpeta "*libraries*" del proyecto y seleccionar el comando "*Add JAR/Folder ...*". Ver la Figura 66.

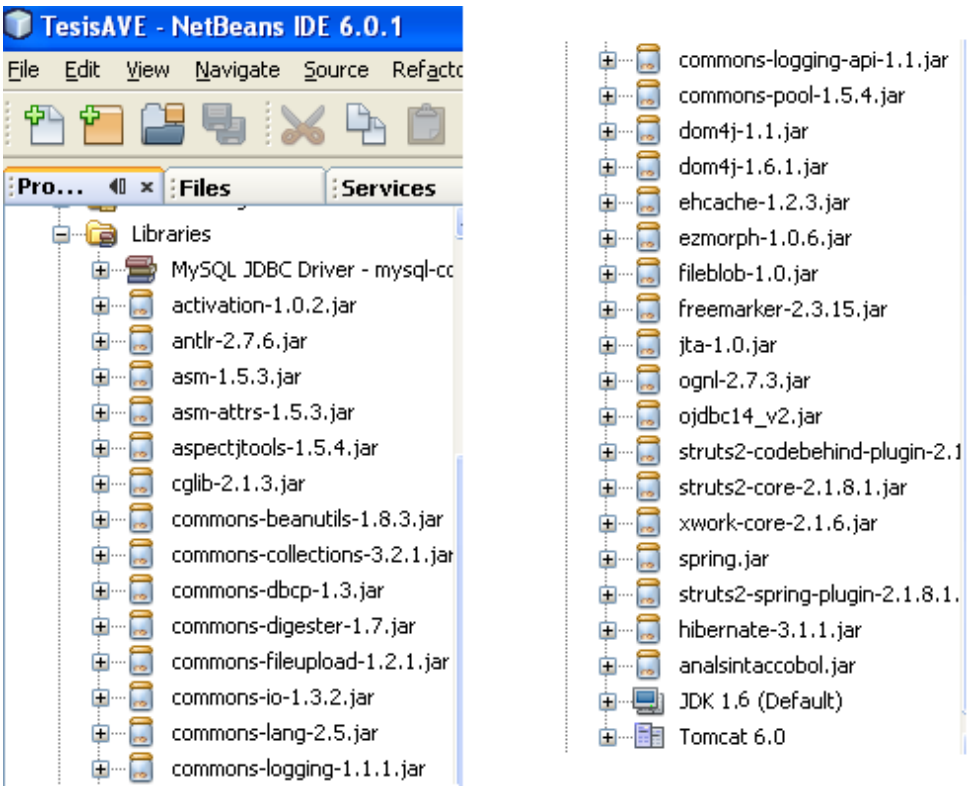


Figura 66 Librerías del proyecto NetBeans TesisAVE.

c. Usar el comando "*Clean and Build*". Poner el apuntador del ratón sobre el ícono del proyecto y pulsar el botón derecho para seleccionar la opción de comando "*Clean and Build*", ver figura 67.

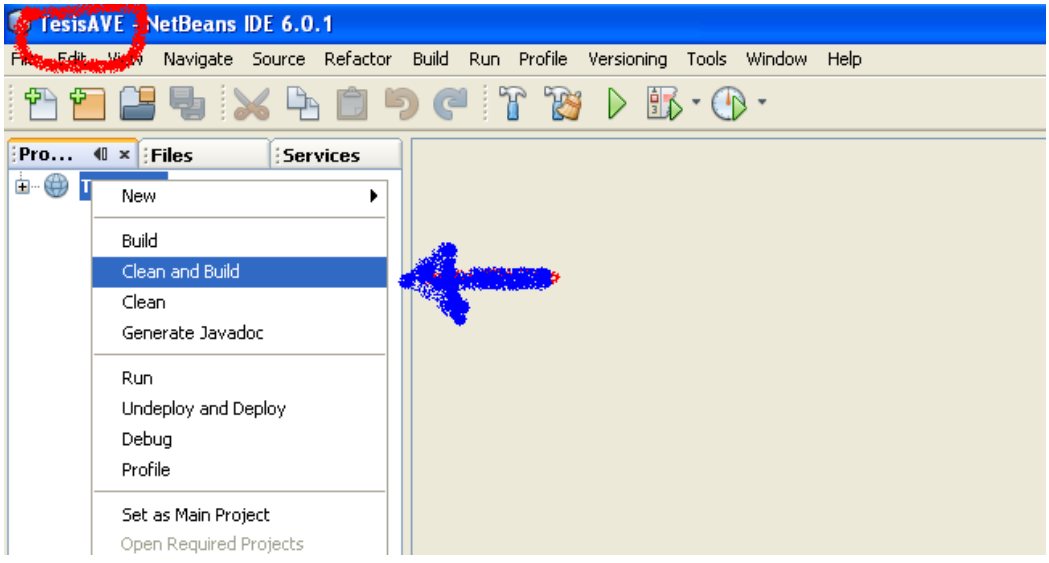


Figura 67 Compilación y construcción del proyecto NetBeans TesisAVE.

d. En el directorio donde estén ubicados los proyectos de *Netbeans* (varía en cada instalación) debe tener ahora una carpeta con el nombre del proyecto y la subcarpeta "*dist*", dentro de esta subcarpeta se debió haber generado el archivo "*TesisAVE.war*", ver figura 68.

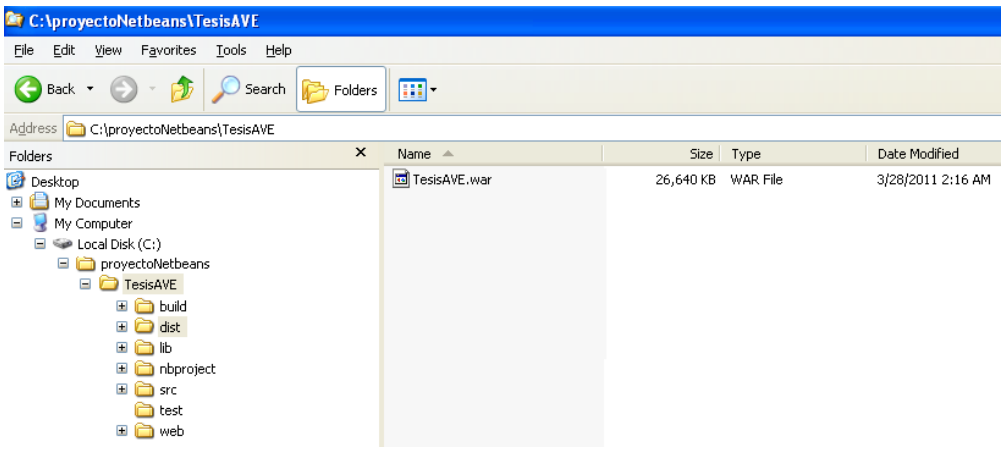


Figura 68 Carpeta de distribución del proyecto NetBeans TesisAVE.

e. Este archivo se puede usar a partir del paso número dos del punto: "4.3.1.2 Desplegar y ejecutar el proyecto en el servidor Web.".

4.3.1.4 Compilar el código fuente con ant.

El código fuente del proyecto Web está en la carpeta llamada "*codigo*", dentro de la misma hay 3 subcarpetas: "*lib*", "*src*" y "*web*". También hay un archivo llamado "*build.xml*" (figura 69).

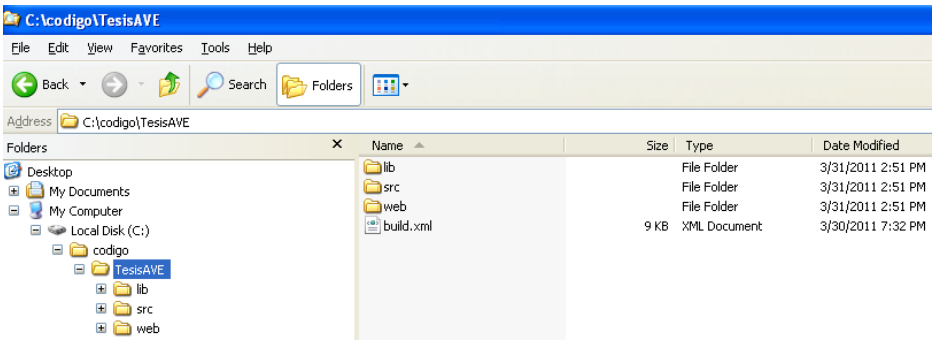


Figura 69 Carpeta de código del proyecto TesisAVE.

Para usar esta opción de compilación se debe tener instalado el software siguiente:

- Apache ant versión1.8.2 [122].

Y además, el Software mencionado en el punto "4.3.1.2 Desplegar y ejecutar el proyecto en el servidor Web."

a. Editar el script de "*ant*" dentro del archivo "*build.xml*". En la etiqueta "*project*" de la línea 1 cambiar el valor del atributo "*basedir*" por el de la ruta de trabajo actual de la carpeta "*/codigo/TesisAVE*" mencionada al inicio de este apartado.

*<**project** name="TesisAVE" default="all" basedir="ubicacion_actual_codigo/TesisAVE">*

b. Editar el script de ant dentro del archivo "*build.xml*". En la etiqueta "*property*" con el atributo "*name*" igual a "*catalina.home*" poner el directorio de trabajo del servidor Web Apache Tomcat en la instalación de la computadora donde se va a compilar.

<**property** name="catalina.home" value="*carpeta_trabajo_tomcat*"/>

c. Cuando se ha terminado la edición del script de ant, usar una ventana de comandos de *Windows* para ir al directorio donde se encuentra la carpeta: "*/codigo/TesisAVE*" y digitar el comando "*ant*" (figura 70).

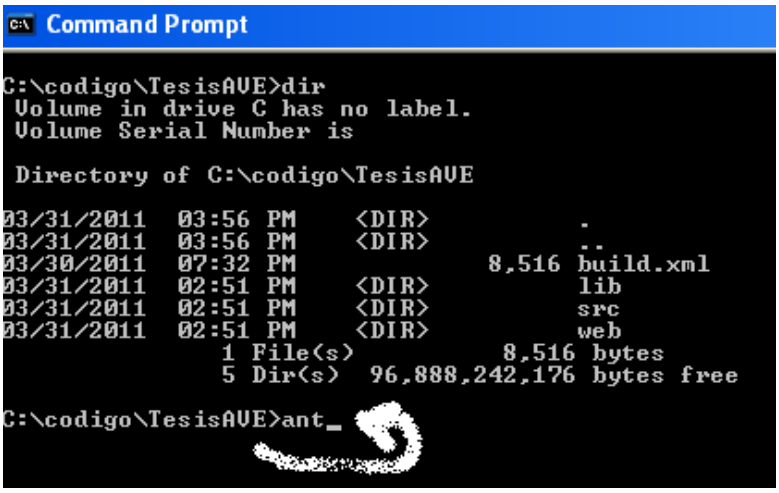


Figura 70 Comando "ant" en ventana de comandos DOS para compilar el proyecto TesisAVE.

d. Cuando el script *ant* termina de ejecutarse, se genera dentro de la estructura de directorios de "*codigo/TesisAVE*" la subcarpeta "*dist*", dentro de la misma se debe generar el archivo: "*TesisAVE.war*" (figura 71).

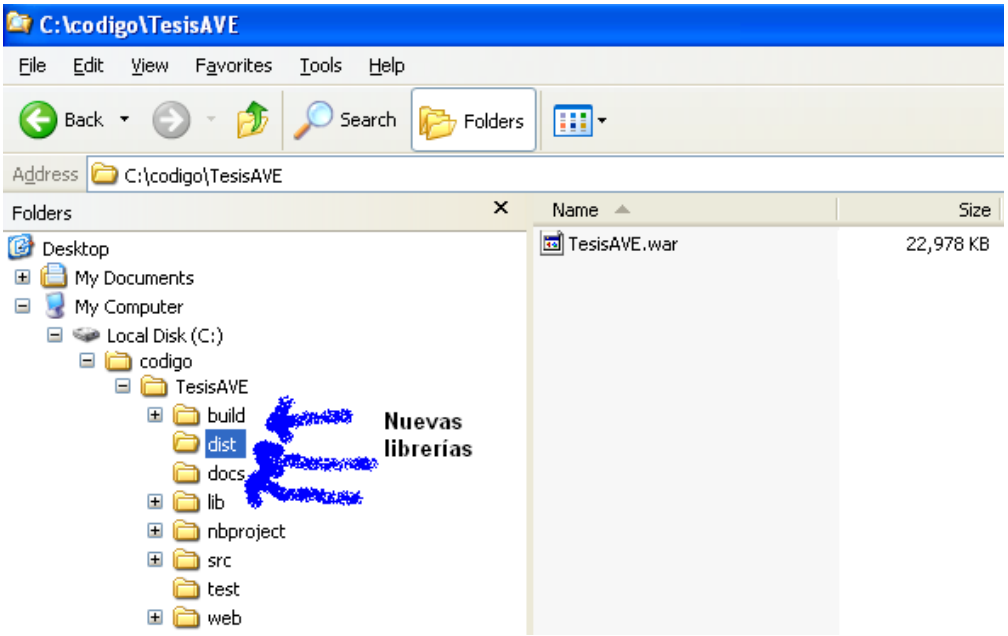


Figura 71 Carpetas generadas por el script "ant" de compilación del proyecto TesisAVE.

e. El script despliega la aplicación en el servidor Web por lo que a partir de aquí se puede retomar el paso número cuatro del punto: "4.3.1.2 Desplegar y ejecutar el proyecto en el servidor Web.".

4.3.2 CARACTERÍSTICAS DE LA INTERFAZ CON EL USUARIO.

La página de firma tiene dos campos: "*Usuario*" y "*Palabra de Paso*", para los cuales los valores son *antonio*/*antonio*. No se implementó una base de datos con usuarios autorizados, por lo que este usuario está configurado con código duro, ver la figura 72.

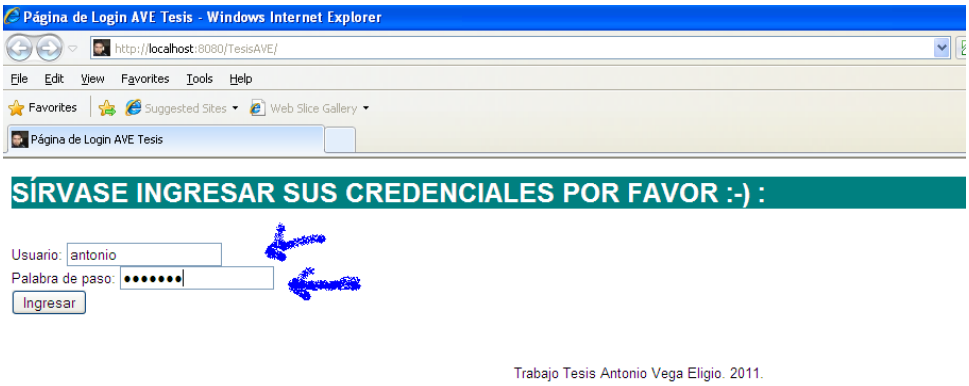


Figura 72 Página de firma de la aplicación TesisAVE.

La pantalla de inicio de la aplicación tiene siete campos de entrada funcionales. Se implementó una pantalla sencilla de error, pero no una validación rigurosa del valor de edición de los campos de entrada, ni tampoco una validación contra catálogos de valores válidos por lo que si un valor es ingresado erróneamente la aplicación simplemente lo tomará como nulo o auto-asignará un valor de defecto, de la misma manera si cumple las reglas de edición lo tomará como válido. Ver la figura 73.

1. "*Introducir el nombre del componente a analizar*". Es un campo de entrada de texto destinado a recibir el nombre del programa *COBOL* a analizar, este nombre de programa se debe introducir de 8 caracteres de longitud y además el archivo debe tener la extensión "txt" la cuál no se pone como parte del texto ingresado. Por ejemplo el programa con nombre: *HA3CMAME* debe estar almacenado en alguna carpeta de la computadora servidor donde se ejecuta el proyecto y su nombre físico debe ser: *HA3CMAME.txt*.

2. "*Sistema*". Es el nombre del aplicativo mainframe al que pertenece el componente a analizar este parámetro es opcional y el analizador pone el valor de defecto "*Sistema*", el campo tiene únicamente funcionalidad de prototipo.

3. "*Subsistema*" Es el nombre del sub aplicativo mainframe al que pertenece el componente a analizar este parámetro es opcional y el analizador pone el valor de defecto "*Subsistema*", el campo tiene únicamente funcionalidad de prototipo.

4. "*¿Sobre escribir análisis previo?*" Es un botón de control que si no se selecciona busca en la base de datos de conocimiento datos que pertenecen al programa que se ha solicitado analizar, al encontrarlos genera a partir de ellos el reporte de análisis. En caso de que no haya datos previos y el botón no esté seleccionado o bien que el botón este seleccionado, el aplicativo limpia cualquier hipotético dato existente en la base de datos para el programa y ejecuta el análisis sintáctico para extraer conocimiento funcional del código fuente.

5. "*Profundidad de ciclos a reportar*" Es un valor numérico que en caso de no ser alimentado o tener un número entero inválido se ajusta con el valor de defecto 1. El parámetro indica cuanta profundidad lógica en el flujo imperativo del programa *COBOL* se reporta.

6. "*Directorio de trabajo*". Es una ruta dentro del servidor donde el componente a analizar se encuentra (su nombre debe tener la extensión "*.txt*"). En caso de que no se introduzca un valor ahí, la aplicación busca el código en la carpeta "*/home/trabajo*" del proyecto Web. Si el archivo con el código del programa no existe, el aplicativo regresa a la pantalla de "*inicio*" y no hace algo.

7. "*Analizar*". Al dar clic a este botón se ejecuta el análisis sintáctico del programa y se presenta el reporte con los resultados de análisis.

Los demás elementos de entrada de la pantalla inicial no tienen funcionalidad implementada.

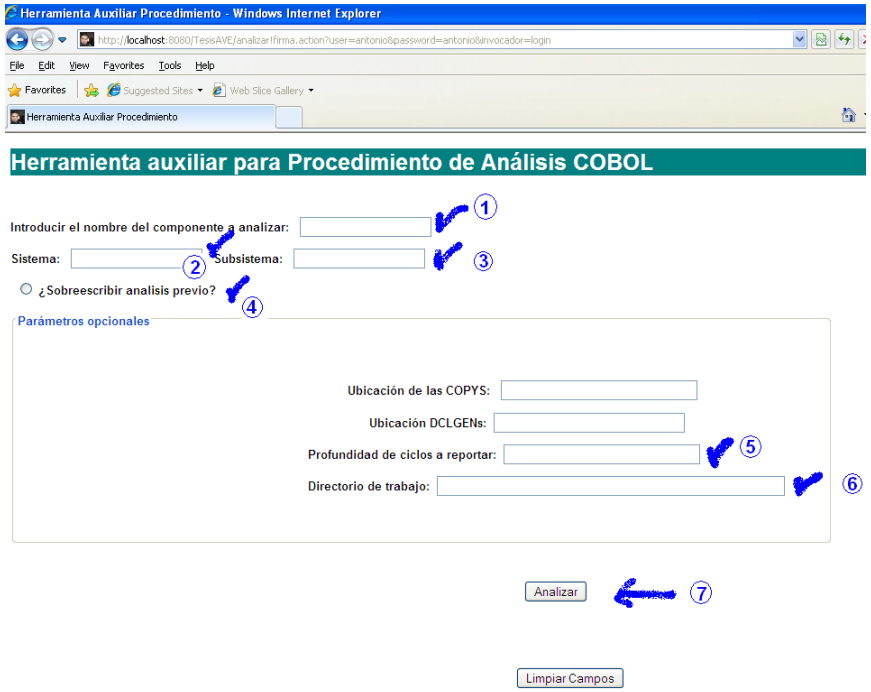


Figura 73 Página de inicio de la aplicación TesisAVE y sus campos de entrada.

El resultado del análisis se presenta en una nueva página, después de solicitar el análisis del componente. La página del reporte generado tiene 7 secciones, ver figuras 74 -77.

1. Sección "*Diagrama de entidades de entrada/salida de información*". La implementación se hizo únicamente para reconocer archivos de entrada y salida secuenciales (texto plano) declarados y usados en el programa.

2. Sección "*Estructura del programa*". Presentación en forma de resumen del flujo imperativo del programa.

3. Sección "*Funcionalidad*". A cada nodo dentro del árbol de ejecución imperativo del programa se le implementó la funcionalidad de poder actualizar por parte del analista cuál es la funcionalidad del nodo dentro del flujo.

4. Sección "*Variables de nivel 01*". Todas las estructuras de datos de más alto nivel (01) se reportan como información útil en esta sección.

5. Sección "*Instrucciones de Fin de Programa y Salto incondicional*". Se reportan las instrucciones que finalizan el programa, así como las instrucciones de salto incondicional. Estas son de interés para proporcionar información adicional sobre el flujo imperativo del programa.

6. Sección "*Llamada a módulos ejecutables externos*". Contiene los programas externos detectados en las instrucciones *CALL*.

7. Sección "*Buenas prácticas de programación*". Se incluyó un ejemplo de cómo el analizador puede sugerir reingeniería con validaciones tan simples como la de detectar que una instrucción *IF* se termina con un símbolo implícito (un punto), siendo que lo recomendable es usar el terminador explícito: *END-IF*.

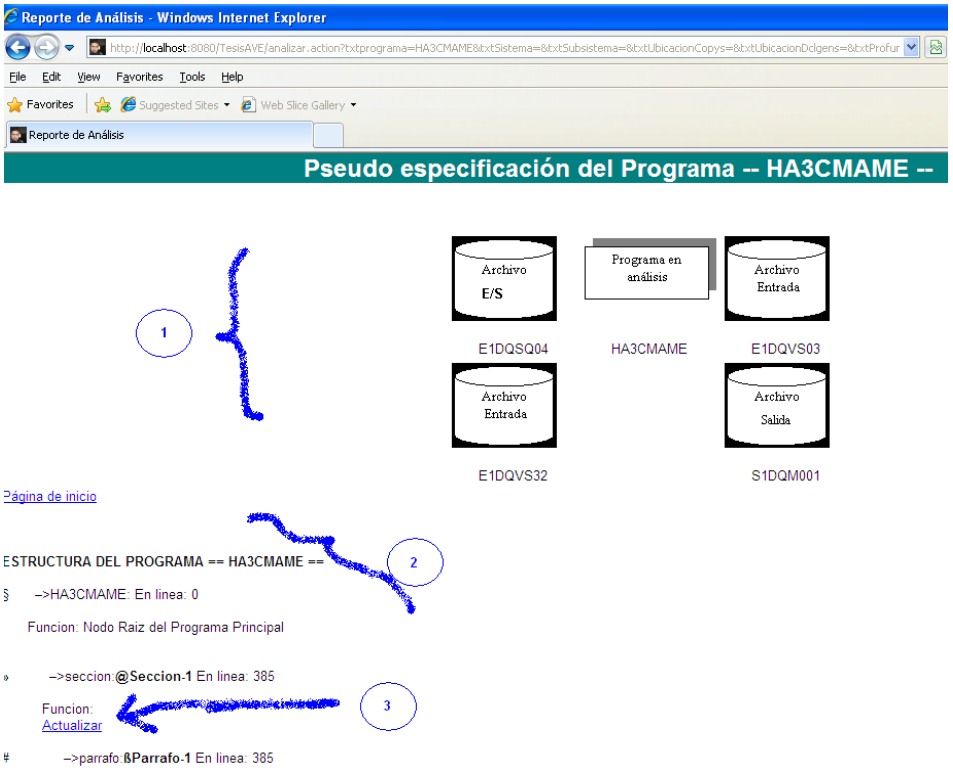


Figura 74 Secciones 1, 2 y 3 de la página de resultados de la aplicación TesisAVE.

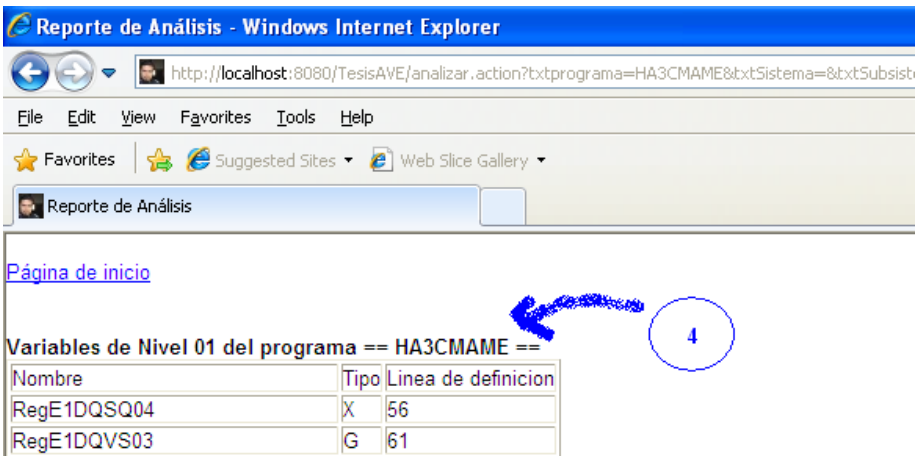


Figura 75 Sección 4 de la página de resultados de la aplicación TesisAVE.

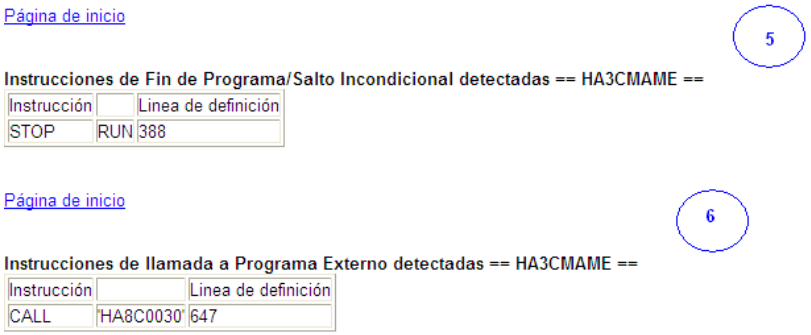


Figura 76 Secciones 5 y 6 de la página de resultados de la aplicación TesisAVE.

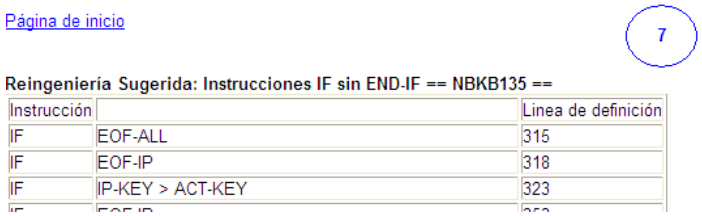


Figura 77 Sección 7 de la página de resultados de la aplicación TesisAVE.

Se implementó una pantalla para introducir descripción de funcionalidad de los nodos del árbol de flujo imperativo del programa, la ventana se muestra en la figura 78. Esta función tiene interacción con el usuario analista del aplicativo, en la página de reporte de resultados en cada nodo del flujo de ejecución (candidato a tener funcionalidad aplicativa) se agregó una hiperliga con la etiqueta "*Actualizar*", misma que al ser seleccionada con clic del ratón abre una ventana que permite incluir los comentarios del analista, esta ventana tiene 4 elementos de entrada.

1. Campo tipo caja de texto. "*Funcionalidad de nodo*". Este campo es de entrada/salida, al cargarse la ventana muestra el valor actual del campo, la funcionalidad del nodo del flujo de ejecución. Cuando el usuario introduce texto en este campo tiene la opción de salvar o descartar el nuevo valor.

2. Botón "*Actualizar*". Al momento de pulsar con el ratón el botón, el contenido de la caja de texto "Funcionalidad de nodo" se actualiza en la base de datos, se cierra la ventana y el nuevo valor descriptivo se actualiza en el reporte principal.

3. Botón "*Limpiar Campos*". Se pone un valor de defecto en la caja de texto que describe la funcionalidad.

4. Botón "*Regresar a Resultados*". Es equivalente a dar clic en la marca que cierra la ventana en la esquina superior derecha, cierra la ventana sin actualizar algo.

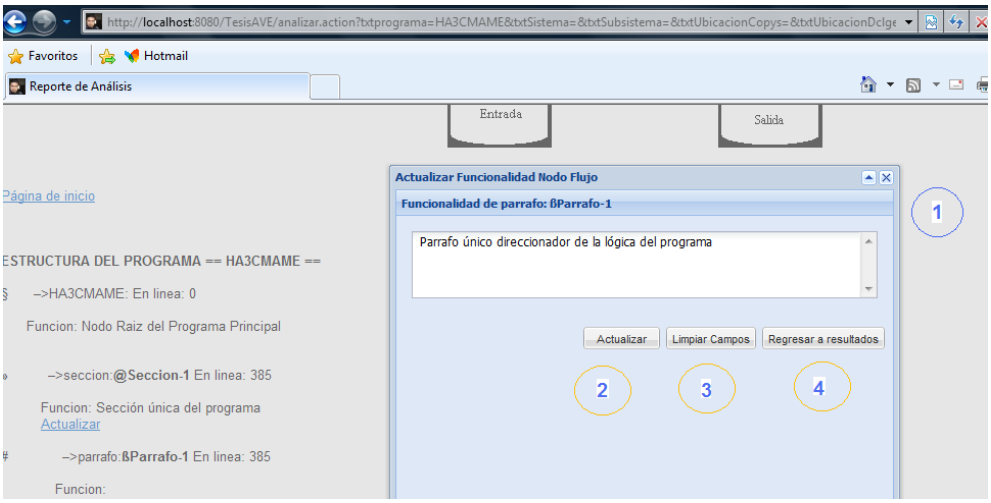


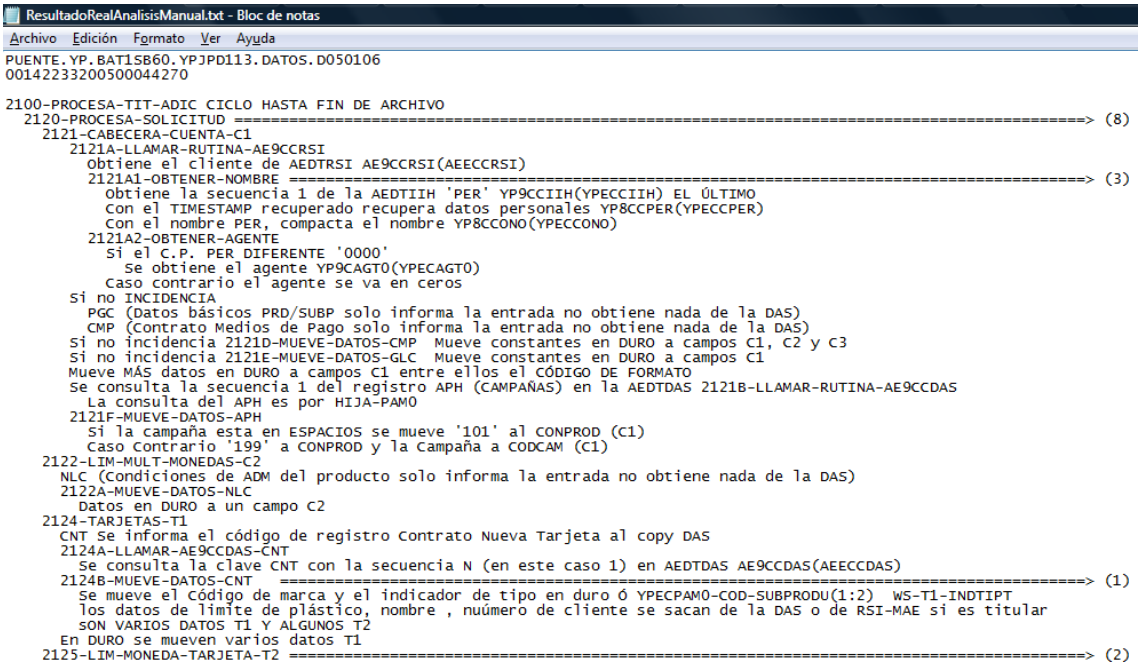
Figura 78 Ventana de actualización de funcionalidad de nodo en la página de resultados.

4.4 EVALUACIÓN DEL PRODUCTO.

Un procedimiento como el descrito en los primeros tres capítulos de esta investigación solo puede ser evaluado de manera rigurosa después de su utilización exhaustiva sobre diferentes situaciones de entrada y después de haber colectado una serie de datos estadísticos para poder concluir si su seguimiento es benéfico comparado con la manera anterior de hacer las cosas. Este tipo de evaluación rigurosa no fue implementado en esta investigación debido a las limitaciones de tiempo que se tienen. Tampoco es factible ya que es necesario tramitar muchos permisos para ejecutar el procedimiento propuesto en ambientes de producción de organizaciones reales.

Sin embargo, como todo resultado en una investigación de naturaleza aplicativa, es de interés evaluar en cierto nivel mínimo si la misma cumplió el objetivo proyectado. Con base en la experiencia del autor: si en varios proyectos en la plataforma mainframe para los cuales el análisis se hizo manualmente, se hubiese tenido la posibilidad de ayudarse de un procedimiento que proporcionara elementos extraídos de manera automática desde el código, tales como las entidades de entrada y salida y el flujo imperativo de ejecución al nivel de secciones y párrafos, esto hubiese sido una ayuda significativa.

Muchas estrategias para analizar código de manera manual inician con la tarea visual de navegación en el código para ubicar los componentes significativos que aportan conocimiento funcional y de arquitectura del aplicativo heredado. La figura 79 presenta el fragmento de resultado de un análisis manual realizado sobre un programa que genera una interfaz para producir los plásticos de un contrato de tarjeta de crédito. El código exacto del componente no se puede presentar en este trabajo, pero este resultado de su análisis fue la inspiración de la presente investigación.


ResultadoRealAnalisisManual.txt - Bloc de notas

Archivo Edición Formato Ver Ayuda

```
PUENTE.YP.BAT1SB60.YPJPD113.DATOS.D050106
00142233200500044270

2100-PROCESA-TIT-ADIC CICLO HASTA FIN DE ARCHIVO
  2120-PROCESA-SOLICITUD ==========================================================================================> (8)
    2121-CABECERA-CUENTA-C1
       2121A-LLAMAR-RUTINA-AE9CCRSI
         Obtiene el cliente de AEDTRSI AE9CCRSI(AEECCRSI)
         2121A1-OBTENER-NOMBRE ======================================================================================> (3)
           Obtiene la secuencia 1 de la AEDTIIH 'PER' YP9CCIIH(YPECCIIH) EL ÚLTIMO
           Con el TIMESTAMP recuperado recupera datos personales YP8CCPER(YPECCPER)
           Con el nombre PER, compacta el nombre YP8CCONO(YPECCONO)
         2121A2-OBTENER-AGENTE
           Si el C.P. PER DIFERENTE '0000'
             Se obtiene el agente YP9CAGT0(YPECAGT0)
           Caso contrario el agente se va en ceros
       Si no INCIDENCIA
         PGC (Datos básicos PRD/SUBP solo informa la entrada no obtiene nada de la DAS)
         CMP (Contrato Medios de Pago solo informa la entrada no obtiene nada de la DAS)
       Si no incidencia 2121D-MUEVE-DATOS-CMP  Mueve constantes en DURO a campos C1, C2 y C3
       Si no incidencia 2121E-MUEVE-DATOS-GLC  Mueve constantes en DURO a campos C1
       Mueve MÁS datos en DURO a campos C1 entre ellos el CÓDIGO DE FORMATO
       Se consulta la secuencia 1 del registro APH (CAMPAÑAS) en la AEDTDAS 2121B-LLAMAR-RUTINA-AE9CCDAS
         La consulta del APH es por HIJA-PAM0
       2121F-MUEVE-DATOS-APH
         Si la campaña esta en ESPACIOS se mueve '101' al CONPROD (C1)
         Caso Contrario '199' a CONPROD y la Campaña a CODCAM (C1)
    2122-LIM-MULT-MONEDAS-C2
      NLC (Condiciones de ADM del producto solo informa la entrada no obtiene nada de la DAS)
      2122A-MUEVE-DATOS-NLC
        Datos en DURO a un campo C2
    2124-TARJETAS-T1
      CNT Se informa el código de registro Contrato Nueva Tarjeta al copy DAS
      2124A-LLAMAR-AE9CCDAS-CNT
        Se consulta la clave CNT con la secuencia N (en este caso 1) en AEDTDAS AE9CCDAS(AEECCDAS)
      2124B-MUEVE-DATOS-CNT   ====================================================================================> (1)
        Se mueve el Código de marca y el indicador de tipo en duro ó YPECPAM0-COD-SUBPRODU(1:2)  WS-T1-INDTIPT
        los datos de limite de plástico, nombre , nuúmero de cliente se sacan de la DAS o de RSI-MAE si es titular
        SON VARIOS DATOS T1 Y ALGUNOS T2
      En DURO se mueven varios datos T1
    2125-LIM-MONEDA-TARJETA-T2 ==================================================================================> (2)
```

Figura 79 Ejemplo de resultado de análisis manual de un programa COBOL.

En primer lugar: el resultado no pareciera ser digerible, sin embargo se debe tener en cuenta que el nicho de usuarios de un resultado como estos es un analista o un arquitecto de aplicaciones mainframe. El propósito de miles de líneas de código *COBOL* muchas veces puede ser reducido a pocas expresiones en lenguaje natural, cosa que se tiene ilustrada en la figura. Se obvian también en esta síntesis muchos aspectos concernientes a la implementación técnica de fondo, por ejemplo el nombre de las bases de datos, la conexión con el motor de base de datos o la conexión con el monitor de transacciones o comandos con el sistema operativo, de todas maneras en un programa real dichos detalles de la arquitectura no se reflejan en el código sino en la configuración de los servidores y parámetros del sistema operativo.

El usuario de un resumen de este tipo es alguien que tiene familiaridad con los conceptos, si estamos en el entorno de una organización bancaria, se asume que un analista ya tiene varias nociones de qué cosa es "el módulo de clientes" el concepto involucra que hay una base de datos y una serie de programas que hacen operaciones *CRUD* sobre esos datos y además que la información tiene un cierto perfil: son datos única y exclusivamente relacionados al manejo de la entidad de información *Cliente* (de un banco). Como se puede ver, aún el resultado del análisis manual tiene abreviaciones. Y aún con esas abreviaciones, el resumen ayuda de manera significativa y directa cuando se quiere implementar una nueva funcionalidad, o bien cuando se está buscando el origen de un funcionamiento no esperado (error).

Los detalles mostrados relativos a qué concepto de negocio está implementado en cada grupo de atributos de la información al nivel de código fuente, pertenecen a un nivel mucho más abstracto que no fue atacado de manera aplicativa en este trabajo. Se discutieron varias posibilidades y enfoques en el marco teórico, lo cual puede servir de guía para futuras investigaciones y son entre otros los algoritmos heurísticos de selección de reglas de negocio implementadas, algoritmos de traducción al lenguaje natural, minería de datos o inclusive algoritmos de inteligencia artificial los cuales en un determinado momento llegasen a enunciar frases del tipo: "*En la línea de código N se está aplicando la comisión por uso de cajero automático al cliente*" de manera totalmente automática. En el diseño de la herramienta se propusieron algoritmos muy sencillos, se pudiera decir que hasta primitivos, que podrían identificar este tipo de gemas de conocimiento, sin embargo esto ya no se alcanzó a cubrir la implementación.

Con esto se concluye que: El motor generador de conocimiento (el "compilador" parcial) fue el mayor aporte y el mismo sirvió de base para la implementación de un primer prototipo limitado de como explotar los datos generados, esto sería una primera iteración de un entregable más ambicioso.

4.4.1 COMENTARIOS SOBRE LOS RESULTADOS OBTENIDOS.

Gran parte del procedimiento propuesto en el capítulo 3 se basa en el artefacto de software que se diseñó y que fue parcialmente implementado. Al inicio de la investigación se propuso obtener algo con utilidad práctica. Se ha documentado también que el tiempo límite para finalizar la investigación fue rebasado por la desviación en tiempo que tuvo la valoración inicial propuesta para la implementación del analizador sintáctico. La medida de control para atenuar la desviación sin perder el objetivo fue la de recortar la funcionalidad de los módulos del artefacto de software. En cierto momento de la implementación se detuvo la tarea de depurar los errores de los analizadores léxico y sintáctico, el hito que marcó el cierre de tema fue cuando el módulo de recuperación de errores pasó la prueba de recuperarse de errores agregados intencionalmente y cuando además fue capaz de reconocer en un rango superior al 60% del total de instrucciones *IF* anidadas, *PERFORM*, *OPEN* y *MOVE* presentes en los programas de prueba. La decisión sobre estas instrucciones en específico fue porque ya se iniciaba la proyección de una interfaz gráfica recortada. El reporte final de conocimiento en realidad quedó como una pequeña muestra de lo que se puede hacer con la información que obtiene un compilador de *COBOL*. Se escogieron estratégicamente esas instrucciones porque, desde el punto de vista del autor, cuando las mismas aparecen en cualquier lugar dentro del código dan una idea “a primera vista” de lo que el programa hace. Se le dio prioridad también al flujo imperativo de ejecución del programa a analizar, esto quiere decir que se tuvo cuidado que la estructura de alto nivel imperativo fuese reconocida por el analizador. Lo mismo aplicó en el caso del uso de un tipo especial de archivos de entrada/salida: los archivos de acceso secuencial, presentes en una gran cantidad de programas de producción.

Habiendo mencionado el panorama general del entregable, se enuncian las siguientes consideraciones sobre la sección medular del resultado: El reporte del flujo imperativo.

1. Del total de instrucciones que el analizador sintáctico pueda reconocer, el módulo de normalización de conocimiento está implementado para almacenar en la base de datos

símbolos terminales de únicamente 18 lexemas para su posterior explotación. Estos lexemas son: Las reglas de producción de la gramática que definen la macro estructura imperativa, la definición de variables, las instrucciones de fin de programa y los verbos mencionados en el párrafo inicial de este tema.

2. El reporte del flujo imperativo se basó en una premisa muy simple, *COBOL* se ejecuta en bloques de código que contienen de una a *N* instrucciones. Y esos bloques están contenidos en un nodo/bloque padre llamado "*PROCEDURE DIVISION*" entonces, desde el punto de vista de más alto nivel, cualquier programa *COBOL* ejecuta el "nivel lógico cero": La "*PROCEDURE DIVISION*". A partir de ahí el nivel de profundidad del detalle es como sigue:

   PROCEDURE DIVISION – *Nivel lógico de ejecución 0.*
   - ➔ SECTION (1,N) – *Nivel lógico de ejecución 1.*
     - PARAGRAPH (1,N) – *Nivel lógico de ejecución 2.*
       - SENTENCE (1,N) – *Nivel lógico de ejecución 3.*
         - STATEMENT (1,N) – *Nivel lógico de ejecución 4.*
           - [STATEMENT] (0,N) …
             - … [STATEMENT] (0,N) …
               - …

   Este es el flujo imperativo secuencial de todo programa *COBOL* si y solo sí no hubiese instrucciones de bifurcación del flujo (*GO TO*, *PERFORM*). La primera sección con sus uno a *N* instrucciones se ejecuta y después la segunda sección y así hasta el fin de la *PROCEDURE DIVISION*. Lo mismo aplica para los elementos contenidos dentro de otro, es decir dentro de una sección se ejecuta el primer párrafo, después el segundo y así hasta el fin de la sección. Los elementos de más bajo nivel en esta macro estructura son las instrucciones "derivadas", es decir instrucciones contenidas dentro de otra instrucción, por ejemplo una instrucción *IF*, puede tener en el cuerpo de *IF* una serie de instrucciones de uno a *N* de nivel lógico "inferior", dentro de este cuerpo de la instrucción *IF* se ejecuta la primera instrucción, después la segunda que aparezca de manera secuencial y así hasta ejecutar la última instrucción en el cuerpo de *IF*. Y así por el estilo con todos los elementos estructurales.

3. El reporte se implementó con la estrategia de presentar un resumen de "nombres de nodos", que son los elementos de la macro estructura imperativa presentados arriba. Esto para un nivel lógico de ejecución dinámico a petición del usuario. El nivel cero no está implementado por considerarse irrelevante (solo aparecería algo como "nodo cero – *PROCEDURE DIVISION*"). Hasta el elemento "*SENTENCE*" todo se almacena en la base de datos y se presenta en el reporte cuando es solicitado en el campo "Profundidad de ciclos a reportar" de la *GUI*, que es la profundidad en el nivel lógico de ejecución secuencial. Para "*STATEMENT*" solo se almacenan (y reportan) los verbos mencionados al inicio de este punto.

4. No se implementó la presentación de la gráfica para el caso de saltos con instrucciones de bifurcación. Cuando se analiza el programa, se almacenan estos saltos en la base de datos de conocimiento, pero no se reportan de manera visual en la *GUI*. En su lugar son reportados, a partir del nivel lógico 3, los verbos *GO TO* y *PERFORM* almacenados por el analizador sintáctico, esto debería dar una señal al analista, el usuario, que hay saltos, con o sin condicionante (y el nodo destino del salto), sin que estén explícitamente "dibujados" en el reporte de flujo.

5. La estrategia anteriormente descrita proporciona en la práctica un nivel bastante parecido al resultado manual presentado en la figura 79, este resultado durante la prueba con programas reales del orden de 600 líneas dio una idea básica acerca de lo que el programa hacía "a primera vista", sin embargo todavía necesita que lo interprete alguien que está familiarizado con los programas *COBOL (*los verbos y el flujo imperativo de ejecución*)*, es decir no hay garantía que alguien que no conozca el lenguaje pueda inferir cual es la funcionalidad del programa al primer vistazo del reporte que genera la herramienta. El traductor de verbos para reportar las instrucciones en un lenguaje coloquial tampoco se implementó (por ejemplo cambiar –traducir- "*MOVE*" por "Mueve valor a …" o "*CALL*" por "Llama a modulo externo …", etcétera)

6. Se agregó a la interfaz de usuario la posibilidad de que el usuario agregue sus comentarios que describen de forma más abstracta la funcionalidad de los bloques de código –nodos– de la macro estructura de flujo imperativo. Esto ilustra cómo se puede refinar la herramienta implementando funciones similares en otras secciones del reporte, lo que quedó fuera de alcance.

7. Las demás secciones del reporte son el diagrama de archivos de entrada y salida, de funcionalidad entendible a primera vista, la lista de variables de tipo estructura con nivel 01, la lista de instrucciones de fin de programa y de salto incondicional detectadas, la lista de instrucciones de llamado a módulos compilados de forma externa y la lista de instrucciones *IF* detectadas sin su clausula *END-IF*. Esta información es también de gran utilidad cuando se analiza manualmente un programa y son solo listas, es decir no hay algo especial que se deba interpretar, es información útil para el analista a primera vista.

8. A continuación se presenta el resultado obtenido por la herramienta desarrollada para tres programas de prueba y algunos comentarios.

4.4.2 CASO DE PRUEBA 1. COMPORTAMIENTO DEL ANALIZADOR SINTÁCTICO.

- Prueba ejecutada: Programa 1.

Funcionalidad real del programa: Existe un libro de contabilidad, el mayor mensual de una organización que muchas veces requiere ajustes basado en el mayor diario, el programa genera un archivo secuencial con una instrucción de actualización *SQL* que acumula los importes al debe y haber del mayor diario en su mes correspondiente del mayor mensual.

Exactitud sintáctica reconocida.
Instrucciones totales del programa: 134
Instrucciones que se esperaba fueran reconocidas: 134
Instrucciones efectivamente reconocidas: 133 (Figura 80).
Porcentaje de efectividad: 99.25%
Líneas del programa: 652
Elementos léxicos reconocidos: 2342

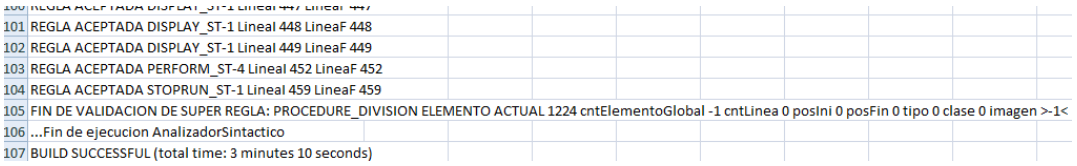


Figura 80 Ejecución del analizador sintáctico programa 1.

Comentarios de la prueba: Instrucción no reconocida línea 373. Lectura de un archivo *VSAM (Virtual Storage Access Method)* indexado, la clausula *INVALID KEY* tiene error en su implementación en el código del analizador léxico:

READ E1DQVS03
INVALID KEY
Perform WtiteNotFound
NOT INVALID KEY
Perform ForEach03
END-READ

- Prueba ejecutada: Programa 2.

Funcionalidad real del programa: Es un proceso intermedio de manejo de interfaces (archivos planos de texto). Toma un historial de "tecleos" (cargos/abonos) y selecciona solo renglones que tienen el valor "DR" en uno de sus campos, además se da un formato especial al renglón de salida.

Exactitud sintáctica reconocida.
Instrucciones totales del programa: 134
Instrucciones que se esperaba fueran reconocidas: 134
Instrucciones efectivamente reconocidas: 96 (Figura 81).
Porcentaje de efectividad: 71.64%
Líneas del programa: 501
Elementos léxicos reconocidos: 1991

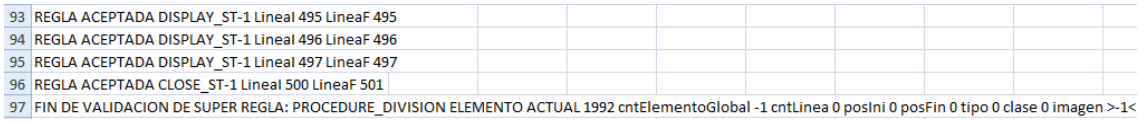


Figura 81 Ejecución del analizador sintáctico programa 2.

Comentarios de la prueba: 38 instrucciones no reconocidas, debido a lo siguiente:

1. Instrucción *CONTINUE* no implementada y donde se encuentre en medio de una instrucción de nivel lógico superior hace fallar el reconocimiento de las instrucciones en cascada.
2. Tipo de elemento léxico no implementado de forma completa: una variable con nombre "*A*" ó "*Z*" ó "*X*" es reconocida por el analizador léxico únicamente como cadena *PIC* (en la definición de tipo de una variable) y debe ser también reconocida como palabra *COBOL* definida por el usuario (el nombre de la variable) y usada en instrucciones. Todas las instrucciones con esta carencia incluidas en otra de nivel lógico superior son no reconocidas en cascada, 21 en total.

IF WF-STATUS-TLF-OK

```
  CONTINUE
ELSE
  MOVE  WF-STATUS-TLF           TO WE-FILE-STATUS
END-IF

EVALUATE WF-STATUS-TLF
  WHEN '00'
    CONTINUE
  WHEN OTHER
     DISPLAY 'STATUS ARCHIVO DE ENTRADA: ' WF-STATUS-TLF
END-EVALUATE
```

- Prueba ejecutada: Programa 3.

Funcionalidad real del programa: Programa que lee un archivo cuyos renglones tienen campos con un formato especial de número comprimido (*COMP*) en el código de caracteres usado en la plataforma mainframe *IBM®*, *EBCDIC* (Extended Binary Coded Decimal Interchange Code, basado en el código de tarjetas perforadas) y los convierte a números editados (código *ASCII* con formato de moneda, separador de miles o supresión de ceros)

Exactitud sintáctica reconocida.
Instrucciones totales del programa: 113
Instrucciones que se esperaba fueran reconocidas: 113
Instrucciones efectivamente reconocidas: 104 (Figura 82).
Porcentaje de efectividad: 92.03%
Líneas del programa: 462
Elementos léxicos reconocidos: 1223

```
99  REGLA ACEPTADA DISPLAY_ST-1 Lineal 446 LineaF 446
100 REGLA ACEPTADA DISPLAY_ST-1 Lineal 447 LineaF 447
101 REGLA ACEPTADA DISPLAY_ST-1 Lineal 448 LineaF 448
102 REGLA ACEPTADA DISPLAY_ST-1 Lineal 449 LineaF 449
103 REGLA ACEPTADA PERFORM_ST-4 Lineal 452 LineaF 452
104 REGLA ACEPTADA STOPRUN_ST-1 Lineal 459 LineaF 459
105 FIN DE VALIDACION DE SUPER REGLA: PROCEDURE_DIVISION ELEMENTO ACTUAL 1224 cntElementoGlobal -1 cntLinea 0 posIni 0 posFin 0 tipo 0 clase 0 imagen >-1<
106 ...Fin de ejecucion AnalizadorSintactico
107 BUILD SUCCESSFUL (total time: 3 minutes 10 seconds)
```

Figura 82 Ejecución del analizador sintáctico programa 3.

Comentarios de la prueba: Instrucción *CONTINUE* no implementada y donde se encuentre en medio de una instrucción de nivel lógico superior hace fallar el reconocimiento de las instrucciones en cascada, 8 instrucciones no reconocidas.

Instrucción no reconocida línea 451. Variable de registro especial *RETURN-CODE* no implementada como variable destino de la instrucción *MOVE*.

```
MOVE  WX-CODRET               TO  RETURN-CODE.
```

4.4.3 CASO DE PRUEBA 2. ENTIDADES ENTRADA/SALIDA.

- Prueba ejecutada: Programa 1.

Archivos secuenciales declarados en el programa: 4.
Archivos secuenciales reconocidos por el analizador: 4 (Figura 83).
Efectividad: 100%

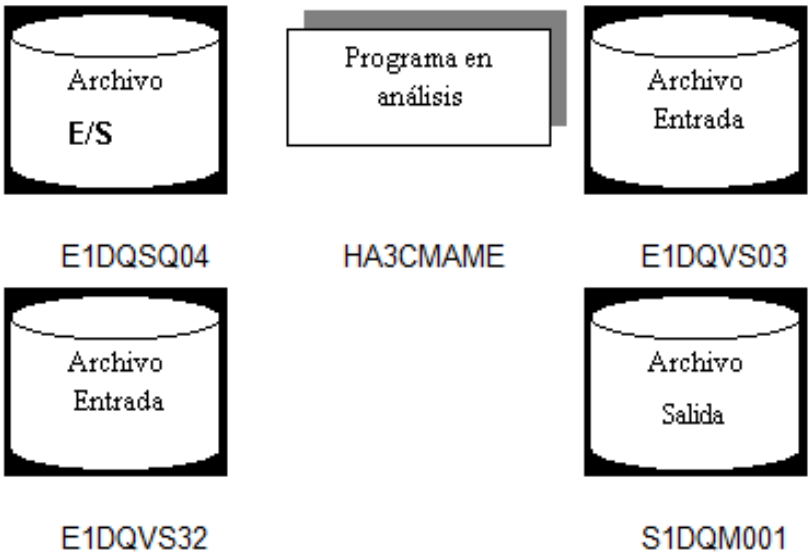


Figura 83 Archivos de entrada y salida del programa 1.

- Prueba ejecutada: Programa 2.

Archivos secuenciales declarados en el programa: 2.
Archivos secuenciales reconocidos por el analizador: 2 (Figura 84).
Efectividad: 100%

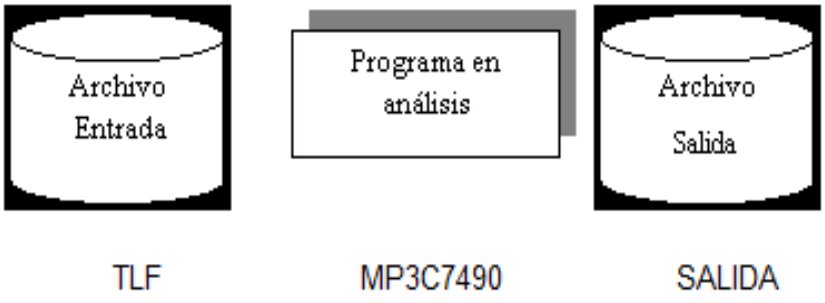


Figura 84 Archivos de entrada y salida del programa 2.

- Prueba ejecutada: Programa 3.

Archivos secuenciales declarados en el programa: 2.
Archivos secuenciales reconocidos por el analizador: 2 (Figura 85).
Efectividad: 100%

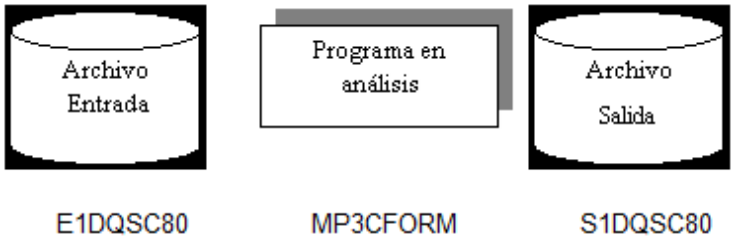


Figura 85 Archivos de entrada y salida del programa 3.

4.4.4 CASO DE PRUEBA 3. FLUJO IMPERATIVO DE EJECUCIÓN.

- Prueba ejecutada: Programa 1.

Se solicitó el flujo imperativo del programa con profundidad lógica 3. El reporte proporcionado al nivel uno de instrucción ejecutada en el flujo proporciona un resumen adecuado de la estructura imperativa. Se observa un primer bloque que direcciona el flujo a varios bloques definidos posteriormente y uno de ellos es un ciclo iterativo (instrucción *PERFORM UNTIL*), ver la figura 86. El programa está estructurado de modo aceptable porque posteriormente se puede apreciar que cada bloque definido tiene una funcionalidad descrita al nivel general por las instrucciones reportadas, el reporte proporciona el perfil de una lectura de renglones de un archivo, un procesamiento con los campo de información leídos y la escritura de un archivo de salida, el nivel de conocimiento acerca de lo que hace el programa quedaría complementado si se sabe que información representan el archivo de entrada y salida al nivel funcional, lo que se puede agregar de manera manual con la funcionalidad de actualizar la funcionalidad de los nodos de flujo imperativo.

Figura 86 Flujo imperativo de ejecución del programa 1.

- Prueba ejecutada: Programa 2 (Figura 87).

La estructura del programa es reportada de manera muy ilustrativa cuando se usa el nivel lógico 3. Se puede asumir que este nivel lógico reporta de manera adecuada el flujo imperativo en programas razonablemente estructurados.

Figura 87 Flujo imperativo de ejecución del programa 2.

- Prueba ejecutada: Programa 3.

Este programa tiene una estructura sencilla, por lo que se intentó un reporte sencillo en la sección de reporte del flujo imperativo. Sin embargo cuando se reporta nivel lógico 1 en los ciclos de ejecución, el flujo generado parece no tener sentido. Se solicitó entonces el reporte de nivel lógico 2, de igual forma no se aprecia a primera vista algo que de pista de qué es lo que hace el programa (la funcionalidad real está descrita en el caso de prueba 1), por lo que se solicitó el nivel lógico 3, se observa que las instrucciones seleccionadas para ser consideradas en el reporte de conocimiento ayudan a presentar de manera visual un reporte que a da una idea resumida del perfil inicial de la funcionalidad del programa analizado (Figura 88).

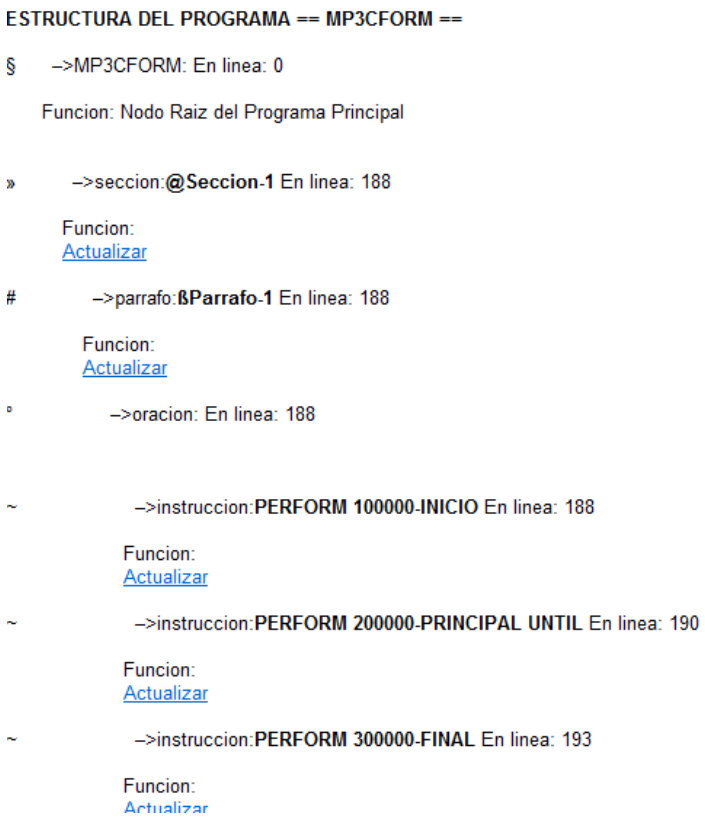
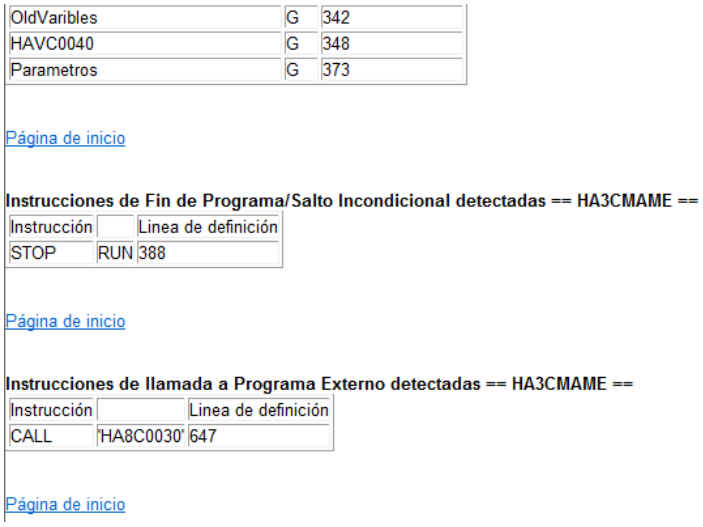


Figura 88 Flujo imperativo de ejecución del programa 3.

4.4.5 CASO DE PRUEBA 4. SECCIONES ADICIONALES DEL REPORTE.

- Prueba ejecutada: Programa 1.

Secciones de información generadas se ilustran en la figura 89.



OldVaribles	G	342
HAVC0040	G	348
Parametros	G	373

Página de inicio

Instrucciones de Fin de Programa/Salto Incondicional detectadas == HA3CMAME ==

Instrucción		Linea de definición
STOP	RUN	388

Página de inicio

Instrucciones de llamada a Programa Externo detectadas == HA3CMAME ==

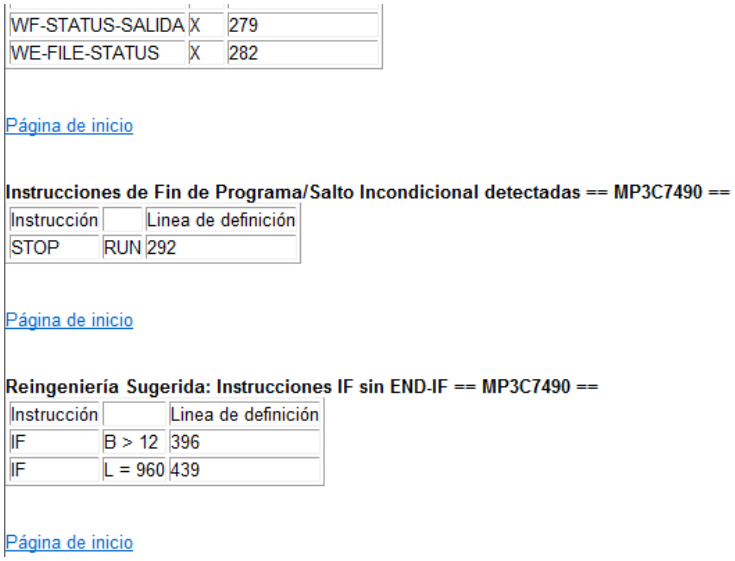
Instrucción		Linea de definición
CALL	'HA8C0030'	647

Página de inicio

Figura 89 Secciones adicionales reporte de análisis del programa 1.

- Prueba ejecutada: Programa 2.

Secciones de información generadas se ilustran en la figura 90.



WF-STATUS-SALIDA	X	279
WE-FILE-STATUS	X	282

Página de inicio

Instrucciones de Fin de Programa/Salto Incondicional detectadas == MP3C7490 ==

Instrucción		Linea de definición
STOP	RUN	292

Página de inicio

Reingeniería Sugerida: Instrucciones IF sin END-IF == MP3C7490 ==

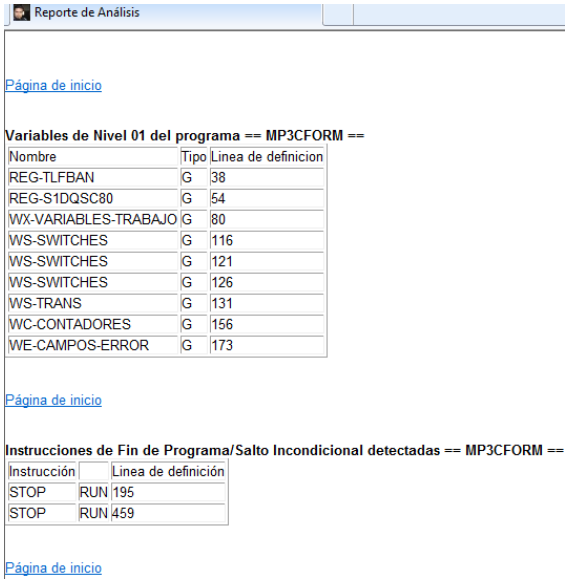
Instrucción		Linea de definición
IF	B > 12	396
IF	L = 960	439

Página de inicio

Figura 90 Secciones adicionales reporte de análisis del programa 2.

- Prueba ejecutada: Programa 3.

Secciones de información generadas se ilustran en la figura 91.



Reporte de Análisis

Página de inicio

Variables de Nivel 01 del programa == MP3CFORM ==

Nombre	Tipo	Linea de definicion
REG-TLFBAN	G	38
REG-S1DQSC80	G	54
WX-VARIABLES-TRABAJO	G	80
WS-SWITCHES	G	116
WS-SWITCHES	G	121
WS-SWITCHES	G	126
WS-TRANS	G	131
WC-CONTADORES	G	156
WE-CAMPOS-ERROR	G	173

Página de inicio

Instrucciones de Fin de Programa/Salto Incondicional detectadas == MP3CFORM ==

Instrucción		Linea de definición
STOP	RUN	195
STOP	RUN	459

Página de inicio

Figura 91 Secciones adicionales reporte de análisis del programa 3.

4.4.6 CASO DE PRUEBA 5. INTRODUCCIÓN DE CONOCIMIENTO POR PARTE DEL USUARIO.

- Prueba ejecutada: Programa 1.

Funcionalidad ejecutada sin errores, ver la figura 92.

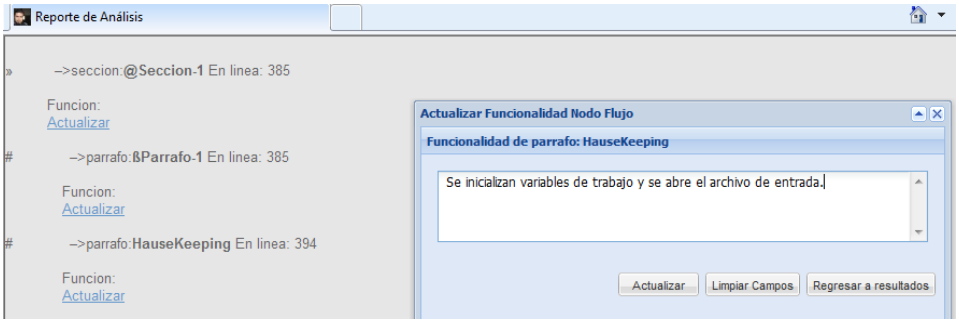


Figura 92 Pantalla de actualización en funcionalidad de nodo de flujo imperativo programa 1.

- Prueba ejecutada: Programa 2.

Funcionalidad ejecutada sin errores, ver la figura 93.

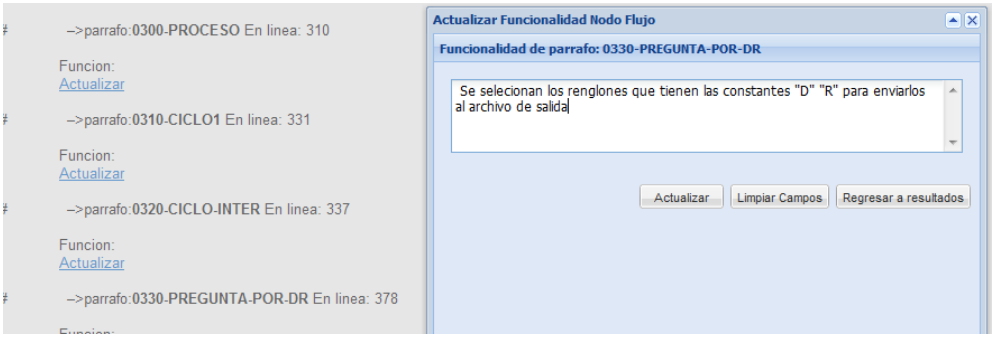


Figura 93 Pantalla de actualización en funcionalidad de nodo de flujo imperativo programa 2.

- Prueba ejecutada: Programa 3.

Funcionalidad ejecutada sin errores, ver la figura 94.

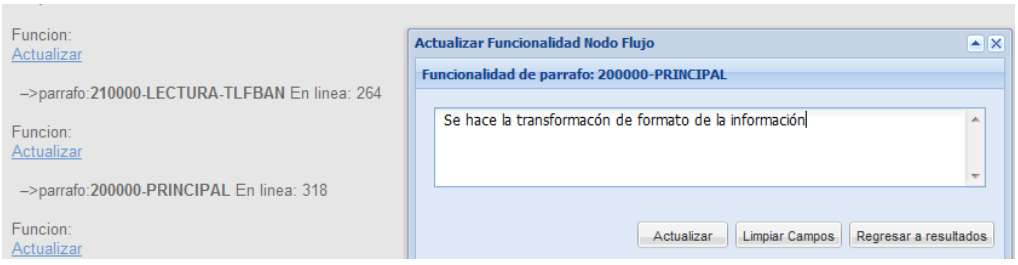


Figura 94 Pantalla de actualización en funcionalidad de nodo de flujo imperativo programa 3.

CONCLUSIÓNES.

La automatización como coloquialmente la entendemos es el propósito final de varios proyectos tecnológicos. El objetivo que se busca es delegar tareas que los seres humanos realizamos, a un conjunto de entes tecnológicos que en todo momento buscan imitar paradójicamente al "ente biológico automático más complejo" conocido hasta ahora: nosotros mismos. Se busca que la tarea delegada se ejecute por el ente tecnológico y que éste funcione en todo o en parte por sí solo. ¿Por qué se busca esto? Si bien es cierto que el ser humano puede hacer muchas de las tareas que desea automatizar, él mismo bajo ciertas condiciones está sujeto a su naturaleza, tenemos defectos y limitaciones, hay características en las tareas que en determinados momentos ponen en evidencia los defectos humanos. Lo que se busca es pues que el autómata no tenga esos defectos y ejecute las tareas de manera "más eficiente". Otras tareas son automatizadas porqué nosotros simplemente no las podemos hacer, por ejemplo volar, mover miles de toneladas de carga en poco tiempo o ejecutar millones de operaciones aritméticas es fracciones de segundo. A lo largo de la historia humana sin embargo, los opositores a la automatización siempre han alegado que: entonces que vamos a hacer al final nosotros los defectuosos humanos cuando "todo" esté automatizado. Probablemente tal suceso nunca pasará y se tenga todavía un largo camino por recorrer en el tema de "automatizar el mundo".

La investigación aplicada que se ha desarrollado tiene relación con el párrafo anterior. La idea de automatizar el análisis funcional de aplicativos en la plataforma mainframe surgió al observar en la vida real como varios proyectos informáticos estuvieron al borde del fracaso o tuvieron severas fallas y afectaciones teniendo como causa los así llamados "defectos humanos", varias veces se observó durante la experiencia profesional que un solo error de análisis se tradujo en miles de horas hombre adicionales en diseño, codificación, pruebas, estrés (con la consiguiente afectación en la calidad de vida de las personas), trámites burocráticos, dinero en penalizaciones y consecuencias asociadas (casos nada graciosos como por ejemplo que un cliente se encuentre con la sorpresa una mañana que le cobraron 10 veces la mensualidad de su crédito en su cuenta de dinero o que vaya a una sucursal bancaria porque algo le urge y simplemente le informen que "no hay sistema"). A veces el mejor de los analistas puede llegar a perder la brújula abrumado entre la cantidad enorme de tareas a completar bajo situaciones de presión, tiempo limitado y demanda de exactitud y calidad en el trabajo. Varios intentos de atacar los problemas asociados a la implementación errónea de proyectos en la plataforma de estudio y los así llamados sistemas heredados que son característicos en la misma, se han centrado en la administración del mismo, sin embargo se puede decir que para el año 2011 se sigue observando a personas salir del trabajo a altas horas de la noche, personas que mueven líneas de código sin tener idea que eso tal vez estará ocasionando que a un cliente le cobren 3 o 4 veces el costo de su seguro o su factura telefónica y otro tipo de "errores humanos" sucederse uno tras otro. Tal vez sea el momento de atacar los problemas desde otra perspectiva, si los desarrolladores de hoy no son ni ingenieros ni matemáticos como se comentaba en una de las revisiones de la historia de la ingeniería de software presentada, sino personas más "enfocadas a la funcionalidad final", si la demanda del mercado actual no da para rígidos esquemas de administración y se necesitan cambios implementados en ambientes de producción a veces sin una sola prueba unitaria ejecutada, ya no

digamos perfectamente documentados sobre qué fue lo que cambió, cosa que ni el mejor de los administradores de proyectos puede controlar desde el punto de vista del autor. Tal vez sea la hora de mover el control de calidad a la incepción del proyecto y no esperanzarse a que el ciclo de pruebas detecte los errores u omisiones de los arquitectos, diseñadores, analistas y programadores.

El procedimiento de análisis propuesto no fue probado de manera rigurosa, pero se anticipa que el entregable parcial logrado es una ayuda valiosa en materia de automatización para reducir tiempo en análisis en la plataforma destino y también contribuye a reducir la probabilidad de materialización de otro tipo de riesgos asociados al trabajo manual. De manera personal el autor piensa evaluar el entregable actual en proyectos reales y también finalizar la implementación del diseño propuesto con un mejor rendimiento. Como en toda la problemática discutida para la plataforma mainframe, la implementación del presente proyecto en otra plataforma distinta tuvo sus errores de planeación, de diseño proyectado, de análisis y de medidas de control. Lo que se planeaba como una tarea bastante obvia y auxiliar terminó consumiendo gran parte de los recursos asociados al proyecto. El entregable resultante, el analizador sintáctico, sirve de manera no inmediata porque no es la materialización del entregable final, pero si es una base importante sobre la cual se puede tomar una investigación subsecuente que implemente el diseño propuesto al 100%.

Al discutir el tema de la automatización se comentó que tal vez el camino hacia la "automatización total" del mundo era probablemente un camino sin fin. El presente trabajo aplicativo es una pequeña muestra. En el apartado de limitantes, temas inconclusos o temas abiertos la lista es extensa. Solo por dar un panorama general se comenta que quedó pendiente el algoritmo predictivo *LL(4)* requerido para evitar el uso de la técnica de fuerza bruta en muchas de las producciones de la gramática del lenguaje *COBOL*, quedó pendiente hacer análisis léxico y sintáctico de los lenguajes incrustados, pendiente quedó también el tema de pasar más elementos del análisis gramático a la base de datos de conocimiento así como el algoritmo que selecciona líneas de comentarios, el flujo de los datos y la extracción básica de las reglas de negocio. En materia del proyecto Web, el mismo no se implementó para ser usado por una gran cantidad de usuarios, no se configuró para usar un "pool" (manejo por grupos de usuarios) de conexiones en la base de datos. La *GUI* puede mejorarse con algoritmos del tipo análisis de datos o presentaciones tipo cubos *OLAP* para ver diferentes presentaciones del reporte en tiempo real, como por ejemplo, si se detecta una funcionalidad de ciclo al nivel alto, es posible que se desee "ver" esa sección del código real sin salir de la aplicación, es posible que se desease también ver exactamente en qué secciones, párrafos una variable sufre modificación en su valor. La lista de temas pendientes es muy grande, pero la idea está planteada y se ha concluido con un prototipo bastante ilustrativo como resultado.

Por último, se recapitulan los principales aportes de esta investigación aplicativa: La tarea de colección de investigaciones previas relacionadas; la conceptualización del diseño adaptado para la plataforma mainframe en específico para uno de los lenguajes más usados en la misma; la propuesta del uso de tecnologías de código abierto para automatizar el análisis funcional de los aplicativos heredados y el primer prototipo que perfila algunos ejemplos del uso de la información que genera un analizador léxico-sintáctico de *COBOL*, que se deja en una etapa de completitud muy avanzada. Pudiera parecer que el proyecto planteó cosas muy ambiciosas y no viables en el tema aplicativo, pero en realidad la implementación del prototipo final es lo que fue adaptado a las limitantes en tiempo del proyecto y aún con este recorte de alcance el resultado es bastante ilustrativo. Los demás entregables (marco teórico y el diseño) son también una

aportación importante en materia de investigación y propuesta de contribución a solucionar varios problemas observados en la industria.

REFERENCIAS Y BIBLIOGRAFÍA.

[1] Boehm B. A View of 20th and 21st Century Software Engineering. ICSE '06 Proceedings of the 28th international conference on Software engineering 2006;49(10):19-20.
[2] Reviews. Journal [IEEE Annals of the History of Computing]. 1980 Date [cited 2008; 2(2): Available from: http://doi.ieeecomputersociety.org/10.1109/MAHC.1980.10016.
[3] Everett RR, Zraket CA, Benington HD. SAGE-A Data Processing System for Air Defense. Journal [IEEE Annals of the History of Computing]. 1983 Date; 5(4): Available from: http://doi.ieeecomputersociety.org/10.1109/MAHC.1983.10096.
[4] Stepka SA, Gasser C. Burroughs Corporation Records. System Development Corporation Records. Finding Aid.: University of Minnesota Libraries. Charles Babbage Institute; 1997 [updated 1997; cited 2008]; System Development Corporation Collection of archive files.]. Available from: http://special.lib.umn.edu/findaid/xml/cbi00090-098.xml.
[5] Brooks FPJ. The Mythical Man Month Essays On Software Engineering. Third Printing 1st ed. North Carolina, USA: Addison Wesley; 1975.
[6] Dijkstra E. Cooperating Sequential Processes. Programming Languages; 1968; New York USA. New York: Academic Press; 1968.
[7] Royce DWW. Managing the development of large software systems. Journal. 1970 Date [cited 2008: Available from: http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf.
[8] Humphrey WS. Managing the Software Process. USA: Addison-Wesley; 1989.
[9] Ehn P. Work-Oriented Design of Computer Artifacts. L. Erlbaum Associates Inc.; 1990.
[10] Gamma E, Helm R, Johnson R, Vlissides JM. Design Patterns: Elements of Reusable Object Oriented Software. USA: Addison Wesley; 1994.
[11] Kruchten P. The Rational Unified Process: An Introduction 3rd ed. USA: Addison-Wesley; 2003.
[12] Oliveira MFS, Bri EW, Nascimento FA, Wagner R, FI. Model driven engineering for MPSOC design space exploration. Proceedings of the 20th annual conference on Integrated circuits and systems design; 2007; Copacabana, Rio de Janeiro. ACM; 2007.
[13] Maier MW. Architecting Principles for Systems-of-Systems. Virginia, USA: John Wiley & Sons, Inc; 1998.
[14] Robertson P. Integrating legacy systems with modern corporate applications. Commun ACM. 1997;40(5):39-46.
[15] Mark M, William ME. The Y2K problem: technological risk and professional responsibility. SIGCAS Comput Soc. 1999;29(4):24-9.
[16] Taft DK. Maximizing the Mainframe. 2007 [updated 2007; cited May 10th, 2008]; Available from: http://www.cioinsight.com/c/a/Past-News/Maximizing-the-Mainframe/.
[17] Zinkowski C, Breuer T, IBM MR. IBM launches New “System z10” Mainframe. 2008 [updated 2008; cited 2008 May 10th, 2008]; Available from: http://www-03.ibm.com/press/us/en/pressrelease/23592.wss.
[18] Palmisano SJ, IBM chairman PCEO. Quarterly earnings (presentation). 2007 [updated 2007; cited May 10th, 2008]; Available from: http://www.ibm.com/investor/2q07/2q07earnings.phtml.

[19] Graham BJ. Financial Matters: IBM Pricing Strategies Fuel Used Processor Market. 2005 [updated 2005 01/05/2005; cited 13/05/2008]; Available from: http://www.zjournal.com/index.cfm?section=article&aid=296.
[20] Graham BJ. Financial Matters: Building an IBM ESSO or ELA Software Contract to Save. 2005 [updated 2005 01/12/2005; cited 13/05/2008]; Available from: http://www.zjournal.com/index.cfm?section=article&aid=203.
[21] Graham BJ. Financial Matters: Mainframe Processor Pricing History. 2006 [updated 2006 01/03/2006; cited 13/05/2008]; Available from: http://www.zjournal.com/index.cfm?section=article&aid=346.
[22] Swanson M. Financial Matters: Trends in Software Costs–Your Mileage Will Vary. 2005 [updated 2005 01/09/2005; cited 13/05/2008]; Available from: http://www.zjournal.com/index.cfm?section=article&aid=322.
[23] Busqueda de empleo OCC. 2008 [updated 2008 15/05/2008; cited 14/05/2008]; Available from: http://www.occ.com.mx.
[24] Fontecchio M. NYSE undertakes IBM mainframe migration to Unix and Linux. 2007 [updated 2007; cited]; Available from: http://searchdatacenter.techtarget.com/news/article/0,289142,sid80_gci1254860,00.html.
[25] Binkley D. Source Code Analysis: A Road Map. 2007 Future of Software Engineering; 2007. IEEE Computer Society; 2007.
[26] Mitchell RL. Cobol: Not Dead Yet. 2006 [updated 2006; cited]; Available from: http://www.computerworld.com/action/article.do?command=viewArticleBasic&taxonomyName=Development&articleId=266156&taxonomyId=11&pageNumber=1.
[27] David AP, David RD. The case for the reduced instruction set computer. SIGARCH Comput Archit News. 1980;8(6):25-33.
[28] Jerome AO. Predicting potential COBOL performance on low level machine architectures. SIGPLAN Not. 1985;20(10):72-8.
[29] Cobol Portal. 2009 [updated 2009; cited 2009]; Available from: http://www.cobolportal.com/.
[30] MVS Forums. 2009 [updated 2009; cited 2009]; Available from: http://www.mvsforums.com/.
[31] What does the future hold for COBOL? ; 2002 [updated 2002; cited 2009]; Available from: http://articles.techrepublic.com.com/5100-10878_11-1050406.html.
[32] IBM. Summary of changes to the COBOL compilers. 2009 [updated 2009; cited 2009]; Available from: http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp?topic=/com.ibm.entcobol.doc_4.1/MG/igymsoacb41.htm.
[33] Dijkstra E. How do we tell truths that might hurt? ; 1975 [updated 1975; cited]; Available from: http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD498.html.
[34] IBM. Enterprise COBOL for z/OSLanguage Reference. San Jose, CA, USA: IBM corporation; 2007 [cited 2008. Available from: http://publibfp.boulder.ibm.com/epubs/pdf/igy3lr40.pdf.
[35] Hoare CAR. An axiomatic basis for computer programming. Commun ACM. 1969;12(10):576-80.
[36] Floyd RW. Assigning meaning to programs. In Proceedings of the Symposium on Applied Mathematics; 1967. American Math Society; 1967. p. 19-32.
[37] Ball T, Rajamani SK. The SLAM Toolkit. Proceedings of the 13th International Conference on Computer Aided Verification; 2001. Springer-Verlag; 2001. p. 260-4.
[38] Clarke EMJ, Grumberg O, Peled DA. Model checking. Cambridge: The MIT Press 1999.
[39] Holzmann GJ. The Model Checker SPIN. IEEE Trans Softw Eng. 1997;23(5):279-95.

[40] Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages; 1977; Los Angeles USA. ACM Press, New York, NY; 1977. p. 238-52.
[41] Jones ND, Flemming N. Abstract interpretation: a semantics-based tool for program analysis. Handbook of logic in computer science (vol 4): semantic modelling: Oxford University Press; 1995. p. 527-636.
[42] Kim M, Peter S, Vincent T, Weiqing H. Removing Node Overlapping in Graph Layout Using Constrained Optimization. Constraints. 2003;8(2):143-71.
[43] Graf S, Sa H. Construction of Abstract State Graphs with PVS. Proceedings of the 9th International Conference on Computer Aided Verification; 1997. Springer-Verlag; 1997.
[44] Sagiv M, Reps T, Wilhelm R. Parametric shape analysis via 3-valued logic. ACM Trans Program Lang Syst. 2002;24(3):217-98.
[45] Yahav E. Verifying safety properties of concurrent Java programs using 3-valued logic. Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages; 2001; London, United Kingdom. ACM; 2001.
[46] Godefroid P. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Leeuwen Jv, Hartmanis J, Goos G, editors.: Springer-Verlag New York, Inc.; 1996.
[47] Chiara B, Pierpaolo D, Flemming N, Hanne Riis N. Control Flow Analysis for the pi-calculus. Proceedings of the 9th International Conference on Concurrency Theory; 1998. Springer-Verlag; 1998.
[48] Marlowe TJ, Ryder BG. Properties of data flow frameworks: a unified model. Acta Inf. 1990;28(2):121-63.
[49] Shivers O. Control flow analysis in scheme. Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation; 1988; Atlanta, Georgia, United States. ACM; 1988.
[50] Jens P, Michael IS. Object-oriented type systems. John Wiley and Sons Ltd.; 1994.
[51] Kirsten LSG, Flemming N, Hanne Riis N. Systematic realisation of control flow analyses for CML. SIGPLAN Not. 1997;32(8):38-51.
[52] Aho AV, Sethi R, Ullman JD. Compiladores Principios, técnicas y herramientas. 1st ed.: Pearson Addison Wesley; 1998.
[53] Matthew SH. Flow Analysis of Computer Programs. Elsevier Science Inc.; 1977.
[54] Robert T. Testing flow graph reducibility. Proceedings of the fifth annual ACM symposium on Theory of computing; 1973; Austin, Texas, United States. ACM; 1973.
[55] Vugranam CS, Guang RG, Yong-Fong L. Identifying loops using DJ graphs. ACM Trans Program Lang Syst. 1996;18(6):649-58.
[56] Zahira A. A Control-Flow Normalization Algorithm and its Complexity. IEEE Trans Softw Eng. 1992;18(3):237-51.
[57] Mark GJvdB, Alex S, Chris V. Generation of components for software renovation factories from context-free grammars. Sci Comput Program. 2000;36(2-3):209-66.
[58] Mark van den B, Eelco V. Generation of formatters for context-free languages. ACM Trans Softw Eng Methodol. 1996;5(1):1-41.
[59] Proefschrift A, Rekers J. Parser Generation for Interactive Environments [PhD Thesis]: University of Amsterdam; 1992.
[60] Mark GJvdB, Sellink A, Verhoef C. Control Flow Normalization for COBOL/CICS Legacy System. Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98); 1998. IEEE Computer Society; 1998.
[61] Dock VT. Structured COBOL: American National Standard. West Publishing Co.; 1978.

[62] Proposed revision of ISO 1989: 1985, Programming Language Cobol. (1989).
[63] IBM C. IBM Cobol for MVS & VM Language Reference. IBM Corporation, San Jose, CA, 1995; 1995 [cited.
[64] John F, Ramalingam G. Identifying procedural structure in Cobol programs. SIGSOFT Softw Eng Notes. 1999;24(5):1-10.
[65] Flemming N, Hanne Riis N. Infinitary control flow analysis: a collecting semantics for closure analysis. Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages; 1997; Paris, France. ACM; 1997.
[66] Jagannathan S, Weeks S. A Unified Treatment of Flow Analysis in Higher-Order Languages. Rec 22nd Ann ACM Symp Princ of Prog Langs; 1995. ACM; 1995. p. 393--407.
[67] Olin S. The semantics of Scheme control-flow analysis. Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation; 1991; New Haven, Connecticut, United States. ACM; 1991.
[68] Sharir M, Pnueli A. Two approaches to interprocedural data flow analysis. In: Jones SSMaND, editor. In Program Flow Analysis: Theory and Applications. Englewood Cliffs, NJ: Prentice-Hall; 1981. p. 189-233.
[69] Neil DJ, Steven SM. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages; 1982; Albuquerque, Mexico. ACM; 1982. p. 121–63.
[70] Nevin H. Set-based analysis of ML programs. Proceedings of the 1994 ACM conference on LISP and functional programming; 1994; Orlando, Florida, United States. ACM; 1994.
[71] Heintze N, Jaffar J. An engine for logic program analysis. Logic in Computer Science, 1992 LICS '92, Proceedings of the Seventh Annual IEEE Symposium on; 1992; Santa Cruz, CA, USA. IEEE; 1992. p. 318-28.
[72] Palsberg J, editor. Global program analysis in constraint form. CAAP '94 1994. Springer.
[73] Palsberg J. Closure analysis in constraint form. ACM Trans Program Lang Syst. 1995;17(1):47-62.
[74] Suresh J, Stephen W. Analyzing stores and references in a parallel symbolic language. SIGPLAN Lisp Pointers. 1994;VII(3):294-305.
[75] Jens K, Bernhard S. The Interprocedural Coincidence Theorem. Proceedings of the 4th International Conference on Compiler Construction; 1992. Springer-Verlag; 1992.
[76] Hemant DP, Barbara GR. Data-Flow-Based Virtual Function Resolution. Proceedings of the Third International Symposium on Static Analysis; 1996. Springer-Verlag; 1996.
[77] Cousot P, Cousot R. Static determination of dynamic properties of recursive procedures. In: Neuhold EJ, editor. IFIP Conference on Formal Description of Programming Concepts; 1977; St-Andrews, N.B., Canada. North-Holland Publishing Company 1977. p. 237-77.
[78] William L, Barbara GR. A safe approximate algorithm for interprocedural pointer aliasing. SIGPLAN Not. 2004;39(4):473-89.
[79] Thomas R, Susan H, Mooly S. Precise interprocedural dataflow analysis via graph reachability. Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages; 1995; San Francisco, California, United States. ACM; 1995.
[80] Hai H, Wei-Tek T, Sourav B, Xiaoping C, Yamin W, Jianhua S. Business rule extraction techniques for COBOL programs. Journal of Software Maintenance. 1998;10(1):3-35.
[81] Robert TG. Structured COBOL programming. Prentice-Hall, Inc.; 1985.
[82] Chen XP, Tsai WT, Joiner JK, Gandamaneni H, Sun J. Automatic variable classification for cobol programs. In Proceedings of IEEE COMPSAC, 1994; 1994; Taipei, Taiwan. IEEE Computer Society Press, Los Alamitos CA; 1994. p. pp. 432–7.
[83] Mark W. Programmers use slices when debugging. Commun ACM. 1982;25(7):446-52.

[84] Mark W. Program slicing. Proceedings of the 5th international conference on Software engineering; 1981; San Diego, California, United States. IEEE Press; 1981.
[85] Susan H, Thomas R, David B. Interprocedural slicing using dependence graphs. ACM Trans Program Lang Syst. 1990;12(1):26-60.
[86] Gopal R, editor. Dynamic program slicing based on dependence relations. Proceedings of the Conference on Software Maintenance; 1991; Sorrento, Italy. IEEE Computer Society Press, Los Alamitos CA.
[87] Chen X, Tsai W-T, Huang H, Poonawala M, Rayadurgam S, Wang Y. Omega an integrated environment for C++ program maintenance. Proceedings of the International Conference on Software Maintenance; 1996; Monterey CA. IEEE Computer Society Press, Los Alamitos CA; 1996. p. pp. 114–23.
[88] Huang T, Subramanian S. Generalized program slicing for software maintenance. Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering (SEKE 96); 1996; Lake Tahoe NV. Knowledge Systems Institute, Skokie IL; 1996. p. pp. 261–8.
[89] Joiner JK, Tsai WT. Re-engineering legacy Cobol programs. Commun ACM. 1998;41(5es):185-97.
[90] Breuer PT, Lano KC. Creating specifications from code: Reverse engineering techniques. J Softw Maint Res Pract 3, 3; 1991. 1991. p. 145-62.
[91] Kazimiras L, Norman W, Scott S, Tim H. TraceGraph: Immediate Visual Location of Software Features. Proceedings of the International Conference on Software Maintenance (ICSM'00); 2000. IEEE Computer Society; 2000.
[92] Jeanne F, Karl JO, Joe DW. The program dependence graph and its use in optimization. ACM Trans Program Lang Syst. 1987;9(3):319-49.
[93] Chi EH. Expressiveness of the data flow and data state models in visualization systems. Advanced Visual Interfaces (AVI 2002); 2002; Trento, Italy. 2002.
[94] Misue K, Eades P, Lai W, Sugiyama K. Layout adjustment and the mental map. Visual Languages and Computing; 1995. 1995. p. 183–210.
[95] Kamada T, Kawai S. An algorithm for drawing general undirected graphs. Inf Process Lett. 1989;31(1):7-15.
[96] Kruskal J, Seery J. Designing network diagrams. Proceedings of the First General Conference on Social Graphics; 1980. 1980. p. 22–50.
[97] Cohen JD. Drawing graphs to convey proximity: an incremental arrangement method. ACM Trans Comput-Hum Interact. 1997;4(3):197-229.
[98] Thomas MJF, Edward MR. Graph drawing by force-directed placement. Softw Pract Exper. 1991;21(11):1129-64.
[99] Kelly AL. Cluster busting in anchored graph drawing. Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research - Volume 1; 1992; Toronto, Ontario, Canada. IBM Press; 1992.
[100] Eades P. Drawing free trees. Bulletin of the Institute for Combinatorics and its Applications. 1992;5:10-36.
[101] Graham JW. NicheWorks - Interactive Visualization of Very Large Graphs. In: Science LNiC, editor. Proceedings of the 5th International Symposium on Graph Drawing; 1997. Springer-Verlag; 1997. p. 403-14.
[102] Andrew SG. Graphics Gems. An algorithm for automatically fitting digitized curves: Academic Press, Inc.; 1990. p. 612-26.
[103] Freivalds K, Dogrus U, Kikusts P. Disconnected Graph Layout and the Polyomino Packing Approach. In: Science LNiC, editor. Revised Papers from the 9th International Symposium on Graph Drawing; 2002. Springer-Verlag; 2002. p. 378-91.

[104] Lipton RJ, North SC, Sandberg JS. A method for drawing graphs. Proceedings of the first annual symposium on Computational geometry; 1985; Baltimore, Maryland, United States. ACM; 1985. p. 153-60.
[105] Shiloach Y. Arrangements of Planar Graphs on the Planar Lattice. Rehovot, Israel: PhD Thesis Weizmann Institute of Science; 1976.
[106] Valiant L. Considerations in VLSI Circuits. Transactions on Computers; 1981. IEEE; 1981. p. 135-40.
[107] Spivey JM. An introduction to Z and formal specifications. Softw Eng J. 1989;4(1):40-50.
[108] UML® Resource Page. 1997 [updated 1997; cited 2008]; Available from: http://www.uml.org/#UML2.0.
[109] Johnson SC. YACC: Yet Another Compiler-Compiler. 1979 [updated 1979; cited 2011]; Available from: http://dinosaur.compilertools.net/yacc/.
[110] Viswanadha S, Sankar S. Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator. 1996 [updated 1996; cited 2011]; Available from: http://java.net/projects/javacc.
[111] The BSD License. University of California, Berkeley; 1998 [updated 1998; cited 2008]; Available from: http://www.opensource.org/licenses/bsd-license.php.
[112] Lämmel R, Verhoef C. VS COBOL II grammar Version 1.0.4. 1999 [updated 1999; cited 2011]; Available from: http://www.cs.vu.nl/grammarware/vs-cobol-ii/.
[113] Java SE Downloads. Sun ORACLE Corp; 2009 [updated 2009; cited 2008]; Available from: http://www.oracle.com/technetwork/java/javase/downloads/index.html.
[114] Java. Sun ORACLE Corp; 2009 [updated 2009; cited 2011]; Available from: http://www.java.com/en/.
[115] Apache Tomcat. The Apache Software Foundation; 1999 [updated 1999; cited 2011]; Available from: http://tomcat.apache.org/.
[116] MySQL The world's most popular open source database. ORACLE Corp; 2010 [updated 2010; cited 2008]; Available from: http://dev.mysql.com/.
[117] NetBeans. ORACLE Corp; 2011 [updated 2011; cited 2007]; Available from: http://netbeans.org/.
[118] Ext JS Cross-Browser Rich Internet Application Framework. Sencha; 2011 [updated 2011; cited 2011]; Available from: http://www.sencha.com/products/extjs/.
[119] Struts 2. The Apache Software Foundation; 2011 [updated 2011; cited 2011]; Available from: http://struts.apache.org/2.2.1/index.html.
[120] Spring Source. SpringSource; 2010 [updated 2010; cited 2011]; Available from: http://www.springsource.org/.
[121] Hibernate JBoss Comunity. RedHat Inc; 2011 [updated 2011; cited 2011]; Available from: http://www.hibernate.org/.
[122] The Apache ANT project.: The Apache Software Foundation; 2011 [updated 2011; cited 2011]; Available from: http://ant.apache.org/.

ANEXO A. ENCUESTA SOBRE PRÁCTICAS MANUALES DE ANÁLISIS DE SOFTWARE EN COBOL.

El presente documento contiene una serie de preguntas encaminadas a evaluar la situación actual de cómo se efectúa la actividad de análisis de aplicativos en la plataforma mainframe, específicamente para componentes unitarios en lenguaje COBOL.

1. En la mayoría de las preguntas se solicita por favor colocar una X donde considere que la opción responde a la pregunta.
2. En otras preguntas se busca un número concreto (por ejemplo un porcentaje aproximado).
3. Cuando se solicita su respuesta de manera abierta, por favor exponer brevemente la misma (una o 2 oraciones cortas que considera describen la situación de manera general).
4. Cuando la pregunta depende de una respuesta afirmativa previa, si dicha respuesta fue negativa, la pregunta dependiente queda en blanco.
5. Si considera que no tiene respuesta a una pregunta dejarla en blanco.

Definición: para efectos de este documento se entiende como análisis de un componente (o sistema) la actividad de identificar cada una de sus partes y describir sus características para que con la ayuda de los mismos, se pueda entender la funcionalidad actual.

- ¿Cuál es su rango de edad?

___0-25 ___25-35 ___35-45 ___45 o más.

- ¿Cuánto tiempo lleva **desarrollando** o lo hizo en la plataforma mainframe?

___Años

- ¿Cuánto tiempo lleva **diseñando** o lo hizo en la plataforma mainframe?

___Años

- ¿Cuánto tiempo lleva efectuando **análisis** o lo hizo en la plataforma mainframe?

___Años

- Cuando ha analizado componentes en COBOL, según su experiencia ¿**cuántas líneas** de código considera que tiene un programa en promedio?

____ líneas.

- ¿Depende el tiempo que se tarda en analizar un componente del tamaño del programa?

___SI ___NO

- ¿Qué factores considera que **complican** el análisis de un componente en forma manual? Considerando que el componente **no está documentado**, o sospecha que la documentación no está actualizada.

__Número de líneas de código.
__Cantidad de párrafos.
__Modularidad, excesiva cantidad de llamados a módulos externos (rutinas).
__Cantidad de entidades externas de información (Tablas de bases de datos, archivos).
__Ciclos controlados por instrucciones GOTO.
__Nombres de variables poco descriptivos.
__Programa no estructurado.
__Programa no indentado.
__Excesivo uso de instrucciones condicionantes anidadas (IF por ejemplo).
__Cuando el programa fue generado por un generador automático de código (reportes EZTrieve por ejemplo). Con estilo de programación autómata tipo ensamblador.
__No seguir un patrón de diseño (nombres de variables o párrafos con un patrón/prefijos según su función)
__Párrafos o secciones que contienen demasiadas instrucciones no relacionadas a una funcionalidad común.
__Código no seccionado en seudo-funciones.
__Múltiples instrucciones de finalización del programa distribuidas a lo largo del código. (Finalización con poco control)
__Otros

_____
_____
_____

- ¿Qué **actividades ejecuta** cuando tiene que hacer el **análisis** de un programa que debe ser modificado para cambiar su funcionalidad, agregar funcionalidad o corregir un error?

___ Lectura de la especificación.
___ Revisión detallada del código fuente.
___ Ejecución repetitiva con datos controlados hasta darme una idea de cómo funciona.
___Identificación de procedimientos o funciones críticas.
___Pregunto a los expertos técnicos del sistema.
___Consulto con los usuarios del sistema (negocio) y hago deducciones técnicas.
___Elaboro diagramas en papel, de procesos que considero importantes en la estructura del programa.
___Otros

_____
_____
_____

- Cuando se realiza **análisis de componentes** COBOL en la empresa donde trabaja, el mismo tiene como **causal**:

___Corrección de errores / mantenimiento.
___Agregar nuevas funciones / crecimiento o evolución del sistema.
___Migrar a otra plataforma.

- Cuando consulta la especificación o **documentación** del sistema informático donde trabaja ¿Cuántas veces encuentra que la misma esta **desactualizada**? Es decir, que lo que indica el documento de especificación no es lo mismo que el código hace.

__Nunca

__Algunas veces
__Casi Siempre
__Siempre

- ¿Con qué frecuencia durante su trabajo en proyectos informáticos de la plataforma mainframe se ha encontrado con que un componente **no fue documentado** para su liberación a producción y por lo tanto no existe ni siquiera un punto inicial de referencia para entender qué cosa hace dicho componente?

__Nunca
__Algunas veces
__Casi Siempre
__Siempre

- Si debido a cualquier circunstancia resulta que tiene que inspeccionar código fuente de forma manual para hacer mantenimiento o bien modificar evolutivamente un componente, normalmente ¿**Cuánto tiempo** le lleva concluir el **análisis** con un nivel de detalle importante del mismo, o por lo menos para solucionar el problema que quiere resolver? (Si considera que no tiene respuesta deje en blanco el espacio).

Si utilizáramos como medida el **tamaño en líneas de código**
Menos de 1000 líneas. Me tardo ____ días de 8 horas laborables en analizarlo.
Entre 1000 y 4000 líneas. Me tardo ____ días de 8 horas laborables en analizarlo.
Entre 4000 y 8000 líneas. Me tardo ____ días de 8 horas laborables en analizarlo.
Más 8000 líneas. Me tardo ____ días de 8 horas laborables en analizarlo.

Si el parámetro subjetivo fuese la **complejidad** del mismo:
Un programa sencillo. Me tardo ____ días de 8 horas laborables en analizarlo.
Un programa medianamente complejo. Me tardo ____ días de 8 horas laborables en analizarlo.
Un programa complejo. Me tardo ____ días de 8 horas laborables en analizarlo.
Un programa muy complejo. Me tardo ____ días de 8 horas laborables en analizarlo.
Un programa sensible ó núcleo del sistema (que ejecuta funcionalidad muy crítica para el sistema donde trabaja y en donde un error sería de consecuencias muy graves). Me tardo ____ días de 8 horas laborables en analizarlo.

- ¿Qué tan seguido considera que el avance con **desviación negativa en tiempo** de los **proyectos** informáticos en su empresa tiene que ver con una forma inadecuada de analizar componentes unitarios de los sistemas actuales o incluso del sistema global?

__Nunca, son otros factores las causas.
__Algunas veces
__Casi siempre
__Siempre

- ¿Considera que la **dependencia** de **personas expertas** en un sistema a modificar es determinante para el éxito o fracaso de un proyecto?

__Nunca
__Algunas veces
__Casi siempre
__Siempre

- Según su experiencia en el ambiente mainframe, ¿qué tan **costoso** es, de acuerdo a su estimación personal, que un grupo de **personas que no conocen** un aplicativo se involucre en **proyectos** de mantenimiento o evolución de un sistema?

__No tiene costo
__Tiene bajo costo
__Tiene mediano costo
__Tiene costo elevado

- Según su experiencia en el ambiente mainframe, ¿qué tan seguido un **análisis inadecuado**, con omisiones o hecho de manera apresurada es la **causa directa** de **errores en producción** durante la liberación de proyectos en la empresa donde trabaja?

__Nunca ocurre
__Algunas veces pasa
__Pasa seguido
__Siempre pasa

- ¿Considera usted que la etapa de **análisis** de un proyecto informático en mainframe, debe tener las siguientes **características**? (Marque con una X las que considere que apliquen)

__Realizado por expertos.
__Planeado con holgura y valorado con ayuda de expertos.
__Debe ser realizado en el menor tiempo posible, pues todos los requerimientos son urgentes.
__Debe contar con un método o procedimiento estándar.
__Debe ser realizado de forma manual.
__Debe considerar márgenes de error funcional.
__Debe ser realizado de forma completamente automática.
__Se debe apoyar en el uso de herramientas automáticas de documentación
__En él se debe considerar más tiempo, que el invertido para el diseño o la construcción.
__Otras características que debe tener el análisis de aplicativos según su apreciación:

_____
_____
_____

- Cuando realiza **análisis manual** de componentes (cuando ha determinado que no puede obtener la funcionalidad por otros medios, como preguntarle a un experto o consultar un documento de especificación confiable), ¿**qué es lo que hace** para comprender como funciona?

__Reviso el reporte de la compilación.
__Investigo como lo puedo ejecutar en modo de prueba unitaria.
__Busco archivos de entrada y salida.
__Obtengo el formato de las entradas y salidas (Lay-out o COPY)
__Identifico tablas de base de datos utilizadas.
__ Ubico las instrucciones CRUD (insertar, leer, actualizar, borrar).
__Identifico la comunicación con sistemas externos (CICS, DB2, JCL)
__Elaboro un flujo general del programa. Principales flujos de ejecución (párrafos, secciones).
__Elaboro un flujo detallado de cada ruta de ejecución.
__Identifico ciclos repetitivos de instrucciones.
__Identifico mensajes de error.

__Busco operaciones aritméticas de riesgo o críticas (divisiones, manejo de índices de arreglos, cálculos)
__Identifico las estructuras de datos utilizadas (COPYs, grupos 01 de variables)
__Investigo sobre módulos externos utilizados y su funcionalidad
__Obtengo el tamaño del área de almacenamiento en memoria del programa (Working Storage).
__Describo el flujo de la información a través de entidades (tablas, archivos)
__Identifico donde se hace lectura a fuentes externas de datos.
__Identifico donde se hacen las actualizaciones a fuentes externas de datos.
__Identifico secciones párrafos o instrucciones de finalización en la ejecución del programa.
__Hago un mapeo de las variables que son leídas o actualizadas.
__Identifico el uso de variables no inicializadas.
__Otras cosas que identifico **dentro del código** del componente:

_____
_____
_____

- Cuando realiza **análisis manual** de componentes, ¿qué es lo que busca en el **ambiente externo** al componente?

__Busco en los repositorios de conocimiento (si los hay) cómo funciona el componente
__Identifico qué proceso genera la entrada del componente en análisis
__Identifico qué proceso recibe la salida del componente en análisis
__Identifico dónde se ubica al nivel global en el flujo de procesos el componente en análisis. (Diagrama de flujo del sistema).
__Investigo qué hacen exactamente las rutinas externas llamadas.
__Otras cosas que identifico en el **ambiente externo** al componente:

_____
_____
_____

- Mencione de manera breve (una oración) cuáles son sus mejores **3 estrategias de análisis manual** de componentes.

_____
_____
_____

- ¿Cree usted que una herramienta que ayude a **analizar** componentes debe ser **totalmente automatizada**?

__SI __NO

- Actualmente existen **productos** de análisis automático de componentes en COBOL en plataforma mainframe, por favor indique **si ha usado alguno** y en este caso, como se llama dicho producto.

_____

- Señale si el **producto que ha usado** tiene alguna de las siguientes **características negativas**.

___Reportes de salida poco entendibles
___Diagramas de flujo inmanejables.
___Nivel de detalle muy complejo.

___Muy lento.
___Muy caro.
___No lo uso porque en vez de ayudar complica más el análisis.
___Necesito entrenamiento excesivo para poderlo usar.

- Si por el contrario (a la pregunta anterior) el producto que ha usado tiene **ventajas**, señale por favor 1 o 2 que considera de ayuda sumamente importante a la hora de analizar aplicativos o componentes.

_____
_____

- Basado en su experiencia como analista de sistemas, si hubiese una **herramienta o procedimiento nuevo** para analizar componentes de sistemas heredados al nivel unitario (basado principalmente en el compilador) ¿Qué es lo que considera fundamental que **debería entregar** la parte automática para que la considerara **útil y práctica**?

__Un breve resumen de entidades de información de entrada / salida.
__Lista detallada de todas las variables y párrafo donde se usan.
__Lista parcial de variables más utilizadas.
__Diagrama de Flujo de ejecución de párrafos hasta un cierto nivel de profundidad.
__Lista de rutinas externas utilizadas.
__Lista de COPYS y los párrafos donde se usa alguna variable de las mismas.
__Bosquejo de instrucciones en ciclo, no a detalle.

- De la pregunta anterior, si usted considera que hay **una salida de tal herramienta** "nueva" que definitivamente le ayudaría a disminuir el riesgo de error, costo o tiempo de análisis de un componente COBOL (del cual no está seguro si el documento de especificación disponible esta actualizado o definitivamente no está documentado) ¿cuál sería esta característica?, es decir ¿**qué le debería entregar la mencionada herramienta** para facilitarle el análisis manual?

Considerar que la herramienta únicamente podría explotar información del compilador en forma unitaria.

_____
_____
_____
_____

- Para la actividad de **análisis de componentes** si pudiera expresar un **porcentaje aproximado del tiempo que ahorraría** si pudiera obtener un documento con la seudo-especificación del componente a analizar, **generado automáticamente** y de manera **confiable**, ¿cuál sería ese valor? Cero si no considera que sería un ahorro o negativo si considera que incrementa el tiempo.

___ %

- Para la actividad de **análisis de componentes**, si pudiera expresar un **porcentaje aproximado de disminución del riesgo** resultante de comparar un análisis completamente manual contra un análisis auxiliado por la obtención de un documento con la seudo-especificación del componente (**generado automáticamente** y de manera **confiable**), ¿cuál sería el valor? Cero si considera que el riesgo no disminuye o negativo si considera que aumenta.

___ %

Muchas gracias por el tiempo que dedicó para responder de este cuestionario.

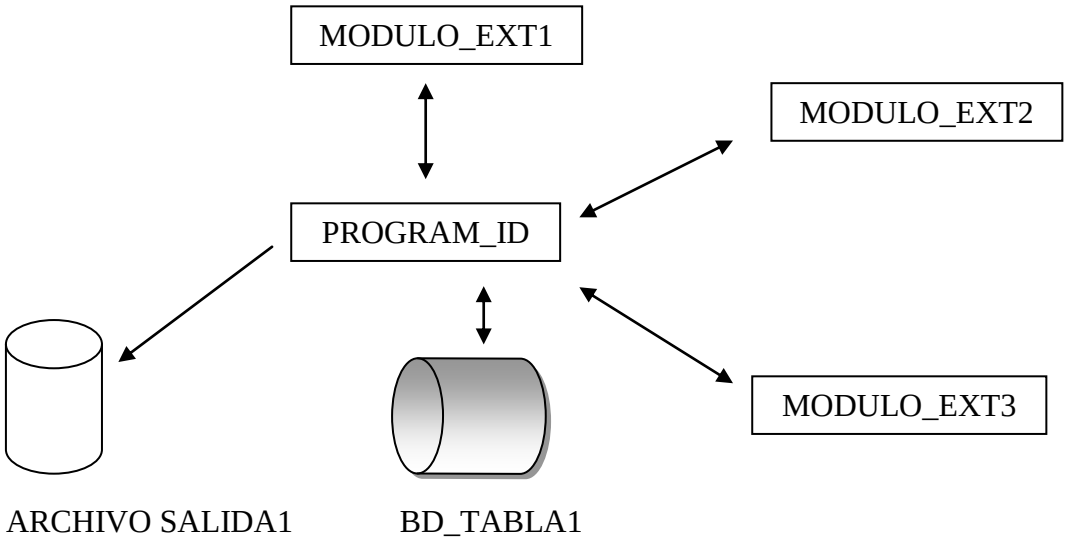
ANEXO B. EJEMPLO DE PLANTILLA DE ESPECIFICACIÓN DE DISEÑO PARA COMPONENTE EN COBOL.

[Encabezado de página]

ORGANIZACIÓN				TIPO_DOC_Diseño	
APLICACIÓN-XXXXX			Plantilla_Especificacion_Componente.doc		
EJECUTABLES-COBOL-NombreComponente					
PREPARADO :	AVE	FECHA :	17/06/2009 8:30	VERSIÓN: ??	PÁG N
APROBADO :	???	FECHA :	MMDDAAAA	ESTADO:Revisión	

PROGRAMA: Nombre_Programa
Especificación de diseño.

DIAGRAMA



REFERENCIAS ASOCIADAS

FUNCIONES (referirse a la Revisión Funcional del Módulo)

TIPO Y NOMBRE	DESCRIPCIÓN

TIPO Y NOMBRE	DESCRIPCIÓN
Función	Descripción

TABLAS

TIPO Y NOMBRE	DESCRIPCIÓN
TABLA1	Tabla de Ejercicios Contables
TABLA2	Descripción tabla

COPYS

TIPO Y NOMBRE	DESCRIPCIÓN
COPY1	Copy de la tabla de Ejercicio Contable
COPY2	Copy de la rutina para el cálculo de Ratios
COPYN	Descripción de COPYN
SQLCA	Copy de los retornos DB2
TC1400	Copy de la rutina TC9C1400
COPYA1	Copy del formato del archivo de salida1
COPYT1	Copy de acceso a la tabla1
EIDBC	Copy de retornos para errores CICS

ARCHIVOS DE ENTRADA

TIPO Y NOMBRE	DESCRIPCIÓN
Archivo1	Descripción

ARCHIVOS DE SALIDA

TIPO Y NOMBRE	DESCRIPCIÓN
Archivo2	Descripción

RUTINAS Y MÓDULOS

TIPO Y NOMBRE	DESCRIPCIÓN
Rutina1	Rutina para el cálculo de Resumen
Rutina2	Rutina para el cálculo de Resumen
RutinaN	Rutina de Control de Entidad (Copy XXXXXXXX)

PROCESO EN EL QUE SE EJECUTA

TIPO Y NOMBRE	DESCRIPCIÓN
NombreProceso	Descripción

CÓDIGOS DE RETORNO

CÓDIGO	DESCRIPCIÓN
Código1	Se utilizarán los códigos de retorno descritos en el documento D610W.AEECRET0

DESCRIPCIÓN – Reglas de negocio – Diagrama de flujo.

El programa realiza la generación de alertas económicas en función de la comparación de los balances en periodos similares.

INICIO
Inicializar variables y COPIES de trabajo, así como, los valores de los switchs utilizados.

Abrir el archivo de salida

Declaración de 2 variables Working para aceptar campos por SYSIN:
WS-FECHA-DIA(formato AAAA-MM-DD)
WS-ENTIDAD
…

PROCESO
Se escribe en el archivo de salida los campos:

COD-ENTIGEN = WS-ENTIDAD
NUM-CLIENTE = NUM-CLIENTE(con el que se informó a la rutina)
COD-ORIGALER = '**E**'
COD-ALERTA = '**15'** ("Balance Vencido")
COD-SUBALERT = '**01**'
FEC-ALERTA = WS-FECHA-DIA
IND-ALTBAJA = '**B**'

Se abre un cursor a la tabla TABLA1 recuperando todos los registros del día, para ello accederemos con la fecha del día y la entidad que se han recuperado de SYSIN (WS-FECHA-DIA(para FEC-ALTA) y WS-ENTIDAD)ordenado por los campos IND-TIP-PER, NUM-CLIENTE, TIP-EJERCICIO, ANO-BALANCE, MES-BALANCE y TIP-BALANCE.

Evaluamos el SQLCODE:
Si es igual a cero
    Continuar
En cualquier otro caso:
    Mover a la copy de retorno **(98) "Error DB2**".Informar los campos de DB2-LOG de la COPY de retorno.
…

FIN
Cerrar el archivo de salida.
Se finalizará la ejecución del programa devolviendo el control al programa que lo llamó.

…

VALIDACIONES

OTRAS CONSIDERACIONES
La devolución de un error tendrá como consecuencia la finalización inmediata del programa.

Si es un ERROR-DB2:
Se activa el campo ERROR-DB2 con ‘S’.
PGRNAME ---→ El nombre del programa
SQLCA ---→ Datos DB2
OBJETO ---→ Tabla
SENTENCIA ---→ Instrucción DB2

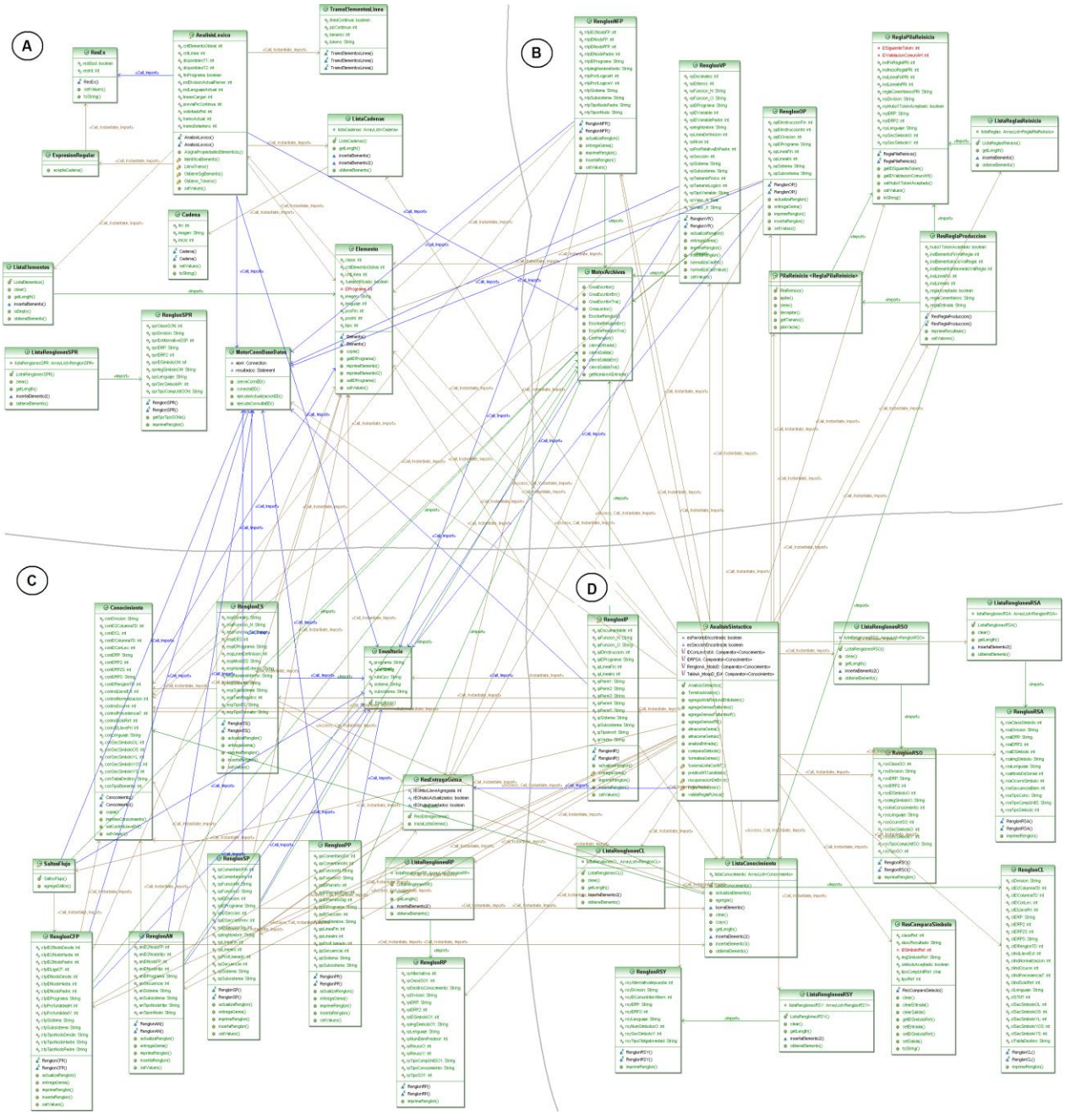
Si el error es otro y no es DB2:
Se activa el campo ERROR-DB2 con ‘N’.
PGRNAME ---→ El nombre del programa
RUTINA ---→ Nombre de la rutina si el error se ha producido en alguna.
CODIGO-RETORNO ---→ Código de retorno de la rutina
REFERNCIA ---→ Datos referenciales.

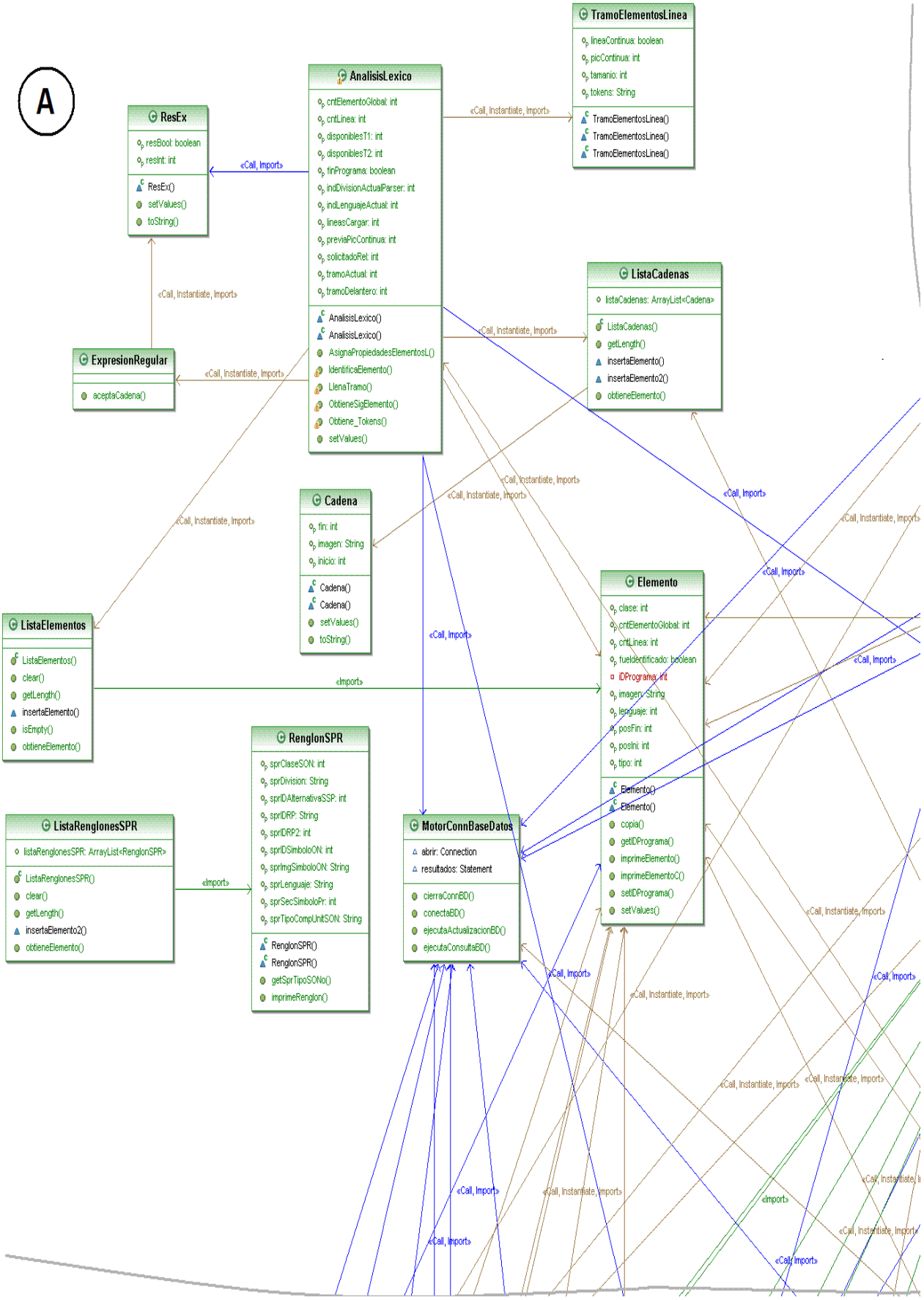
ERRORES Y AVISOS.

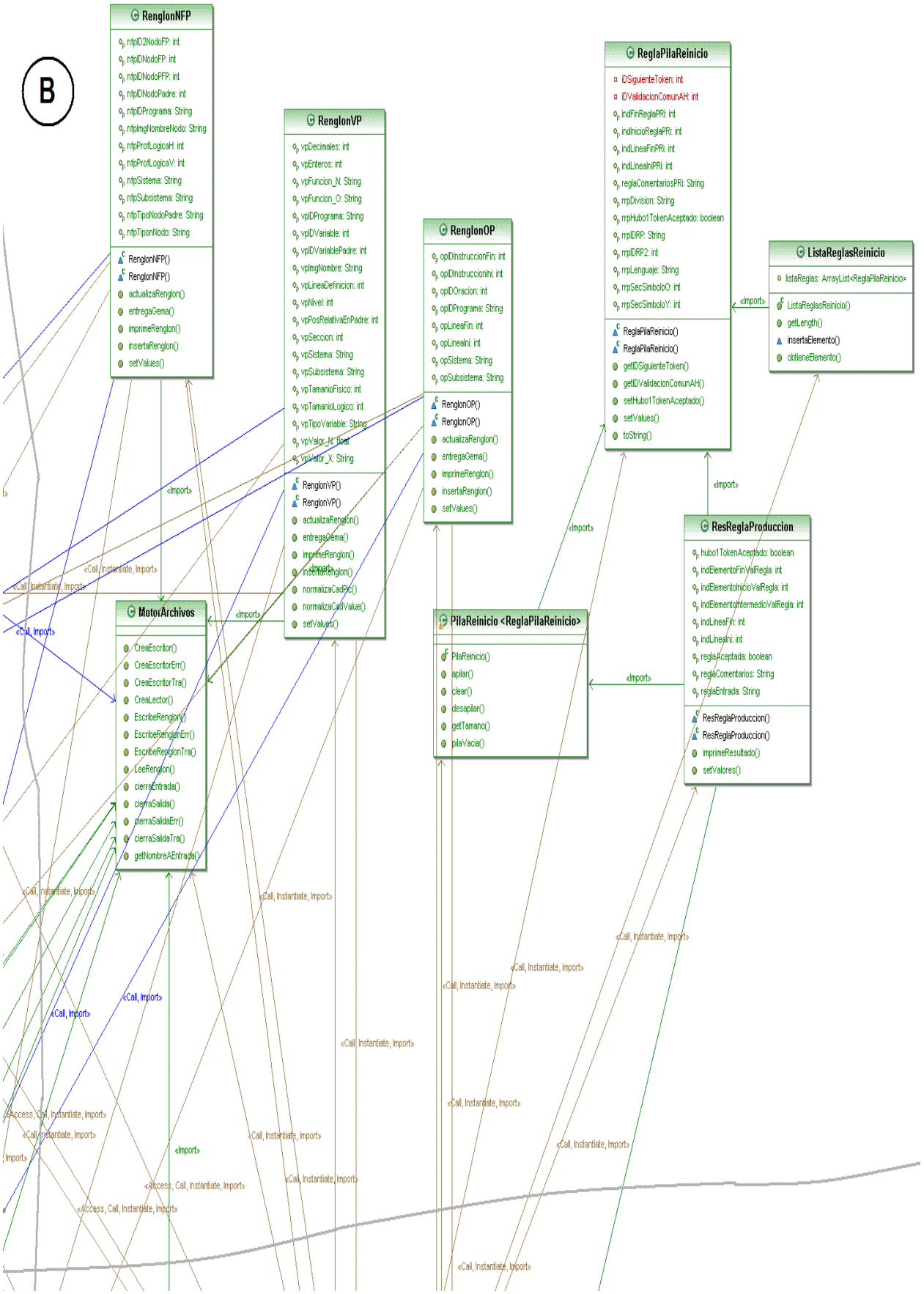
CODIGO	DESCRIPCIÓN
Error1	@@@@@@@@@@@@@@@@@@@@ INEXISTENTE
Error2	@@@@@@@@@@@@@@@@@@@@ **ERRONEO**
ErrorN	ERROR DB2

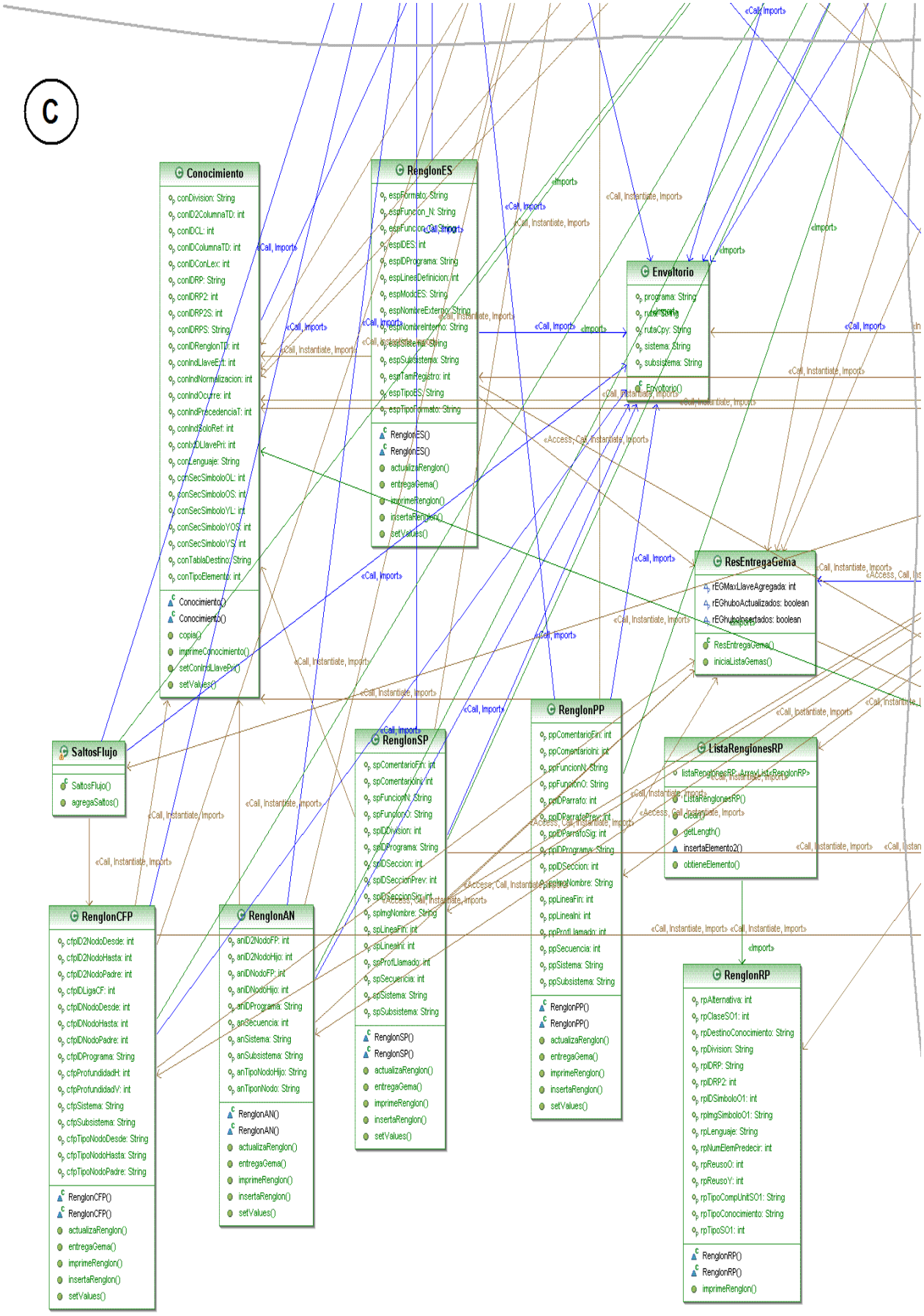
ANEXO C. DETALLE DEL DIAGRAMA DE CLASES DEL ANALIZADOR SINTÁCTICO Y NORMALIZADOR DE CONOCIMIENTO IMPLEMENTADO.

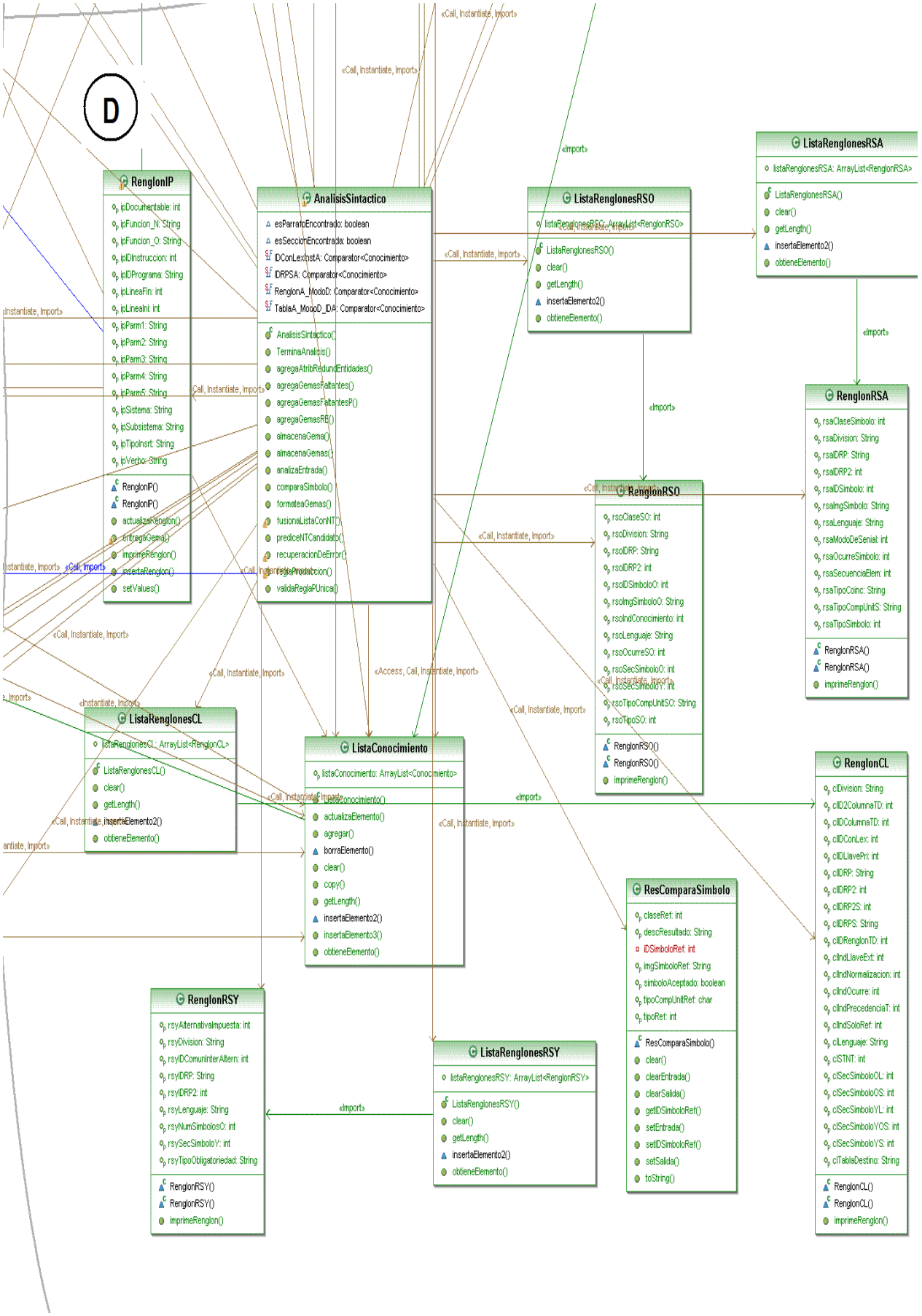
Se presenta a continuación el diagrama de clases del software implementado primero en forma de mapa seccionado en 4 partes, posteriormente se presenta el detalle ampliado de cada una de las 4 secciones: *A*, *B*, *C* y *D*.




A
ResEx
resBool: boolean
resInt: int
ResEx()
setValues()
toString()
AnalisisLexico
cntElementoGlobal: int
cntLinea: int
disponiblesT1: int
disponiblesT2: int
finPrograma: boolean
indDivisionActualParser: int
indLenguajeActual: int
lineasCargar: int
previaPicContinua: int
solicitadoRel: int
tramoActual: int
tramoDelantero: int
AnalisisLexico()
AnalisisLexico()
AsignaPropiedadesElementosL()
IdentificaElemento()
LlenaTramo()
ObtieneSigElemento()
Obtiene_Tokens()
setValues()
TramoElementosLinea
lineaContinua: boolean
picContinua: int
tamanio: int
tokens: String
TramoElementosLinea()
TramoElementosLinea()
TramoElementosLinea()
ListaCadenas
listaCadenas: ArrayList<Cadena>
ListaCadenas()
getLength()
insertaElemento()
insertaElemento2()
obtieneElemento()
ExpresionRegular
aceptaCadena()
Cadena
fin: int
imagen: String
inicio: int
Cadena()
Cadena()
setValues()
toString()
ListaElementos
ListaElementos()
clear()
getLength()
insertaElemento()
isEmpty()
obtieneElemento()
Elemento
clase: int
cntElementoGlobal: int
cntLinea: int
fueIdentificado: boolean
iDPrograma: int
imagen: String
lenguaje: int
posFin: int
posIni: int
tipo: int
Elemento()
Elemento()
copia()
getIDPrograma()
imprimeElemento()
imprimeElementoC()
setIDPrograma()
setValues()
RenglonSPR
sprClaseSON: int
sprDivision: String
sprIDAlternativaSSP: int
sprIDRP: String
sprIDRP2: int
sprIDSimboloON: int
sprImgSimboloON: String
sprLenguaje: String
sprSecSimboloPr: int
sprTipoCompUnitSON: String
RenglonSPR()
RenglonSPR()
getSprTipoSONo()
imprimeRenglon()
ListaRenglonesSPR
listaRenglonesSPR: ArrayList<RenglonSPR>
ListaRenglonesSPR()
clear()
getLength()
insertaElemento2()
obtieneElemento()
MotorConnBaseDatos
abrir: Connection
resultados: Statement
cierraConnBD()
conectaBD()
ejecutaActualizacionBD()
ejecutaConsultaBD()
«Call, Instantiate, Import»
«Call, Import»
«import»


B
RenglonNFP
nfpID2NodoFP: int
nfpIDNodoFP: int
nfpIDNodoPFP: int
nfpIDNodoPadre: int
nfpIDPrograma: String
nfpImgNombreNodo: String
nfpProfLogicaH: int
nfpProfLogicaV: int
nfpSistema: String
nfpSubsistema: String
nfpTipoNodoPadre: String
nfpTiponNodo: String
RenglonNFP()
RenglonNFP()
actualizaRenglon()
entregaGema()
imprimeRenglon()
insertaRenglon()
setValues()
RenglonVP
vpDecimales: int
vpEnteros: int
vpFuncion_N: String
vpFuncion_O: String
vpIDPrograma: String
vpIDVariable: int
vpIDVariablePadre: int
vpImgNombre: String
vpLineaDefinicion: int
vpNivel: int
vpPosRelativaEnPadre: int
vpSeccion: int
vpSistema: String
vpSubsistema: String
vpTamanioFisico: int
vpTamanioLogico: int
vpTipoVariable: String
vpValor_N: float
vpValor_X: String
RenglonVP()
RenglonVP()
actualizaRenglon()
entregaGema()
imprimeRenglon()
insertaRenglon()
normalizaCadPic()
normalizaCadValue()
setValues()
RenglonOP
opIDInstruccionFin: int
opIDInstruccionIni: int
opIDOracion: int
opIDPrograma: String
opLineaFin: int
opLineaIni: int
opSistema: String
opSubsistema: String
RenglonOP()
RenglonOP()
actualizaRenglon()
entregaGema()
imprimeRenglon()
insertaRenglon()
setValues()
ReglaPilaReinicio
iDSiguienteToken: int
iDValidacionComunAH: int
indFinReglaPRi: int
indInicioReglaPRi: int
indLineaFinPRi: int
indLineaIniPRi: int
reglaComentariosPRi: String
rrpDivision: String
rrpHubo1TokenAceptado: boolean
rrpIDRP: String
rrpIDRP2: int
rrpLenguaje: String
rrpSecSimboloO: int
rrpSecSimboloY: int
ReglaPilaReinicio()
ReglaPilaReinicio()
getIDSiguienteToken()
getIDValidacionComunAH()
setHubo1TokenAceptado()
setValues()
toString()
ListaReglasReinicio
listaReglas: ArrayList<ReglaPilaReinicio>
ListaReglasReinicio()
getLength()
insertaElemento()
obtieneElemento()
ResReglaProduccion
hubo1TokenAceptado: boolean
indElementoFinValRegla: int
indElementoInicioValRegla: int
indElementoIntermedioValRegla: int
indLineaFin: int
indLineaIni: int
reglaAceptada: boolean
reglaComentarios: String
reglaEntrada: String
ResReglaProduccion()
ResReglaProduccion()
imprimeResultado()
setValores()
MotorArchivos
CreaEscritor()
CreaEscritorErr()
CreaEscritorTra()
CreaLector()
EscribeRenglon()
EscribeRenglonErr()
EscribeRenglonTra()
LeeRenglon()
cierraEntrada()
cierraSalida()
cierraSalidaErr()
cierraSalidaTra()
getNombreAEntrada()
PilaReinicio <ReglaPilaReinicio>
PilaReinicio()
apilar()
clear()
desapilar()
getTamano()
pilaVacia()
«Import»
«Call, Instantiate, Import»
«Call, Import»
«Access, Call, Instantiate, Import»


C
Conocimiento
conDivision: String
conID2ColumnaTD: int
conIDCL: int
conIDColumnaTD: int
conIDConLex: int
conIDRP: String
conIDRP2: int
conIDRP2S: int
conIDRPS: String
conIDRenglonTD: int
conIndLlaveEnt: int
conIndNormalizacion: int
conIndOcurre: int
conIndPrecedenciaT: int
conIndSoloRef: int
conIxDLlavePri: int
conLenguaje: String
conSecSimboloOL: int
conSecSimboloOS: int
conSecSimboloYL: int
conSecSimboloYOS: int
conSecSimboloYS: int
conTablaDestino: String
conTipoElemento: int
Conocimiento()
Conocimiento()
copia()
imprimeConocimiento()
setConIndLlavePri()
setValues()
RenglonES
espFormato: String
espFuncion_N: String
espFuncion_O: String
espIDES: int
espIDPrograma: String
espLineaDefinicion: int
espModES: String
espNombreExterno: String
espNombreInterno: String
espSistema: String
espSubsistema: String
espTamRegistro: int
espTipoES: String
espTipoFormato: String
RenglonES()
RenglonES()
actualizaRenglon()
entregaGema()
imprimeRenglon()
insertaRenglon()
setValues()
Envoltorio
programa: String
rutaCpy: String
sistema: String
subsistema: String
Envoltorio()
ResEntregaGema
rEGMaxLlaveAgregada: int
rEGhuboActualizados: boolean
rEGhuboInsertados: boolean
ResEntregaGema()
iniciaListaGemas()
SaltosFlujo
SaltosFlujo()
agregaSaltos()
RenglonSP
spComentarioFin: int
spComentarioIni: int
spFuncionN: String
spFuncionO: String
spIDDivision: int
spIDPrograma: String
spIDSeccion: int
spIDSeccionPrev: int
spIDSeccionSig: int
spImgNombre: String
spLineaFin: int
spLineaIni: int
spProfLlamado: int
spSecuencia: int
spSistema: String
spSubsistema: String
RenglonSP()
RenglonSP()
actualizaRenglon()
entregaGema()
imprimeRenglon()
insertaRenglon()
setValues()
RenglonPP
ppComentarioFin: int
ppComentarioIni: int
ppFuncionN: String
ppFuncionO: String
ppIDParrafo: int
ppIDParrafoPrev: int
ppIDParrafoSig: int
ppIDPrograma: String
ppIDSeccion: int
ppImgNombre: String
ppLineaFin: int
ppLineaIni: int
ppProfLlamado: int
ppSecuencia: int
ppSistema: String
ppSubsistema: String
RenglonPP()
RenglonPP()
actualizaRenglon()
entregaGema()
imprimeRenglon()
insertaRenglon()
setValues()
ListaRenglonesRP
listaRenglonesRP: ArrayList<RenglonRP>
ListaRenglonesRP()
clear()
getLength()
insertaElemento2()
obtieneElemento()
RenglonCFP
cfpID2NodoDesde: int
cfpID2NodoHasta: int
cfpID2NodoPadre: int
cfpIDLigaCF: int
cfpIDNodoDesde: int
cfpIDNodoHasta: int
cfpIDNodoPadre: int
cfpIDPrograma: String
cfpProfundidadH: int
cfpProfundidadV: int
cfpSistema: String
cfpSubsistema: String
cfpTipoNodoDesde: String
cfpTipoNodoHasta: String
cfpTipoNodoPadre: String
RenglonCFP()
RenglonCFP()
actualizaRenglon()
entregaGema()
imprimeRenglon()
insertaRenglon()
setValues()
RenglonAN
anID2NodoFP: int
anID2NodoHijo: int
anIDNodoFP: int
anIDNodoHijo: int
anIDPrograma: String
anSecuencia: int
anSistema: String
anSubsistema: String
anTipoNodoHijo: String
anTiponNodo: String
RenglonAN()
RenglonAN()
actualizaRenglon()
entregaGema()
imprimeRenglon()
insertaRenglon()
setValues()
RenglonRP
rpAlternativa: int
rpClaseSO1: int
rpDestinoConocimiento: String
rpDivision: String
rpIDRP: String
rpIDRP2: int
rpIDSimboloO1: int
rpImgSimboloO1: String
rpLenguaje: String
rpNumElemPredecir: int
rpReusoO: int
rpReusoY: int
rpTipoCompUnitSO1: String
rpTipoConocimiento: String
rpTipoSO1: int
RenglonRP()
RenglonRP()
imprimeRenglon()
«Call, Import»
«Call, Instantiate, Import»
«Access, Call, Instantiate, Import»
«import»


D
RenglonIP
ipDocumentable: int
ipFuncion_N: String
ipFuncion_O: String
ipIDInstruccion: int
ipIDPrograma: String
ipLineaFin: int
ipLineaIni: int
ipParm1: String
ipParm2: String
ipParm3: String
ipParm4: String
ipParm5: String
ipSistema: String
ipSubsistema: String
ipTipoInsrt: String
ipVerbo: String
RenglonIP()
RenglonIP()
actualizaRenglon()
entregaGema()
imprimeRenglon()
insertaRenglon()
setValues()
AnalisisSintactico
esParrafoEncontrado: boolean
esSeccionEncontrada: boolean
IDConLexInstA: Comparator<Conocimiento>
IDRPSA: Comparator<Conocimiento>
RenglonA_ModoD: Comparator<Conocimiento>
TablaA_ModoD_IDA: Comparator<Conocimiento>
AnalisisSintactico()
TerminaAnalisis()
agregaAtribRedundEntidades()
agregaGemasFaltantes()
agregaGemasFaltantesP()
agregaGemasRE()
almacenaGema()
almacenaGemas()
analizaEntrada()
comparaSimbolo()
formateaGemas()
fusionaListaConNT()
prediceNTCandidato()
recuperacionDeError()
reglaProduccion()
validaReglaPUnica()
ListaRenglonesRSO
listaRenglonesRSO: ArrayList<RenglonRSO>
ListaRenglonesRSO()
clear()
getLength()
insertaElemento2()
obtieneElemento()
ListaRenglonesRSA
listaRenglonesRSA: ArrayList<RenglonRSA>
ListaRenglonesRSA()
clear()
getLength()
insertaElemento2()
obtieneElemento()
RenglonRSA
rsaClaseSimbolo: int
rsaDivision: String
rsaIDRP: String
rsaIDRP2: int
rsaIDSimbolo: int
rsaImgSimbolo: String
rsaLenguaje: String
rsaModoDeSerial: int
rsaOcurreSimbolo: int
rsaSecuenciaElem: int
rsaTipoCoinc: String
rsaTipoCompUnitS: String
rsaTipoSimbolo: int
RenglonRSA()
RenglonRSA()
imprimeRenglon()
RenglonRSO
rsoClaseSO: int
rsoDivision: String
rsoIDRP: String
rsoIDRP2: int
rsoIDSimboloO: int
rsoImgSimboloO: String
rsoIndConocimiento: int
rsoLenguaje: String
rsoOcurreSO: int
rsoSecSimboloO: int
rsoSecSimboloY: int
rsoTipoCompUnitSO: String
rsoTipoSO: int
RenglonRSO()
RenglonRSO()
imprimeRenglon()
ListaRenglonesCL
listaRenglonesCL: ArrayList<RenglonCL>
ListaRenglonesCL()
clear()
getLength()
insertaElemento2()
obtieneElemento()
ListaConocimiento
listaConocimiento: ArrayList<Conocimiento>
ListaConocimiento()
actualizaElemento()
agregar()
borraElemento()
clear()
copy()
getLength()
insertaElemento2()
insertaElemento3()
obtieneElemento()
RenglonCL
clDivision: String
clID2ColumnaTD: int
clIDColumnaTD: int
clIDConLex: int
clIDLlavePri: int
clIDRP: String
clIDRP2: int
clIDRP2S: int
clIDRPS: String
clIDRenglonTD: int
clIndLlaveExt: int
clIndNormalizacion: int
clIndOcurre: int
clIndPrecedenciaT: int
clIndSoloRef: int
clLenguaje: String
clSTNT: int
clSecSimboloOL: int
clSecSimboloOS: int
clSecSimboloYL: int
clSecSimboloYOS: int
clSecSimboloYS: int
clTablaDestino: String
RenglonCL()
RenglonCL()
imprimeRenglon()
ResComparaSimbolo
claseRef: int
descResultado: String
IDSimboloRef: int
imgSimboloRef: String
simboloAceptado: boolean
tipoCompUnitRef: char
tipoRef: int
ResComparaSimbolo()
clear()
clearEntrada()
clearSalida()
getIDSimboloRef()
setEntrada()
setIDSimboloRef()
setSalida()
toString()
RenglonRSY
rsyAlternativaImpuesta: int
rsyDivision: String
rsyIDComunInterAltern: int
rsyIDRP: String
rsyIDRP2: int
rsyLenguaje: String
rsyNumSimbolosO: int
rsySecSimboloY: int
rsyTipoObligatoriedad: String
RenglonRSY()
RenglonRSY()
imprimeRenglon()
ListaRenglonesRSY
listaRenglonesRSY: ArrayList<RenglonRSY>
ListaRenglonesRSY()
clear()
getLength()
insertaElemento2()
obtieneElemento()
«Call, Instantiate, Import»
«import»
«Access, Call, Instantiate, Import»