VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN
FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN

# "DESARROLLO DE UN SISTEMA DE CÓMPUTO INTEGRAL APLICADO A PROBLEMAS DEL MEDIO AMBIENTE"

## T E S I S

QUE PARA OBTENER EL GRADO DE:

## MAESTRA EN CIENCIAS (COMPUTACIÓN)

## P R E S E N T A:

ANGÉLICA ESPINOZA GODÍNEZ

DIRECTOR DE TESIS: DR. VLADIMIR TCHIJOV

México, D.F. 2005.

# Agradecimientos

A la UNAM y en particular a la FES-Cuautitlán por darme la oportunidad de adquirir conocimientos y de compartirlos.

Al CONACYT por el apoyo económico otorgado.

A mis sinodales por sus acertadas sugerencias:

Dr. Ismael Herrera Revilla
Dr. Gustavo Amado Ayala Milián
Dr. Vladimir Tchijov
Dr. Vladislav Khartchenko
M. en C. Ricardo Paramont Hernández García

Y un agradecimiento muy especial a mi director de tesis, el Dr. Tchijov, por su valiosa dedicación y al profesor Carlos Pineda por todo su apoyo y sus palabras de ánimo.

Al Consejo Mexiquense de Ciencia y Tecnología (COMECYT) por haberme apoyado para concluir la última fase de esta tesis.

Dedicada con amor y cariño a mis hijos:
Itzel y Pepito.

Y con mucho amor a José Luis,
porque el logro de esta meta
fue un esfuerzo de todos.

# ÍNDICE

# OBJETIVO

Desarrollar un sistema de cómputo integral que constituya la implementación original de un *analizador sintáctico (parser en inglés)* para el cálculo analítico y exacto de la matriz *Jacobiana* en el modelo matemático de reacciones químicas del medio ambiente (con referncia a contaminación de atmosférica); la resolución del sistema rígido de *Ecuaciones Diferenciales Ordinarias (EDO)* de este modelo; utilizando dicha matriz en el paquete computacional *CVODE* y, la representación gráfica de los resultados en tiempo de ejecución. Para ello, utilizar programación orientada a objetos en *C++*, *STL (por sus siglas en inglés Standard Template Library)* y herramientas de programación visual y multihilos proporcionadas por el *Borland C++ Builder*.

# INTRODUCCIÓN

En simulación computacional de problemas del medio ambiente (con referencia a contaminación atmosférica), surge la necesidad de resolver numéricamente sistemas rígidos de Ecuaciones Diferenciales Ordinarias *(EDO)* de reacciones químicas de tamaño importante de manera eficiente y eficaz. Para ello, se usan sofisticados paquetes computacionales de alto rendimiento tales como *CVODE*. La aplicación eficiente de *CVODE* implica el uso de la matriz *Jacobiana* de ecuaciones de reacciones químicas en su forma analítica. Por el gran tamaño de esta matriz resulta difícil obtenerla manualmente, lo que hace indispensable el uso de un analizador sintáctico (*parser en inglés)* para su cálculo exacto. Por otro lado, paquetes como *CVODE* no incluyen la graficación de los resultados en tiempo de ejecución. Sin embargo, herramientas computacionales de programación visual y multihilos *(multithread en inglés)* de *Borland C++ Builder* permiten el desarrollo de sistemas de cómputo integrales para cálculos numéricos y graficación que facilitan notablemente la interacción entre el usuario y el programa.

El objetivo de esta tesis es desarrollar un sistema de cómputo integral que constituya la implementación original de un *analizador sintáctico* para el cálculo analítico y exacto de la matriz *Jacobiana* en el modelo matemático de reacciones químicas del medio ambiente; la resolución del sistema rígido de *EDO* de este modelo utilizando dicha matriz en el paquete computacional *CVODE* y, la representación gráfica de los resultados en tiempo de ejecución. Para ello, fue utilizada la programación orientada a objetos en *C++*, la *STL (por sus siglas en inglés de Standard Template Library)* y las herramientas de programación visual y multihilos proporcionadas por el *Borland C++ Builder*.

La obtención de la matriz *Jacobiana* requiere del análisis carácter por carácter del sistema de ecuaciones de reacciones químicas, para diferenciar una unidad llamada símbolo (*token en inglés)*, luego clasificar el símbolo por su tipo e interpretarlo basándose en una gramática válida. Con el procedimiento anterior, tenemos la posibilidad de distinguir cada parte de los términos involucrados en el sistema de ecuaciones y calcular sus derivadas parciales, dando origen a la matriz *Jacobiana*. Considerando lo repetitivo del procedimiento y obedeciendo a una gramática convenientemente definida, se ha desarrollado un programa donde se permite manipular un archivo de entrada conteniendo el sistema de ecuaciones. El sistema de cómputo integral que se desarrolla en esta tesis está dividido en tres procesos: el primero, implementa un modelo de *análisis sintáctico* proporcionando como salida un archivo con la matriz *Jacobiana* obtenida en el formato reconocido por *CVODE*; el segundo, toma como entrada el archivo con la matriz *Jacobiana* y lo usa en *CVODE* y; el tercero, recibe los resultados numéricos proporcionados por *CVODE* dando como salida los resultados gráficamente.

Desde el punto de vista técnico, el contar con un sistema de cómputo integral donde sólo se requiera de un *archivo de texto* con un formato sencillo como entrada; un *analizador sintáctico* escrito en un lenguaje de programación tan versátil como es *C++* y utilizando la *STL* para obtener mayor eficiencia en la manipulación de los datos; el uso de software reconocido y de uso libre para solución numérica de sistemas rígidos de *EDO* como es *CVODE* y, el empleo de la técnica de programación multihilos del *Borland C++ Builder* en la graficación de los resultados, todo esto le brinda al usuario (un químico por ejemplo) una solución confiable, rápida y gráfica con el distintivo de emplear un software desarrollado específicamente para satisfacer sus necesidades.

Una contribución importante de esta tesis es su integración a los trabajos de investigación de la FES Cuautitlán, específicamente con relación al proyecto *PAPIIT* No. 105401 "Modelación matemática de sistemas multifásicos y multicomponentes en problemas del medio ambiente" (investigador responsable Dr. Vladimir Tchijov). Por lo tanto, los programas desarrollados se aplican de manera directa e inmediata a la investigación realizada en esta facultad.

# CAPÍTULO 1. ANÁLISIS SINTÁCTICO EN LA COMPILACIÓN DE PROGRAMAS

El *análisis sintáctico (parsing en inglés)* también conocido como *análisis gramatical,* tiene principal aplicación en la compilación de programas de cómputo, por lo cual se aborda como introducción en este capítulo las fases de compilación, poniendo mayor atención a las tres primeras fases (*análisis lexicográfico, análisis sintáctico y análisis semántico*) y a algunos métodos para la creación del modelo de análisis sintáctico de esta tesis.

En el presente capítulo, se explican algunos fundamentos de operación de las primeras fases de compilación y posteriormente en el **Capítulo 4**, es expuesto el modelo de análisis sintáctico adoptado en esta tesis para el cálculo analítico y exacto de la *matriz Jacobiana* en el modelo matemático de reacciones químicas del medio ambiente (con referencia a contaminación atmosférica); logrando tener una implementación original de un analizador sintáctico.

## 1.1 FASES DE UN COMPILADOR

El proceso de compilación puede variar en cuanto al número de fases y su interacción con el manejador de tablas y de errores, para los autores **[AHO1986, p. 10] Figura 1.1.1** y **[LEMON1996, p. 3] Figura 1.1.2**, dicho proceso consta de 6 fases principales.

Conceptualmente, un compilador trabaja en fases, donde éstas transforman el programa fuente de una representación a otra **[AHO1986, p. 10]**. En la **Figura 1.1.1** el proceso inicia con la entrada del programa fuente, luego las seis fases (*analizador léxico, analizador sintáctico, analizador semántico, generador de código intermedio, optimización de código, y generador de código*), ejecutan su función trabajando conjuntamente con el manejador de tabla de símbolos y el manejador de errores, hasta producir el código ensamblado y finalmente producir el programa destino.

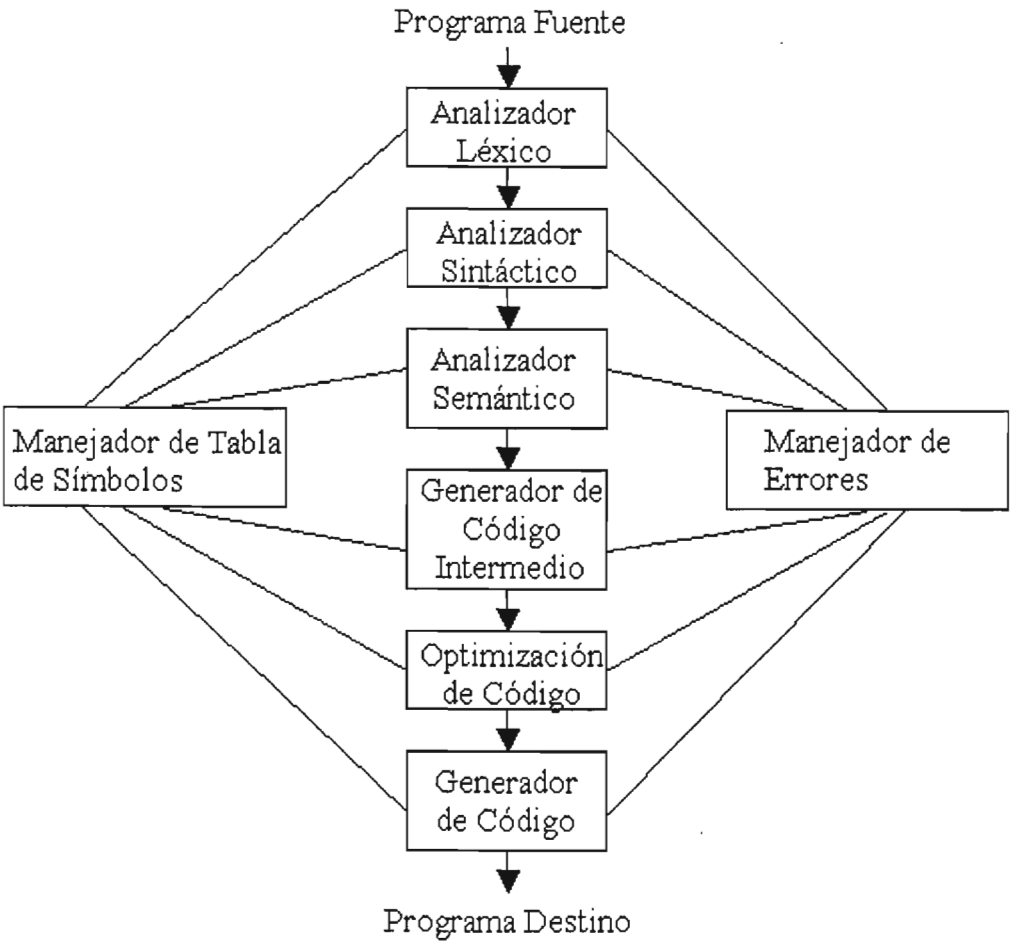



Figura 1.1.1 Fases de un Compilador [AHO1986, p. 10]

El proceso de compilación puede describirse como una secuencia de fases seriadas que comienzan con el *análisis lexicográfico* y finalizan con la *generación del código*, con el *manejo de tabla, módulos de error y de E/S* que interactúan con más de una fase **[LEMON1996, p. 3]**.

En la **Figura 1.1.2 Fases del compilador [LEMON1996, p. 3]** se puede apreciar una simplificación del proceso de compilación que consta de seis fases (*análisis lexicográfico, análisis sintáctico, análisis semántico, optimización, preparación para la generación del código, y generación del código*), además de tres subprocesos operando paralelamente (*manejo de tablas, manejo de errores y, entrada/salida*). El subproceso del manejo de tablas, opera la tabla de símbolos, la tabla de literales o identificadores usados en el código fuente, la tabla de ciclos iterativos y la tabla de representación intermedia. En la práctica las fases

frecuentemente interactúan con alguna de las otras o no siempre se realizan en el orden mostrado en las figuras.

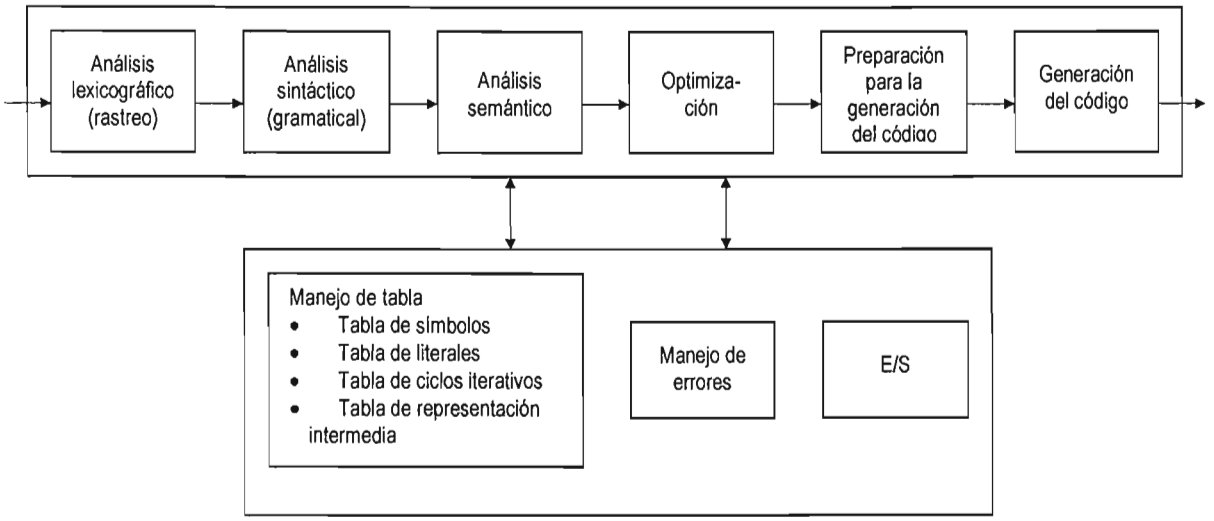



Figura 1.1.2 Fases del compilador [LEMON1996, p. 3]

## 1.2 ANÁLISIS LEXICOGRAFICO

El *análisis lexicográfico* es el proceso mediante el cual se rastrea y filtra una cadena de caracteres, para identificar las componentes léxicas conocidas como símbolos (*tokens en inglés)*. El rastreo y filtrado de caracteres se realiza en el programa fuente, moviendo un puntero una posición a la vez de izquierda a derecha, desde el inicio del programa hasta llegar al final. Para lo anterior, el programa fuente es visto como una cadena de caracteres finitos. El resultado del *análisis lexicográfico* es la determinación de símbolos que son enviados a la fase de *análisis sintáctico* para su evaluación.

Un símbolo es la unidad léxica básica de un lenguaje, puede constar de un solo caracter o una secuencia de ellos; como un signo de puntuación o una palabra **(véase Tabla 1.2.1 Ejemplos de símbolos)**. Los símbolos varían de lenguaje en lenguaje, e incluso de compilador en compilador para el mismo lenguaje. La elección de los símbolos es una de las tareas del diseñador de compiladores **[LEMON1996, p. 5]**.

| Símbolo | Significado |
|---|---|
| If | Inicio de una condición |
| <, <=, =, <>, >, >= | Símbolos de relación |
| 3.1416 | Numero |
| Var, Var1 | Nombre de Variable |
| ; | Final de sentencia |

Tabla 1.2.1 Ejemplos de símbolos

Los símbolos se forman a partir de expresiones regulares. Para generar expresiones regulares se usan frecuentemente diagramas de transición, en los cuales pueden determinarse las cadenas de caracteres que forman un símbolo. Los diagramas de transición son conocidos formalmente como autómatas de estados finitos. Existen los autómatas finitos deterministas y no deterministas. La diferencia más notable entre los dos tipos de autómatas, consiste en que los autómatas finitos no deterministas pueden pasar de un solo estado de transición a otros estados con el mismo símbolo de entrada; mientras que en los autómatas deterministas con un símbolo de entrada se puede transitar desde un estado a otro único estado. En esta tesis, se emplearon autómatas finitos deterministas por la facilidad de su codificación en un lenguaje de programación.

Debido a lo extenso del tema sobre expresiones regulares y lenguajes formales no será explicado en detalle su contenido, para ello se recomienda consultar **[AHO1986, p. 83-145]** y **[HOPCROFT1993, p. 15-48]**.

En el apartado **4.1.1. Análisis Lexicográfico del Capítulo 4**, se muestran los símbolo*s* válidos para el sistema de cómputo integral desarrollado en esta tesis.

## 1.3 ANÁLISIS SINTÁCTICO

El *análisis sintáctico (parsing en inglés)* es el proceso mediante el cual se evalúa un símbolo en un lenguaje y una gramática para ese lenguaje, como se hace con las palabras al formar una oración de un lenguaje, el resultado es un árbol derivado en estructuras sintácticas y símbolos.

A continuación se proporcionan algunas de las concepciones acerca del *análisis sintáctico*:

**[AHO1986, p. 10]** "El *parser* obtiene una cadena de *tokens* desde el analizador léxico, [...] y verifica que la cadena pueda ser generada por la gramática del lenguaje fuente".

**[AHO1986, p. 40]** "*Parsing* es el proceso de determinar si una cadena de *tokens* puede ser generada por una gramática".

**[LEMON1996, p. 7]** "El *análisis sintáctico* o fase gramatical de un compilador, agrupa los *tokens* en estructuras sintácticas en forma muy similar a como teníamos que estructurar las oraciones en la primaria".

El *analizador sintáctico* es una fase del proceso de compilación de un programa escrito en un lenguaje de computación, es también conocido como *analizador gramatical*, debido a que la sintaxis forma parte de la gramática de un lenguaje. En inglés, la fase de *análisis sintáctico* es conocida como *parsing*.

Es muy usual emplear *gramáticas libres de contexto o BNF* (*abreviatura en inglés de Backus-Naur Form*) para definir lenguajes que puedan crear un *árbol de análisis sintáctico*. Una *gramática libre de contexto* consta de terminales, no terminales, un símbolo inicial y producciones **[AHO1986, p. 26-27,170]**. Por otro lado, un *árbol de análisis sintáctico* indica gráficamente como del símbolo inicial de una gramática deriva una cadena del lenguaje y tiene las siguientes propiedades:

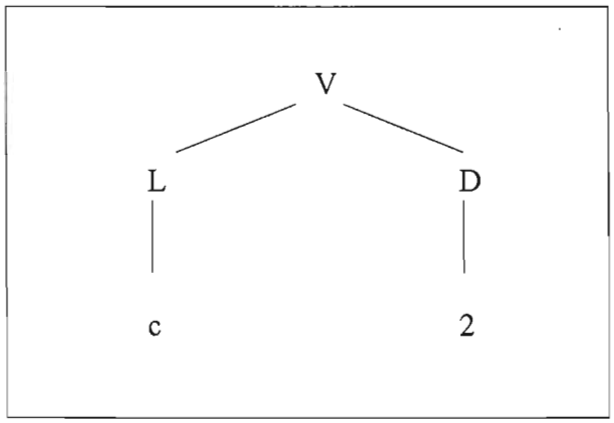
1. La raíz o nodo inicial esta etiquetada con el símbolo inicial
2. Cada hoja está etiquetada con un componente léxico o con $\epsilon$ (significa cadena vacía)
3. Cada nodo interior está etiquetado con un no terminal
4. Si $A$ es el no terminal que etiqueta a algún nodo interior y $X_1, X_2, ..., X_n$ son las etiquetas de los hijos de ese nodo, de izquierda a derecha, entonces $A->X_1, X_2, ..., X_n$ es una producción. Aquí, $X_1, X_2, ..., X_n$ representa un símbolo que es un terminal o un no terminal. Para el caso especial, si $A->\epsilon$, entonces un nodo etiquetado con $A$ tiene un solo hijo etiquetado con $\epsilon$.

**Ejemplo 1.3.1** Sea la siguiente *gramática libre de contexto* para nombrar variables que empiecen con la letra *a* o *b* y después tengan un número entre *1* y *3*. Construir el *árbol de análisis sintáctico* de la cadena *c2*.

*V -> LD*
*L -> a | b | c*
*D -> 1 | 2 | 3*

- El símbolo inicial es *V*
- Las producciones son *V->LD, L->a, L->b, L->c, D->1, D->2 y D->3*
- Los terminales son: *a, b, c, 1, 2 y 3*
- Los no terminales son: *L y D*

El *árbol de análisis sintáctico* quedaría constituido como se muestra a continuación




Existen diferentes *métodos de análisis sintáctico*, la elección de uno de ellos para su implementación en compiladores o en sistemas de cómputo varía según la facilidad y eficiencia del algoritmo en relación al tipo de problema. El tipo de problema podría caracterizarse por una gramática que lo represente.

## 1.4 MÉTODOS DE ANÁLSIS SINTÁCTICO

Existen varios métodos de análisis sintáctico, pudiendo ser de dos tipos: descendentes o ascendentes. La clasificación del método corresponde al orden en que se forman los nodos del árbol de análisis sintáctico. En los métodos descendentes, la construcción del árbol empieza por un nodo raíz y se expande hasta las hojas; mientras que en los ascendentes, la construcción del árbol empieza con las hojas y se expande hasta el nodo raíz. La **Figura 1.4.1** muestra una clasificación de los métodos de análisis sintáctico.

### Método de análisis sintáctico descendente

El *método de análisis sintáctico descendente* crea el *árbol de análisis sintáctico* partiendo de un nodo raíz, etiquetado como el nodo inicial no terminal y generando los nodos hijos, empezando por el hijo situado más a la izquierda. Este método se efectúa repitiendo los siguientes dos pasos:

1. En el nodo *n*, etiquetado con el no terminal *A*, seleccionar una de las producciones para *A* y expandir con los hijos de *n* para los símbolos del lado derecho de la producción.
2. Encontrar el siguiente nodo donde ha de construirse un subárbol.

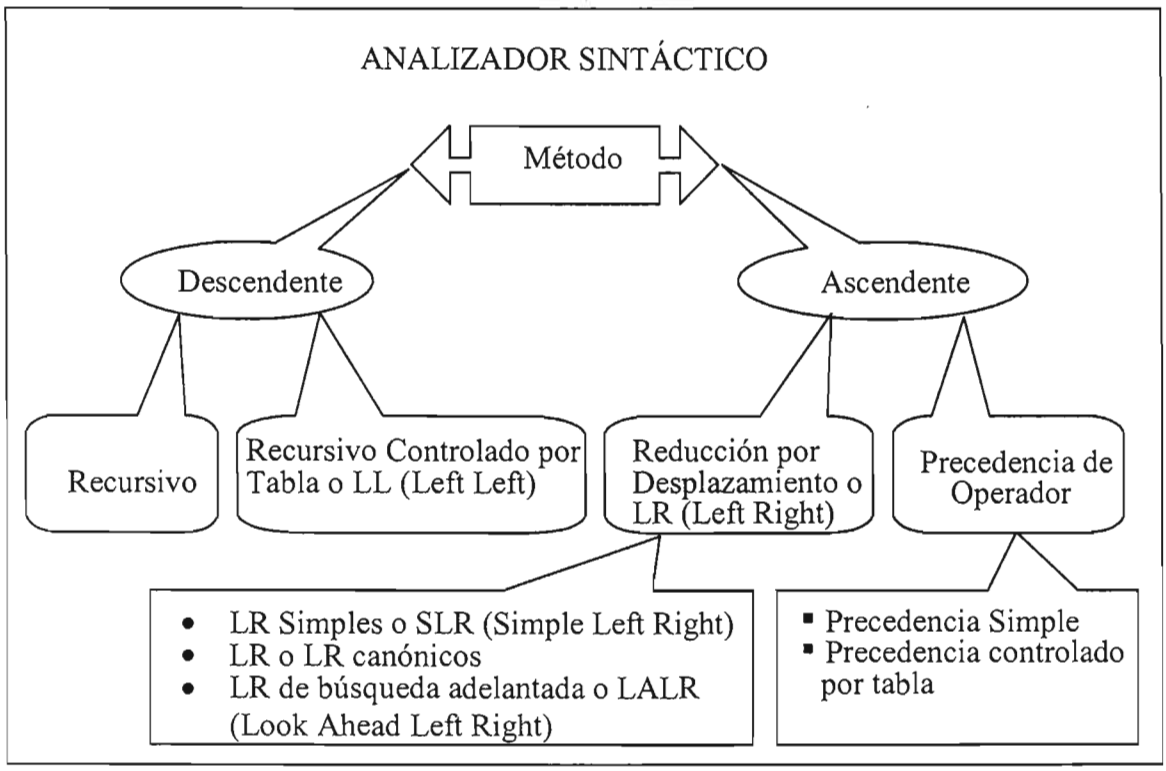



Figura 1.4.1 Clasificación de métodos de análisis sintáctico

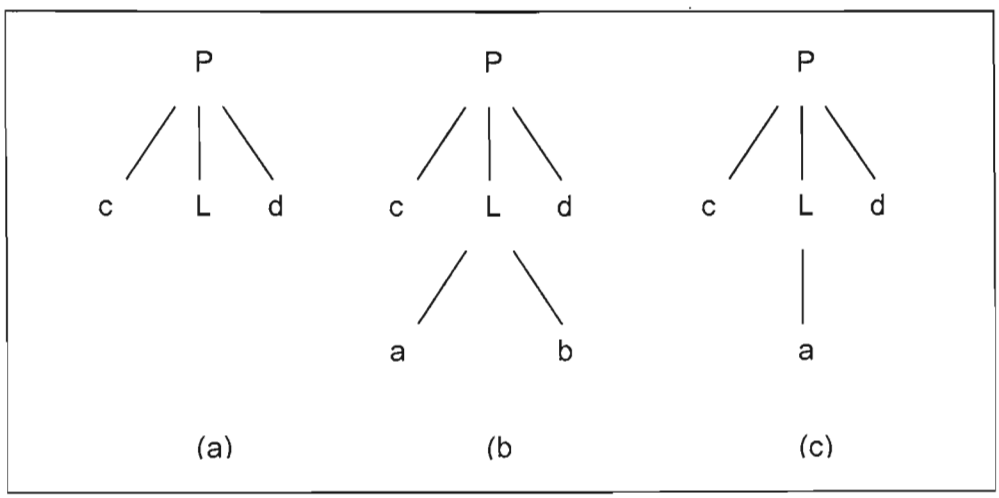



Figura 1.4.2 Pasos para el *Método de Análisis Sintáctico Descendente*

**Ejemplo 1.4.1** Construcción del árbol sintáctico descendente considerando la gramática siguiente:

*P -> cLd*
*L -> ab | a*

y el símbolo de entrada *u= cad*. Para crear el *árbol de análisis sintáctico descendente* del símbolo, primero se crea el nodo raíz, en este caso el nodo raíz es *P*, un puntero debe apuntar a la entrada *c*, el primer carácter del símbolo *u*. Después se utiliza la primer producción de *P* para expandir el árbol y obtener el árbol de la **Figura 1.4.2 (a).** Se emparejan la hoja más a la izquierda, con la etiqueta *c*, con el primer carácter de *u*, luego se cambia puntero de la entrada al carácter *b*, el segundo carácter de *u*, y se evalúa la siguiente hoja etiquetada con *L*. Como *L* es un nodo no terminal se puede expandir utilizando la primer alternativa de *L* para obtener el árbol de la **Figura 1.4.2 (b).** Al realizarse la comparación se obtiene la concordancia de los dos caracteres y se avanza el puntero de entrada a *d*, el tercer carácter de *u*, con la hoja siguiente, etiquetada con *b*. Como *b* no concuerda con *d*, se indica un fallo y se regresa a *L* para ver si existe otra alternativa de expansión que no se haya intentado, pero que pueda dar lugar a un emparejamiento.

Al regresar a *L*, se debe restablecer el puntero a la posición 2, aquella que tenía al evaluarse *L* por primera vez. Se intenta a continuación la segunda alternativa de *L* para obtener el árbol de la **Figura 1.4.2 (c)**. Se compara y empareja *a* con el segundo carácter de *u*, y la hoja *d*, con el tercer carácter de *u*. Como ya se ha producido el *árbol de análisis sintáctico* para *u*, se para y se anuncia del éxito del proceso.

Existen otros métodos que se derivan del método descendente, uno de ellos implementa el concepto de *recursividad* para la generación del *árbol de análisis sintáctico*. El ***método de análisis sintáctico recursivo descendente*** es un método en el que se ejecuta un conjunto de procedimientos recursivos para procesar la entrada y modelar el *árbol de análisis sintáctico* por construirse. Para cada no terminal de una gramática es asociado un procedimiento que lo analiza gramaticalmente. Cada uno de estos procedimientos puede leer la entrada de caracteres, hacer concordar o emparejar símbolos terminales o llamar otros procedimientos para leer entradas (*símbolos*) y hacer coincidir terminales en el lado derecho de la producción.

Para el ***método de análisis sintáctico recursivo descendente LL(k),*** el procedimiento es muy similar al método recursivo descendente. La primer *L (por left-to-right)* significa que la cadena de caracteres se evalúa de izquierda a derecha, moviendo un puntero en ese sentido. La segunda *L (por leftmost derivation)*, se interpreta como la construcción del *árbol de análisis sintáctico* con derivación por la izquierda y, el símbolo *k* indica el número de caracteres de entrada evaluándolos en cada paso anticipadamente antes de emparejarlo con un símbolo terminal o un no terminal. Si *k* es omitida, entonces *k* es igual a *1*. La característica particular del *método recursivo LL*, radica en el empleo de una tabla para el *análisis sintáctico*, esta tabla consta de renglones y columnas; en los renglones se definen los nodos no terminales y en las columnas los símbolos de

entrada, las decisiones de emparejamiento se toman en base a las coincidencias existentes en la tabla, tomando como coordenadas el no terminal y el símbolo de entrada. Esta coordenada debe contener una única producción válida para esos valores definidos en la gramática del lenguaje.

Un método recursivo descendente simple, está constituido por procedimientos (o módulos de programación) recursivos para evaluar la gramática del lenguaje; mientras que los métodos *LL* recurren a una tabla predefinida para ese mismo propósito.

### Método de *análisis sintáctico* ascendente

El *método de análisis sintáctico ascendente* crea el *árbol de análisis sintáctico* partiendo de las hojas o nodos terminales (desde el fondo del árbol) y avanzando hacia al nodo inicial o raíz (cima). Los pasos son:

1. Comenzar con la cadena de caracteres de entrada (las hojas o terminales del árbol que será creado)
2. Intentar reducir al símbolo de inicio o raíz encontrando el controlador actual: el controlador es
    a) la colección más extensa de terminales y no terminales en la parte extrema izquierda de la entrada que coincida en el lado derecho de una producción y
    b) tal que todos los símbolos a la derecha del controlador sean terminales y
    c) tal que el reemplazo del controlador con el lado izquierdo de la producción eventualmente (en caso de encontrar más controladores) conduce de regreso al símbolo de inicio o raíz.

**Ejemplo 1.4.2** Construcción del árbol sintáctico ascendente considerando la gramática siguiente:

*S -> aABe*
*A -> Abc | b*
*B -> d*

La cadena *abbcde* se puede reducir a *S* como sigue:

*abbcde => aAbcde => aAde => aABe => S*

Se examina la cadena *abbcde* buscando una subcadena que concuerde con el lado derecho de alguna producción. Las subcadenas *b* y *d* pueden servir, sin embargo; debe elegirse la que se encuentra más a la izquierda en la cadena. Al elegirse *b,* se

sustituye *b* por *A* el lado izquierdo de la producción *A->b*; así se obtiene la cadena *aAbcde* **(véase Figura 1.4.3 (b))**. Luego, las subcadenas *Abc, b y d* concuerdan con el lado derecho de dos producciones (*A->Abc, A->b y B->d*), siendo *b* la subcadena situada más a la izquierda de la entrada que concuerda con el lado derecho de una producción, pero en este caso; debe elegirse la colección más extensa de terminales y no terminales; por tanto, sustituir la subcadena *Abc* por *A*, que es el lado derecho de la producción *A->Abc*. Ahora se obtiene *aAde* (**véase Figura 1.4.3 (c)**). Después, sustituyendo *d* por *B*, que es el lado derecho de la producción *B->b*, se obtiene *aABe* **(véase Figura 1.4.3 (d))**, para finalmente sustituir toda esta cadena por *S* (el símbolo de inicio o raíz) **(véase Figura 1.4.3 (e))**.

El proceso anterior podría verse como una derivación derecha en orden inverso:

*S->aABe=>aAde=>aAbcde=>abbcde*

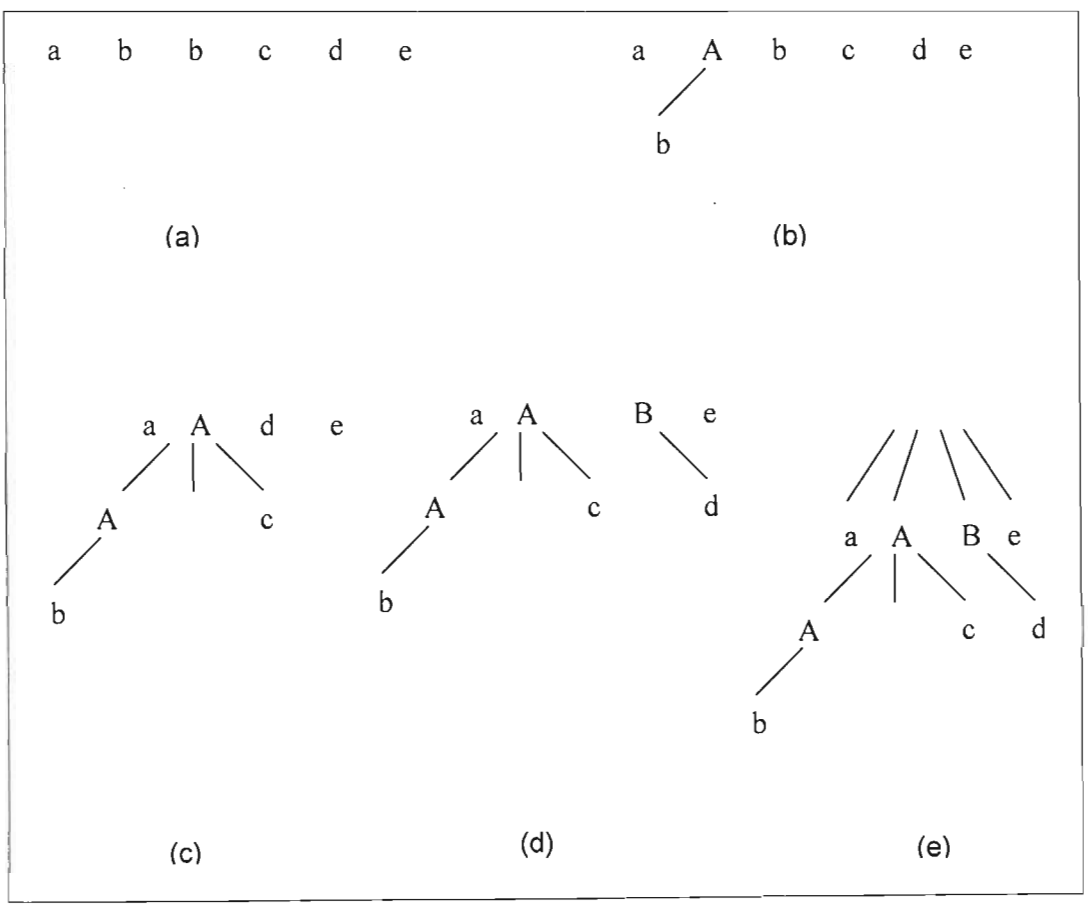



Figura 1.4.3 Pasos para el *Método de Análisis Sintáctico Ascendente*

El ***método de análisis sintáctico ascendente LR(k)***, es un método eficiente pero difícil de implementar manualmente, por lo que existen algunas herramientas que crean el *árbol de análisis sintáctico* partiendo de una gramática específica para un lenguaje. El nombre del método contiene las letras *LR*, *la letra L(por left-to-right)* significa que la cadena de caracteres se evalúa de izquierda a derecha, moviendo un puntero en ese sentido. La letra *R(por rightmost derivation)*, se interpreta como la construcción del *árbol de análisis sintáctico* con derivación por la derecha en orden inverso (es decir desde los no terminales hacia el nodo raíz) y, el símbolo *k* indica el número de caracteres de entrada evaluándolos en cada paso anticipadamente antes de emparejarlo con un símbolo terminal o un no terminal. Si *k* es omitida, entonces *k* es igual a *1*.

El método de *análisis sintáctico LR* está constituido por tres tipos: ***LR Simple*** *(Simple Left-to-Right Rightmost Derivation o SLR abreviado)*, ***LR*** *(o LR Canónico)* y ***LR de Búsqueda Adelantada*** *(LookAhead Left-to-Rigth Rightmost Derivation o LALR abreviado)*. El método *SLR* es más fácil de implementar pero no es tan eficiente, podría no producir una tabla de *análisis sintáctico* que otros métodos sí; el método *LR Canónico* es el más eficiente pero su implementación es costosa y, el método *LALR* ocupa un lugar intermedio entre los dos métodos anteriores en cuanto a facilidad de uso y términos de eficiencia, funciona con las gramáticas de la mayoría de los lenguajes **[AHO1986, p. 222]**.

La diferencia básica entre los tres métodos anteriores consiste en la manera que se construye la *tabla de análisis sintáctico*, así como la dimensión de ésta en cada tipo de método. La *tabla de análisis sintáctico* creada en el método *LALR* es más pequeña que la generada por el método *LR Canónico*. Las *tablas SLR y LALR* para una gramática siempre tienen el mismo número de estados, normalmente cientos de estados, en cambio la *tabla del análisis LR Canónico* podría tener miles de estados para un lenguaje del mismo tamaño. Por lo anterior, es más fácil y económico construir *tablas SLR y LALR* que *tablas del análisis LR Canónico* **[AHO1986, p. 243]**.

Los *métodos ascendentes LR* implican cierta dificultad para crear *árboles de análisis sintáctico* manualmente, sin embargo; son ampliamente implementados por su eficiencia, por esto último; existen varios generadores de analizadores sintácticos, uno de los más conocidos es *YACC (Yet Another Compiler-Compiler, que significa "Otro Compilador de Compiladores Mas")*, el cual utiliza el *método LALR(1)*.

Existen otros *métodos de análisis ascendente*: *precedencia de operador simple y precedencia de operador controlado por tabla*. Los dos métodos anteriores son empleados para gramáticas de operador, las cuales generan un subconjunto

específico de símbolos de un lenguaje más general, dichos métodos tienen como propósito analizar cadenas del lenguaje que puedan clasificarse como operador u operando.

El ***método ascendente de precedencia de operador simple*** hace uso de dos pilas como estructura de datos, para diferenciar en una los operandos y en otra los operadores. Es recomendado este método para gramáticas que generen expresiones algebráicas con un número reducido de símbolos como operadores.

Para una cantidad grande de símbolos como operadores, es más adecuado implementar el ***método ascendente de precedencia de operadores*** controlado por tabla, debido a que a través de la tabla se puede hacer la consulta directa entre los operandos y la relación de precedencia entre éstos y los operadores.

## 1.5 ANÁLISIS SEMÁNTICO

La fase de *análisis semántico* forma parte de la verificación estática de errores. En esta fase se parte de la estructura generada por el *análisis sintáctico* creando otra estructura que interpreta a la primera y le da significado, es decir;.es aquí donde es analizado el *árbol de análisis sintáctico* y es creada otra estructura que bien puede ser un árbol de sintaxis abstracto, una *notación Polaca posfija* o código en tres direcciones; la cual representa el código intermedio entre el código fuente y el código objeto. Las tareas principales de la tercer fase, consisten en crear la tabla de símbolos y traducir el *árbol de análisis sintáctico* a una representación intermedia más apropiada para apoyar las siguientes fases de compilación.

Las tres primeras fases de compilación se encargan de realizar la verificación estática de errores durante el tiempo de compilación de un programa; mientras que, el resto de las fases se encargan de detectar otros errores generados en la ejecución del mismo programa.

Existen ciertos errores que las *gramáticas libres de contexto* a través de la *BNF* no pueden detectar durante la verificación estática, por ejemplo; en algunos lenguajes de programación es necesario declarar una variable para poder usarla después, si no se hace de este modo se produce un error, éste tipo de error no es sintácticamente incorrecto, porque se puede producir una estructura válida en la *BNF*, sin embargo; es semánticamente incorrecto porque no se podría asignar a la variable su valor según el tipo de dato correspondiente puesto que se desconoce.

Las *gramáticas de atributo* **[LEMON1996, p. 126-130]** buscan concluir la verificación estática de errores de un programa como uno de los propósitos de la fase de *análisis semántico*; son también libres de contexto, agregan atributos y

funciones semánticas para lograr dicho propósito. Los atributos son variables que toman valores y están relacionados con uno o más no terminales o terminales de la gramática. Por otro lado, las funciones semánticas son ecuaciones mediante las cuales se asignan los valores a los atributos locales, siendo éstos últimos los que caen dentro del ámbito de una producción en el *árbol sintáctico*. Las *gramáticas de atributo* también son conocidas como *definiciones dirigidas por la sintaxis* **[AHO1986, p. 288-295]**.

**Ejemplo:**

| **Producción** (Sintaxis) | **Reglas o Funciones Semánticas** |
|---|---|
| *Exp -> Exp + Ter* | *Exp.Valor = Exp.Valor + Ter.Valor* |

La creación de la tabla de símbolos podría realizarse desde la fase de *análisis lexicográfico*, sin embargo; algunos compiladores están diseñados para crearla después de tener el *árbol de análisis sintáctico*; es decir, una vez que se conoce toda la información relevante de la estructura general del programa. La tabla de símbolos por lo regular incluye información de tipo de datos, ámbito de nombres (variables, procedimientos y funciones) e información acerca de la ubicación posible en la memoria.

Se realizan otras verificaciones estáticas durante esta fase, como la verificación de tipos para asegurar la compatibilidad entre operador y operandos; la verificación de arreglos para no rebasar sus dimensiones al utilizar sus índices; la verificación por duplicidad de nombres de variables, procedimientos o funciones, entre otros. Las restricciones del *análisis semántico* varían en cada lenguaje de programación.

Para el propósito de esta tesis, se toman en consideración las tres primeras fases de compilación (*análisis lexicográfico*, *análisis sintáctico* y análisis semántico), sobre las cuales se rige el **modelo de *análisis sintáctico*** empleado; siendo parte fundamental en el diseño del sistema de cómputo integral expuesto en el **Capítulo 4**.

En esta tesis se eligió la implementación del *método de análisis recursivo descendente* por su sencillez y por su gran acoplamiento al problema. La sencillez del método elegido se basa en la estrecha similitud entre el desarrollo manual y el desarrollo en un lenguaje de programación; la recursividad del método permite generar la gramática requerida evitando el uso de tablas de símbolos complejas o de gran tamaño y, favorece el uso de *gramáticas libres de contexto* representadas

en *EBNF (Extended Backus Naur Form)* de gran ayuda para la codificación del programa de *análisis sintáctico*. Por otra parte, el método mencionado presenta un gran acoplamiento al problema; con su característica recursiva atiende a la necesidad de procesar cada término de una ecuación, de igual manera que cada término del resto de las ecuaciones del sistema de EDO; con lo cuál, el problema se concibe como un problema de análisis recursivo en cada término de cada una de las ecuaciones.

Asimismo, el *análisis semántico* en el sistema de cómputo integral de esta tesis, se representa a través de un autómata, donde la secuencia de números 1, 2 y 3 determinan el tipo de símbolo de cada término en el sistema EDO (delimitador, número y variable, respectivamente). La evaluación de la secuencia generada, permite saber si el término es correcto, de ser así; el término es aceptado como válido y guardado en memoria.

# CAPÍTULO 2. MULTIHILOS Y GRAFICACIÓN

En la ejecución secuencial, concurrente y paralela de programas puede hacerse uso de herramientas de programación visual para ofrecer mayor acoplamiento del usuario con la interfaz gráfica y también para mostrar resultados con dibujo de imágenes, texto o líneas. En el caso de programas de ejecución concurrente y/o ejecución paralela, es necesario dividir el programa en tareas; de tal modo que se produzca mayor eficiencia en el procesamiento de datos y la obtención de resultados; para lo cual, es necesario diferenciar las tareas más complejas de las que no los son; así como también, se deben distinguir las dependencias de datos entre tareas.

En *Borland C++ Builder*, que se usó como herramienta principal para el desarrollo de esta tesis, es posible crear aplicaciones de ejecución concurrente o aplicaciones multihilos, asignando a cada hilo existente una tarea específica para obtener el propósito final de la aplicación.

La *VCL (Visual Component Library)* proporciona mecanismos de coordinación de hilos para evitar colisiones entre ellos que produzcan resultados erróneos o el colapso del sistema operativo *Windows*.

Las herramientas de programación visual de *Borland C++ Builder* permiten integrar a las aplicaciones el dibujo de texto, imágenes y líneas. Al combinar las características de desarrollo de aplicaciones multihilos y las herramientas visuales para graficación pueden obtenerse aplicaciones muy eficientes con una interfaz gráfica atractiva para el usuario.

## 2.1 MULTIHILOS

Al conjunto de instrucciones que son ejecutadas en un orden determinado para lograr un fin específico se le conoce como programa. Dicho conjunto de instrucciones puede ser ejecutado de forma secuencial, concurrente o paralela. Así como un sistema operativo multitarea tiene la capacidad de mantener en ejecución más de un proceso en forma concurrente y/o paralela, un proceso lo hace similarmente manteniendo en ejecución más de un hilo. Cada hilo representa una

línea de ejecución diferente con un flujo de control, pudiendo ejecutar sus instrucciones independientemente de los otros hilos presentes en el proceso.

Multihilos es un paradigma de programación en el cuál un solo proceso puede tener más de una línea de ejecución **[LEWIS1996, pag. 297]**.

Un programa multihilos puede lograr un rendimiento significativo a través del uso de la ejecución concurrente y/o paralela de hilos. La ejecución concurrente de hilos (o concurrencia) significa que dos o más hilos están en progreso al mismo tiempo **[NORTON1997, pag. 17]**.

Cada línea de ejecución es un hilo de una aplicación multihilos. Cuando una tarea involucra dos o más subtareas independientes, es posible resolverla a través de múltiples hilos de control representados en un programa multihilos. De este modo, un hilo podría encargarse de mostrar la *GUI (Graphical User Interface)*, un segundo hilo podría realizar cálculos matemáticos y un tercer hilo podría enviar resultados a una impresora.

En un programa de ejecución secuencial las instrucciones son ejecutadas una a la vez y el control del flujo de éstas instrucciones lo tiene a su cargo un solo proceso de un solo hilo *(thread en inglés)*, el cuál controla también las llamadas a las rutinas o funciones. El sistema operativo representa al programa secuencial como un solo proceso, el cuál contiene código, datos y demanda recursos de sistema para lograr su propósito.

Podemos decir que los programas de ejecución concurrente o paralela involucran tanto procesos como hilos, por lo cuál a partir de esta sección serán explicados conjuntamente los términos de proceso, hilos, concurrencia y paralelismo.

Un proceso es una instancia de la ejecución de un programa y también la unidad básica de ejecución del sistema operativo **[WALL2000, pag. 62]**. Los procesos constan de los siguientes elementos:

- **Código, datos y pila**. Son el texto del programa, los datos que maneja, el espacio del montículo *(heap)*, las bibliotecas compartidas y el espacio de pila que ocupa.
- **Contexto de Programa**. Está integrado por los registros de datos, los códigos de condición, puntero a la pila y el contador de programa.
- **Contexto de Kernel o Núcleo**. Está integrado por el identificador del proceso *(PID)*, la estructura que caracteriza la organización de la memoria virtual, información sobre los archivos abiertos, manejadores de señal y el apuntador de final.

Un hilo es considerado como una "unidad de ejecución, asociada a un proceso pero con su propio identificador de hilo, pila, puntero de pila, contador de programa, códigos de condición y registros de propósito general" **[BRYANT2001, pag. 1]**.

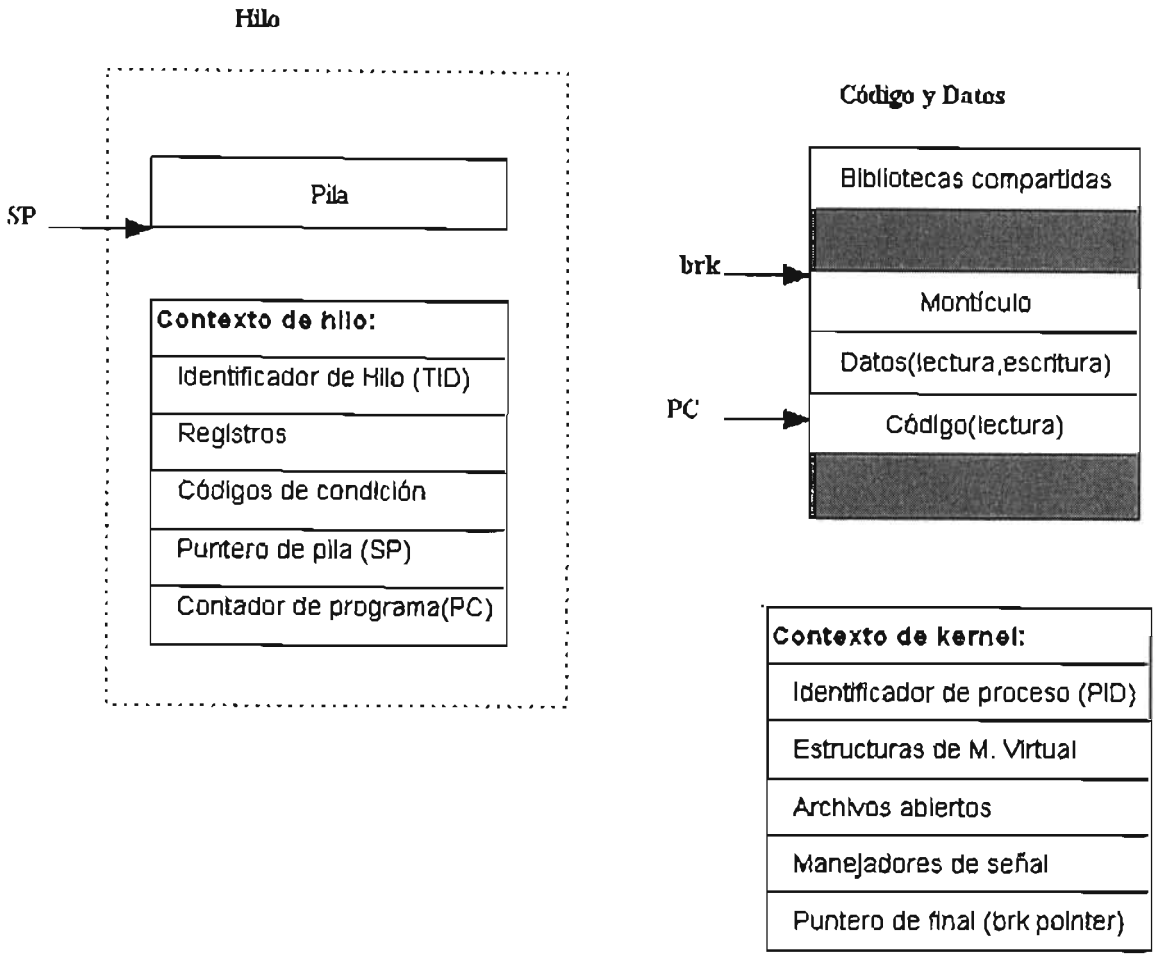



Figura 2.1.1 Relación de un hilo con el proceso donde fue creado

Cuando en un proceso se crean varios hilos, los hilos comparten código, datos, montículo *(heap)*, bibliotecas, manejadores de señales, identificador de proceso *(PID)*, estructuras de memoria virtual, puntero de final y archivos abiertos declarados en el proceso donde son lanzados los hilos **[BRYANT2001, pag. 1-3] (véase Figura 2.1.1)**.

Al existir varios hilos en un mismo proceso, cada uno de ellos tendrá su propia pila y contexto de hilo *(thread context en inglés)* integrado por registros de propósito general, códigos de condición, puntero de pila *(SP)* y contador de programa *(PC)*,

así como un identificador de hilo *(TID)* **[BRYANT2001, pag. 1-3] (véase Figura 2.1.2).**

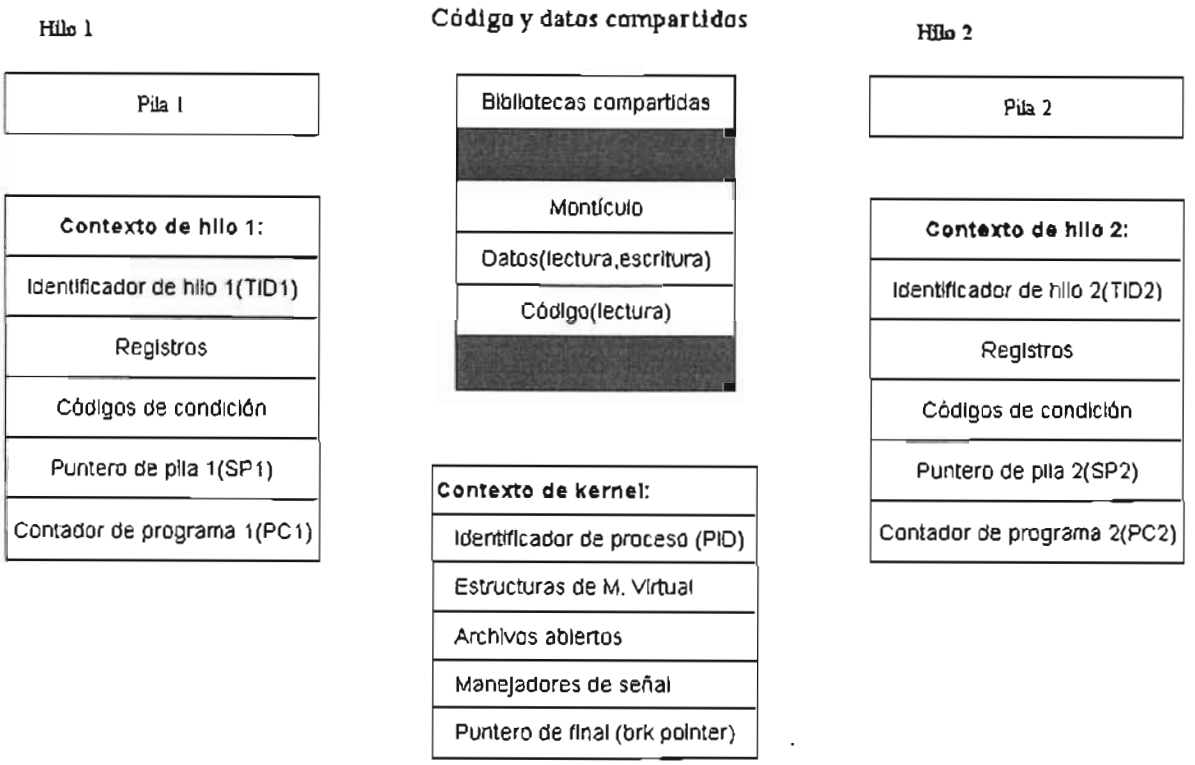



Figura 2.1.2 Relación de dos hilos con el proceso donde fueron creados.

El cambio entre procesos es similar al cambio entre hilos. A continuación se muestra en la **Figura 2.1.3** el cambio entre un proceso y otro, conocido como *"process context switch"*, el cuál consiste en modificar el estado activo de un proceso y activar algún otro. En la **Figura 2.1.3** el Proceso 1 inicia su ejecución, después en algún tiempo es activado el Proceso 2 y el contexto cambia, en este punto los dos procesos son ejecutados concurrentemente. De forma similar, la **Figura 2.1.4** muestra el cambio de contexto desde un hilo a otro conocido como "*thread context switch*".

Como se observa en la **Figura 2.1.4**, al crear varios hilos dentro de un proceso se tienen varias líneas de ejecución concurrente relativamente independientes con algunos recursos propios y otros definidos en el proceso; produciéndose así, una estructura de memoria compartida a través de los recursos del proceso, determinando así un mecanismo sencillo para compartir datos.

El utllizar un proceso con varios hilos consume menos recursos que el crear varios procesos concurrentes bajo el esquema clásico de los procesos. Asimismo, debido a que el contexto de un hilo es más pequeño que el de un proceso, el cambio entre un hilo y otro *"thread context switch"* es más rápido que si se llevara a cabo entre procesos. Observar la **Figura 2.1.3** y **Figura 2.1.4**.

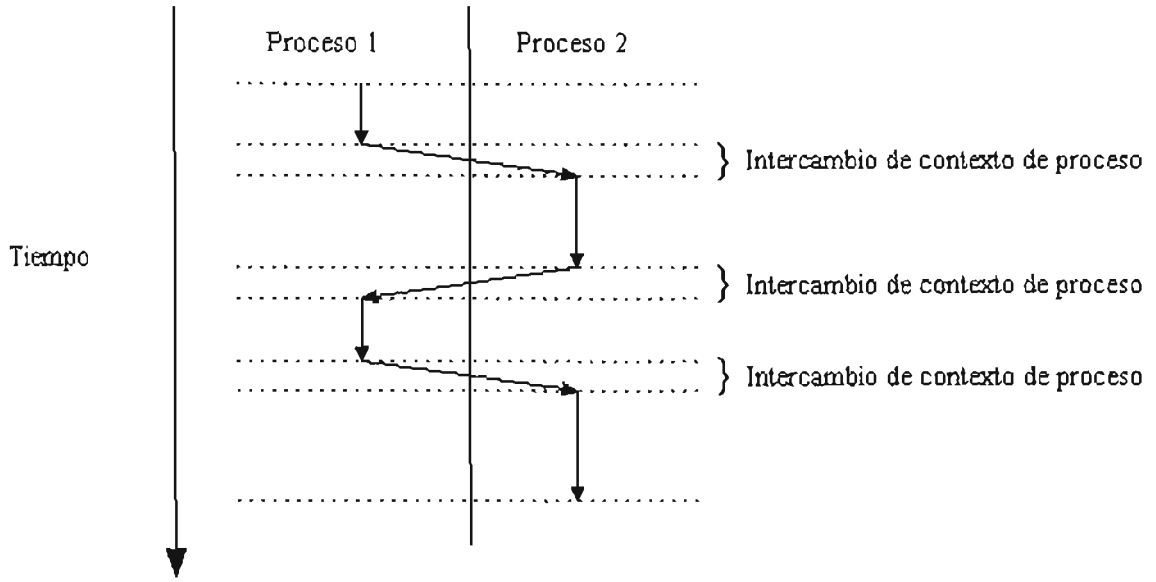



Figura 2.1.3 Intercambio de contexto de un proceso a otro *"process context switch"*.

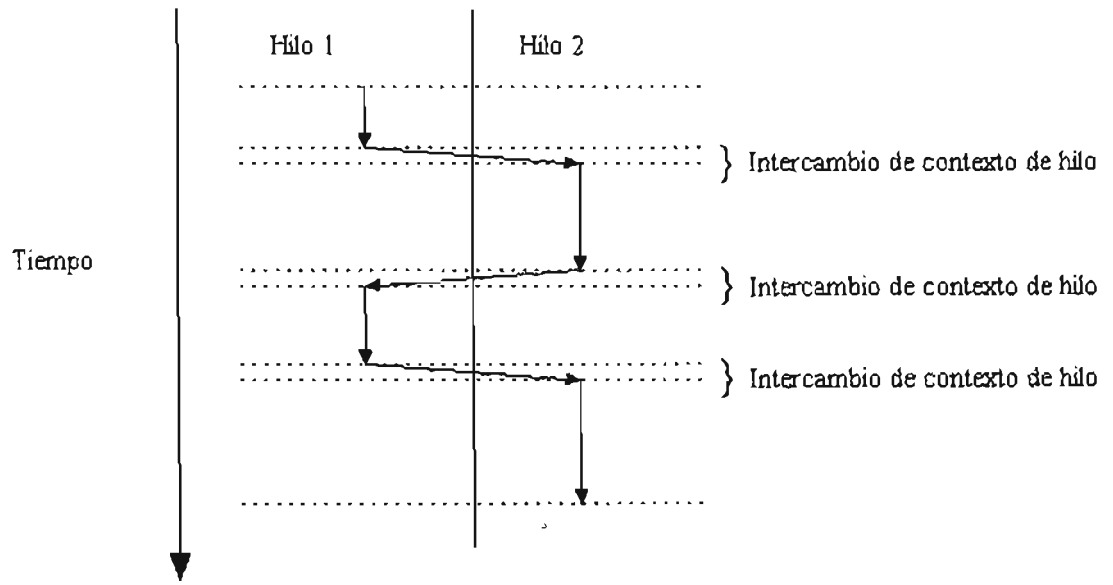



Figura 2.1.4 Intercambio de contexto de un hilo a otro *"thread context switch"*.

En un programa de ejecución tanto concurrente como paralela es necesario conocer el estado que guardan los hilos o los procesos. En el modelo de ejecución de multiproceso, un proceso puede estar en alguno de los seis estados durante su ciclo de vida: creación, ejecutable(listo para pasar al estado activo), activo(o en ejecución), dormido, detenido o terminado **[NORTON1997, pag. 9]**. En el modelo de ejecución multihilos se cuenta con cinco estados: ejecutable, detenido, activo, dormido y terminado **[LEWIS1996, pag. 48]**.

### 2.1.1 Ejecución Concurrente y Ejecución Paralela

Un programa de ejecución concurrente puede producirse con más de un proceso o con más de un hilo. Desde el punto de vista de los sistemas basados en procesos, un programa es concurrente si involucra varios procesos ejecutándose independientemente e interactuando entre sí para lograr un propósito en común. Por otro lado, un programa multihilos, es un programa concurrente si en lugar de intervenir varios procesos, es usado sólo uno con varios hilos. En el caso de los programas multihilos, la ejecución concurrente y paralela es realizada en equipos de memoria compartida.

Concurrencia significa que dos o más hilos (o procesos en el caso general) pueden estar al mismo tiempo en medio del código de ejecución; en el mismo código o en diferente **[LEWIS1996, pag. 88]**. Lo anterior quiere decir que todos los hilos han iniciado su ciclo de vida y pueden o no estar ejecutándose al mismo tiempo, pero aún no han alcanzado el estado terminado **(véase Figura 2.1.5)**. Un sistema operativo multitarea tiene la capacidad de mantener en ejecución más de un proceso; sin embargo sólo uno a la vez puede permanecer en un procesador.

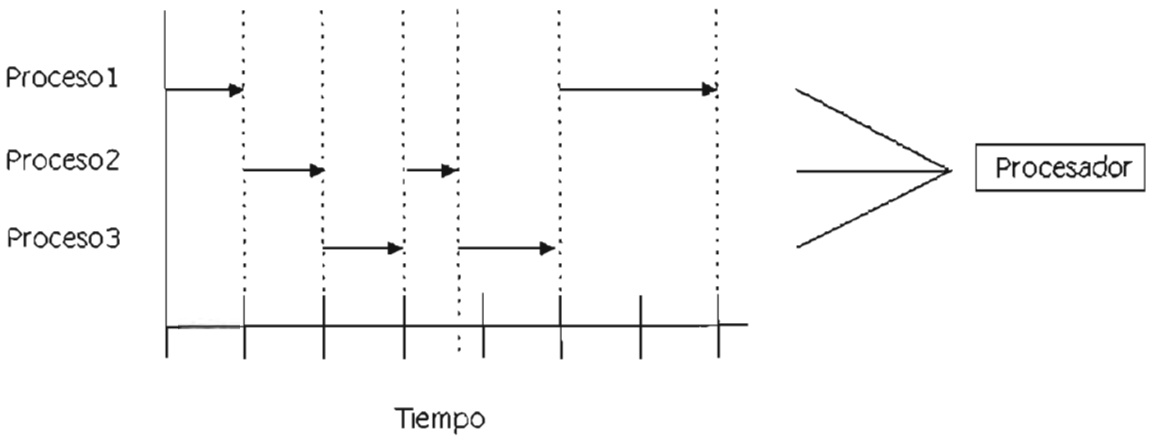



Figura 2.1.5 Tres procesos ejecutándose concurrentemente en un procesador

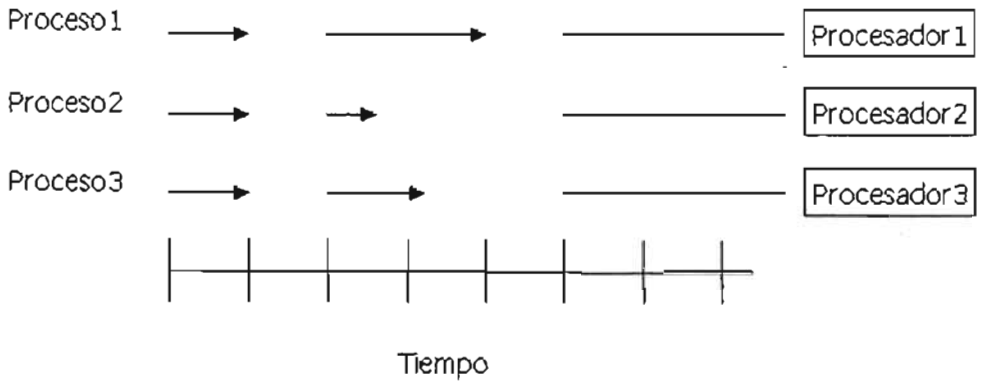



Figura 2.1.6 Tres procesos ejecutándose paralelamente en tres procesadores

Paralelismo significa que dos o más hilos son ejecutados en realidad al mismo tiempo en diferentes procesadores **[LEWIS1996, pag. 89]**, como muestra la **Figura 2.1.6**. Asimismo, si un equipo cuenta con varios procesadores, es posible ejecutar un programa multihilos en ambos modos de ejecución: paralelo y/o concurrente. Lo anterior aplica también para varios equipos que trabajen independientemente y se encuentren interconectados por redes de comunicaciones (lo que se conoce como procesamiento distribuido).

En el caso de sistemas basados en procesos, la mayoría de los sistemas operativos se basan en el modelo jerárquico de creación de procesos. En dicho modelo, dentro de cada proceso pueden crearse procesos hijos; los cuáles, al momento de ser creados, se conforman de una copia del proceso padre, un número distinto de *PID (Process Identificator)* y un identificador del proceso padre *PPID (Parent Process Identificator)*. Un proceso hijo existe sólo si su proceso padre existe también; sí éste último termina, todos los procesos hijos terminarán.

A diferencia de los procesos, los hilos no cuentan con una relacionan jerárquica padre-hijo; todos ellos tienen acceso a los recursos compartidos en el proceso de la misma forma; pueden terminar la ejecución de otros hilos y sincronizarse entre sí no importando su orden de creación y cual haya sido el hilo que los creara dentro del proceso.

Bajo el esquema anterior de hilos, dentro de todo proceso existe un hilo principal, cuya única diferencia con los demás es la de haber sido el primero en ser creado y ejecutado por el flujo normal del programa, y el primero en crear y lanzar otros hilos.

Los mecanismos de coordinación de hilos mencionados en el **apartado 2.2.3** son necesarios en la ejecución concurrente o paralela por procesos o por hilos en los sistemas operativos modernos.

En un principio, los sistemas operativos definían sus propias bibliotecas *APIs (Application Programming Interface)* diseñando sus propias funciones para la integración de aplicaciones multihilos. Lo anterior, tuvo como consecuencia la dificultad para transportar dichas aplicaciones para ser ejecutadas en múltiples plataformas de sistemas operativos. Un apoyo para evitar esta dificultad, lo ofrecen algunos estándares establecidos por la *IEEE( Institute of Electrical and Electronics Engineers)* surgiendo con ello el estándar *POSIX (Portable Operating System Interface)*. Sin embargo, las marcas de compiladores y los diferentes sistemas operativos siguen creando diferentes herramientas para la programación multihilos o la programación multiprocesos.

### 2.1.2 Características de la Ejecución Concurrente

La ejecución concurrente está caracterizada por ser no-determinista, sujeta a calendarización (o asignación de orden y tiempo) por parte del sistema operativo y requiere de mecanismos de coordinación entre procesos o hilos para evitar regiones inseguras.

- **No-determinismo.** En un programa de ejecución secuencial el orden de ejecución de las instrucciones corresponde a la lógica que guarda el programa permitiendo observar una total determinación de su flujo. En cambio en la ejecución concurrente, las instrucciones no son ejecutadas completamente en forma predeterminada, aunque se cuente con un orden dado; la ejecución final es no-determinista dentro de cada proceso.
- **Calendarización (Scheduling)**. El sistema operativo se encarga de colocar las instrucciones de los procesos o los hilos en estado ejecutable en una cola de espera y de esto depende el orden en el cual serán ejecutadas, además le es asignado un tiempo determinado de uso del procesador. Ésta característica está muy relacionada con la anterior.
- **Técnicas de Prueba**. Cuando en un programa concurrente intervienen varios procesos y varios procesadores las líneas de ejecución de las instrucciones se incrementan y se vuelven más complejas en la medida que aumentan los procesos y/o los procesadores, lo cual podría conducir a trayectorias o rutas de ejecución denominadas como inseguras, las cuáles podrían producir resultados incorrectos o inesperados.

Debido a las características antes mencionadas de la ejecución concurrente, es necesario contar con mecanismos para prevenir colisiones entre procesos cuando pretendan acceder simultáneamente la misma región de memoria o usar algún otro recurso del sistema, así como también procurar el orden correcto de ejecución previendo las posibilidades válidas de las que no lo son y además evitar en lo

posible que un proceso tenga retrasos por la espera de ejecución de otro. Por lo anterior, existen mecanismos de sincronización para evitar conducir la ejecución de un programa a regiones inseguras. Los mecanismos más comunes se basan en el establecimiento de secciones críticas y el bloqueo de variables.

### 2.1.3 Modelos de Concurrencia

Los modelos de concurrencia determinan el nivel de no-determinismo (lo cual se traduce como "determinismo observable") de acuerdo al tipo de problema planteado en un programa concurrente. Dichos modelos, determinan la forma en que los recursos compartidos por los procesos o hilos concurrentes son accedidos, principalmente la memoria. Existen varios modelos de concurrencia los más comunes son:

- Concurrencia declarativa
- Paso de Mensajes

#### 2.1.3.1 Concurrencia Declarativa

Los programas escritos en este modelo son de utilidad en problemas con un no-determinismo no observable. Los procesos generalmente se comunican entre sí mediante acceso directo a áreas de memoria compartida, por lo que, este modelo es aplicado ampliamente en equipos con arquitectura de memoria compartida o bien en equipos de memoria distribuida formados por nodos de varios procesadores con memoria compartida dentro del nodo; en dicho caso, el programa se pude ejecutar en un solo nodo.

La ejecución concurrente multihilos está considerada dentro del modelo de concurrencia declarativa.

**Operaciones con hilos**

Al crearse los hilos dentro de un proceso, éstos pueden escribir o leer datos de áreas memoria comunes, tener dependencia de datos o recursos entre ellos, y en algunos casos correr solamente en ciertos momentos y permanecer inactivos o en espera de otros. Para prevenir errores en tiempo de ejecución o lógicos, además de la creación y terminación de hilos existen otras operaciones sobre los hilos que dan como resultado los siguientes estados:

**Operaciones:**

Detener, continuar, despertar, dormir, activar, despachar, terminar.

**Estados:**

Ejecutable, detenido, activo, dormido, terminado. **[LEWIS1996, pag 48]**.

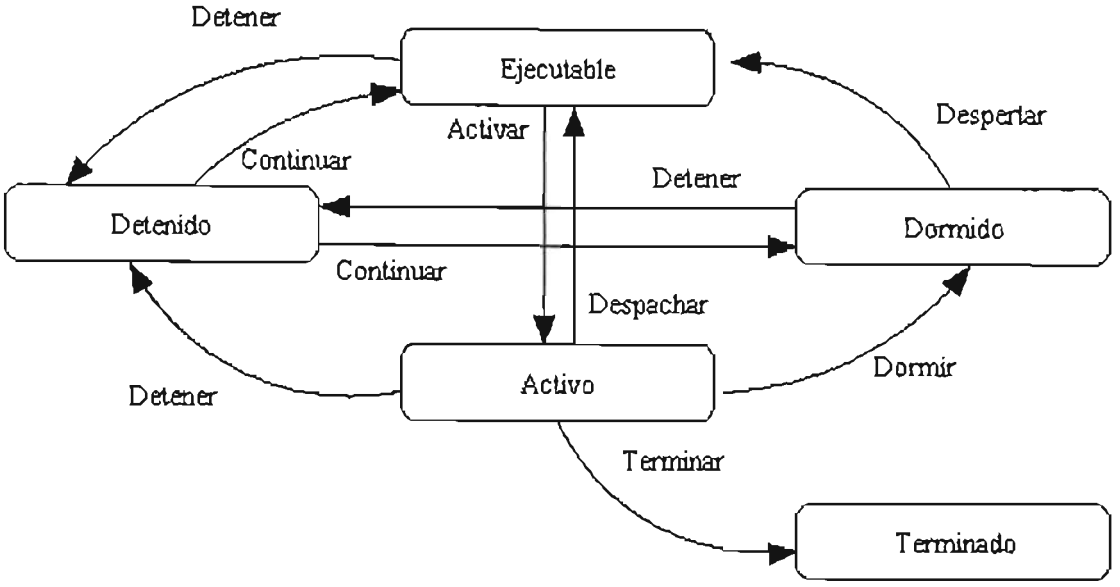



Figura 2.1.7 Operaciones y estados de hilos.

En la **Figura 2.1.7** se muestran las operaciones válidas para los hilos, además de la operación de creación del hilo; donde un hilo en estado *Ejecutable* puede recibir una instrucción *Activar o Detener* pasando al estado *Activo o Detenido* respectivamente. Desde el estado *Ejecutable*, un hilo no puede cambiar directamente al estado *Dormido*, tendrá que recibir primero la orden adecuada para pasar el estado *Detenido o Activo*, y desde cualquiera de éstos estados entonces sí podrá llegar al estado *Dormido*.

### 2.1.3.2 Paso de Mensajes

El modelo de paso de mensajes, es una extensión del modelo de concurrencia, agregando los conceptos de puerto y canal para reducir el no-determinismo en la ejecución de los programas. En el modelo de paso de mensajes, los procesos se comunican entre sí mediante mensajes, que consisten en accesos de lectura y escritura a los puertos.

Un puerto es un tipo de dato abstracto, definido en la implementación del paso de mensajes. Se realizan básicamente tres operaciones con respecto a los puertos:

- Creación del puerto
- Envío de datos al puerto
- Recepción de datos desde el puerto.

Con los canales se busca reducir aún más el no-determinismo presente en el manejo por puertos. Las operaciones de los canales son las mismas que las del modo normal de puertos, con la diferencia de los procesos emisor *E* y receptor *R* establecidos en la creación del puerto.

## 2.2 PROGRAMACIÓN MULTIHILOS EN BORLAND C++ BUILDER

Multihilos es la capacidad de un programa para ejecutar múltiples tareas (hilos) al mismo tiempo **[HOLLINGWORTH2001, pag. 194]**. En un programa multihilos de *Borland C++ Builder*, el hilo primario tiene a su cargo la creación de las ventanas hijo y el procesamiento de mensajes; mientras que, los hilos secundarios ejecutan operaciones como: lectura de archivos grandes, búsqueda de información, cálculos matemáticos, entre otros.

Cuando un programa está en ejecución, el sistema operativo sobre el cual esté trabajando, lo representa como un proceso de uno o más hilos que contienen datos, código y recursos de sistema necesarios para lograr un propósito determinado. A cada hilo le es encomendada la ejecución de una parte del programa y el sistema operativo le reserva un tiempo de CPU para obtener un resultado. Todos los hilos de un proceso comparten el mismo espacio de memoria y tienen alcance a las variables globales de ese proceso.

En *Borland C++ Builder* las propiedades y métodos de los objetos de la *VCL (Visual Component Library)* son regiones inseguras de hilo, lo cual significa que al acceder a las propiedades y métodos pudiera usarse memoria no protegida en otros hilos ejecutados simultáneamente causando conflictos y generando resultados incorrectos. Por lo anterior, existe un hilo principal *VCL* controlando exclusivamente la *VCL*: manipulando y procesando los mensajes recibidos por los controles de la propia *VCL*. Además para evitar dichos conflictos existen varios mecanismos para coordinar la ejecución de hilos, como se verá en el **apartado 2.2.3**.

Con *Borland C++ Builder* se pueden crear hilos de dos formas: usando llamadas a la *API* de *Windows* o usando el objeto *TThread*.

### 2.2.1 Creación de Hilos con la API de Windows

Es posible crear un hilo a partir de otro, llamando a la función *API* de *Windows CreateThread*. Algunos de los parámetros de dicha función son los atributos de seguridad usados para determinar si otros hilos pueden modificar el objeto; la creación de banderas para establecer si el hilo será ejecutado inmediatamente después de su creación o estará detenido y, la función del hilo para determinar el propósito del hilo. A continuación se presenta el prototipo de esta función:

```
HANDLE CreateThread
(       LPSECURITY_ATTRIBUTES lpThreadAttributes,// pointer to thread security
attributes
        DWORD dwStackSize,      // initial thread stack size, in bytes
        LPTHREAD_START_ROUTINE lpStartAddress,  // pointer to thread function
        LPVOID lpParameter,     // argument for new thread
        DWORD dwCreationFlags, // creation flags
        LPDWORD lpThreadId      // pointer to returned thread identifier
);
```

Otra forma de crear hilos es a través de la función *_beginthread*, la cuál es muy usada en aplicaciones multihilos porque necesita pocos parámetros.

### 2.2.2 Creación de Hilos con el Objeto TThread

En *Borland C++ Builder* se pueden crear también hilos de ejecución concurrente con el objeto preconstruido *TThread*. Cada instancia nueva de un descendiente *TThread* se convierte en un nuevo hilo de ejecución; múltiples instancias de una clase derivada de *TThread* logran una aplicación multihilos. A continuación es mostrada la definición de la clase abstracta de *TThread*:

```
class DELPHICLASS TThread;
class PASCALIMPLEMENTACTION TThread : public System::TObject
{
        typedef System::TObject inherited;

private:
        unsigned FHandle;
        unsigned FThreadID;
        bool FTerminated;
        bool FSuspended;
```

```
        bool FFreeOnTerminate;
        bool FFinished;
        int FReturnValue;
        TNotifyEvent FOnTerminate;
        TThreadMethod FMethod;
        System::TObjet* FSynchronizeException;
        void __fastcall SetPriority(TThreadPriority Value);
        void __fastcall SetSuspended(bool Value);

protected:
        virtual void __fastcall DoTerminate(void);
        virtual void __fastcall Execute(void) = 0;
        void __fastcall Synchronize(TThreadMethod Method);
        __property int ReturnValue = {read=FReturnValue,
                write=FReturnValue, nodefault};
        __property bool Terminated = {read=FTerminated, nodefault};

public:
        __fastcall TThread(bool CreateSuspended);
        __fastcall virtual ~TThread(void);
        void __fastcall Resume(void);
        void __fastcall Suspend(void);
        void __fastcall Terminate(void);
        unsigned __fastcall WaitFor(void);
        __property bool FreeOnTerminate = {read=FFreeOnTerminate,
                write= FFreeOnTerminate , nodefault};
        __property unsigned Handle = {read=FHandle, nodefault};
        __property TThreadPriority Priority = {read=GetPriority,
                write= SetPriority, nodefault};
        __property bool Suspended = {read=FSuspended,
                write=SetSuspended, nodefault};
        __property unsigned ThreadID = {read=FThreadID, nodefault};
        __property TNotifyEvent OnTerminate = {read=FOnTerminate,
                write= FOnTerminate };
};
```

Se puede crear un descendiente de *TThread* al seleccionar del menú principal *File*, luego *New* diálogo, después *Thread Object* en el contenedor de objetos y cuando *Borland C++ Builder* lo solicita proporcionar el nombre de la clase para el nuevo descendiente. Hecho lo anterior, es creado automáticamente un nuevo archivo de código conteniendo el nuevo objeto con el nombre dado a la clase.

El nuevo archivo generado en la creación del nuevo hilo, muestra en su interior el método *Execute* conteniendo el código por ejecutarse una vez que el hilo sea puesto a funcionar. Dicho archivo también cuenta con el método constructor, donde un parámetro es una bandera para la ejecución inmediata o la ejecución suspendida del hilo al momento de ser creado. Si la bandera anterior es *true,* debe llamarse primero al método *Resume* para ejecutar el hilo. En la **Tabla 2.2.1** y la **Tabla 2.2.2** se muestran las propiedades y métodos más comunes de la clase *TThread.* **[HOLLINGWORTH2001 pag. 201-202].**

| **Propiedad** | **Descripción** |
|---|---|
| *FreeOnTerminate* | Determina si el objeto *TThread* será destruido automáticamente cuando el hilo termine su ejecución. Si su valor es *true* el objeto es destruido automáticamente y si es *false* por código de la aplicación. |
| *Priority* | Especifica una prioridad alta o baja de calendarización *(scheduling)* del hilo en relación con los otros hilos del proceso, cuando así es requerido. |
| *ReturnValue* | Determina el valor entregado a otros hilos cuando el objeto del hilo actual finaliza. Se emplea para indicar el éxito o fallo, el resultado o la salida numérica hacia la aplicación o a otros hilos. El método *WaitFor* regresa el valor guardado en ésta propiedad. |
| *Suspended* | Indica si el hilo está suspendido o no. |
| *Terminated* | Determina si el hilo está por terminar tan pronto como le sea posible al sistema operativo. |
| *ThreadID* | Determina el identificador del hilo. |

Tabla 2.2.1 Propiedades más comunes de la clase *TThread.*

| **Método** | **Descripción** |
|---|---|
| *DoTerminate()* | Llama al manejador de eventos *OnTerminate* sin terminar el hilo. |
| *Execute()* | Contiene el código por ejecutarse cuando el hilo es puesto a funcionar. Dentro de este método no debe |

| | |
|---|---|
| | hacerse uso de propiedades y métodos de otro objetos directamente, debe disponerse este uso en rutinas separadas llamándolas como parámetro del método *Synchronize*. |
| *Resume()* | Reinicia un hilo suspendido. |
| *Suspend()* | Detiene un hilo en ejecución. |
| *Synchronize()* | Ejecuta una llamada del método dentro del hilo primario *VCL*. Se emplea para evita conflictos entre hilos. La ejecución del hilo es suspendida mientras el método es ejecutado dentro del hilo primario *VCL*. |
| *Terminate()* | Indica al hilo el termino de su ejecución, modifica la propiedad *Terminated* a *true*. Es usado para provocar el fin anticipado de la ejecución de un hilo. |
| *WaitFor()* | Espera el termino de ejecución del hilo y regresa el valor guardado en la propiedad *ReturnValue* |

Tabla 2.2.2 Métodos más comunes de la clase *TThread*.

La propiedad *FreeOnTerminate* con su valor igual a *true* indica que el objeto hilo será destruido tan pronto termine su ejecución; en cambio, si su valor es *false* entonces dicho objeto será destruido explícitamente a través del código de la aplicación. Para asegurar que la memoria ocupada por el objeto hilo ha sido liberada al terminar la ejecución, debe insertarse lo siguiente en el método *Execute*:

```
FreeOnTerminate = true;
```

En ocasiones es necesario terminar la ejecución de un hilo por código, para lo cual es empleado el método *Terminate*. El método anterior le indica al hilo que finalice su ejecución estableciendo la propiedad *Terminated* igual a *true*.

### 2.2.3 Coordinación de Hilos

En la ejecución multihilos se debe evitar que dos o más hilos intenten usar un objeto al mismo tiempo, sobretodo cuando se trata de escribir nuevos valores.

Existen varios mecanismos para coordinar la ejecución de hilos, entre ellos tenemos los siguientes:

- **Evitar accesos simultáneos**
- **Espera de otros hilos**
- **Usar el hilo principal *VCL***

### 2.2.3.1 Evitar Accesos Simultáneos

Para evitar la colisión con otros hilos al intentar acceder a objetos o variables globales, se necesita bloquear la ejecución de otros hilos hasta que el código del hilo haya terminado una operación específica, teniendo el cuidado de no bloquear innecesariamente a los otros hilos. La *VCL* proporciona tres mecanismos para prevenir que otros hilos accedan la misma memoria simultáneamente.

- **Bloqueo de objetos o variables**. Consiste en bloquear un objeto mientras un hilo hace uso de él, negando el acceso a otro hilo hasta que el objeto sea desbloquedo. Algunos objetos definen métodos preconstruidos para bloqueo y desbloqueo, por ejemplo: el objeto *TCanvas* y descendientes cuenta con los métodos *Lock* y *UnLock* y, el objeto *TThreadList* con los métodos *LockList* y *UnLockList*.
- **Secciones Críticas**. Cuando los objetos no cuentan con métodos preconstruidos de bloqueo y desbloqueo, es posible usar la técnica de secciones críticas. Las secciones críticas trabajan como puentes para permitir la entrada a un solo hilo a la vez, dichas secciones son áreas de memoria global que se pretende proteger. Para usar éste mecanismo se hace una instancia global a *TCriticalSection*, el cual consta del método *Acquire* (para bloquear la sección e impedir que el resto de los hilos la ejecuten) y *Release* (para desbloquear la sección). Éste mecanismo funciona bien cuando todos los hilos accesan a la memoria global y para cada hilo debe existir una llamada *Acquire* para bloqueo y otra llamada *Release* para desbloqueo.
- **Sincronización de lectura y escritura.** Si necesitamos proteger un área de memoria global de accesos simultáneos y observamos que la lectura es muy frecuente y la escritura es escasa; entonces podríamos emplear este mecanismo. Para utilizar este mecanismo se crea una instancia global al objeto *TMultiReadExclusiveWriteSynchronizer* asociando la memoria global por proteger, cuando un hilo necesite leer esa memoria llamará al método *BeginRead* asegurando que ningún otro hilo escriba sobre dicha memoria, al terminar de leer el hilo se debe llamar al método *EndRead*. Cuando un hilo requiera escribir sobre la memoria mencionada, llamará al método *BeginWrite* para evitar que cualquier otro hilo lea o escriba en ese momento sobre la memoria, al terminar de escribir sobre la memoria protegida se debe llamar al método *EndWrite*, de este modo si algún hilo está esperando leer sobre la

misma memoria protegida pueda empezar a hacerlo. Al igual que el mecanismo de secciones críticas, éste mecanismo funciona bien cuando todos los hilos accesan a la memoria global protegida y para cada hilo debe existir un par de llamadas de inicio y fin para leer y escribir.

### 2.2.3.2 Espera de Otros Hilos

Si un hilo debe esperar a que otro hilo termine alguna operación, se puede suspender temporalmente su ejecución en dos casos:

- **Esperar que otro hilo termine completamente su ejecución**. Si un hilo requiere para continuar que otro hilo termine su ejecución, es usado el método *WaitFor* en el segundo hilo. El método *WaitFor* regresa su valor hasta que el otro hilo ha terminado de forma natural por su método *Execute* o de forma excepcional, permitiendo entonces que el primer hilo continué su ejecución. Este mecanismo es usado cuando en un hilo existe dependencia de tareas o resultados finales realizados por otro.
- **Esperar que una operación sea terminada**. Algunas ocasiones, un hilo para continuar necesita que otro hilo realice una operación en particular; sin que ello implique el termino completo de su ejecución. Para realizar lo anterior, se usa el objeto *TEvent*, creado con un alcance global y cuando un hilo completa la operación esperada es prendida una señal de termino a través del método *SetEvent*, ante la señal el hilo en espera continua su ejecución. La seña es apagada con el método *ResetEvent*.

### 2.2.3.3 Usar el Hilo Principal VCL.

Al usar objetos de la jerarquía *VCL*, sus propiedades y métodos ocupan regiones de memoria inseguras para accesos simultáneos multihilos. Por lo anterior, esta reservado el hilo principal *VCL* para acceder a los objetos *VCL* y evitar conflictos o colapsos, éste hilo manipula todas las mensajes recibidos por los componentes de la aplicación.

Para usar el hilo principal *VCL*, se debe crear una rutina separada que ejecute las acciones requeridas y en el método *Execute* del hilo llamarla como argumento del método *Synchronize*. El método *Synchronize* espera a que el hilo *VCL* lo ingrese a su ciclo de mensajes y así ejecute la rutina indicada. Algunos objetos ocupan regiones seguras de hilos que no requieren utilizar el hilo *VCL* y son: los componentes de acceso a base de datos, objetos gráficos *(TFont, TPen, TBrush, TBitmap, TMetafile y TIcon)*, objeto hilo para listas *TThreadList* y el objeto *TCanvas* puede ser usado en este sentido bloqueándolo (con los métodos *Lock* y *UnLock*).

## 2.3 PROGRAMACIÓN GRÁFICA EN BORLAND C++ BUILDER

Para integrar figuras sencillas en un formulario de *Borland C++ Builder* se emplean los componentes *Shape*, *Image* y *PaintBox*.

El componente *Shape* sirve para crear círculos, elipses, cuadrados y rectángulos. Para ello, debe colocarse un componente en el formulario y modificar las propiedades *Shape* para seleccionar el tipo de figura, *Brush* para definir el color y el estilo del fondo y, *Pen* para determinar el color y estilo del contorno así como el modo de presentación de la figura con respecto al formulario.

*Image* es el componente utililizado para presentar un mapa de bits. Este componente puede seleccionar una imagen en tiempo de diseño o cargarla en tiempo de ejecución. La propiedad *Strech* de este componente permite ajustar la imagen de mapa de bits al tamaño del componente, *Center* establece si la imagen aparecerá o no centrada en el componente y *Autosize* hace que el componente modifique su tamaño al de la imagen.

Por otro lado, el componente *PaintBox* permite definir un área de dibujo estableciendo límites de un espacio del formulario, esta área es considerada un lienzo de dibujo. La propiedad *Canvas* es la más importante de este componente, siendo una instancia de la clase *TCanvas*, con esta clase son realizadas la mayoría de la funciones de dibujo de una aplicación *C++ Builder*.

### 2.3.1 Contexto de Dispositivo y la Clase TCanvas

*Windows* usa el término Contexto de Dispositivo para describir un área donde se puede dibujar. Un contexto de dispositivo es un lienzo donde los programas de *Windows* pueden dibujar texto, líneas, mapas de bits, rectángulos, elipses, etc. La manipulación de contextos de dispositivos es realizada normalmente a través de funciones de la *API* de *Windows*. El número de contextos de dispositivos en *Windows* es limitado por lo que se debe tener cuidado de liberar la memoria empleada por ellos al terminar de usarlos; de no hacerlo, pueden ocurrir fallas en el programa o en el propio *Windows*.

El trabajo con contextos de dispositivo también está disponible a través de la *VCL* de forma más sencilla. Para facilitar el uso de contextos de dispositivos, la *VCL* encapsula los contextos de dispositivo de *Windows* en la clase *TCanvas*, evitando al programador conocer el detalle de éstos. La propiedad *Canvas* representa el lienzo de cada formulario de *C++ Builder*, por lo cual, si se requiere dibujar directamente sobre el formulario se debe acceder a la propiedad *Canvas*. La *VCL*

se encarga de obtener el contexto de dispositivo, seleccionar los objetos (plumas, pinceles o fuentes) apropiados en él y liberarlo cuando ya no se requiera. Un contexto de dispositivo se puede utilizar para dibujar en superficies como:

- En el marco de una ventana
- En el escritorio
- En la memoria
- En la impresora

Haciendo uso de la *VCL* es posible dibujar un rectángulo en un formulario, como sigue:

```
Canvas->Brush->Color = clBlue;
Canvas->Pen->Width = 10;
Canvas->Pen->Color = clYellow;
Canvas->Rectangle(300,300, 400, 400);
```

El código anterior mostrará en el formulario un rectángulo azul con un contorno amarillo de un grosor de 10 pixeles. Aquí la *VCL* a través de la clase *TCanvas* se encarga de liberar recursos del contexto de dispositivo adecuado, según se requiera y no es necesario preocuparse por realizarlo explícitamente.

La clase *TCanvas* cuenta con muchas propiedades y métodos en **la Tabla 2.3.1** y la **Tabla 2.3.2** se encuentran algunas.

| **Propiedad** | **Descripción** |
|---|---|
| *Font* | Especifica las características del tipo de letra al escribir texto sobre una imagen. Configura las propiedades del objeto *TFont* para determinar tipo, color, tamaño y estilo de letra. |
| *Brush* | El color del pincel o patrón utilizado para rellenar las figuras |
| *ClipRect* | El rectángulo de recorte actual para el lienzo. Todo dibujo estará limitado a este rectángulo. |
| *Pen* | Determina el estilo y color de las líneas que se dibujan sobre el lienzo. |
| *PenPos* | La posición actual de dibujo, en coordenadas *x,y.* |

Tabla 2.3.1 Algunas propiedades de la clase *TCanvas*

| Método | Descripción |
|---|---|
| *CopyRect* | Copia al lienzo parte de una imagen |
| *Draw* | Copia al lienzo una imagen desde la memoria. La imagen puede ser un mapa de bits, un icono o un archivo *metafile*. |
| *Ellipse* | Dibuja una elipse usando la pluma y el pincel actuales. |
| *LineTo* | Dibuja una línea desde la posición actual del dibujo hasta la posición indicada en los parámetrso *x,y*. |
| *MoveTo* | Establece la posición actual de dibujo. |
| *Rectangle* | Dibuja sobre el lienzo un rectángulo de acuerdo a la pluma y pincel actuales. |
| *StrechDraw* | Copia un grafico desde la memoria hasta lienzo. El tamaño del grafico se amplía o reduce de acuerdo con el tamaño del rectángulo de destino. El gráfico puede ser un mapa de bits, un Icono o un archivo *metafile* |
| *TextHelght, TextWidth* | Regresa el valor del alto y ancho del tipo de letra actual, respectivamente. |
| *TextOut* | Emplea la fuente actual para dibujar texto en la posición especificada del lienzo. |
| *TextRect* | Dibuja texto en un rectángulo de recorte. |

Tabla 2.3.2 Algunos métodos de la clase *TCanvas*

### 2.3.2 Objetos GDI

La *GDI (Graphic Device Interface)* de *Windows* cuenta con objetos que definen el funcionamiento de un contexto de dispositivo. Dichos objetos son las plumas, los pinceles, las fuentes, las paletas, los mapas de bits y las regiones.

#### 2.3.2.1 Plumas, Pinceles y Fuentes

Una pluma es el objeto utilizado para dibujar líneas, puede ser el borde de una figura o una simple recta. Se logra el acceso a este objeto a través de la propiedad *Pen* de la clase *TCanvas*. La propiedad *Pen* es una instancia de la clase *TPen*. Entre

las funciones de la propiedad *Pen* se encuentran la especificación del color, el grosor en pixeles y el estilo sólido, de puntos, transparente, etc.

Un pincel es el objeto que permite establecer el relleno de una figura gráfica. Cuando a una figura le es aplicado el valor actual de un pincel puede ser que la figura sea rellenada con un color sólido, un patrón o un mapa de bits. El acceso al objeto pincel se obtiene con la propiedad *Brush* de la clase *TCanvas*, siendo esta propiedad una instancia de la clase *TBrush*. Entre las funciones de la propiedad *Brush* podemos mencionar la especificación del color ya sea sólido o el color de las líneas de un patrón elegido; la determinación de mapa de bits como fondo del pincel y el estilo sólido, transparente o de un patrón.

Con los pinceles se puede presentar un fondo de mapa de bits, para conseguir esto es necesario crear un objeto *TBitmap* descendiente de la clase *Graphics* y asignarlo a la propiedad *Bitmap* del pincel, luego debe indicarse a ésta propiedad el mapa de bits seleccionado (usualmente indicando el archivo conteniendo el mapa de bits de 8x8 pixeles). Una vez dibujada la figura con el fondo de mapa de bits, debe eliminarse el objeto asignado a la propiedad *Brush* debido a que la memoria asignada no es liberada automáticamente en este caso. El siguiente código muestra la forma de usar un mapa de bits como pincel:

```
/* Mapa de bits como fondo */
Canvas->Brush->Bitmap = new Graphics::TBitmap;
Canvas->Brush->Bitmap->LoadFromFile("C:||WINDOWS||Greca.bmp");
Canvas->Ellipse(170,170,270,270);
delete Canvas->Brush->Bitmap;
```

El objeto fuente sirve para dibujar texto en un espacio especificado del lienzo, determinando un tipo de fuente, su tamaño, su estilo , su color y la posición donde es colocado.

### 2.3.2.2 Mapas de Bits, Paletas y Regiones de Recorte

Una paleta se encarga de controlar el color de un mapa bits, para ello cuenta con 256 colores que pueden ser usados para mostrar el gráfico en pantalla.

La clase *TBitmap* encapsula un objeto de mapa de bits así como su paleta. El objeto *TBitmap* contiene una imagen de un gráfico como mapa de bits y se encarga del manejo automático de la paleta cuando la imagen es dibujada.

Las regiones de recorte son áreas de la pantalla que pueden utilizarse para indicar las partes del lienzo donde es posible dibujar, sirven para establecer límites de

dibujo dentro del lienzo. A través de la propiedad *ClipRect* es posible establecer una región de dibujo para el lienzo, de modo tal que; si se dibuja fuera de las coordenadas de esa región sólo será mostrada la parte de la imagen que ocupe la región de recorte.

### 2.3.3 Dibujo de Texto

El dibujo de texto puede realizarse con el método *TextOut* de la clase *TCanvas* o bien con la función *DrawText* de la *API* de *Windows*. El método *TextOut* es utilizado para dibujar texto donde no se requiere de mucha precisión y presentación para ello; en cambio con *DrawText* es posible tener mayor control sobre la forma de presentar un dibujo de texto como por ejemplo: el texto puede aparecer centrado sobre el lienzo.

El método *TextOut* dibuja el texto indicado sobre el lienzo, proporcionando el valor de las coordenadas *x,y* donde empezará el dibujo del texto. El tipo de fuente dibujada y su apariencia estarán de acuerdo a la fuente, color y estilo actuales.

Para dibujar texto en un área de recorte es empleado el método *TextRect*, con el cual el texto está restringido al limite establecido por el método, de no ser suficiente el espacio rectangular el texto es truncado. Los prototipos de los metodos *TextOut* y *TextRect* son los siguientes:

```
void __fastcall TextOut(int X, int Y, const AnsiString Text);
void __fastcall TextRect(const Windows::TRect &Rect, int X, int Y,
                         const AnsiString Text);
```

El fondo de un dibujo de texto puede ser establecido como un color sólido, de mapa de bits o transparente, lo anterior; de acuerdo con los valores dados en el pincel actual (el fondo de color sólido blanco es el predeterminado).

### 2.3.4 Dibujo de Mapa de Bits

El método más usado para dibujar mapas de bits es *Draw*. Dicho método dibuja en el lienzo el gráfico especificado en la posición correspondiente de las coordenadas *x,y* suministradas. Si el gráfico a dibujar es un mapa de bits debe crearse un objeto de la clase *TBitmap*.

Con el método *StrechDraw* puede colocarse un gráfico sobre el lienzo cambiando el tamaño del gráfico acorde al tamaño del rectángulo especificado. Se debe tener

cuidado de determinar el tamaño del rectángulo que conserve la mejor proporción del grafico original. Al igual que el método *Draw*, si el gráfico a dibujar es un mapa de bits debe crearse un objeto de la clase *TBitmap*

Otro método para manipular imágenes de mapa de bits es CopyRect, el cual sirve para transferir una parte de la Imagen desde un lienzo a otro. Este método es de utilidad cuando se desea mostrar solo una parte de la imagen en una parte determinada de la pantalla. Enseguida se encuentran los prototipos de los métodos mencionados para el dibujo de mapas de bits:

```
void __fastcall Draw(int X, Int Y, TGraphic* Graphic);
void __fastcall StretchDraw(const Windows::TRect &Rect,
                            TGraphic* Graphic);
void __fastcall CopyRect(const Windows::TRect &Dest,
                         TCanvas* Canvas, const Windows::TRect &Source);
```

### 2.3.5 Mapas de Bits Fuera de Pantalla

Los mapas de bits fuera de pantalla, tamblén son conocidos como mapas de bits de memoria. Este tipo de Imágenes son creadas en memoria y después son mostradas en la pantalla con el método *Draw* (ya antes mencionado). Esta forma de trabajar las imágenes de mapa de bits ayuda a que el despliegue sea más rápido; puesto que es mostrado hasta que la imagen esta lista y completa. Los mapas de bits fuera de pantalla son también útiles en los programas de dibujo complejos. El proceso que siguen los mapas de bits fuera de pantalla es el siguiente:

1. Crear el mapa de bits de memoria
2. Dibujar en el mapa de bits de memoria
3. Copiar el mapa de bits de memoria a la pantalla

La creación del mapa de bits es realizada cuando se crea el objeto, por ejemplo cuando es creado el objeto de la clase *TBitmap.* Para dibujar en el mapa de bits se puede partir de la carga de un archivo ya definido o bien; se pueden especificar su tamaño y luego dibujar en él. Finalmente, una vez que el mapa de bits está terminado se llama al método *Draw*, el cuál copia el mapa y lo muestra en el lienzo. Para guardar el mapa de bits de memoria es necesario emplear el método *SaveFile*, el cual copia la imagen en un archivo con el nombre indicado en su parámetro. El siguiente código muestra un ejemplo de creación de mapa de bits de memoria y la forma de guardarlo en un archivo:

```
Graphics::TBitmap* MapaBitsMem = new Graphics::TBitmap();
  MapaBitsMem->Width = 300;
  MapaBitsMem->Height = 300;
  for (int i=0; i<20; i++)
  {
    int x = random(400);
    int y = random(400);
    int l = random(100) + 50;
    int a = random(100) + 50;
    int rojo = random(255);
    int verde = random(255);
    int azul = random(255);
    MapaBitsMem->Canvas->Brush->Color = RGB(rojo, verde, azul);
    MapaBitsMem->Canvas->Rectangle(x,y,l,a);
  }
  MapaBitsMem->Canvas->Draw(0,0,MapaBitsMem);
  MapaBitsMem->SaveToFile("Rectangulos.bmp");
  delete MapaBitsMem;
```

El uso del paquete computacional *CVODE* para resolver sistemas de EDO de tipo *stiff* (rígido), no proporciona herramientas para graficar los resultados en tiempo de ejecución. Para resolver el problema anterior, en esta tesis se desarrolló un sistema de cómputo integral reuniendo un *analizador sintáctico (parser en inglés)* con *CVODE*, y junto con las herramientas de programación visual y de hilos proporcionadas por *Borland C++ Builder* son efectuados los cálculos numéricos y la graficación de los resultados; facilitando notablemente la interacción entre el usuario y el programa.

Por lo anterior, el propósito de emplear herramientas de programación visual y de hilos, pretende facilitar la interacción entre el usuario y el programa, ofreciendo en un mismo sistema de cómputo la implementación de un *analizador sintáctico*, el uso de *CVODE* y la graficación de resultados. Con la programación visual, se obtiene por un lado; la interfaz del sistema de cómputo integral, permitiendo al usuario introducir el sistema de EDO para efectuar los cálculos numéricos a través del *analizador sintáctico* y de *CVODE*. Por otro lado; las herramientas de hilos son combinadas con la programación visual para graficar los resultados, donde la graficación de resultados es ejecutada al mismo tiempo en un segundo hilo dentro del mismo proceso.

La interfaz del sistema de cómputo integral consta de dos partes fundamentales. La primer parte, es aquella donde se han utilizado herramientas visuales de uso común de *Borland C++ Builder* como son: menús, etiquetas, cajas de texto, editor

de texto memo, listas desplegables, entre otras. La segunda parte, es aquella donde se grafican los resultados de la solución del sistema de EDO, haciendo uso de las herramientas multihilos y de graficación.

Las propiedades y métodos empleadas en el sistema de cómputo integral de esta tesis; con respecto al manejo multihilos y graficación, son mostradas a continuación en la **Tabla 2.3.3**. En el **Capítulo 4** serán nuevamente planteadas con mayor detalle.

| | **Propiedades** | **Métodos** |
|---|---|---|
| **MULTIHILOS** | | |
| | *FreeOnTerminate* | *Execute* |
| | *Terminated* | *Synchronize* |
| **GRAFICACIÓN** | | |
| | *Font* | *Draw* |
| | *Brush* | *LineTo* |
| | *Pen* | *MoveTo* |
| | | *Rectangle* |
| | | *TextHeight* |
| | | *TextWidht* |
| | | *TextOut* |

Tabla 2.3.3 Propiedades y Métodos de *Borland C++ Builder* empleadas en el sistema de cómputo integral

# CAPÍTULO 3. CVODE

*CVODE* es un paquete de cómputo que sirve para resolver *Problemas de Valores Iniciales* (*PVI*), en sistemas rígidos y no rígidos de *Ecuaciones Diferenciales Ordinarias* (*EDO*); anteriormente codificado en *lenguaje C* y basado en otros dos paquetes antecesores escritos en lenguaje *Fortran*, *VODE* **[BROWN1989]** y *VODPK* **[BYRNE1992]**, creados por el *LLNL*[1]. El paquete *CVODE* utiliza el método de diferenciación en retroceso o *BDF* (por sus siglas en inglés *Backward Differentiation Formula*) y el *Adams-Moulton* para resolver sistemas *EDO* rígidos y no rígidos.

El propósito principal de *CVODE* es encontrar la solución a sistemas *EDO* rígidos, para lo cual dispone de diferentes resolvedores lineales: los de método directo (nombrados *dense*, *band* y *aproximación de la diagonal de la matriz Jacobiana*) y el de método iterativo *Krylov* precondicionado llamado *SPGMR* (por sus siglas en inglés *Scaled Preconditioned GMRES –Generalized Minimal RESidual*). La organización de *CVODE* es modular, está caracterizada por contar con un módulo integrador principal independiente de los resolvedores y cuenta con un módulo adicional para realizar operaciones con vectores.

En la actualidad, existe en desarrollo un conjunto de elementos para procesamiento en paralelo llamado *PVODE*. Asimismo, existe la interfaz para poder usar *CVODE* en programas escritos en C++. El paquete *CVODE* y la guía del usuario pueden encontrarse en la página web de dominio público *http://www.netlib.org.*

## 3.1. ANTECEDENTES HISTÓRICOS DE CVODE

Muchos paquetes escritos en *Fortran* son usados para resolver *PVI* en sistemas *EDO*. Dos paquetes creados por el en los últimos años son *VODE* y *VODPK* escritos en lenguaje *Fortran*. El paquete *VODE* es un resolvedor de propósito general que incluye métodos para resolver sistemas rígidos y no rígidos, emplea resolvedores directos para los sistemas rígidos para una matriz de banda y para una matriz densa (nombrados respectivamente resolvedor *band* y resolvedor *dense*). Por otra

[1] Lawrence Livermore National Laboratory de la Universidad de California en Estados Unidos de Norteamérica

parte, el paquete *VODPK* es una variante de *VODE* que implementa el método iterativo *Krylov* precondicionado para resolver sistemas lineales; es un resolvedor para sistemas rígidos de gran tamaño; es muy potente porque combina métodos reconocidos en la integración de sistemas rígidos, utiliza la iteración no lineal y la iteración (lineal) *Krylov* para resolver problemas específicos preponderantemente de origen rígido, a través de una matriz de condiciones previas proporcionada por el usuario.

Para la creación de *CVODE* se tradujeron los algoritmos de *VODE* y *VODPK* de *Fortran* a C y adicionalmente se optimizó el algoritmo por completo. La elección del *lenguaje C* estuvo motivada primero, por la gran popularidad que ocupa el lenguaje en el área de cómputo científico y segundo, porque los punteros de las estructuras y las características del manejo dinámico de memoria en C son muy útiles en software de esta complejidad.

## 3.2. REPRESENTACIÓN MATEMÁTICA

Un sistema *EDO* en *PVI* se puede representar como sigue:

$$\dot{y} = f(t, y), \quad y(t_0) = y_0, \quad y \in R^N \tag{1}$$

donde $\dot{y}$ denota la derivada $\frac{dy}{dt}$.

En *CVODE* están disponibles dos tipos de métodos lineales multipaso y son:

- Métodos de orden variable y paso variable de *Adams* (*Adams-Moulton*),y
- Métodos *BDF* (por sus siglas en inglés *Backward Differentiation Formula*) de orden variable, paso variable y coeficiente principal fijo.

La solución numérica de (1) se obtiene por valores discretos $y_n$ en los puntos $t_n$. Los valores calculados de $y_n$ obedecen a la fórmula lineal multipaso

$$\sum_{i=0}^{k1} \alpha_{n,i} y_{n-i} + h_n \sum_{i=0}^{k2} \beta_{n,i} \dot{y}_{n-i} = 0 \tag{2}$$

donde $y_n$ son aproximaciones a $y(t_n)$, $h_n = t_n - t_{n-1}$ es el tamaño del paso y $\alpha_{n,0} = -1$. Ambos métodos lineales multipaso pueden ser escritos en la forma (2). Para el caso de problemas no rígidos, la fórmula del método de *Adams-Moulton* está caracterizado porque $k_1 = 1$, $k_2 = q$, y el orden de $q$ varía entre 1 y 12. En el caso de problemas rígidos, la fórmula *BDF* considera a $k_1 = q$ y $k_2 = 0$, mientras que el orden de $q$ varía entre 1 y 5. En cualquiera de los dos casos anteriores, debe resolverse una aproximación del siguiente sistema no lineal en cada paso:

$$G(y_n) \equiv y_n - h_n \beta_{n,0} f(t_n, y_n) - a_n = 0, \quad \text{donde} \quad a_n \equiv \sum_{i>0} (\alpha_{n,i} y_{n-i} + h_n \beta_{n,i} \dot{y}_{n-i}) \qquad (3)$$

Para los problemas no rígidos, la solución de (3) se logra con una iteración funcional simple (o iteración de punto fijo). Mientras que, para los problemas rígidos, la solución se obtiene mediante una la iteración de *Newton* modificada involucrando a su vez la solución de sistemas lineales de la forma

$$M\left[y_{n(m+1)} - y_{n(m)}\right] = -G(y_{n(m)}) \qquad (4)$$

donde *M* es una aproximación a la matriz de *Newton* $I - h_n \beta_{n,0} J$ y $J = \partial f / \partial y$ es la matriz *Jacobiana* del sistema *EDO*. Los sistemas lineales que va originando la iteración de *Newton* son resueltos por un método directo o un método iterativo. Los resolvedores llamados *dense, band* y *aproximación de la diagonal de la matriz Jacobiana* emplean métodos directos en *CVODE;* el primer resolvedor trata con matrices densas y el segundo con matrices de banda. El método iterativo implementado en *CVODE* es conocido como *GMRES* (*por sus siglas en inglés Generalized Minimal RESidual*) incluyendo escala y precondición, el resolvedor es llamado *SPGMR* (por sus siglas en inglés *Scaled Preconditioned Generalized Minimal Residual*)

En la **Figura 3.2.1** se muestra la relación de los *PVI* no rígidos y rígidos en relación con el método empleado en la solución de los sistemas *EDO* de acuerdo a los resolvedores disponibles en *CVODE*.

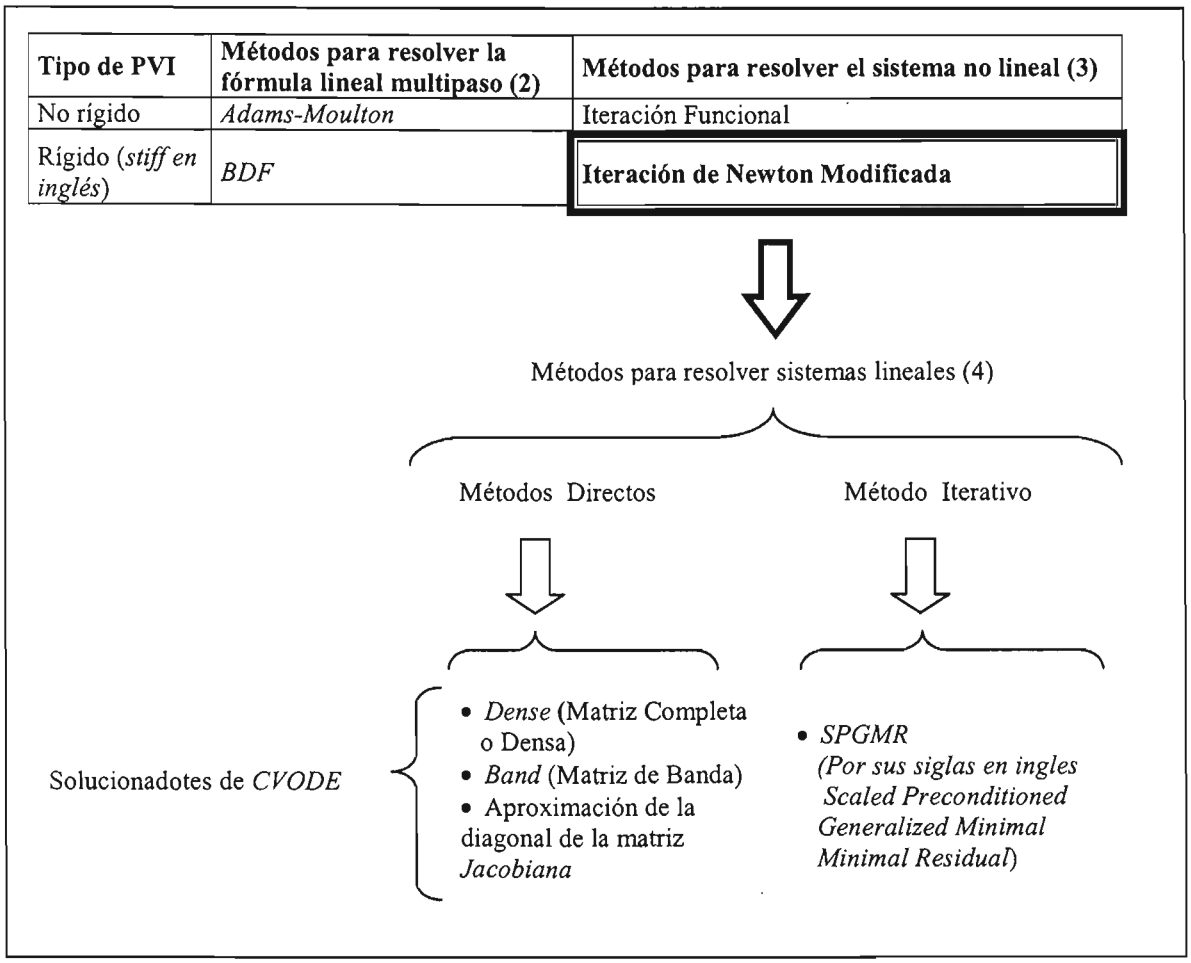



Figura 3.2.1 Relación de los resolvedores de *CVODE* de acuerdo al método implementado

## 3.3. ORGANIZACIÓN DE CVODE

Los paquetes *VODE* y *VODPK* tienen algunas desventajas, algunas de ellas heredadas del lenguaje *Fortran*, las cuales motivaron la creación de *CVODE*. Primeramente, se cuenta con dos paquetes distintos, uno (*VODE*) conteniendo los métodos directos para sistemas lineales y otro (*VODPK*) incluyendo los métodos de *Krylov*. Además, existen dos versiones de cada paquete, una versión para la precisión simple y otra para la precisión doble. Las diferencias entre los cuatro paquetes tienen que ver con las declaraciones y los resolvedores lineales. La subrutina principal de cada paquete emplea lógica específica para los resolvedores lineales además de la lógica para la integración. En *VODPK*, la implementación del método precondicionado *GMRES* involucra una mezcla de lógica para el método genérico *GMRES* y una lógica específica en el contexto de la iteración de *Newton* al momento de la integración.

En contraste, el diseño de *CVODE* fue planeado para evitar las desventajas mencionadas anteriormente. Los módulos del paquete *CVODE* se muestran en la **Figura 3.4.1 Diagrama por bloques del paquete *CVODE***. El módulo integrador principal, también llamado *CVode*, trabaja estrictamente con cuestiones de integración y es completamente independiente del método usado para resolver los sistemas lineales (4).

Un conjunto de módulos resolvedores de sistemas lineales es parte del paquete, el módulo integrador principal se conecta con alguno de éstos módulos de acuerdo a la elección del usuario. El conjunto contiene cuatro resolvedores lineales hasta este momento, explicados en los **apartados 3.5 RESOLVEDORES LINEALES DE *CVODE*** y **3.6 MÓDULOS GENÉRICOS DE LOS RESOLVEDORES**) aunque el número de los resolvedores puede incrementarse. El incremento del número de resolvedores no afectaría al modulo integrador principal.

El paquete de *CVODE* incluye un módulo llamado *VECTOR*, tratado con mayor detalle en el **apartado 3.7 NÚCLEOS BÁSICOS DE VECTORES .** Dicho módulo, contiene rutinas para manipular vectores, todas las operaciones de *N_Vectors* se realizan con llamadas a este módulo.

Dos módulos pequeños se encargan de realizar las tareas relacionadas con la precisión aritmética y las operaciones matemáticas de bajo nivel. El módulo *LLNLTYPS* contiene las declaraciones para los tipos *real* e *integer,* facilitando las conversiones de doble precisión a precisión simple o viceversa. El módulo *LLNLMATH* contiene macros para operaciones tales como *MAX*, *MIN* y potencia de funciones.

## 3.4. INTERFAZ DE USUARIO DE CVODE

Para utilizar *CVODE*, el usuario debe proporcionar ciertas rutinas de definición del problema y hacer llamadas al paquete. Para comenzar, el programa del usuario debe llamar a tres o cuatro funciones de *CVODE*. Al contrario de *VODE* y las versiones anteriores, en *CVODE* las llamada son explícitamente separadas para la integración, la carga de los datos de entrada fijos, la reservación y liberación de memoria, así como la especificación del resolvedor lineal. En la **Figura 3.4.2 Paquete Computacional *CVODE***, se muestran los nombres de los archivos de cabecera para cada uno de los módulos que constituyen *CVODE*, así como también el nombre de las rutinas empleadas para las llamadas desde el programa del usuario.

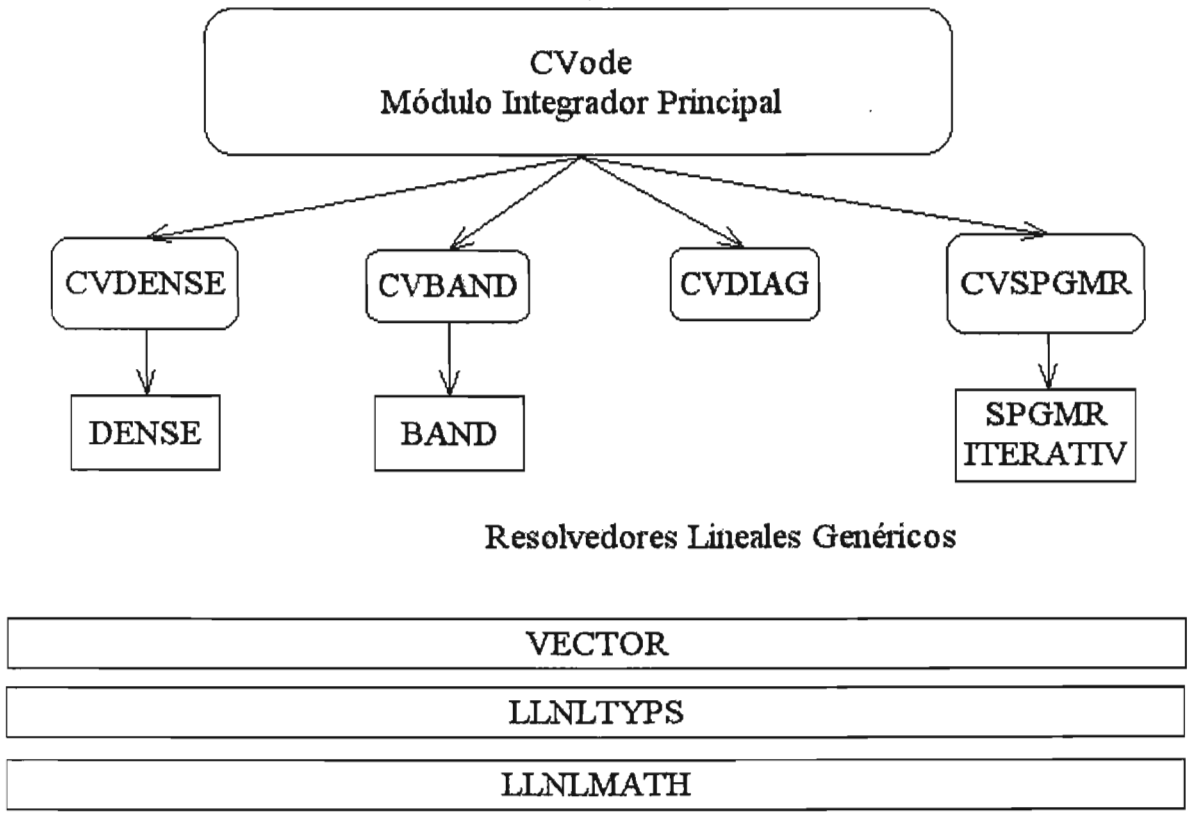



Figura 3.4.1 Diagrama por bloques del paquete *CVODE*

El programa del usuario deberá contener las siguientes partes fundamentales:

- lineas *#include* para acceder a los archivos de cabecera;
- reservación y liberación de memoria para los datos de entrada de *CVODE*, incluyendo los datos de entrada *y* del tipo *N_Vector* (utilizando la macro *N_VNew*);
- una llamada a la función *CVodeMalloc* para reservar memoria para *CVODE*;
- si es seleccionada la iteración de *Newton* deberá llamarse una rutina *CVDense*, *CVBand*, *CVDiag* o *CVSpgmr* para especificar cual módulo será empleado como resolvedor del sistema lineal;
- un ciclo de llamadas a *CVode* para integrar los puntos de salida señalados;
- una llamada a *CVodeFree* para liberar toda la memoria ocupada por *CVODE*;

- liberación de la memoria solicitada por el usuario después de haberse realizado la integración, por ejemplo la liberación de *y* con *N_VFree*;

- la función *f* proporcionada por el usuario, definiendo *f(t,y)*;

- una función opcional *Jac* elaborada por el usuario para una aproximación de la matriz *Jacobiana* (resolvedores *CVDENSE* y *CVBAND*) o un par de rutinas, *Precond* y *PSolve*; para el método *Krylov* precondicionado;

- funciones del usuario que definen el problema *EDO* y la función *Jac* opcional (se dispone de una función *Jac* si no es proporcionada una por el usuario); y

- llamadas para las operaciones con vectores. Todos los vectores de longitud *N*, así como *y*, están comunicados con objetos del tipo *N_Vector* (tipo definido en el paquete *CVODE*).

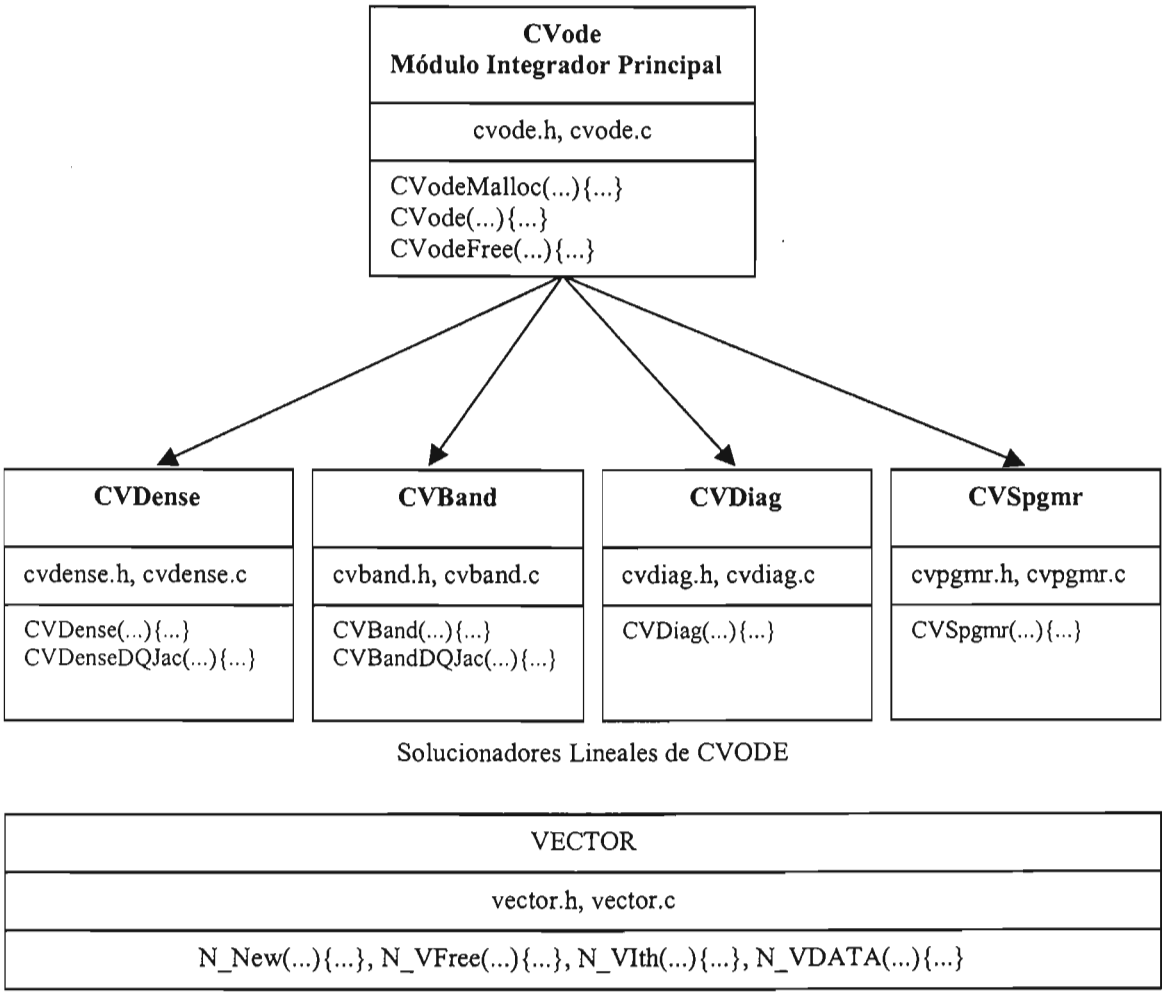



Figura 3.4.2 Paquete Computacional *CVODE*

La estructura del programa del usuario tendrá la forma siguiente, en este caso; se ilustra el uso de salidas opcionales accediendo el número de pasos especificados en *iopt[Paso].*

```
main( )
{
    y = N_VNew(...);

    cvode_mem = CVodeMalloc(N, f, t0, y, ..., iopt, ...);

    CVDense, CVBand, CVDiag o CVSpgmr(...);

    for (tout = ...)
        Bandera = CVode(cvode_mem, tout, y, ...);
        printf("Paso = %d", iopt[Paso]);

    CVodeFree(...);

    N_VFree(y);
}

f( ... ) { ... }

Jac( ... ) { ... }      ó

Precond( ... ) { ... }      y      PSolve( ... ) { ... }
```

Para lograr el enlace con *CVODE*, el usuario debe establecer una comunicación de datos entre el programa de llamada y sus rutinas, complementando con algunos punteros pasados a *CVODE* y regresados a las rutinas del usuario. Los punteros pueden apuntar a cualquier estructura diseñada por el usuario, de una manera apropiada para la aplicación. Existen tres de éstos punteros:

- *f_data*. El usuario pasa este puntero a *CVodeMalloc* y *CVODE* lo pasa a la rutina *f* del usuario.
- *jac_data*. El usuario pasa este puntero a *CVDense* o *CVBand* y *CVODE* lo pasa a la rutina *Jac* del usuario.
- *P_data*. El usuario pasa este puntero a *CVSpgmr* y *CVODE* lo pasa a las rutinas *Precond* y *PSolve* del usuario.

Cabe señalar que, en los casos *dense* y *band, jac_data* puede ser idéntico a *f_data*, mientras que en el caso *Krylov, P_data* puede ser idéntico a *f_data*.

## 3.5. RESOLVEDORES LINEALES DE CVODE

Los resolvedores lineales en *CVODE*, forman un arreglo expandible de código modular; lo cuál juega un papel importante en el éxito de *CVODE* para resolver sistemas *EDO* rígidos. Los módulos son cuatro: *CVDENSE*, *CVBAND*, *CVDIAG* y *CVSPGMR*. Cada módulo debe interactuar con el usuario, con el módulo integrador principal, con el resolvedor lineal genérico que le da soporte, con el módulo *VECTOR* y otros módulos de bajo nivel.

Como se indicó anteriormente, el usuario especifica cual resolvedor lineal usará, haciendo una llamada a la rutina CVxxx, el nombre puede ser *CVDense*, *CVBand*, *CVDiag* o *CVSpgmr*; esta llamada proporciona datos de entrada específicos al resolvedor lineal, tales como las rutinas del usuario asociadas.

Además de la función *CVxxx*, cada resolvedor lineal cuenta con cuatro partes:

1. **Función de inicialización**. Reserva memoria para los datos propios del resolvedor (por ejemplo, una matriz *M* y un arreglo de pivotes) e inicializa contadores específicos del resolvedor.
2. **Función de configuración de matriz**. Manipula los cálculos de datos relacionados con la matriz *Jacobiana*, si es llamada para el contexto de la iteración de *Newton*. Esta función debe ejecutar cualquier preproceso necesario en la función de solución del sistema (por ejemplo, la *factorización LU*).
3. **Función de solución del sistema**. Se encarga de obtener la solución del sistema lineal $Mx = b$, dentro del contexto de la *iteración* de *Newton*; emplea la memoria reservada por la función de inicialización y puede llamar a la rutina del resolvedor lineal genérico.
4. **Función de liberación**. Libera la memoria empleada por el resolvedor especificado.

Las cuatro funciones anteriores, son llamadas desde el módulo principal *CVode*, para que éste módulo sea independiente del resolvedor lineal elegido, las secuencias de llamada de las cuatro funciones son fijas. La liga desde el módulo integrador principal hasta el módulo del resolvedor lineal es creada por la función CVxxx, la cual establece los punteros alojados en el bloque de memoria reservada por *CVODE*. Las funciones del resolvedor lineal también reciben un puntero al bloque de memoria de *CVODE* para poder compartir datos acerca del estado de la integración y para tener alcance al bloque de memoria específica del resolvedor.

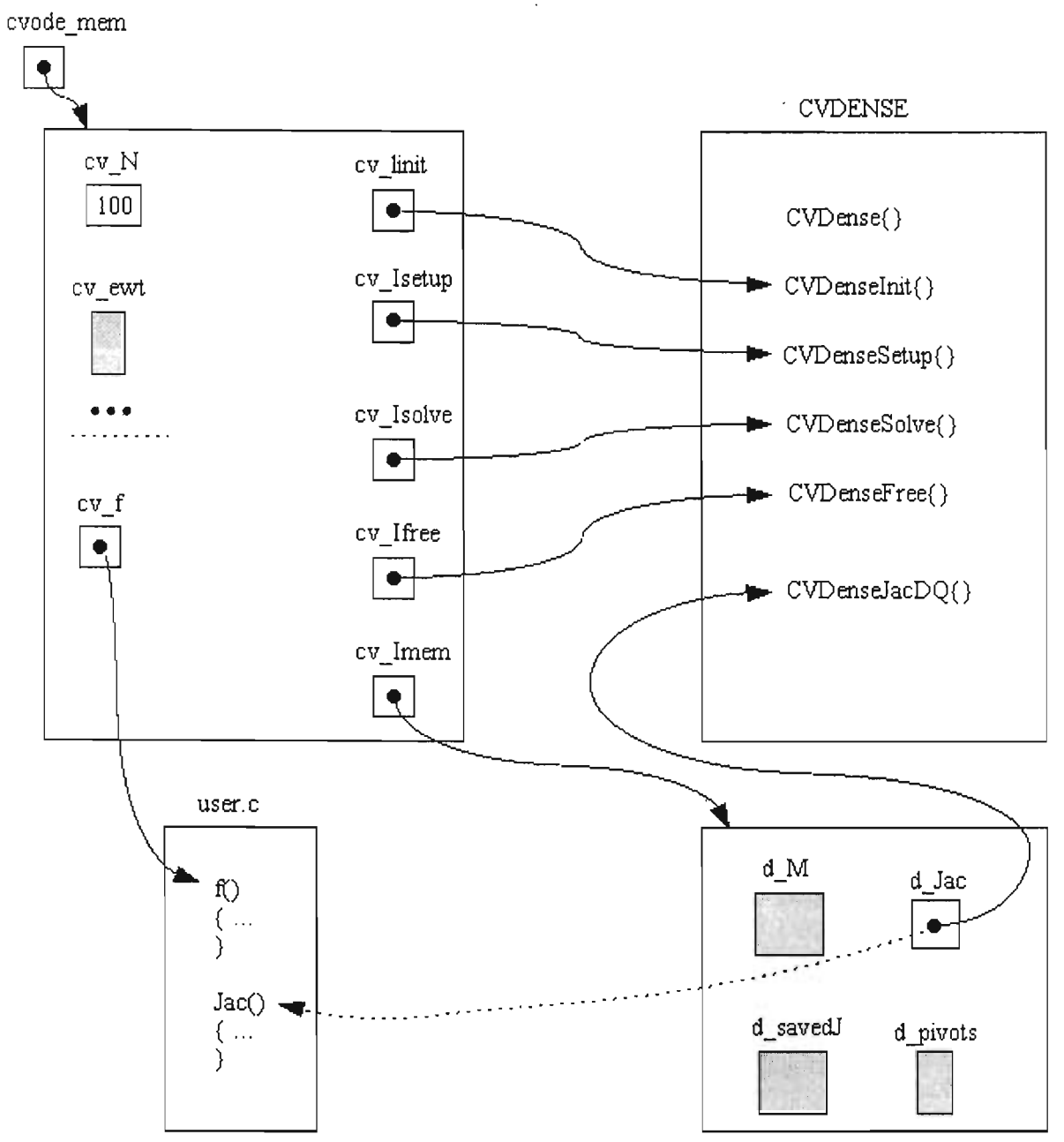



Figura 3.5.1 Enlaces entre la memoria de *CVODE* y el resolvedor lineal *dense*

Los alcances a la memoria, se muestran en la **Figura 3.5.1 Enlaces entre la memoria de *CVODE* y el resolvedor lineal *dense***, en la cual el resolvedor lineal es *CVDENSE*. El bloque de memoria de *CVODE*, el cual esta señalado por el puntero *cvode_mem* regresado por la función *CVodeMalloc*; incluye valores como *N*, arreglos para trabajo adicional, un puntero a la función *f* del usuario, punteros a las cuatro partes del módulo del resolvedor lineal (*cv_linit* puntero a la función de inicialización, etc.) y un puntero *cv_lmem* a la memoria específica del resolvedor. Los últimos cinco punteros señalados anteriormente, son establecidos en el módulo

*CVDense*. La memoria del resolvedor lineal, en este caso; incluye la matriz *M*, una copia guardada de la matriz *Jacobiana J*, un arreglo de pivotes, y un puntero *d_jac* a la rutina del cálculo de la *Jacobiana*. Si el argumento *Jac* de *CVDense* no fue *NULL*, *d_jac* apunta a la función dada por el usuario; en caso contrario *d_jac* apunta a la rutina proporcionada en *CVDENSE*, llamada *CVDenseJacDQ*.

## 3.6. MÓDULOS GENÉRICOS DE LOS RESOLVEDORES

Un criterio importante en el diseño de *CVODE,* fue usar los resolvedores de sistemas lineales de forma genérica, el código modular permite la selección intercambiable de los resolvedores por separado, sin la referencia del sistema *EDO* o la iteración de *Newton*. El paquete *CVODE* incluye tres módulos de resolvedores lineales genéricos, mostrados en la **Figura 3.4.1 Diagrama por bloques del paquete *CVODE***, (el módulo *CVDIAG* no requiere un resolvedor genérico), éstos módulos son explicados brevemente a continuación:

- ***DENSE*** resuelve sistemas lineales densos por la *factorización LU* y por solución en retroceso con pivoteo parcial, fue recodificado en *C* a partir de rutinas del paquete *LINPACK*, contiene funciones para el manejo de memoria y otras para operaciones auxiliares. Las funciones de *DENSE* son empleadas en cálculos de matrices del tipo *DenseMat* (definido en *DENSE*) y vectores del tipo *N_Vector* (definido en *VECTOR*).
- ***BAND*** resuelve sistemas lineales de banda a través de la factorización LU y por solución en retroceso soluciòn con pivoteo parcial, fue recodificado en *C* a partir de rutinas del paquete *LINPACK*, contiene funciones para el manejo de memoria y otras para operaciones auxiliares. Las funciones de *BAND* son empleadas en cálculos de matrices del tipo *BandMat* (definido en *BAND*) y vectores del tipo *N_Vector* (definido en *VECTOR*).
- ***SPGMR + ITERATIV*** resuelve sistemas lineales arbitrarios mediante el algoritmo *GMRES* de precondición escalar. El módulo *SPGMR* contiene tres funciones de llamada por el usuario (para reservar memoria, resolver un sistema lineal y liberar memoria). El módulo *ITERATIV* contiene funciones de apoyo para procedimientos *Gram-Schmidt* clásico y modificado; *factorización QR* y solución "*least-squares*" de un sistema *Hessenberg* **[HARRIS1998, p. 416]**.

## 3.7. NÚCLEOS BÁSICOS DE VECTORES

El módulo *VECTOR* del paquete *CVODE,* es una colección de núcleos (en inglés *kernel)* para realizar operaciones con vectores *N-Vectors,* ejecutadas dentro de

*CVODE*. Las operaciones disponibles incluyen la reservación y liberación de memoria (con los nombres *N_VNew, N_VFree*); aritmética de vectores (con los nombres *N_VLinearSum, N_VScale, N_VProd, N_VDiv, N_VConst*, etc.) y cálculos escalares (con el nombre *N_VDotProd, N_VWrmsNorm, N_VMaxNorm, N_VMin*).

Todos los núcleos en el módulo *VECTOR* operan con vectores del tipo *N_Vector*, el cual está definido en el propio módulo.

El capítulo actual ha mostrado las características del paquete *CVODE*, a través de cada apartado. En esta tesis, se usó *CVODE* por estar diseñado para resolver problemas rígidos; ya que los sistemas *EDO* tratados son rígidos; por ser un software de alto rendimiento, reconocido y de uso libre (o gratuito) y, por poseer una interfaz en *C++*, lo cuál permite implementarlo en el sistema de cómputo integral desarrollado.

Para encontrar información con mayor detalle técnico acerca de *CVODE*, se puede consultar **[COHEN1994]**. En el **APÉNDICE A: FUNCIONES PRINCIPALES DE CVODE**, se muestran las especificaciones particulares de las cuatro funciones principales de *CVODE*, que constituyen la interfaz con *C++*.

# CAPÍTULO 4. SISTEMA DE CÓMPUTO INTEGRAL IMPLEMENTANDO UN PARSER

El sistema de cómputo integral desarrollado en esta tesis, está constituido por la implementación de un *analizador sintáctico (parser en inglés)* para el cálculo analítico y exacto de la matriz *Jacobiana* en el modelo matemático de reacciones químicas del medio ambiente (con referencia a contaminación atmosférica); por la resolución del sistema *EDO (Ecuaciones Diferenciales Ordinarias)* de tipo rigido utilizando el paquete computacional *CVODE* y, por la representación gráfica de los resultados en tiempo de ejecución. Además, el sistema de cómputo integral cuenta con una interfaz gráfica para facilitar la interacción con el usuario.

Para el cálculo analítico y exacto de la matriz *Jacobiana* se diseñó un modelo de *análisis sintáctico* incluyendo las tres primeras fases de compiladores, tratadas en el **Capítulo 1**: *análisis lexicográfico*, *análisis sintáctico* y *análisis semántico*. Para desarrollar el modelo, se utilizó programación orientada a objetos y herramientas de programación visual y multihilos proporcionadas por *Borland C++ Builder* (descritas en el **Capítulo 2**), incorporando la *STL (Standard Template Library)* que forma parte del estándar de *C++*.

La *STL* es un conjunto de bibliotecas, diseñada para ser eficiente, segura, portable y genérica; la integran cuatro conceptos: contenedores, iteradores, algoritmos y objetos de función. Los contenedores de *STL* utilizados en esta tesis fueron: *vector*, *map*, *set* y *string*. Para mayores detalles acerca de la *STL* se pueden consultar **[STROUSTRUP2002, p. 443-680] y [MARTÍN2001, p. 56-61]**.

Son tres los procesos principales del sistema de cómputo integral: ***el primero***, implementa un modelo de *análisis sintáctico*, el cuál necesita como entrada el archivo con el sistema *EDO* y proporciona como salida un archivo con la matriz *Jacobiana* obtenida en el formato reconocido por *CVODE*; ***el segundo***, toma como entrada el archivo con la matriz *Jacobiana* y lo usa en *CVODE* para resolver el sistema *EDO* y; ***el tercero***, recibe los resultados numéricos proporcionados por *CVODE* dando como salida los resultados gráficamente.

En los siguientes apartados se explica a través de tres fases el modelo de *análisis sintáctico* implementado; el uso del paquete computacional *CVODE* y, el empleo de programación visual y las características multihilos utilizadas para obtener el sistema de cómputo integral. También se muestra una parte de la intefaz del usuario desarrollada para el sistema de cómputo integral aplicando el modelo de 11 substancias de *Seinfeld* **[SEINFELD1998].**

## 4.1. MODELO DE ANÁLISIS SINTÁCTICO

El sistema de cómputo integral cuenta con tres procesos principales en su implementación; el primer proceso lo constituye el modelo de *análisis sintáctico*. Dicho modelo está formado por ***tres fases***: la de *análisis lexicográfico*, la de *análisis sintáctico* y la de *análisis semántico*. El propósito fundamental de éste primer proceso es obtener la matriz *Jacobiana* y almacenarla en un *archivo de texto* **(véase APÉNDICE C: EJEMPLO DE ARCHIVO CONTENIENDO LA MATRIZ JACOBIANA)**. El *archivo de texto* mencionado, es requerido por el segundo proceso de implementación con *CVODE*. Para lograr el propósito de este primer proceso, es necesario seguir el modelo de *análisis sintáctico* mostrado en la **Figura 4.1.1.**

El modelo de *análisis sintáctico* inicia suministrando un *archivo de texto* conteniendo el sistema *EDO;* como el ejemplo indicado en la **Tabla 4.1.2.** ***La primera fase*** o *análisis lexicográfico* evalúa cada carácter, ignorando los espacios en blanco e integrando los símbolo*s*. En esta fase, a cada símbolo le es asociado un tipo de símbolo. Los símbolo*s* encontrados son recibidos en ***la segunda fase*** o *análisis sintáctico*, donde son unidos para formar estructuras más complejas representando los términos y al mismo tiempo las ecuaciones del sistema *EDO*. Al formar un término se establece una representación de los componentes del término y su orden (como son signo, coeficiente, variables y sus potencias) a través de una cadena de números con el símbolo $ al final. La cadena de números creada en la fase de *análisis sintáctico* contiene los números 1, 2 y 3 asociando el tipo de símbolo delimitador, número y variable respectivamente. ***En la tercera fase*** o *análisis semántico* se verifica la cadena de números enviados por la fase anterior y se interpreta a través de un autómata finito determinista si es correcta; al ser correcta, se regresa a la segunda fase una bandera de aceptación.

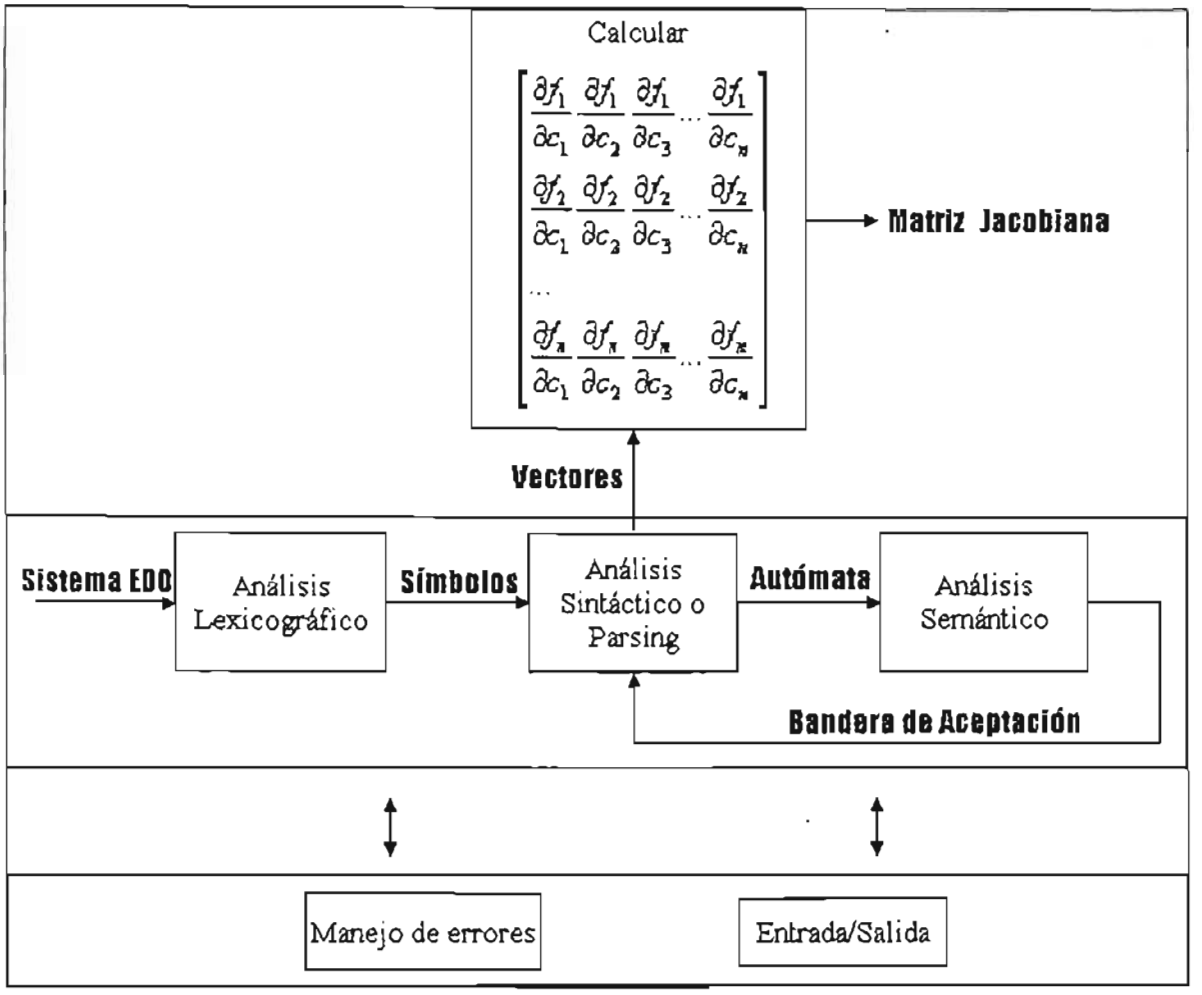



Figura 4.1.1 Modelo de *anállsis sintáctico* implementado

Cuando un término es aceptado, entonces se guarda en variables de vectores de términos hasta concluir la revisión completa y correcta del sistema *EDO* proporcionado al inicio del modelo. Finalmente, los vectores de términos del sistema *EDO* son recuperados de la memoria para realizar los cálculos necesarios para obtener la matriz *Jacobiana* y crear un *archivo de texto* conteniéndola.

El modelo de *anállsis sintáctico* está implementado en los archivos de la **Tabla 4.1.1.** En las tres fases del modelo se interactúa con varias rutinas de control de error y la interfaz de Entrada/Salida para mostrar los mensajes acordes.

| **Archivo** | **Descripción** |
|---|---|
| *Interprete.cpp* | Guarda los términos de cada ecuación del sistema *EDO* en vectores de la *STL*. |
| *Parser.cpp* | Forma los símbolos y su tipo. |
| *MatrizJac.cpp* | Calcula la matriz *Jacobiana* de acuerdo con los datos guardados en los vectores de la *STL*. |
| *Gramatica.cpp* | Realiza el *análisis semántico* de cada término del sistema *EDO*. |
| *ClaseTer* | Define una clase para los términos de las ecuaciones. |
| *ManError.cpp* | Controla el envío de mensajes a la pantalla cuando se produce un error en el modelo de *análisis sintáctico* **implementado** (véase ***Tabla 4.1.3*** ). |

Tabla 4.1.1 Archivos empleados en la implementación del modelo de *análisis sintáctico*

En la **Tabla 4.1.2** se muestra un ejemplo de un sistema *EDO* para ser analizado por el modelo de *análisis sintáctico* aquí planteado. Las letras *c1, c2, c3*, etc. representan a las substancias que intervienen en las ecuaciones químicas, las cuales corresponden al modelo de 11 substancias de *Seinfeld* de la **Tabla 4.1.3**.

| |
|---|
| *+k1c2 -k3c1c3 -k7c1c10 -k8c1c7 –k9c1c9* |
| *-k1c2 +k3c3c1 +k7c1c10 +k8c1c7 +k9c1c9 -k10c2c6 -k11c2c9 +k12c11* |
| *+k2c4c_O2c_M -k3c3c1* |
| *+k1c2 -k2c4c_O2c_M* |
| *-k4c5c6* |
| *-k4c5c6 -k5c6c8 +k7c1c10 -k10c2c6* |
| *+k4c5c6 +k6c8 -k8c1c7 +k9c1c9* |
| *-k5c6c8 -k6c8 +k8c1c7* |
| *+k5c6c8 -k9c1c9 -k11c2c9 +k12c11* |
| *+k6c8 -k7c1c10 +k8c1c7* |
| *+k11c2c9 –k12c11* |

Tabla 4.1.2 Ejemplo de un sistema de *EDO* del modelo de 11 substancias de *Seinfeld*

| **Variable** | **Nomenclatura** |
|---|---|
| *c1* | *NO* |
| *c2* | *NO2* |
| *c3* | *O3* |
| *c4* | *O* |
| *c5* | *RH* |
| *c6* | *OH* |
| *c7* | *RO2* |
| *c8* | *RCHO* |
| *c9* | *RC[O]O2* |
| *c10* | *HO2* |
| *c11* | *RC[O]O2NO2* |

Tabla 4.1.3 Modelo de 11 substancias de *Selnfeld*

El valor de los coeficientes constantes representados por la letra *k*, se muestran a continuación en la **Tabla 4.1.4**; y más adelante, se indica el valor de las concentraciones constantes *c_O2* y *c_M*.

| Coeficiente Constante | Valores | Valores Aproximados |
|---|---|---|
| *K1* | 0.533/60.0 | 0.008883333 |
| *K2* | 2.183e-5/60.0 | 0.000000363 |
| *K3* | 26.59/60.0 | 0.443166666 |
| *K4* | 3.775e3/60.0 | 62.91666667 |
| *K5* | 2.341e4/60.0 | 390.1666667 |
| *K6* | 1.91e-4/60.0 | 0.000003183 |
| *K7* | 1.214e4/60.0 | 202.3333333 |
| *K8* | 1.127e4/60.0 | 187.8333333 |
| *K9* | 1.127e4/60.0 | 187.8333333 |
| *K10* | 1.613e4/60.0 | 268.8333333 |
| *K11* | 6.893e3/60.0 | 114.8833333 |
| *K12* | 2.143e-2/60.0 | 0.000357166 |

Tabla 4.1.4 Valor de los coeficientes constantes

**Valor de las concentraciones constantes**

*c_M*=79.0e4;
*c_O2*=21.0e4;

En la **Tabla 4.1.2 Ejemplo de un sistema de EDO del modelo de 11 substancias de Seinfeld**, se puede apreciar que los coeficientes toman la forma de coeficientes constantes representados por la letra *k* y un número, aunque ésta es la forma más usual; también la siguiente forma sería válida para la primera y la cuarta ecuaciones:

*(Ecuación Química 1)*
+0.008883333 C2
-0.443166666 C1 C3
-202.3333333 C1 C10
-187.8333333 C1 C7
-187.8333333 C1 C9

*(Ecuación Química 4)*
+0.008883333 C2
-0.000000363 C4 C_O2 C_M

En el **apartado 4.4. DESCRIPCIÓN DE LA INTERFAZ DEL USUARIO**, se muestra la ejecución del sistema de cómputo integral y se explica con mayor detalle la interfaz del usuario.

### 4.1.1 Análisis Lexicográfico

El modelo de *análisis sintáctico* implementado consta de una primera fase llamada *análisis lexicográfico*. Esta fase tiene como misión determinar los símbolo*s* válidos y su tipo. Para hacer lo anterior, se debe proporcionar un *archivo de texto* (en ASCII con la extensión txt de tipo texto) conteniendo el sistema *EDO* a evaluar, cada vez que se logre obtener un símbolo, se enviará el símbolo y su tipo (delimitador, números o variables) a la fase de *análisis sintáctico*. El archivo de texto deberá corresponder a las características determinadas por el modelo de *análisis sintáctico* en cada una de sus fases.

El *análisis lexicográfico* del sistema de cómputo integral de esta tesis, se lleva a cabo a través de las características de los símbolo*s* mostradas en la **Tabla 4.1.5**. Un símbolo puede ser delimitador, número o variable. Los delimitadores son los símbolo*s* formados por caracteres especiales indicando el signo del término de la ecuación; la potencia de una variable; el fin de una ecuación y, el final del archivo. Los números que pueden formar un símbolo pueden ser de un solo dígito, de varios dígitos, reales con punto decimal y representados en notación científica en formato exponencial. Las variables pueden ser de tres tipos, substancias

representadas por la letra *c* y un número; valor predefinido de substancias representadas por *c_m* y *c_o2*, y el valor del coeficiente del término, ya conocido previamente representado por la letra *k* y un número.

| Tipo de Símbolo | Símbolo | Ejemplo de Símbolo | Descripción del Símbolo |
|---|---|---|---|
| Delimitador | Signo | - | Signo negativo del término |
| Delimitador | Signo | + | Signo positivo del término |
| Delimitador | Gato | # | Seguido de un número significa la potencia de la variable anterior |
| Delimitador | Enter | \n | *Enter*, indica el fin de una ecuación |
| Delimitador | Fin | \0 | Marca el fin del archivo |
| Números | Coefi | 9, 200, 0.443166, 2.143e-2, 1.214e4 | Un dígito, secuencia de dígitos, secuencia de uno o más dígitos donde puede incluirse el símbolo "." o bien; incluir la letra *e* (o también *e- e+)* seguida de otros dígitos |
| Números | Potencia | #2, #8, #200 | Número entero al cual le antecede el símbolo #. |
| Variables | Concentra | c1, c2, c12, c200 | Letra *c* seguida de uno o más dígitos |
| Variables | ConCons | c_o2, c_m | Letra *c* seguida del símbolo " _ " |
| Variables | CoefiConst | k1, k9, k11 | Letra *k* seguida de uno o más dígitos |

Tabla 4.1.5 Características de símbolo para el sistema de cómputo integral

Los símbolos se pueden definir a través de expresiones regulares o por autómatas finitos deterministas. Para definir todos los símbolos válidos es necesario tomar en consideración las abreviaturas empleadas en las expresiones regulares.

**Abreviaturas empleadas en las expresiones regulares:**

1. * Es un operador que representa "cero o más casos de", es decir; el conjunto de todas las cadenas de uno o más símbolos, incluyendo la cadena vacía.

2. ? Es un operador que representa "cero o un caso de", es decir; uno o ningún símbolo.
3. | Es un operador que significa "o", es decir; uno u otro símbolo acotados entre el operador.

**Tipo de Símbolo Delimitador**

-|+|\n|#|\0

- **Símbolo SIGNO**
  +|-

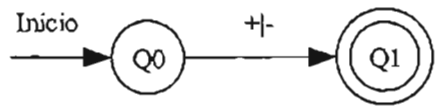



Figura 4.1.2 Autómata finito determinista para el símbolo *signo* del tipo delimitador

- **Símbolo ENTER**
  \n

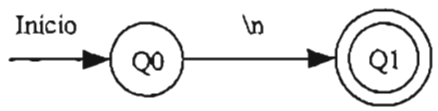



Figura 4.1.3 Autómata finito determinista para el símbolo enter del tipo delimitador

- **Símbolo GATO**
  #

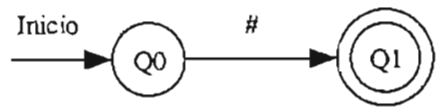



Figura 4.1.4 Autómata finito determinista para el símbolo gato del tipo delimitador

- **Símbolo FIN**
  \0

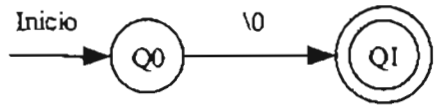



Figura 4.1.5 Autómata finito determinista para el símbolo fin del tipo delimitador

Como muestra la **Figura 4.1.2, Figura 4.1.3, Figura 4.1.4 y Figura 4.1.5**, puede formarse un símbolo del tipo delimitador con cualquiera de los caracteres -, *+, #, \n* ó *\0*; donde los dos primeros se utilizan para dar el signo negativo o positivo de cualquier número contenido en los términos de una ecuación; el símbolo *#* aparece para especificar la potencia de la variable anterior en el término; los símbolos \n indican que se oprimió la tecla *enter* especificando el fin de la ecuación, y \0 representa el fin del archivo.

**Tipo de Símbolo Números**

El tipo de símbolo números está formado por diferentes formatos de números, por lo cual se define cada uno con una expresión regular y posteriormente se representan en conjunto en un autómata finito determinista. Cabe mencionar que los símbolo*s* de número pueden ser coeficientes de un término o potencias de una variable de concentración; en la tercera fase del modelo de *análisis sintáctico* implementado es donde se interpreta el orden correcto para los componentes (signo, coeficiente, variables y potencias) de cada término del sistema *EDO* y se determina la validez de dichos componentes.

- **Símbolo DIGITO**

9|8|7|6|5|4|3|2|1|0

- **Símbolo COEFI entero**

**(DIGITO)(DIGITO)***

(9|8|7|6|5|4|3|2|1|0)(9|8|7|6|5|4|3|2|1|0)*

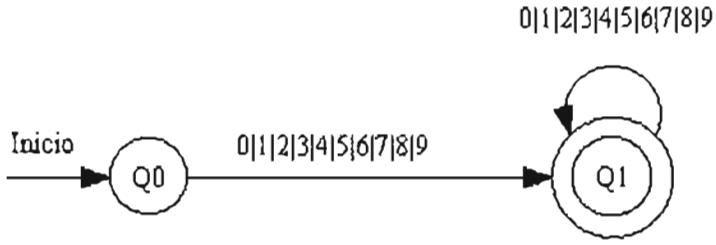



Figura 4.1.6 Autómata finito determinista para el símbolo número entero

- **Símbolo COEFI real**

**COEFI entero (. (COEFI entero))?**

(9|8|7|6|5|4|3|2|1|0)(9|8|7|6|5|4|3|2|1|0)*
(.(9|8|7|6|5|4|3|2|1|0)(9|8|7|6|5|4|3|2|1|0)*)?

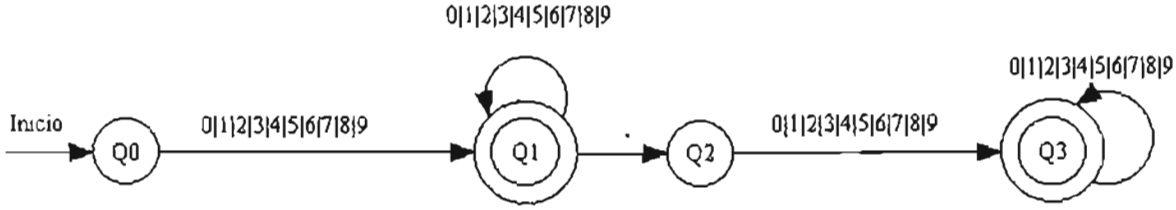



Figura 4.1.7 Autómata finito determinista para el símbolo número real

- **Símbolo COEFI real en notación exponencial**

**(COEFIentero)**
**(.(COEFIentero)(e(DIGITO|SIGNO(DIGITO))(DIGITO)*)?)?**

(9|8|7|6|5|4|3|2|1|0)(9|8|7|6|5|4|3|2|1|0)*
(.(9|8|7|6|5|4|3|2|1|0)(9|8|7|6|5|4|3|2|1|0)*
(e(9|8|7|6|5|4|3|2|1|0|(-|+)(9|8|7|6|5|4|3|2|1|0))(9|8|7|6|5|4|3|2|1|0)*)?)?

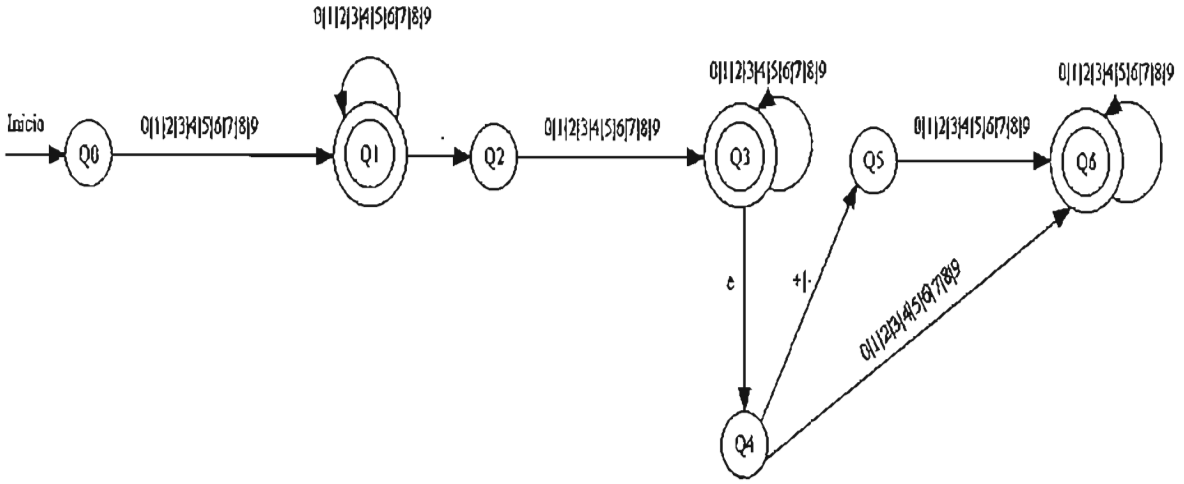



Figura 4.1.8 Autómata finito determinista para el símbolo número exponencial

En el caso de los números, un símbolo se puede formar por un número entero, un real o un exponencial. En los números exponenciales esta previsto que empiecen con un número real (si es un entero debe agregarse un punto y el 0), a continuación la letra *e,* y de ser necesario; puede especificarse el signo del siguiente número;este último valor puede ser positivo y no es obligatorio incluir el signo +. La **Figura 4.1.6, Figura 4.1.7 y Figura 4.1.8** muestran un autómata para los números.

- Símbolo **POTENCIA**

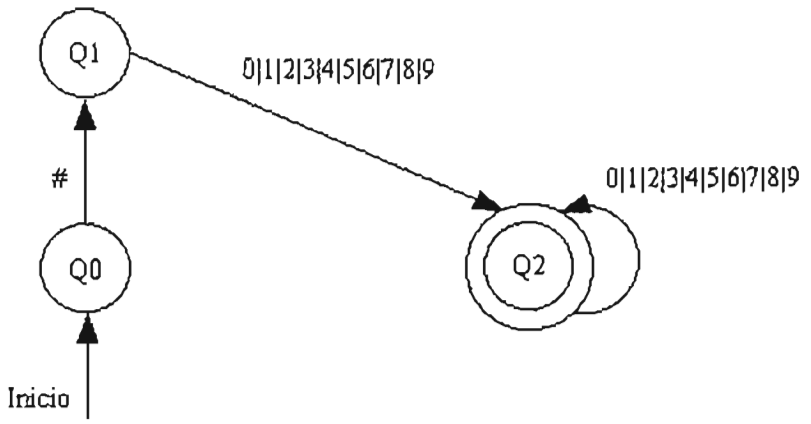



Figura 4.1.9 Autómata finito determinista para el tipo de símbolo numérico especial potencia

En el grupo de los símbolos numéricos está incluido el símbolo numérico especial potencia, dicho símbolo lo distinguimos por tener el símbolo # al inicio, como se podrá apreciar en la **Figura 4.1.9**. El valor de la potencia se emplea para indicar la potencia a la cual es elevado el valor de una concentración o variable, todo esto aplica para las variables de la forma *c1, c12, c200* por ejemplo.

**Tipo de Símbolo Variables**

c(9|8|7|6|5|4|3|2|1|0)(9|8|7|6|5|4|3|2|1|0)*|c_o2|c_m
|k(9|8|7|6|5|4|3|2|1|0)(9|8|7|6|5|4|3|2|1|0)*

- Símbolo **CONCENTRA**

c(9|8|7|6|5|4|3|2|1|0)(9|8|7|6|5|4|3|2|1|0)*

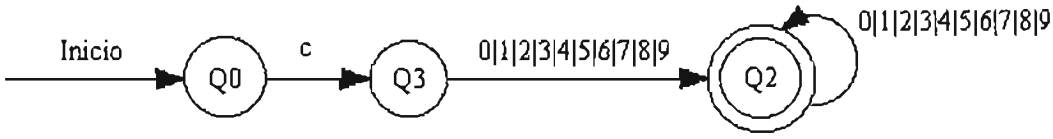



Figura 4.1.10 Autómata finito determinista para el tipo de símbolo variables para el símbolo CONCENTRA

- Símbolo **CONCONS**

c_m|c_o2

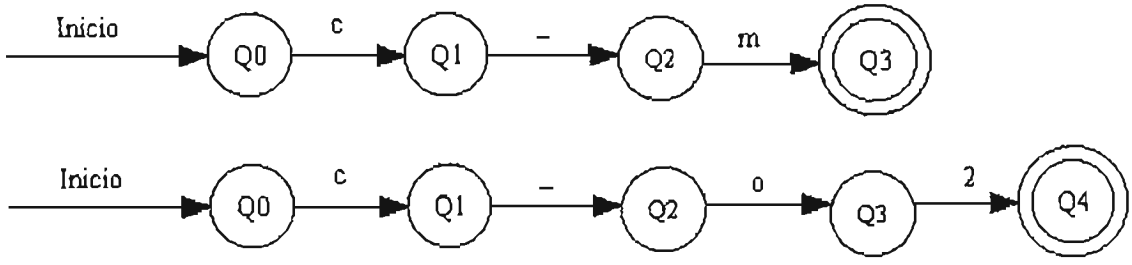



Figura 4.1.11 Autómata finito determinista para el tipo de símbolo variables para el símbolo CONCONS

- Símbolo **COEFICONS**

k(9|8|7|6|5|4|3|2|1|0)(9|8|7|6|5|4|3|2|1|0)*

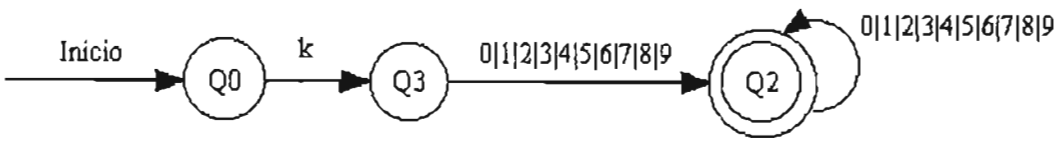



Figura 4.1.12 Autómata finito determinista para el tipo de símbolo variables para el símbolo COEFICONS

Las variables pueden tener cuatro formas distintas, como se puede observar en la **Figura 4.1.10, Figura 4.1.11 y Figura 4.1.12**. Las variables para representar las concentraciones de elementos químicos serán *c1, c2, c3, ..., cN*; mientras que las variables *k1, k2, k3*, etc., en realidad representarán valores numéricos definidos anticipadamente.

El *análisis lexicográfico* como primera fase del modelo de *análisis sintáctico* implementado, envía los símbolos a la segunda fase (*análisis sintáctico*), correspondiendo cada símbolo con alguna de las definiciones anteriores y con su tipo, de lo contrario; se genera un error. Además, en esta primera fase no se consideran los espacios en blanco ni las tabulaciones, son ignorados para solo crear símbolos con al menos un símbolo y sin espacios en blanco. Los símbolos del tipo delimitador, como son el *Enter* y el *Fin* de archivo son caracteres especiales que no se reflejan en pantalla pero si pueden ser detectados en la implementación del sistema de cómputo.

### 4.1.2 Análisis Sintáctico o Parsing

El propósito del *análisis sintáctico* o gramatical *(parsing en inglés)*, es evaluar un orden específico para los símbolos definidos y generados en la primera fase o *análisis lexicográfico*, es decir; en esta fase se aplicarán reglas de sintaxis a los símbolos para unirlos y formar términos, ecuaciones y finalmente el sistema *EDO* completo; empleando los contenedores *vector, map y string* de la *STL*.

El modelo de *análisis sintáctico* desarrollado aplica el método recursivo descendente el cual se ve reflejado a través de una *Gramática Libre de Contexto* y a su vez representada en el código de programación del sistema de cómputo integral.

Para lograr tener un sistema *EDO* sintácticamente correcto, se debe cumplir con la siguiente *Gramática Libre de Contexto* mostrada en la **Tabla 4.1.6 *Gramática Libre de Contexto* expresada con notación *EBNF***.

<SistemaEDO> ::= <FormaEc> | <***FIN***>

<FormaEc> ::= <FormaTer> | <***ENTER***> <***FIN***> | <***ENTER***> <FormaEc>
<FormaTer> ::= <***SIGNO***> { <Variables> | <Números> } <SistemaEDO>
<Variables> ::= <***COEFICONS***> | <***CONCENTRA***> | <***CONCONS***>
<Números> ::= <***COEFI***> | <***POTENCIA***>

Tabla 4.1.6 *Gramática Libre de Contexto* expresada con notación *EBNF*

Las reglas gramaticales de la **Tabla 4.1.6** están expresadas con notación *BNF extendida* o *EBNF (Extended Backus Naur Form)*, los nombres de los símbolos definidos en la primera fase o *análisis lexicográfico* están con mayúsculas, remarcados y con letras itálicas para diferenciarlos de los demás nombres que son no terminales. La selección está indicada en las reglas gramaticales por el símbolo | y la repetición por los símbolos { y }. La recursividad de la gramática empleada es por la derecha en la regla gramatical o de producción *FormaEc;* donde se agrupan cada uno de los términos de la ecuación y de manera recursiva se forman cada una de las ecuaciones del sistema *EDO*.

**Ejemplo**: Sea un sistema *EDO* de 2 ecuaciones como se muestra a continuación

```
-k1 c1#2 c2#3 c_o2 c_m +k2 c1#4 \n
+k3 c4#1                -k4 \n\0
```

Empleando la gramática antes mostrada con el modelo de *análisis sintáctico* recursivo descendente, se obtiene el árbol de análisis gramatical de la **Figura 4.1.13** .

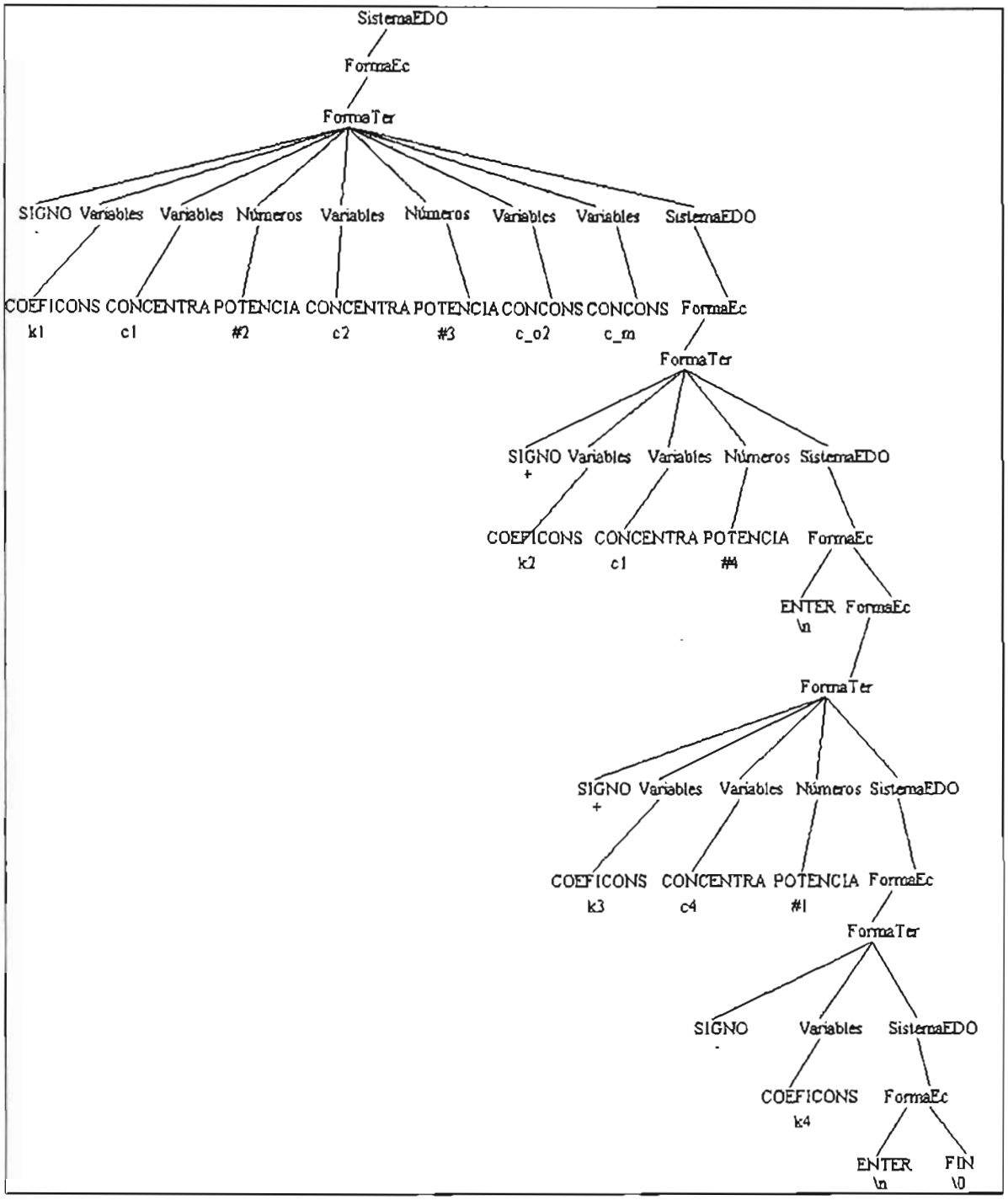



Figura 4.1.13 Un Árbol de Análisis Gramatical Recursivo Descendente

La codificación del sistema de cómputo integral en lenguaje *C++,* se realizó creando métodos similares a las reglas de producción *SistemaEDO*, *FormaEc* y *FormaTer* siendo de gran ayuda la simbología de la *EBNF* para definir sentencias de selección y repetición.

La *Gramática es Libre de Contexto* porque todas las reglas de producción anteriores cumplen con la siguiente definición:

$A \rightarrow \beta$
*con*
$A \in VNT$
$B \in (VNT \cup VT)^*$

donde VNT es el Vocabulario de los No Terminales y
VT es el Vocabulario de los Terminales

**Definición Formal de la Gramática Libre de Contexto**

G = ( VNT, VT, P, S )

Vocabulario No Terminal: VNT = { SistemaEDO, FormaEc, FormaTer, Variables, Números}

Vocabulario Terminal: VT = { FIN, ENTER, SIGNO, COEFICONS, CONCENTRA, CONCONS, COEFI, POTENCIA }

Producciones: P = {
( SistemaEDO , FormaEc ),
( SistemaEDO , FIN ),
( FormaEc, FormaTer ),
( FormaEc, ENTER FIN ),
( FormaEc, ENTER FormaEc ),
( FormaTer, SIGNO Variables SistemaEDO ),
( FormaTer, SIGNO Números SistemaEDO ),
( Variables, COEFICONS ),
( Variables, CONCENTRA ),
( Variables, CONCONS ),
( Números , COEFI ),
( Números , POTENCIA ),
}

Símbolo Inicial: S = SistemaEDO

Como se puede observar, en la definición formal de la *Gramática Libre de Contexto* empleada; se agrupan los no terminales con los terminales y la combinación de ambos genera el lenguaje del sistema de cómputo integral. Cabe señalar que el conjunto de teminales, es en realidad el conjunto de símbolos definidos en la primera fase del modelo de *análisis sintáctico* aquí implementado.

En esta segunda fase del modelo de *análisis sintáctico*, se logra formar estructuras más complejas partiendo de las estructuras simples o símbolos del *análisis lexicográfico*; los símbolos al unirse determinan la sintaxis de cada término y a su vez la unión de cada término logra formar las ecuaciones del sistema *EDO*.

Las estructuras empleadas para guardar en memoria cada término de las ecuaciones son las siguientes:

```
vector < Termino > Ecuacion;
vector < vector < Termino > > EDO;
```

donde;

***vector*** es un contenedor de *STL*,
***Termino*** es una clase que contiene el valor del coeficiente, variables y su potencia

Para las variables de las concentraciones se empleó la estructura *map* de *STL* para identificar el nombre de la variable y asociar su potencia como sigue:

```
typedef map<string, int, less<string> > TipoMap;
TipoMap VarExp;
```

donde;

*TipoMap* es el tipo de dato que contiene en el primer campo una cadena de caracteres y en el segundo campo un entero

*VarExp* es la variable definida para reconocer el nombre de las concentraciones *c1, c2, c3*, etc. y su respectiva potencia entera.

En el **apartado 4.1.3** se presenta la tercer fase del modelo de *análisis sintáctico*, donde se verifica la sintaxis generada buscando encontrar un significado particular y correcto en el contexto planteado.

### 4.1.3 Análisis Semántico

La tercera y última fase del modelo de *análisis sintáctico* es el *análisis semántico,* donde se revisa que los componentes de cada término y su orden correspondan con el diseño especificado en la **Figura 4.1.14 Autómata Finito Determinista para un término**; si existe correspondencia entonces el término es correcto y se almacena en la estructura de datos correspondiente (como se indicó en el apartado anterior del *análisis sintáctico*). Los componentes de cada término son el signo positivo o negativo, el coeficiente y las variables con sus respectivas potencias.

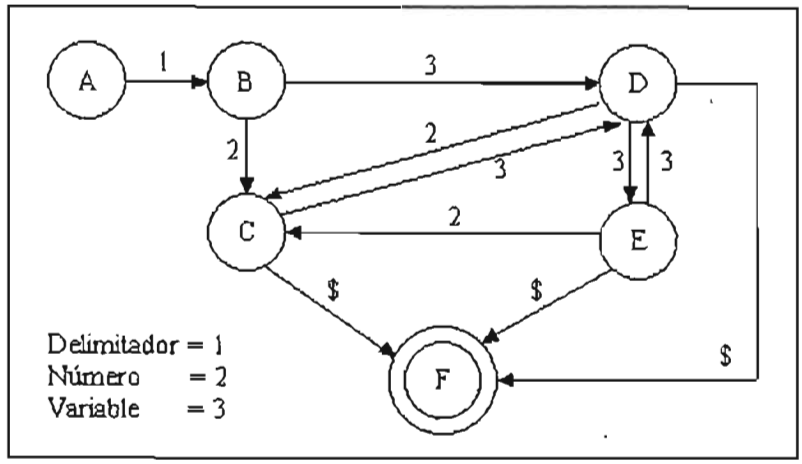



Figura 4.1.14 Autómata Finito Determinista para un término

El autómata mostrado en la **Figura 4.1.14**, define los componentes de cada término de una ecuación y su orden válidos para el *análisis semántico*. Los números 1, 2 y 3 representan los tipos de símbolo delimitador, número y variable. Dicho autómata indica que cada término debe comenzar en el estado *A* y transitar al estado *B* con el carácter – o + (caracteres definidos como símbolos de tipo delimitador); luego, del estado *B* se transita al estado *C* con el tipo de símbolo número (lo que representa un coeficiente en forma numérica) y desde el estado *C* se puede llegar al estado de aceptación *F* con el símbolo $. En caso de que el término tenga variables y potencias, las transiciones se realizan en los estado *D* y *E* hasta encontrar el símbolo $ para alcanzar el estado de *F*.

La otra posibilidad de transitar desde el estado *B*, es con el tipo de símbolo variable de acuerdo a las diferentes formas especificadas. Las formas especificadas pueden ser coeficiente constante *k1, k2, k3*, etc. ; variables de concentraciones *c1, c2, c3*, etc. o concentraciones constantes *c_o2 o c_m*.

Una vez alcanzado el estado de aceptación *F* significa que los componentes del término y su orden son correctos y el término es guardado en las variables de memoria determinadas. Hasta aquí se termina el modelo de *análisis sintáctico* en su sentido estricto, después de esto es calculada la matriz *Jacobiana* y guardada en un *archivo de texto* **(véase APÉNDICE C: EJEMPLO DE ARCHIVO CONTENIENDO LA MATRIZ JACOBIANA)**.

La matriz *Jacobiana* se calcula obteniendo las derivadas parciales del sistema *EDO* **(véase *Figura 4.1.1 Modelo de análisis sintáctico implementado*)**. Para lograr lo anterior, el modelo dispone y diferencia cada parte del sistema *EDO*, constituido éste último por cada ecuación, cada término y en cada término: el coeficiente, las variables y sus potencias. Una vez que el modelo establece en memoria las partes del sistema *EDO* en estructuras de datos, es posible calcular las derivadas parciales.

El resultado del primer proceso del sistema de cómputo integral, es el archivo *Jacob.Inc* conteniendo las asignaciones para la variable empleada como matriz, a la cual se hace referencia en los cálculos posteriores, los de *CVODE*. A continuación se muestra una parte del contenido del archivo *Jacob.Inc*, correspondiente a la primera y segunda filas de la matriz *Jacobiana*.

```
IJth(J,1,1)=-k3*c[3]-k7*c[10]-k8*c[7]-k8*c[9];
IJth(J,1,2)=k1;
IJth(J,1,3)=-k3*c[1];
IJth(J,1,4)=0;
IJth(J,1,5)=0;
IJth(J,1,6)=0;
IJth(J,1,7)=-k8*c[1];
IJth(J,1,8)=0;
IJth(J,1,9)=-k8*c[1];
IJth(J,1,10)=-k7*c[1];
IJth(J,1,11)=0;

IJth(J,2,1)=k3*c[3]+k7*c[10]+k8*c[7]+k8*c[9];
IJth(J,2,2)=-k1-k10*c[6]-k11*c[9];
IJth(J,2,3)=k3*c[1];
IJth(J,2,4)=0;
IJth(J,2,5)=0;
IJth(J,2,6)=-k10*c[2];
IJth(J,2,7)=k8*c[1];
IJth(J,2,8)=0;
IJth(J,2,9)=k8*c[1]-k11*c[2];
IJth(J,2,10)=k7*c[1];
IJth(J,2,11)=k12;
```

En el **APÉNDICE C: EJEMPLO DE ARCHIVO CONTENIENDO LA MATRIZ JACOBIANA,** se muestra el contenido completo del archivo *Jacob.inc*.

## 4.2. IMPLEMENTACIÓN DE CVODE

El sistema de cómputo integral cuenta con tres procesos principales en su implementación, como ya se ha mencionado anteriormente. En este apartado es abordado el segundo proceso, cuyo propósito consiste en la resolución del sistema rígido del sistema *EDO* en el modelo matemático de reacciones químicas del medio ambiente (con referencia a contaminación atmosférica), utilizando la matriz *Jacoblana* obtenida durante el proceso anterior, en el paquete computacional *CVODE*.

Para resolver el sistema *EDO* rígido en *CVODE* fueron incluidas llamadas a las cuatro funciones principales de *CVODE*, como se muestran a continuación y se describen los argumentos empleados en las funciones creadas en el sistema de cómputo integral, *f* y *Jac*. En la **Tabla 4.2.1**, **Tabla 4.2.2** y **Tabla 4.2.3** se describen los argumentos de tres de las funciones principales de *CVODE*.

### 1. *CVodeMalloc( )*, reservación de memoria de *CVODE* para el problema

*cvode_mem = CVodeMalloc(NumChem, f, t0, y, BDF, NEWTON, SV, &reltol, abstol, NULL, NULL, FALSE, iopt, ropt, NULL);*

| Argumento | Significado |
|---|---|
| *NumChem* | Número de ecuaciones que determinan el tamaño del sistema *EDO*. Su valor es 11. |
| *F* | Definición de la función del usuario $\dot{y} = f(t, y)$ (lado derecho). (***Véase Tabla 4.2.4 Argumentos usados en la llamada a la rutina f.).*** |
| *T0* | Valor inicial del tiempo. Su valor inicial es 0. |
| *Y* | Vector de los valores iniciales de la variable dependiente, en este caso el valor de *c1, c2, ..., c11*. |

| | |
|---|---|
| | C0[1] = 0.00954;<br>C0[2] = 0.042;<br>C0[3] = 0.01951;<br>C0[4] = 0.0;<br>C0[5] = 0.17192;<br>C0[6] = 0.0;<br>C0[7] = 0.0;<br>C0[8] = 0.17192;<br>C0[9] = 0.0;<br>C0[10] = 0.0;<br>C0[11] = 0.0; |
| *BDF* | Tipo del método lineal multipaso. El valor debe ser *ADAMS o BDF.* |
| *NEWTON* | Tipo de iteración empleada para resolver el sistema no lineal. Los valores permitidos son *FUNCTIONAL* o *NEWTON.* |
| *SV* | Tipo de tolerancia de error. El valor es *SV* (tolerancia relativa escalar y tolerancia absoluta en *vector*) |
| *&reltol* | Puntero de tolerancia relativa escalar. El valor es 1.0e-3. |
| *abstol* | Puntero de tolerancia absoluta escalar o de vector. El valor es 1.0e-6. |
| *NULL* | Puntero de datos del usuario. Este puntero es enviado a la función *f* cada vez que es llamada. El valor es nulo. |
| *NULL* | El emplear *NULL* los mensajes de error serán escritos en la biblioteca *stdout.* |
| *FALSE* | El valor *FALSE* indica que no existen entradas opcionales. |
| *iopt* | Arreglo opcional que almacena números enteros de entrada y salida. |
| *ropt* | Arreglo opcional que almacena números reales de entrada y salida. |
| *NULL* | Es un puntero a información específica del ambiente de la máquina. |

Tabla 4.2.1 Argumentos usados en la llamada a *CVodeMalloc*

## 2. *CVode( )*, integración de la solución en varios puntos

*CVode* es una función que se encarga de calcular la solución del sistema *EDO* en varios puntos específicos *t*. Por lo cual, es necesario incluirla en un ciclo externo que integre cada función y en otro ciclo interno para resolver cada ecuación en varios puntos llamados *t*, los cuáles serán parte del resultado junto con los valores de las concentraciones, de acuerdo con el modelo de 11 substancias de *Seinfeld.*

*CVode* es una función ejecutada en el segundo hilo del sistema de cómputo integral, sincronizada con los mecanismos de operación y control de la *VCL* de *Borland C++ Bullder.*

*flag = CVode(cvode_mem, tout, y, &t, NORMAL);*

| **Argumento** | **Significado** |
|---|---|
| *cvode_mem* | Puntero de memoria de *CVODE* regresado por la rutina *CVodeMalloc* |
| *tout* | Tiempo siguiente en el cual se busca una solución. Es calculado antes de la llamada a *CVode* a través de la operación *T0+i*dt*, donde *T0* es el tiempo donde se inicia, *i* toma el valor de 1 hasta el valor de *NOUT* (número de soluciones por mostrar) y *dt* es obtenida por *((TMAX)-(T0))/NOUT.* (*TMAX* es el valor máximo del tiempo donde se busca una solución) |
| *Y* | Vector de la solución. En este caso el valor que toman cada una de las variables *c1, c2, ..., c11.* |
| *T* | Puntero a una localidad real. *CVode* asigna el valor de (**t*) de acuerdo al tiempo alcanzado por el resolvedor y establece *y=y(*t).* |
| *NORMAL* | Si este valor es *NORMAL* el resolvedor integra desde el valor interno de t hasta otro punto cercano o hasta alcanzar el valor dado por *tout*, en este proceso se interpola *t=tout* y es regresado el valor *y(tout)* y almacenado en el vector *y.* |

Tabla 4.2.2 Argumentos usados en la llamada a *CVode*

### 3. *CVDense( )*, llamada al resolvedor lineal *dense*

*CVDense(cvode_mem, Jac, NULL);*

| Argumento | Significado |
|---|---|
| *cvode_mem* | Puntero de memoria de *CVODE* regresado por la rutina *CVodeMalloc* |
| *Jac* | Rutina que realiza la aproximación de la matriz *Jacobiana.* (***Véase* Tabla 4.2.5**) |
| *NULL* | Puntero de datos del usuario pasados a la función *Jac* cada vez que es llamada. El valor es nulo. |

Tabla 4.2.3 Argumentos usados en la llamada a *CVDense*

### 4. *CvodeFree( )*, liberación de memoria asignada a *CVODE* para el problema

*CVodeFree(cvode_mem);*

La llamada a la función *CVodeFree* libera la memoria asignada previamente en la función *CVodeMalloc* a través del puntero *cvode_mem*.

También es necesaria la reservación y liberación de memoria para los vectores de tipo *N_Vector* para los datos de entrada de *y* o variables de concentraciones *c1, c2, c3, ..., c11*. Además, se emplea otro vector para los valores de la tolerancia absoluta.

| Reservación | Liberación |
|---|---|
| *y = N_VNew(NumChem, NULL);* | *N_VFree(y);* |
| *abstol = N_VNew(NumChem, NULL);* | *N_VFree(abstol);* |

### Funciones creadas en la aplicación para ser llamadas por *CVODE*

Son dos las funciones creadas en el sistema de cómputo integral, para ser llamadas por *CVODE* e integrarlas a las suyas y obtener la solución del sistema *EDO*. Éstas funciones son *f* y *Jac*.

La función o rutina *f* es empleada como argumento de la función *CVodeMalloc* y tiene como propósito calcular *f(t,y)*, o bien el lado derecho de *f(t, c)* en este caso; donde *t* son los puntos del tiempo o variable independiente; mientras que *c* son los valores de las concentraciones *c1, c2, c3, ..., c11* o variables dependientes. La cabecera de la función *f* es presentada posteriormente y sus argumentos se describen en la **Tabla 4.2.4.**

*static void f(integer N, real t, N_Vector y, N_Vector ydot, void *f_data);*

| **Argumento** | **Significado** |
|---|---|
| *N* | Número de ecuaciones que determinan el tamaño del sistema *EDO*. Su valor es 11. |
| *t* | Valor de la variable independiente *t*. |
| *Y* | Vector de la variable dependiente. |
| *Ydot* | Vector que recibe el resultado de *f(t,y)*. La reservación y liberación de memoria la realiza *CVODE* para este vector. |
| **f_data* | Es un puntero a datos de usuario enviados a *f* cada vez que es llamada. |

Tabla 4.2.4 Argumentos usados en la llamada a la rutina *f*.

Por otro lado, la rutina o función *Jac* es un argumento de la función *CVDense* y obtiene una matriz *dense J*, tomando como punto de partida la matriz *Jacobiana* obtenida en el primer proceso (explicado en el **apartado 4.1 MODELO DE ANÁLISIS SINTÁCTICO**). La cabecera de la función *Jac* es presentada enseguida y sus argumentos se describen brevemente en la **Tabla 4.2.5** .

*static void Jac(integer N, DenseMat J, RhsFn f, void *f_data, real t,*
*N_Vector y, N_Vector fy, N_Vector ewt, real h, real uround,*
*void *jac_data, long int *nfePtr, N_Vector vtemp1,*

*N_Vector vtemp2, N_Vector vtemp3)*

| **Argumento** | **Significado** |
|---|---|
| *N* | Longitud de los argumentos vector |

| | |
|---|---|
| | *(vtemp1, vtemp2 o vtemp3).* |
| *J* | Matriz *dense* donde se guardan los valores de la matriz *Jacobiana* obtenida en el primer proceso (**apartado 4.1 MODELO DE ANÁLISIS SINTÁCTICO**). |
| *F* | Rutina que calcula el lado derecho de las ecuaciones del sistema *EDO.* |
| **f_data* | Puntero a datos de usuario enviados a *f.* Este dato es el mismo transmitido a *CVodeMalloc*, solo que en esta implementación no es necesario su uso y está indicado como *NULL.* |
| *T* | Valor de la actual de la variable independiente t. |
| *Y* | Valor actual del vector de la variable dependiente, llamado el valor predecido de *y(t).* |
| *Fy* | Vector *f(t,y).* |
| *Ewt* | Vector de pesos o magnitudes de error. |
| *H* | Tamaño tentativo del paso en *t.* |
| *Uround* | Unidad de Término de la máquina |
| **jac_data* | Puntero de datos del usuario pasados a la función *Jac* . Este dato es el mismo transmitido a *CVDense*, solo que en esta implementación no es necesario su uso y está indicado como *NULL.* |
| **nfePtr* | Puntero que contiene el número de llamadas a *f.* |
| *vtemp1, vtemp2 y vtemp3* | Puntero a memoria reservada para vectores de longitud *N (indicada en el primer argumento)*, la cuál puede ser usada por la rutina *Jac* para almacenamiento temporal o espacio de trabajo. |

Tabla 4.2.5 Argumentos usados en la llamada a la rutina *Jac.*

El segundo proceso termina al obtener el valor de las variables dependientes *c1, c2, c3, ..., c11*, evaluadas en varios puntos de tiempo *t* desde un valor inicial hasta

la solución del sistema *EDO* completo; pudiendo entonces, continuar con el tercer proceso donde se grafican los resultados haciendo uso de herramientas de programación visual y multihilos.

## 4.3. MULTIHILOS Y GRAFICACIÓN

En el sistema de cómputo integral se combinaron las características de *Borland Builder C++* para crear una aplicación multihilos, con las herramientas de programación visual disponibles para la graficación. En la **Figura 4.3.1** se muestra el esquema general de la implementación.

De acuerdo con la **Figura 4.3.1**, el sistema de cómputo integral consta de dos hilos: el primero, es el hilo principal *VCL (Visual Component Library)* predefinido por *Borland Builder C++* y el segundo, es el hilo creado para la graficación de resultados.

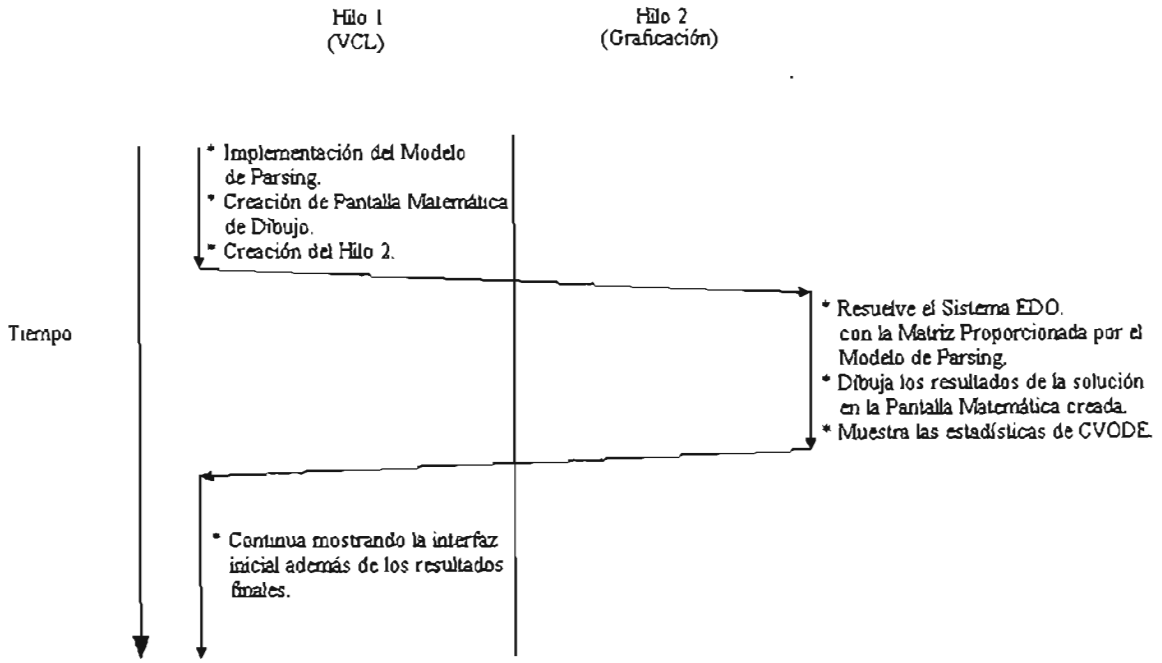



Figura 4.3.1 Esquema general del sistema de cómputo integral con características multihilos.

En el hilo *VCL* es creada y desplegada la interfaz inicial del usuario, donde entre otras cosas; es implementado el modelo de *análisis sintáctico* al analizar el archivo conteniendo el sistema *EDO*; es creada la pantalla matemática de dibujo simulando un plano cartesiano de dos ejes y, es definido el segundo hilo.

En el segundo hilo (hilo de graficación), es resuelto el sistema *EDO* empleando la matriz *Jacobiana* proporcionada por el modelo de *análisis sintáctico*, siendo éste ejecutado sobre el hilo *VCL*; son dibujados los valores obtenidos para las concentraciones representado la solución al problema en el sistema *EDO*, coordinando la ejecución del hilo a través del método *Synchronize* al momento de dibujar los resultados sobre la pantalla matemática creada anteriormente y, finalmente muestra las estadísticas generadas por el paquete *CVODE*.

La interfaz del usuario está presente durante la ejecución del hilo *VCL* y no cesa su despliegue en la ejecución del hilo de graficación. Al final de la ejecución multihilos (o concurrente), el hilo *VCL* permite mostrar y conservar la interfaz inicial junto con los resultados generados por el hilo de graficación.

La pantalla matemática simula un plano cartesiano, donde el eje de las *Xs* representa el tiempo en segundos y el eje de las *Ys* representa el valor de las 11 concentraciones del modelo de *Seinfeld*. Para su creación, fue empleado el componente *PaintBox* para establecer los límites y a su vez definir el área o lienzo de dibujo; las características del fondo y las líneas son dibujadas con la propiedad *Pen* de la clase *TCanvas* a través del método *LineTo*; el texto sobre la imagen es dibujado con la propiedad *Font* de la clase *TCanvas*, siendo necesario primero definir el valor del estilo, el tamaño, el nombre, etc. del tipo de letra a mostrar para cada concepto de la imagen.

Las propiedades y métodos de *Borland C++ Builder* empleadas en el sistema de cómputo integral desarrollado en esta tesis; con respecto al manejo multihilos y graficación, son mostradas a continuación en la **Tabla 4.3.1 Funciones empleadas de *Borland C++ Builder***.

| ***MULTIHILOS*** | ***FUNCIÓN DONDE SE USA*** | ***ARCHIVO*** |
|---|---|---|
| **Propiedad** | | |
| FreeOnTerminate | GrThread() | VenHilo1.cpp |
| Terminated | RevTerminac() | VenHilo1.cpp<br>VenHiloDibujar1.cpp |
| **Método** | | |
| Execute() | Execute() | VenHilo1.cpp |
| *Synchronize*() | MuestraLin() | VenHilo1.cpp<br>VenHiloDibujar1.cpp |

| ***GRAFICACIÓN*** | ***FUNCIÓN DONDE SE USA*** | ***ARCHIVO*** |
|---|---|---|
| **Propiedad** | | |
| Font | ConfiFontPar() | Dibujos.cpp |
| Brush | ColorLienzo() | Dibujos.cpp |
| Pen | ColorLienzo()<br>DibujaLinea()<br>DibujaLineaEn()<br>WrNumVert()<br>WrNumHor()<br>VentanaMate()<br>DibujaCuadro() | Dibujos.cpp |
| **Método** | | |
| Draw | PintaVenMate()<br>PintaVentanaMate ()<br>PintaGrLienzo99() | Principal.cpp |
| LineTo | DibujaLinea()<br>DibujaLineaEn()<br>WrNumVert()<br>WrNumHor() | Dibujos.cpp |
| MoveTo | DibujaLinea()<br>MuevePixelA ()<br>WrNumVert()<br>WrNumHor() | Dibujos.cpp<br>VenHilo1.cpp |
| Rectangle | ColorLienzo()<br>DibujaCuadro() | Dibujos.cpp |
| TextHeight | WrNumVert()<br>VentanaMate () | Dibujos.cpp |
| TextWidht | WrNumVert()<br>WrNumHor()<br>VentanaMate () | Dibujos.cpp |
| TextOut | WrNumVert()<br>WrNumHor()<br>VentanaMate () | Dibujos.cpp |

Tabla 4.3.1 Funciones empleadas de *Borland C++ Builder*

## 4.4. DESCRIPCIÓN DE LA INTERFAZ DEL USUARIO

El sistema de cómputo integral es ejecutado con el archivo *SisParser.exe*. La primera vista a la interfaz del usuario es la siguiente:

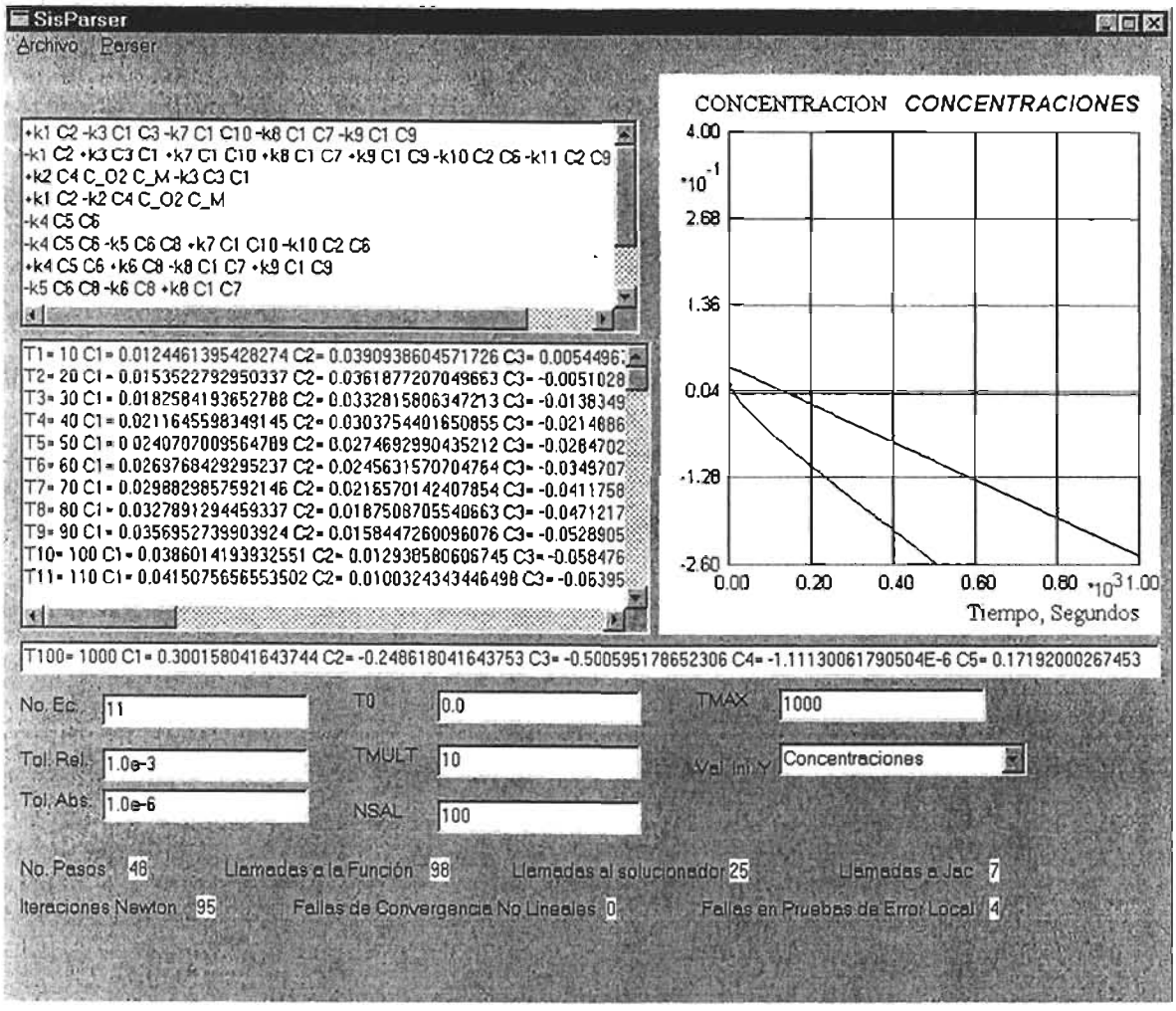



Figura 4.4.1 Interfaz del Sistema de Cómputo Integral Desarrollado

El sistema de cómputo integral consta de cuatro áreas. La primer área despliega el sistema EDO, proporcionado en un *archivo de texto*, en el cuál se encuentran los coeficientes indicados por *Ks* y las concentraciones por *Cs* **(véase Figura 4.4.1 Interfaz del Sistema de Cómputo Integral Desarrollado)**. Cabe destacar, la creación del archivo de texto, en la cual los términos deben seguir un formato como el mostrado en la **Figura 4.4.1**, presionando en cada ecuación la tecla Enter para marcar el fin de la ecuación y deberá ser guardado como *archivo de texto*.

La segunda área, despliega la solución del sistema *EDO*, mostrando el valor de cada una de las concentraciones del Modelo de 11 substancias de *Seinfeld* para ciertos valores en *T* (*Tiempo*).

La tercer área, muestra gráficamente los resultados de algunas concentraciones que resuelven el sistema *EDO* proporcionado originalmente. Y finalmente, en la cuarta área, están señalados algunos datos informativos de entrada y otros sobre la estadística de desempeño durante el proceso de cálculo realizado por *CVODE*.

### 4.4.1 Operación del Sistema Cómputo Integral

La operación del sistema de cómputo integral debe ser con cierto procedimiento. El procedimiento está formado por dos partes principales:

- *Análisis Sintáctico o Parsing*
- Graficación

En una parte del procedemiento se lleva a cabo la implementación del modelo de *análisis sintáctico* en sus tres fases: *análisis lexicográfico*, *análisis sintáctico* y *análisis semántico*. Por lo anterior, es necesario abrir un *archivo de texto* conteniendo las ecuaciones del sistema *EDO* a evaluar o resolver. Cuando el archivo es abierto desde el sistema de cómputo integral, su contenido es desplegado en la pantalla de la aplicación como se muestra en la **Figura 4.4.2 Despliegue del sistema EDO por analizar**.

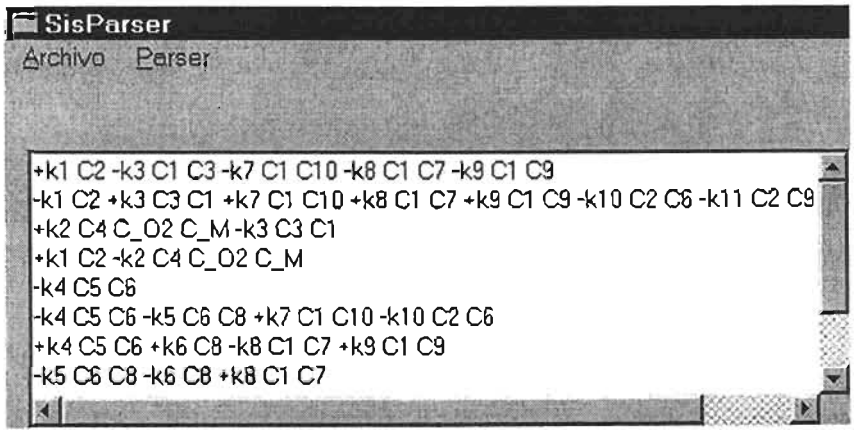



Figura 4.4.2 Despliegue del sistema *EDO* por analizar

El sistema de cómputo integral aplica el modelo de 11 substancias de *Seinfeld* para su ejecución. La **Figura 4.4.2 Despliegue del sistema *EDO* por analizar**, muestra las substancias o concentraciones involucradas en reacciones químicas del medio ambiente donde ocurre contaminación atmosférica, indicadas por la letra inicial *C*, las cuáles presentan la siguiente relación:

c1 - NO
c2 - NO2
c3 - O3
c4 – O
c5 – RH
c6 - OH
c7 – RO2
c8 - RCHO
c9 – RC[O]O2
C10 – HO2
C11 – RC[O]O2NO2

Una vez desplegado el sistema EDO en pantalla, a continuación es realizado el análisis completo del archivo (basado en el modelo planteado en el **apartado 4.1 MODELO DE ANÁLISIS SINTÁCTICO**), esto a través del menú; en la opción *Parser* luego, la opción *Analizar.* Si el sistema *EDO* es correcto, se crea la matriz *Jacobiana* y se puede proceder a la graficación por la opción del menú *Parser,* seguida de la opción *Graficar.* La pantalla muestra enseguida los resultados obtenidos como en la **Figura 4.4.3 Graficación de resultados de algunas concentraciones**.

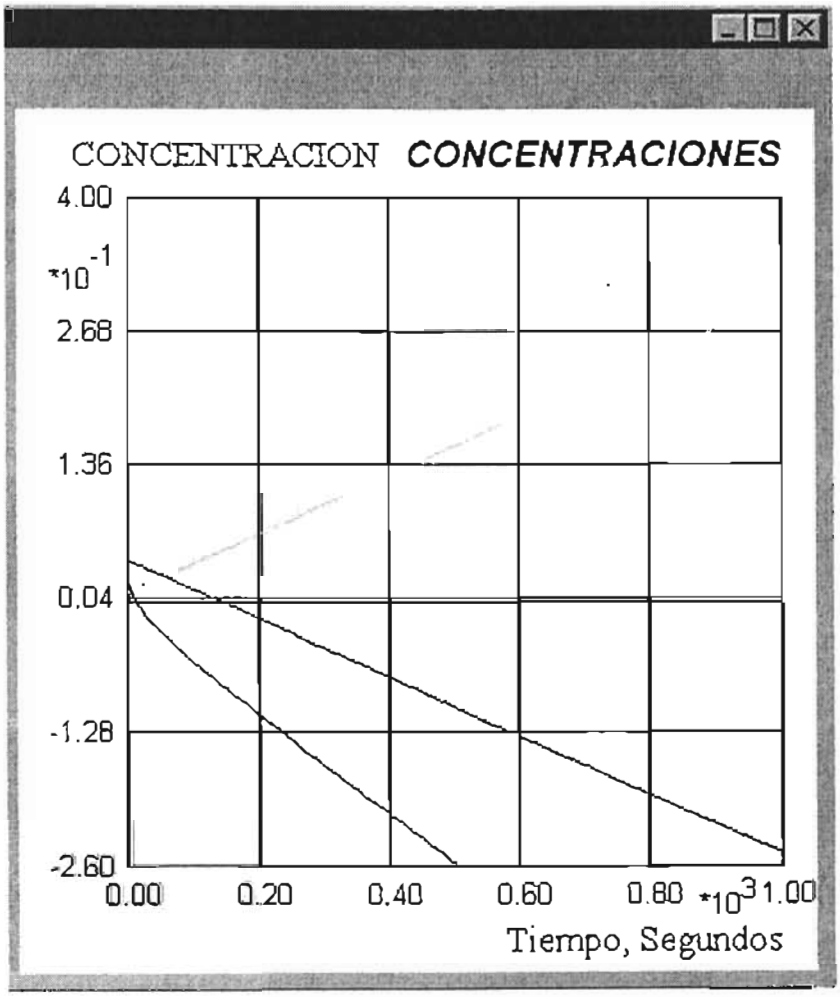



Figura 4.4.3 Graficación de resultados de algunas concentraciones

Conforme *CVODE* resuelve el sistema *EDO* especificado, a través de la tarea (*hilo1*), simultáneamente es efectuada la tarea (*hilo2*) de graficación de la **Figura 4.4.3**. La tarea de cálculo por el resolvedor *Dense* de *CVODE*, pone a disposición de la tarea de graficación los resultados, para que puedan ser mostrados aún sin completar se objetivo. La **Figura 4.4.4 Valores del modelo de 11 substancias de *Seinfeld* por *Dense* de *CVODE***, muestra los valores obtenidos por el resolvedor *Dense* de *CVODE*.

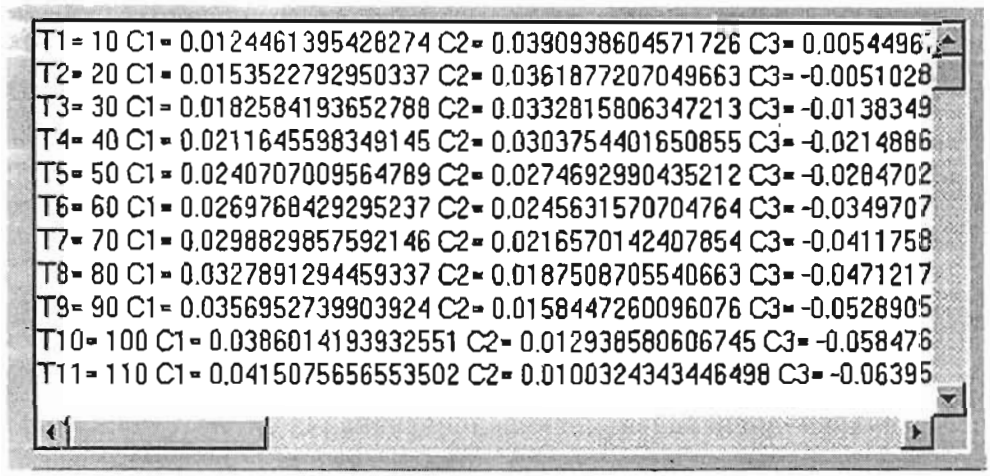



Figura 4.4.4 Valores del modelo de 11 substancias de *Seinfeld* por *Dense* de *CVODE*

Adicionalmente, se presentan en la interfaz del sistema de cómputo integral de esta tesis, algunos datos informativos de entrada y otros más que representan estadísticas del desempeño de los cálculos con el resolvedor *Dense* de *CVODE* (***véase* Figura 4.4.5 Datos de Entrada y Datos Estadísticos de Desempeño de *CVODE*).**

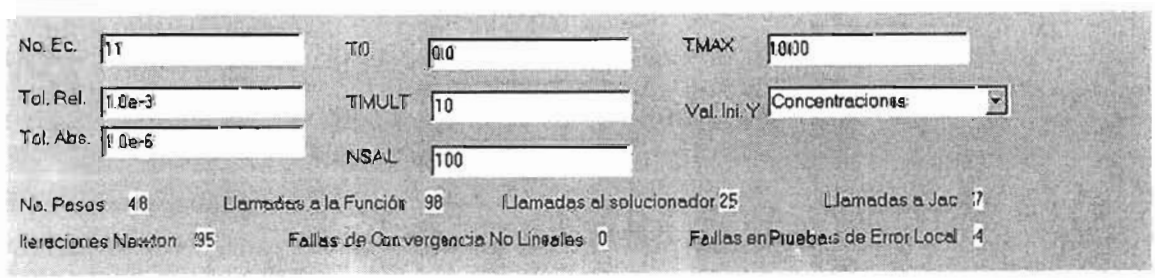



Figura 4.4.5 Datos de Entrada y Datos Estadísticos de Desempeño de *CVODE*

El sistema de cómputo integral desarrollado en esta tesis, aplica la teoría de *análisis sintáctico (parsing en inglés)* frecuentemente empleada en el diseño de compiladores de programas de computación. Dicho sistema, integra un *analizador sintáctico*, el paquete computacional *CVODE* y una interfaz visual para resolver un sistema *EDO* y mostrar gráficamente su solución. Fue desarrollado con técnicas de programamación orientada a objetos de C++, haciendo uso de la *STL(Standard Template Library)* integrada al estándar de *C++* y utilizando herramientas de programación visual y multihilos proporcionadas por el *Borland C++ Builder*.

# CONCLUSIONES

Se ha diseñado y desarrollado en esta tesis, un sistema de cómputo integral caracterizado por la aplicación de la teoría de *análisis sintáctico (parsing en inglés)*, para el cálculo analítico y exacto de la matriz *Jacobiana*, de acuerdo al modelo matemático de reacciones químicas del medio ambiente (modelo de 11 substancias de *Seinfeld*) en lo que toca a contaminación atmosférica.

Aunado a lo anterior, se implementó el paquete computacional *CVODE* gratuito y de libre distribución, por requerirse de cálculos numéricos complejos que proporcionen alta confiabilidad en los resultados, apegado a las características de un sistema *EDO (Ecuaciones Diferenciales Ordinarias)* rígido, como lo son las relacionadas con problemas del medio ambiente en la contaminación atmosférica y por contar con bibliotecas compatibles con *C++*.

Asimismo, el sistema de cómputo integral fue desarrollado con tecnología reconocida y de vanguardia, como lo son el uso de técnicas de programación orientada a objetos, el empleo de la *STL (Standard Template Library)* como estándar de *C++*, la implementación de programación concurrente o programación multihilos, así como también; el diseño de la interfaz del usuario con herramientas visuales.

Al hacer uso del lenguaje de programación *C++* y la *STL*, específicamente de los contenedores *vector, map y string* en esta tesis, se obtiene portabilidad, compatibilidad y uso eficiente de la memoria en varias plataformas de sistema operativo. Aunque el sistema de cómputo integral fue desarrollado para operar en sistemas *Windows*, cuenta con un diseño versátil para poder adecuarlo y usarlo en algún otro sistema.

Cabe destacar, la implementación de programación concurrente en esta tesis, al permitir mostrar los resultados gráficamente en tiempo de ejecución, en un solo programa, sin depender en gran medida de la disposición de los datos de la solución y poderlos a su vez graficar conforme se van generando. Por lo anterior, se espera mayor rendimiento del procesador donde sea ejecutado y poca demora en la apreciación de los resultados; ya sea por los valores de éstos o por la graficación de los mismos.

Con el sistema de cómputo integral desarrollado en esta tesis, se busca aportar a la simulación computacional de problemas del medio ambiente en contaminación atmosférica, la resolución de sistemas *EDO* rígidos que a pesar de su tamaño no

representen problema alguno de computabilidad y confiabilidad. Evitando realizar las operaciones manualmente con susceptibilidad de error, tener que emplear varias herramientas de software para los cálculos y el análisis de los resultados, y a medida que el tamaño del sistema *EDO* crece, poder garantizar la obtención de resultados por la eficiencia de la programación concurrente o multihilos.

Es también importante mencionar, la integración del trabajo de esta tesis a la investigación de la *FES Cuautitlán-UNAM*, por estar vinculada al proyecto *PAPIIT No. 105401 "Modelación matemática de sistemas multifásicos y multicomponentes en problemas del medio ambiente"*.

# APÉNDICE A: FUNCIONES PRINCIPALES DE CVODE

Son cuatro las funciones principales de la interfaz de usuario del paquete *CVODE*. A continuación se explican las especificaciones de cada una.

1. *CVodeMalloc*(...) recibe el problema y las especificaciones del método, además de hacer la reservación de la memoria necesaria. Sus argumentos son las cantidades definidas por el problema (*N, f,* $t_0$, *y*), opciones del método lineal multipaso *(Adams-Moulton o BDF)* para la integración, método de iteración *(funcional o Newton)*, tres argumentos para la tolerancia de error, puntero para datos del usuario, puntero para errores de archivo, bandera de entrada opcional, arreglo opcional de entrada de números enteros y, arreglo opcional de salida de números reales **(véase Tabla A. 1 Argumentos usados en la cabecera de *CVodeMalloc*)**. *CVodeMalloc* regresa un puntero a un bloque de memoria de *CVODE* (bloque empleado para el estado de los datos y los punteros de *CVODE*) y su cabecera es como se muestra a continuación:

*void *CVodeMalloc(integer N, RhsFn f, real to, N_Vector y0, int lmm, int Iter, int itol, real *reltol, void *abstol, void *f_data, FILE *errfp, bool optIn, long int iopt[], real rpot[], void *machEnv)*

| **Argumento** | **Significado** |
|---|---|
| *N* | Número de ecuaciones que determinan el tamaño del sistema *EDO* |
| *F* | Definición de la función del usuario $\dot{y} = f(t, y)$ (lado derecho). |
| *t0* | Valor inicial del tiempo |
| *y0* | Vector de los valores iniciales de la variable dependiente |
| *Lmm* | Tipo del método lineal multipaso. Los valores permitidos son *ADAMS* o *BDF*. |
| *Iter* | Tipo de iteración empleada para resolver el sistema no lineal. Los valores permitidos son *FUNCTIONAL* o *NEWTON*. |
| *Itol* | Tipo de tolerancia de error. Los valores permitidos son: *SS* (tolerancia relativa y absoluta escalares) y *SV* (tolerancia relativa escalar y tolerancia absoluta en |

| | |
|---|---|
| | vector) |
| *Reltol* | Puntero de tolerancia relativa escalar |
| *abstol* | Puntero de tolerancia absoluta escalar o de vector |
| *f_data* | Puntero de datos del usuario. Este puntero es enviado a la función *f* cada vez que es llamada. |
| *errfp* | Puntero de archivo de errores. En el archivo de errores se escriben todos los mensajes de alerta y de error producidos por *CVODE*. Este argumento puede ser *stdout* (salida estándar), *stderr* (salida estándar de errores), un puntero de archivo (correspondiente a un archivo del usuario para errores y abierto para escritura) obtenido por *fopen,* o bien; puede ser *NULL*. Si el usuario emplea *NULL* los mensajes serán escritos a *stdout*. |
| *optIn* | Bandera que sirve para indicar si existen datos opcionales de entrada proporcionados por el usuario en los arreglos *iopt* y *ropt*. El valor *FALSE* indica que no existen entradas opcionales y *TRUE* significa que sí. |
| *iopt* | Arreglo opcional que almacena números enteros de entrada y salida. Si el valor de este argumento es *NULL* significa que no se desea usar el arreglo. Si *opIn* es *TRUE* el arreglo *iopt* debe inicializarse con *0* en todas sus localidades. |
| *ropt* | Arreglo opcional que almacena números reales de entrada y salida. Si el valor de este argumento es *NULL* significa que no se desea usar el arreglo. Si *optIn* es *TRUE* el arreglo *ropt* debe inicializarse con *0.0* en todas sus localidades. |
| *machEnv* | Es un puntero a información específica del ambiente de la máquina. |

Tabla A. 1 Argumentos usados en la cabecera de *CVodeMalloc*

La función *f* debe ser del tipo *RhsFn*. La función *f* necesita datos como el tamaño del problema *N*, el valor de la variable dependiente *t* y el vector de la variable dependiente *y*, para luego almacenar el resultado de *f(t,y)* en el vector *ydot* (la reservación de la memoria para el vector *ydot* la realiza *CVODE*). El parámetro *f_data* es el mismo que el parámetro de *CVodeMalloc*. *CVODE* define *RhsFn* como sigue:

*typedef void(*RhsFn)(integer N, real t, N_Vector y, N_Vector ydot, void *f_data)*

2. Una rutina determina el resolvedor lineal empleado en la iteración de Newton, la rutina puede ser *CVDense, CVBand, CVDiag o CVSpgmr*, las dos primeras emplean una rutina *Jac* elaborada por el usuario. A continuación se muestran las cabeceras correspondientes a cada resolvedor:

*void CVDense(void *cvode_mem, CVDenseJacFn dJac, void *jac_data);*

*void CVBand(void *cvode_mem, integer mupper, integer mlower, CVBandJacFn bjac, void *jac_data);*

*void CVDiag(void *cvode_mem);*

*void CVSpgmr(void *cvode_mem, int pretype, int gstype, int max1, real delt, CVSpgmrPrecondFn precond, CVSpgmrPSolveFn psolve, void *P_data);*

Una sencilla explicación de las rutinas *CVDense, CVBand y CVSpgmr* se encuentra en la **Tabla A. 2, Tabla A. 3** *y* **Tabla A. 4** respectivamente.

| **Argumento** | **Significado** |
|---|---|
| *cvode_mem* | Puntero de memoria de *CVODE* regresado por la rutina *CVodeMalloc* |
| *Djac* | Rutina que realiza la aproximación de la matriz *Jacobiana*. *djac* debe ser del tipo *CVDenseJacFn* definido en *CVODE*. Si se indica *NULL*, significa que se empleará la rutina *Jac* del tipo *CVDenseDQJac* proporcionada por el resolvedor *Dense*. |
| *jac_data* | Puntero de datos del usuario pasados a la función *djac* cada vez que es llamada. |

Tabla A. 2 Argumentos usados en la cabecera de *CVDense*

| **Argumento** | **Significado** |
|---|---|
| *cvode_mem* | Puntero de memoria de *CVODE* regresado por la rutina *CVodeMalloc* |
| *Mupper* | Valor superior del grupo de valores de los cuales se pretende obtener la aproximación de la matriz *Jacobiana* |
| *Mlower* | Valor inferior del grupo de valores de los cuales se pretende obtener la aproximación de la matriz *Jacobiana* |
| *Bjac* | Rutina que realiza la aproximación de la matriz *Jacobiana*. *bjac* debe ser del tipo *CVBandJacFn* definido en *CVODE*. Si se indica *NULL* significa que se empleará la rutina *Jac* del tipo *CVBandDQJac* proporcionada por el resolvedor *Band*. |
| *jac_data* | Puntero de datos del usuario pasados a la función *bjac* cada vez que es llamada. |

Tabla A. 3 Argumentos usados en la cabecera de *CVBand*

| **Argumento** | **Significado** |
|---|---|
| *cvode_mem* | Puntero de memoria de *CVODE* regresado por la rutina *CVodeMalloc* |
| *Pretype* | Tipo de la precondición. Puede tomar uno de los cuatro valores de constantes enumeradas: *NONE, LEFT, RIGHT o BOTH* |
| *Gstype* | Tipo de ortogonalización de *Gram-Schmidt*. Los valores permitidos pueden ser: *MODIFIED_GS o CLASSICAL_GS*. |
| *max1* | Dimensión máxima de *Krylov*. Este valor es opcional, se puede indicar 0 para tomar el valor por omisión |
| *Delt* | Factor por el cual la tolerancia, en una iteración no lineal, es multiplicada para obtener una tolerancia de una iteración lineal. Este valor es opcional, se puede indicar *0* para tomar el valor por omisión |
| *Precond* | Rutina de precondición del usuario. |

| | |
|---|---|
| | Indicar *NULL* si no se requiere de una configuración particular de los datos de la matriz *Jacobiana*. |
| *Psolve* | Rutina de precondición del usuario. El único caso donde se puede indicar *NULL* para este valor es cuando *pretype* es *NONE*. |
| *p_data* | Puntero a los datos de la precondición del usuario. Este puntero es enviado a *precond* y *psolve* cada vez que son llamadas estas rutinas. |

Tabla A. 4 Argumentos usados en la cabecera de *CVSpgmr*

3. *CVode* (...) se encarga de la integración, regresa una bandera de fin o terminación y el vector con la solución en *yout*. Si *itask=NORMAL*, *Paso* se incrementa tomando otro valor o hasta alcanzar el valor de *tout*, aumentando e interpolando valores hasta obtener la solución en *yout*. Si *itask=ONE_STEP* significa que el proceso es efectuado en un solo paso para llegar a *tout* y el control de la ejecución se regresa a la rutina de llamada. *CVode* usa la memoria proporcionada por *CVodeMalloc* para saber su estado actual. La cabecera de *CVode* es la siguiente:

*int CVode (void *cvode_mem, real tout, N_Vector yout, real *t, int itask);*

La **Tabla A. 5** muestra el significado de la rutina *CVode*.

| **Argumento** | **Significado** |
|---|---|
| *cvode_mem* | Puntero de memoria de *CVODE* regresado por la rutina *CVodeMalloc* |
| *Tout* | Tiempo siguiente en el cual se busca una solución |
| *Yout* | Vector de la solución |
| *T* | Puntero a una localidad real. *CVode* asigna el valor de (*t) de acuerdo al tiempo alcanzado por el resolvedor y establece yout=y(*t) |
| *Itask* | Si este valor es *NORMAL* el resolvedor integra desde el valor interno de t hasta otro punto cercano o hasta alcanzar el valor dado por *tout*, en este proceso se interpola *t=tout* y es regresado el valor *y(tout)* y almacenado en el vector yout. Por otra |

| | |
|---|---|
| | parte, si este valor es *ONE_STEP* el resolvedor toma un paso de tiempo interno y regresa en *yout* el valor de *y* en el nuevo tiempo interno; en este caso, *tout* es usado solamente durante la primer llamada a *CVode* para determinar la dirección de integración y la escala de rigidez del problema. |

Tabla A. 5 Argumentos usados en la cabecera de *CVode*

4. *CVodeFree* libera la memoria reservada por *CVodeMalloc*, su cabecera es la siguiente:

*void CVodeFree (void *cvode_mem);*

El programa del usuario debe crear el vector de las variables dependientes *y* y asignarle los valores iniciales. De cualquier modo, el módulo *VECTOR* de *CVODE* incluye una rutina llamada *N_VNew* para la reservación de memoria de un vector *N_Vector* y otra más *N_VFree* para liberar la memoria de ese vector.

# APÉNDICE B: EJEMPLO DE ARCHIVO DE TEXTO CON UN SISTEMA EDO

Las siguientes líneas son un ejemplo de un *archivo de texto*, para resolver un sistema *EDO* en *PVI*, correspondiente al modelo matemático de 11 substancias de *Seinfeld,* en reacciones químicas del medio ambiente:

```
+k1 C2 -k3 C1 C3 -k7 C1 C10 -k8 C1 C7 -k9 C1 C9
-k1 C2 +k3 C3 C1 +k7 C1 C10 +k8 C1 C7 +k9 C1 C9 -k10 C2 C6 -k11 C2 C9
+k12 C11
+k2 C4 C_O2 C_M -k3 C3 C1
+k1 C2 -k2 C4 C_O2 C_M
-k4 C5 C6
-k4 C5 C6 -k5 C6 C8 +k7 C1 C10 -k10 C2 C6
+k4 C5 C6 +k6 C8 -k8 C1 C7 +k9 C1 C9
-k5 C6 C8 -k6 C8 +k8 C1 C7
+k5 C6 C8 -k9 C1 C9 -k11 C2 C9 +k12 C11
+k6 C8 -k7 C1 C10 +k8 C1 C7
+k11 C2 C9 -k12 C11
```

Podemos destacar las siguientes características del sistema EDO mostrado:

- Cada término debe comenzar con el signo + ó -.
- La letra *k* indica un valor constante predefinido. El sistema de cómputo integral también puede realizar los cálculos con valores numéricos en lugar de las *k*.
- Las 11 substancias o concentraciones son representadas con la letra inicial *C*. Adicionalmente, se pueden indicar potencias con valores enteros de cada una de las concentraciones. Por ejemplo, *C1#2* indica que la concentración *C1*, debe ser elevada a la potencia 2.
- Al crear el *archivo de texto*, es importante oprimir la tecla *Enter* para cada una de las ecuaciones, lo que señalará al sistema de cómputo integral el fin de cada ecuación del sistema EDO.

El valor de los coeficientes constantes k, se muestra en la siguiente tabla:

| **Coeficiente Constante** | **Unidades** | **Aproximación de unidades** |
|---|---|---|
| *K1* | 0.533/60.0 | 0.008883333 |
| *K2* | 2.183e-5/60.0 | 0.000000363 |
| *K3* | 26.59/60.0 | 0.443166666 |
| *K4* | 3.775e3/60.0 | 62.91666667 |
| *K5* | 2.341e4/60.0 | 390.1666667 |
| *K6* | 1.91e-4/60.0 | 0.000003183 |
| *K7* | 1.214e4/60.0 | 202.3333333 |
| *K8* | 1.127e4/60.0 | 187.8333333 |
| *K9* | 1.127e4/60.0 | 187.8333333 |
| *K10* | 1.613e4/60.0 | 268.8333333 |
| *K11* | 6.893e3/60.0 | 114.8833333 |
| *K12* | 2.143e-2/60.0 | 0.000357166 |

# APÉNDICE C: EJEMPLO DE ARCHIVO CONTENIENDO LA MATRIZ JACOBIANA

A continuación se presenta el contenido del archivo *Jacob.inc*, creado como resultado del primero de los tres procesos considerados en el sistema de cómputo integral, proceso en el cuál se implementa el modelo del **apartado 4.1 MODELO DE ANÁLISIS SINTÁCTICO**.

```
IJth(J,1,1)=-k3*c[3]-k7*c[10]-k8*c[7]-k9*c[9];
IJth(J,1,2)=k1;
IJth(J,1,3)=-k3*c[1];
IJth(J,1,4)=0;
IJth(J,1,5)=0;
IJth(J,1,6)=0;
IJth(J,1,7)=-k8*c[1];
IJth(J,1,8)=0;
IJth(J,1,9)=-k9*c[1];
IJth(J,1,10)=-k7*c[1];
IJth(J,1,11)=0;

IJth(J,2,1)=k3*c[3]+k7*c[10]+k8*c[7]+k9*c[9];
IJth(J,2,2)=-k1-k10*c[6]-k11*c[9];
IJth(J,2,3)=k3*c[1];
IJth(J,2,4)=0;
IJth(J,2,5)=0;
IJth(J,2,6)=-k10*c[2];
IJth(J,2,7)=k8*c[1];
IJth(J,2,8)=0;
IJth(J,2,9)=k9*c[1]-k11*c[2];
IJth(J,2,10)=k7*c[1];
IJth(J,2,11)=k12;


IJth(J,3,1)=-k3*c[3];
IJth(J,3,2)=0;
IJth(J,3,3)=-k3*c[1];
IJth(J,3,4)=k2*c_M*c_O2;
IJth(J,3,5)=0;
IJth(J,3,6)=0;
IJth(J,3,7)=0;
```

```
IJth(J,3,8)=0;
IJth(J,3,9)=0;
IJth(J,3,10)=0;
IJth(J,3,11)=0;

IJth(J,4,1)=0;
IJth(J,4,2)=k1*c_M*c_O2;
IJth(J,4,3)=0;
IJth(J,4,4)=-k2*c_M*c_O2;
IJth(J,4,5)=0;
IJth(J,4,6)=0;
IJth(J,4,7)=0;
IJth(J,4,8)=0;
IJth(J,4,9)=0;
IJth(J,4,10)=0;
IJth(J,4,11)=0;

IJth(J,5,1)=0;
IJth(J,5,2)=0;
IJth(J,5,3)=0;
IJth(J,5,4)=0;
IJth(J,5,5)=-k4*c_M*c_O2*c[6];
IJth(J,5,6)=-k4*c_M*c_O2*c[5];
IJth(J,5,7)=0;
IJth(J,5,8)=0;
IJth(J,5,9)=0;
IJth(J,5,10)=0;
IJth(J,5,11)=0;

IJth(J,6,1)=k7*c_M*c_O2*c[10];
IJth(J,6,2)=-k10*c_M*c_O2*c[6];
IJth(J,6,3)=0;
IJth(J,6,4)=0;
IJth(J,6,5)=-k4*c_M*c_O2*c[6];
IJth(J,6,6)=-k4*c_M*c_O2*c[5]-k5*c_M*c_O2*c[8]-k10*c_M*c_O2*c[2];
IJth(J,6,7)=0;
IJth(J,6,8)=-k5*c_M*c_O2*c[6];
IJth(J,6,9)=0;
IJth(J,6,10)=k7*c_M*c_O2*c[1];
IJth(J,6,11)=0;

IJth(J,7,1)=-k8*c_M*c_O2*c[7]+k9*c_M*c_O2*c[9];
IJth(J,7,2)=0;
IJth(J,7,3)=0;
```

```
IJth(J,7,4)=0;
IJth(J,7,5)=k4*c_M*c_O2*c[6];
IJth(J,7,6)=k4*c_M*c_O2*c[5];
IJth(J,7,7)=-k8*c_M*c_O2*c[1];
IJth(J,7,8)=k6*c_M*c_O2;
IJth(J,7,9)=k9*c_M*c_O2*c[1];
IJth(J,7,10)=0;
IJth(J,7,11)=0;

IJth(J,8,1)=k8*c_M*c_O2*c[7];
IJth(J,8,2)=0;
IJth(J,8,3)=0;
IJth(J,8,4)=0;
IJth(J,8,5)=0;
IJth(J,8,6)=-k5*c_M*c_O2*c[8];
IJth(J,8,7)=k8*c_M*c_O2*c[1];
IJth(J,8,8)=-k5*c_M*c_O2*c[6]-k6*c_M*c_O2;
IJth(J,8,9)=0;
IJth(J,8,10)=0;
IJth(J,8,11)=0;

IJth(J,9,1)=-k9*c_M*c_O2*c[9];
IJth(J,9,2)=-k11*c_M*c_O2*c[9];
IJth(J,9,3)=0;
IJth(J,9,4)=0;
IJth(J,9,5)=0;
IJth(J,9,6)=k5*c_M*c_O2*c[8];
IJth(J,9,7)=0;
IJth(J,9,8)=k5*c_M*c_O2*c[6];
IJth(J,9,9)=-k9*c_M*c_O2*c[1]-k11*c_M*c_O2*c[2];
IJth(J,9,10)=0;
IJth(J,9,11)=k12*c_M*c_O2;

IJth(J,10,1)=-k7*c_M*c_O2*c[10]+k8*c_M*c_O2*c[7];
IJth(J,10,2)=0;
IJth(J,10,3)=0;
IJth(J,10,4)=0;
IJth(J,10,5)=0;
IJth(J,10,6)=0;
IJth(J,10,7)=k8*c_M*c_O2*c[1];
IJth(J,10,8)=k6*c_M*c_O2;
IJth(J,10,9)=0;
IJth(J,10,10)=-k7*c_M*c_O2*c[1];
IJth(J,10,11)=0;
```

```
IJth(J,11,1)=0;
IJth(J,11,2)=k11*c_M*c_O2*c[9];
IJth(J,11,3)=0;
IJth(J,11,4)=0;
IJth(J,11,5)=0;
IJth(J,11,6)=0;
IJth(J,11,7)=0;
IJth(J,11,8)=0;
IJth(J,11,9)=k11*c_M*c_O2*c[2];
IJth(J,11,10)=0;
IJth(J,11,11)=-k12*c_M*c_O2;
```

El nombre de la variable *IJth* corresponde a un vector definido en *CVODE* para almacenar los valores de la matriz *Jacobiana* y continuar con los cálculos correspondientes para la solución del sistema *EDO*.

El sistema de cómputo integral hace uso del resolvedor *Dense* de *CVODE*, especificando explícitamente un función propia con matriz *Jacobiana* requerida en el proceso.

# REFERENCIAS

## BIBLIOGRÁFICAS


1) [ADAMS1991] ADAMS, Lee, Programación avanzada de gráficos interactivos : Modelado, acabado y animacion en 2d y 3d, Trad. Vicente Sosa Trevino, Madrid: Anaya multimedia, 1991, 505 p.

2) [AHO1988] AHO, Alfred V., SETHI, Ravi y ULLMAN, Jefrey D., Compilers. Principles, Techniques and Tools, EUA: Addison-Wesley, 1986, 796 p.

3) [AHO1972] AHO, Alfred V. y ULLMAN, Jeffrey D., The Theory of Parsing, Translation and Compiling, Vol. I: Parsing, Nueva Jersey: Prentice-Hall, 1972.

4) [ANDREWS2000] ANDREWS, Gregory R, Foundations of multithreaded, parallel, and distributed programming, Mexico: Addison-Wesley, 2000, 664 p.

5) [BEVERIDGE1997] BEVERIDGE, Jim y WIENER, Robert, Multithreading applications in Win32 : the complete guide to threads, Reading, Massachusetts: Addison-Wesley, 1997, 368 p.

6) [BRONSON2000] BRONSON, Gary J., C++ para Ingeniería y Ciencias. México: Thomson, 2000, 966 p.

7) [BURDEN2002] BURDEN, Richard L. y FAIRES, J. Douglas, Análisis numérico, Trad. Oscar Palmas Miguel, Mexico: Thomson Learning, 2002, 839 p.

8) [BURDEN1998] BURDEN, Richard L. y FAIRES, J. Douglas, Análisis numérico, Trad. Efren Alatorre Miguel, Mexico: International Thomson, 1998, 802 p.

9) [BUTENHOF1997] BUTENHOF, David R., Programming with POSIX threads, Mexico: Addison-Wesley, 1997, 381 p.

10) [CHARTE1999] CHARTE OJEDA, Francisco, Programacion con C++ Builder 4, Madrid, España: Anaya Multimedia, 1999, 672 p.

11) [CONTE1974] CONTE, Samuel Daniel y DE BOOR, Carl., Análisis numérico elemental: Un enfoque algoritmico, Trad. H. A. Castillo, Mexico: McGraw-Hill, 1974, 418 p.

12) [DONOVAN1986] DONOVAN, John J., Systems Programming, 17a. ed., Singapore: McGrawHill, 1986, 488 p.

13) [GERALD2000] GERALD, Curtis F y Patrick O. WHEATLEY, Análisis numérico con aplicaciones, Trad. Hugo Villagomez Velazquez, 6ª. ed., Mexico: Pearson Educacion, 2000, 698 p.

14) [HARRIS1998] HARRIS, John W. y STOCKER, Horst, Handbook of Mathematics and Computational Science, Nueva York: Springer-Verlag, 1998, 1027 p.

15) [HOLLINGWORTH2001] HOLLINGWORTH, Jarrod, BUTTERFIELD., Dan, SWART, Bob y ALLSOP, Jamie, C++ Builder Developer's Guide, EUA: Sams Publishing, 2001, 2004 p.

16) [HOPCROFT1993] HOPCROFT, John E. y ULLMAN, Jeffrey D., Introducción a la Teoría de Autómatas, Lenguajes Formales y Computación, México: Continental, 1993, 447 p.

17) [HOROWITZ1984] HOROWITZ, Ellis, Fundamentals of Programming Languages, 2a. ed., Rockville, Maryland: Computer Science, 1984, 446 p.

18) [KLEIMAN1996] KLEIMAN, Steve, SHAH, Devang y SMAALDERS Bart, Programming with threads, Mountain View, California: Sunsoft; Upper Saddle River, Nueva Jersey: Prentice Hall, 1996, 534 p.

19) [LEMON1996] LEMON, Karen A., Fundamentos de Compiladores. Como traducir el lenguaje de computadora, México: Continental, 1996, 210 p.

20) [LEWIS1996] LEWIS, Bil y BERG, Daniel J., Threads Primer. A Guide for Multithread Programming, EUA: Prentice-Hall, 1996, 319 p.

21) [LIBERTY2001] LIBERTY, Jesse y HORVATH, David B.. Aprendiendo C++ para Linux en 21 Días, México: Pearson Education, 2001, 1144 p.

22) [LOUDEN1997] LOUDEN, Kenneth C., Construcción de Compiladores. Principios y Práctica, México: Thomson, 1997, 582 p.

23) [LOUDEN1993] LOUDEN, Kenneth C., Lenguajes de Programación. Principios y práctica, México: Thomson, 1993, 633 p.

24) [MACCRACKEN1966] MACCRACKEN, Daniel D. y DORN, William S., Métodos numéricos y programación en fortran con aplicaciones en ingeniería y ciencias, Mexico: Limusa, 1966, 476 p.

25) [MARON1995] MARON, Melvin J. y LOPEZ, Robert J., Análisis Numérico. Un enfoque práctico, Trad. Rafael Iriarte Vivar-Valderrama, 3ª. ed., México: Continental, 1995, 780 p.

26) [MERLO1996] MERLO, Paola, Parsing with principles and classes of information, Netherlands: Kluwer Academic, 1996, 245 p.

27) [NIEVES1995] NIEVES HURTADO, Antonio y Domínguez Sánchez, Federico C. Métodos Numéricos Aplicados a la Ingeniería. México: Continental, 1995, 607 p.

28) [NORTHRUP1996] NORTHRUP, Charles J., Programming with UNIX Threads, Nueva York: J. Wiley, 1996, 399 p.

29) [NORTON1997] NORTON, Scott J. y DIPASQUALE, Mark D. Thread Time. The Multithread Programming Guide, EUA: Prentice-Hall, 1997, 540 p.

30) [REISDORPH1999] REISDORPH, Kent. Aprendiendo Borland C++ Builder 3 en 21 Días. México: Prentice-Hall, 1999, 856 p.

31) [PRESS2002] PRESS, William H., FLANNERY, Brian P., TEUKOLSKY, Saul A., VETTERLING, William T., Numerical Recipes in C++. The art of scientific computing, EUA: Cambridge, 2002, 1002 p.

32) [SCHEID1972] SCHEID, Francis J., Teoría y problemas de análisis numérico, Trad. H. A. Castillo, Mexico: McGraw-Hill, 1972, 422 p.

33) [SEINFELD1998] SEINFELD, John H. y PANDIS, Spyros N., Atmospheric Chemistry and Physics: from air pollution to climate change, Nueva York: John Wiley, 1998 1326 p.

34) [SCHILDT1996] SCHILDT, Herbert, Schildt´s Expert C++, EUA: McGraw-Hill, 1996, 402 p.

35) [SMITH1999] SMITH, James T. C++ toolkit for Engineers and Scientists, 2ª. Ed. EUA: Springer-Verlag, 1999, 392 p.

36) [STROUSTRUP2002] STROUSTRUP, Bjarne. El Lenguaje de Programación C++. Edición especial, Madrid: Pearson Education, 2002, 1072 p.

37) [WALL2000] WALL, Kurt, Programación en Linux con ejemplos, Buenos Aires: Prentice-Hall, 2000, 568 p.

38) [WALMSLEY2000] WALMSLEY, Mark, Multi-threaded programming in C++, Nueva York: Springer, 2000, 223 p.

## HEMEROGRÁFICAS

1) [BROWN1989] Brown, P. N., Byrne, G. D. y Hindmarsh, A. C. "VODE, a Variable-Coefficient ODE Solver", SIAM J. Sci. Stat. Comput., 10(1989), pp. 1038-1051.

2) [BRYANT2001] Bryant, Randal E. y O´Hallaron, David R. "Concurrent Programming with Threads", 17 de enero de 2001.

3) [BYRNE1992] Byrne, George D. "Pragmatic Experiments with Krylov Methods in the Stiff ODE Settings, in Computacional Ordinary Differential Equations", J. R. Cash and I. Gladwell(Eds.), Oxford University Press, Oxford, 1992, pp. 323-356.

4) [COHEN1994] Cohen, Scott D. y Hindmarsh, Alan C. "CVODE User Guide". Lawrence Livermore National Laboratory report UCRL-MA-118618, Septiembre 1994.

5) [MARTÍN2001] Martín Bragado, Ignacio. "La biblioteca estándar de plantillas". Mundo Linux. (España: 30 de marzo de 2001). Año IV, núm. 36. Págs. 56-61.

## SITIOS WEB

1) http://www.ucse.edu.ar/fma/sepa y www.ucse.edu.ar/fma/compiladores, Proyecto SEPa! (Software para la Enseñanza de Parsing), Desarrollado por la Cátedra de Compiladores e Intérpretes de la Universidad Católica de Santiago del Estero. Sofware Chalchalero versión 06/2003 y Kakuy versión v2003.06.12

2) http://www.netlib.org, colección de software, documentos y bases de datos acerca de temas relacionados con matemáticas.