UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

Unidad Académica de los Ciclos
Profesional y de Posgrado
del C.C.H.

Análisis de Mecanismos de Control de Concurrencia en Lenguajes Paralelos

TESIS DE GRADO

Que para obtener el título de
Maestro en Ciencias de la Computación
p r e s e n t a

Sergio Ramón Villavicencio Fernández

Mayo de 1988

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Análisis de Mecanismos de Control de Concurrencia en Lenguajes Paralelos

INDICE

Introducción

En los inicios de la computación, el CPU (procesador) no era compartible y cuando se asignaba a algún programa, éste no lo liberaba hasta que finalizara su ejecución; aún hoy en día existen sistemas operativos de este tipo a los que se les llama de uniprogramación o uniusuario.

El programa está integrado por una lista de instrucciones que se ejecutan secuencialmente, además de que en todo momento se puede determinar del valor de los datos la siguiente instrucción a procesarse; por tal razón, la ejecución de estos programas es secuencial y determinística. El término programa más bien se refiere a la codificación en lenguaje fuente de alto nivel, el cual constituye una interfase entre la propia computadora y el usuario; sin embargo, existen elementos de software que traducen del lenguaje fuente al de máquina con objeto de que la propia computadora pueda interpretar los programas, a los que se les llama procesos o tareas cuando ya son procesables por la computadora.

En función de lo que se ha escrito respecto a los programas secuenciales, se puede definir a un proceso como la ejecución de una lista secuencial de instrucciones, o lo que es lo mismo, al procesamiento de un programa secuencial.

En una siguiente etapa en la evolución de los sistemas de cómputo se hizo necesario incrementar la velocidad de procesamiento del CPU y de compartirlo entre varios usuarios, lo cual implicaba que varios procesos se repartieran al procesador en función del tiempo, con lo que a cada proceso le correspondía una tajada de tiempo de uso del procesador, esta técnica se conoce como multiprogramación y trae como consecuencia que los procesos compartan además del CPU, recursos tales como información y dispositivos de entrada y salida.

La multiprogramación crea el mismo efecto que si cada proceso estuviera procesándose en un distinto procesador, (pero a una velocidad menor de procesamiento) y se puede considerar que varias partes de los procesos se ejecutan al mismo tiempo, con lo cual se introduce el concepto de programación concurrente o paralela; a los procesos que se procesan simultáneamente se les llama procesos paralelos.

Concluyendo, se puede definir al concepto de programa concurrente como aquél que especifica a dos o más programas secuenciales que pueden ser ejecutados simultáneamente como procesos paralelos.

Con el proceso de multiprogramación con un solo procesador se tiene un paralelismo lógico, pues el efecto es equivalente al de

que cada proceso se ejecute en un distinto procesador y todos compartan una memoria común.

Posteriormente se desarrollaron arquitecturas en las que se tenía más de un procesador, y todos accesaban una memoria común. En este caso existe un paralelismo físico, ya que más de un proceso se está procesando al mismo tiempo en distintos procesadores que comparten una memoria común, este concepto es conocido como multiprocesamiento.

Finalmente, existe la tendencia, en la arquitectura de sistemas de cómputo, de la creación de redes de procesadores, en los que cada uno de éstos posee una memoria propia independiente y se enlaza con los demás mediante canales de comunicación; en cada procesador puede existir uno o más procesos ejecutándose paralelamente; este tipo de procesamiento se llama distribuido. Desde luego que existen combinaciones entre distintos tipos de arquitecturas y con diferentes modos de procesamiento (uniprogramación, multiprogramación).

Como ha podido observarse, la evolución en la arquitectura de los sistemas de cómputo ha ido a la par con el desarrollo de la programación concurrente; como ya se indicó anteriormente, la ejecución de procesos paralelos trae como consecuencia el que se compartan recursos entre ellos, que pueden ser elementos físicos como dispositivos periféricos, o bien de índole informático como estructuras y bases de datos.

Se dice que dos procesos son disjuntos si no comparten recursos, y son por lo tanto independientes.

Cuando más de un proceso comparte algún recurso, es necesario garantizar el acceso exclusivo al mismo por parte de un proceso, hasta que este último ya no requiera seguirlo accesando, para que posteriormente se le asigne a un nuevo proceso, quien a partir de ese momento tenga nuevamente el acceso exclusivo al mismo recurso. A esta propiedad se le conoce como exclusión mutua. Generalmente, no todas las instrucciones que integran un proceso referencian y/o operan en el recurso compartido. Al conjunto de aquellas que sí accesan al recurso se les llama sección crítica del proceso.

Para lograr tener exclusión mutua y coordinar el acceso a recursos compartidos es necesario que los procesos se comuniquen y sincronicen mutuamente. La comunicación entre los procesos posee cierto grado de dependencia con la arquitectura del sistema de cómputo en que residan; en general, se puede clasificar en dos los modos de comunicación existentes entre procesos: el primero es mediante el uso de variables compartidas, el segundo es a través de envío de mensajes.

El primer modo de comunicación referido en el párrafo anterior posee como único canal de comunicación a las variables compartidas, de modo que aquellos procesos que deseen comunicarse deben referirse a estas variables ya sea para leerlas o para

asignarles nuevos valores. Para lograr este modo de comunicación ambos procesos deben accesar una memoria compartida, lo cual implica una arquitectura de un solo procesador con multiprogramación, o bien una de multiprocesamiento.

En este modo de comunicación los procesos que participen en el acceso a las variables compartidas deben sincronizarse, pues siempre un proceso espera a que el otro realice alguna acción, como puede ser la asignación de algunas variables que el primero necesita para continuar con su ejecución, o bien, alguno de los dos espera a que el otro termine de leerlas para modificarlas, con lo cual, se establece un orden en la realización de las acciones o eventos en ambos procesos, que depende de la forma en que ambos están codificados por parte del programador; dicho en otros términos, nó es conveniente ni confiable que los procesos que compartan un recurso lo hagan aleatoria y desorganizadamente, lo cual no permitiría tener exclusión mutua.

A las restricciones en el orden en que deban realizarse los eventos o acciones en los procesos, para el acceso a los recursos compartidos, se le llama sincronización.

Para sincronizar procesos cuyo modo de comunicación es mediante variables compartidas, se utilizan mecanismos que a su vez accesan otras variables de acceso común, cuyo estado indicará si el recurso está libre, con lo que se modifica el valor de la variable, para indicar a los demás procesos que el recurso está nuevamente ocupado, o bien, se hace uso de mecanismos más complejos cuya implementación está igualmente en función de variables compartidas.

Nunca es posible sincronizar a los procesos en función de sus velocidades de procesamiento, pues es impredecible la velocidad con que cada uno es procesado.

Otro modo en que dos procesos pueden establecer comunicación sin compartir variables o memoria común es mediante el uso de envío de mensajes; en este caso se establece el canal de comunicación entre los procesos destino y fuente, en el momento en el que el segundo llama al primero, o bien, ambos se invocan mutuamente, siendo el mensaje el objetivo o resultado del proceso de comunicación, ya que mediante éste se transfieren datos y resultados, o se solicita la activación de una función, acción o servicio.

Los mecanismos que usan envío de mensajes pueden ser síncronos o asíncronos. Los síncronos requieren que ambos procesos se sincronicen para la transferencia del mensaje, mientras que los asíncronos nunca interaccionan entre sí y los mensajes se almacenan en buffers.

Los primeros mecanismos de comunicación que se concibieron e implementaron fueron los correspondientes al modo de variables compartidas, los cuales podían instalarse en una arquitectura tradicional de un procesador con su memoria y con

multiprogramación, o bien, en una de multiprocesamiento.

En [Dijkstra, 1968] se desarrolló un mecanismo de sincronización que opera sobre variables compartidas: el semáforo. Posteriormente se fueron desarrollando otros mecanismos de comunicación más complejos como es el caso de los monitores, cuya aparición data de los primeros años de la década de los 70 [Brinch Hansen, 1973; Hoare, 1974], este mecanismo dio lugar a la implementación de lenguajes tales como Pascal Concurrente [Brinch Hansen, 1975], Modula [Wirth, 1977] y Euclid entre otros; paralelamente, otro mecanismo de este mismo tipo se desarrolló: regiones críticas condicionales. Existen varios más que caen en este modo de comunicación, pero se considera que los expuestos en estas líneas fueron los más significativos.

Con respecto al modo de comunicación por envío de mensajes, cuyos mecanismos son más propios a implementarse en una arquitectura distribuida, se han desarrollado varios modelos o lenguajes.

Entre los más conocidos o destacados, cabe mencionar a Comunicación de Procesos Secuenciales, más comúnmente referido como CSP. Este concepto de lenguaje fue propuesto por Hoare en 1978 [Hoare, 1978]; PLITS (Programming Language In The Sky) desarrollado en la Universidad de Rochester [Feldman, 1979], que fué diseñado para aplicarse en una red de computadoras, su mecanismo de comunicación es asíncrono; ADA [U.S. Department of Defense, 1983] que es un lenguaje diseñado principalmente para aplicaciones de tiempo real, así como para control de dispositivos y procesos físicos, solicitado por el Departamento de Defensa de los Estados Unidos, cuyo mecanismo de comunicación por envío de mensajes es síncrono; SR (Synchronizing Resources) [Andrews, 1981, 1982] es otro lenguaje cuyo mecanismo es de tipo síncrono, y cuyo diseño se hizo pensando en su aplicación a la implementación de sistemas operativos distribuidos.

Otros ejemplos de lenguajes cuyos mecanismos de comunicación son por envío de mensajes son GYPSY [Good et al., 1979], el cual fue uno de los primeros que usó almacenamiento de mensajes; DP (Distributed Processes) [Brinch Hansen, 1978] de tipo síncrono; StartMod [Cook, 1980], Argus [Liskov and Scheifler, 1982], etc.

Como se puede observar, existe mucha diversidad en lo referente a modelos, conceptos y lenguajes de programación concurrente que hacen uso del envío de mensajes.

El objetivo del presente trabajo es analizar al conjunto de los modelos y lenguajes que se consideran más destacados e ilustrativos, así como sus diferencias.

El conjunto de lenguajes al que se hace referencia está constituido por los tres siguientes:

1.- Comunicación de Procesos Secuenciales, CSP [Hoare, 1978]
2.- Sincronización de Recursos, SR [Andrews, 1981, 1982]
3.- ADA

El primero de los tres (CSP) se seleccionó por ser uno de los primeros conceptos de programación concurrente que hace uso del envío de mensajes, y que ha tenido mucha trascendencia en el desarrollo de otros varios lenguajes concurrentes del mismo tipo que han surgido a lo largo de la presente década.

SR (Sincronización de Recursos) no es tan popular como ADA, sin embargo, su diseño e implementación es más reciente, y su mecanismo de comunicación es aún más rico que el de ADA, por tal motivo se decidió incluirlo en el análisis.

ADA se seleccionó por ser un lenguaje muy rico en mecanismos de control de comunicación y sincronización, además de que su uso se ha ido extendiendo por todo el mundo, y es uno de los más populares en la actualidad.

El análisis objeto del presente trabajo consiste en sistematizar conceptos de cada uno de los tres anteriores lenguajes de acuerdo con un esquema, el cual se utiliza en el último capítulo como marco de referencia para realizar la comparación entre los tres lenguajes.

El esquema de análisis consiste en describir en forma sintáctica y semántica los elementos, mecanismos y características de los lenguajes con respecto a los siguientes puntos:

1.- Procesos.
2.- Paralelismo.
3.- Comunicación.
4.- Sincronización.
5.- Indeterminismo.
6.- Calendarización (scheduling).

A continuación se definen estos puntos:

Procesos:
Se considera a un proceso como un conjunto de instrucciones y declaraciones que constituye en sí a un programa independiente, que tiene acceso exclusivo a sus variables locales, y que se ejecuta concurrentemente con otros procesos.

Paralelismo:
El concepto de paralelismo o concurrencia especifica la ejecución simultánea de un conjunto de procesos.

Comunicación:
Especifica la interacción entre procesos con objeto de sincronización y/o la transferencia de un mensaje.

Sincronización:
Se define como el conjunto de restricciones en el orden en que deben realizarse los eventos o instrucciones en los procesos involucrados en la comunicación, para el logro de un objetivo determinado previamente por el programador.

Indeterminismo:
El indeterminismo consiste en la selección aleatoria de una lista de instrucciones, de entre otras que forman un conjunto, para su posterior ejecución, con la característica de que la selección es totalmente aleatoria, y no existe ningún criterio, algoritmo u ordenamiento.

Calendarización (scheduling):
La calendarización consiste en la asignación de recursos compartidos a un conjunto de procesos (recursos tales como archivos, controladores físicos, CPU, etc.).

El último capítulo es la parte central del presente trabajo, ya que en él se hace un análisis comparativo de CSP, SR y ADA fundamentado en los puntos (ya definidos) del esquema propuesto de análisis.

Existen trabajos que se han realizado respecto al análisis comparativo de lenguajes concurrentes.

Dos de ellos son de Welsh y Lister [Welsh, Lister, 1979, 1981] de la Universidad de Queensland en Australia. En el primero hacen un análisis comparativo de los lenguajes CSP [Hoare, 1978] y DP [Brinch Hansen, 1978] (Distributed Processes), y en el segundo analizan a ADA respecto a CSP y SR.

Su análisis lo seccionan en dos partes: una cualitativa y otra cuantitativa. En el análisis cualitativo describen algunas características de los lenguajes, sin seguir ningún esquema en particular, pues las comparaciones las realizan tomando como referencia ejemplos clásicos de programas concurrentes, como por ejemplo, el 'shortest-job-next scheduler', la 'alarma', un ordenador de números, etc.

El análisis cuantitativo lo efectuan analizando el texto del código fuente de los ejemplos programados, y contando el número de operadores diferentes y sus ocurrencias, así como el de los operandos, a lo que aplican los postulados de Halstead [Halstead, 1977] con lo que obtienen medidas aproximadas de esfuerzo de programación, nivel del lenguaje, etc.

Otro trabajo en el que se hizo un análisis comparativo de varios lenguajes concurrentes es [Filman y Friedman, 1984]. En él se describen las diferencias generales entre los lenguajes en función de los siguientes puntos:

1.- Campos de aplicación.
2.- Procesos (creación).
3.- Comunicación (características generales).
4.- Grado de asignación de recursos a procesos (fairness).

5.- Manejo de fallas.

Como se puede observar, en el esquema de análisis que se propone ya se incluyen los puntos de comparación que presentan Filman y Friedman (procesos, comunicación, manejo de fallas como parte de la comunicación, 'fairness' se analiza en calendarización), además, de que el esquema propuesto es más amplio y ambicioso, pues en él se analizan aparte de los cinco puntos anteriores, el paralelismo, la sincronización y el indeterminismo, que son fundamentales en análisis de lenguajes concurrentes.

Por otra parte, el análisis que Filman y Friedman hacen para cada lenguaje no se apega al esquema de los cinco puntos anteriores que utilizan en la parte comparativa, mientras que el esquema que se propone en este trabajo se aplica a cada lenguaje de estudio, y se usa como referencia para el análisis comparativo.

Hasta la fecha no se tiene conocimiento de algún trabajo publicado, que proponga un esquema de comparación similar al propuesto.

Nota:
Se convino en usar la forma de Backus-Naur (BNF) para representar la sintaxis del lenguaje, así como también incluir en la descripción de la misma a los paréntesis '[' y ']'.

A cualquier símbolo encerrado por estos paréntesis (corchetes), se le considera como opcional en la gramática, es decir, puede o no existir.

Su traducción equivalente en la notación BNF es la siguiente:

notación propuesta:	notación BNF equivalente:
[<símbolo>]	<símbolo> | 0

'0' representa la cadena vacía (inexistencia del símbolo).

Por lo anterior, si se llegaran a presentar corchetes como parte de la sintaxis de alguno de los lenguajes bajo estudio, se encerrarían entre comillas para diferenciarlos.

"CSP" Comunicación de Procesos Secuenciales

1.- Origen y Objetivos

CSP son las iniciales de "Comunicación de Procesos Secuenciales" (Communicating Sequential Processes), el cual es un concepto de un lenguaje de programación concurrente en el que la comunicación es mediante el envío de mensajes.

Fue desarrollado por C.A.R. Hoare en la Universidad de Queen's en Belfast, Irlanda del Norte, en el año de 1978 [Hoare, 1978].

Entre los puntos más importantes que dieron origen a CSP están:
1.- La adición de operaciones de entrada y salida como parte del lenguaje.
2.- El advenimiento de arquitecturas integradas por redes de procesadores.
3.- La introducción de no determinismo.

En relación al primer punto, las operaciones de entrada y salida no se concebían como parte de la filosofía del lenguaje en la mayoría de los lenguajes que se consideraban como tradicionales. Estas operaciones se concebían como algo extra, añadido posteriormente al diseño del mismo.

En el punto 2, como consecuencia del desarrollo de arquitecturas modernas, en las cuales se tiene a una red de procesadores, cada uno con su propia memoria, se creó la necesidad de implementar nuevos lenguajes que mejor se acoplaran a estas arquitecturas.

En arquitecturas menos contemporáneas existía únicamente un procesador, o bien, un conjunto de procesadores que compartían memoria, y los procesos instalados en los diferentes procesadores podían comunicarse indirectamente, a través de la memoria compartida (En el caso de que existiera un solo procesador, la memoria es compartida por todos los procesos).

El uso de variables (memoria) compartidas dio lugar al desarrollo de mecanismos de sincronización, que aseguraran el acceso exclusivo a las variables, tales como los semáforos y posteriormente los monitores. En una red de procesadores el uso de una memoria compartida por todos representaría un cuello de botella.

Por las razones expuestas anteriormente, el uso de mensajes como medio de comunicación y sincronización es directamente implementable por los mismos protocolos de comunicación de la red.

En arquitecturas que comparten memoria, también es fácilmente implementable el uso de mensajes, y no representa serias desven-

tajas respecto al uso de variables compartidas. En este último caso se elimina el uso de semáforos, monitores u otro mecanismo similar.

Respecto al tercer punto, el no determinismo consiste en la selección y ejecución de una lista de instrucciones, de entre otras que forman un conjunto, siendo la selección aleatoria o arbitraria, y sin ningún criterio u ordenamiento previo de selección.
Los lenguajes tradicionales (Algol, Pascal, Fortran, etc.) no contemplaban esta propiedad.

Por último, es interesante señalar que CSP fue uno de los primeros conceptos de programación concurrente que hizo uso del envío de mensajes para la comunicación y sincronización entre procesos. Su filosofía influyó en varios lenguajes o conceptos de programación concurrente, que surgieron a principios de la presente década.

CSP
2.- PROCESOS

Se considera que un programa concurrente es aquél que especifica a dos o más secuenciales que pueden ser ejecutados al mismo tiempo.

2.1- Sintaxis de procesos.

En CSP se designa a un proceso de la siguiente forma:

<proceso> ::= <etiqueta> <lista de instrucciones>

Se representa a un proceso con una etiqueta seguida de una lista de instrucciones.

La etiqueta identifica y da nombre al proceso formado por la lista de instrucciones.

Es válido que exista a lo más un proceso que no posea etiqueta, y por lo tanto nombre (sintácticamente se le representa por el símbolo 'vacío'). No es permitido que exista más de un proceso con el mismo nombre, cada uno posee su propia identidad.
La sintaxis del símbolo 'etiqueta' es:

<etiqueta> ::= <vacío> | <identificador>::

Si existe la etiqueta, entonces se le representa por una cadena de caracteres o identificador (nombre), seguido del símbolo '::'.

Se define a una familia de procesos como un arreglo de procesos, los cuales comparten el mismo nombre de etiqueta y la misma lista de instrucciones, pero se diferencian por:

1.- El valor que tiene el índice o índices.
2.- El valor que poseen sus elementos o variables indexadas por los índices del punto 1.

Puede existir más de un índice, lo que permite tener matrices de procesos de n dimensiones.
La sintaxis de una familia de procesos es la siguiente:

<fam_proceso> ::= <fam_etiqueta> <lista de instrucciones>
<fam_etiqueta>::=<identificador>(<ident_índice>{,<ident_índice>})::
<ident_índice> ::= <constante entera>|<rango>
<constante entera> ::= <número>|<índice>
<índice> ::= <identificador>
<rango> ::= <índice>:<límite inferior>..<límite superior>

La etiqueta está representada por el identificador, que es una cadena de caracteres que le proporciona nombre a la familia.
El identificador de la etiqueta está seguido de un par de paréntesis entre los que se indican los valores numéricos que pueden tomar los índices, cuya representación puede ser cualquiera de las dos siguientes:

1.- Como constante de tipo entero (número o identificador).
2.- Como rango.

En el primer punto se pueden tener los siguientes casos:

nombre_familia(5,7,3,..,10)

o bien:

constantes i1, i2, i3, ..., in
i1 = 5, i2 = 7, i3 = 3,.., in = 10
nombre_familia(i1,i2,i3,...,in)

De modo que si los índices se designan por j1, j2, j3,..,jn; j1 únicamente puede tomar valores del 0 al 5, j2 los correspondientes del 0 al 7, j3 del 0 al 3, etc., por lo que en este ejemplo, el proceso "nombre_familia(6,5,1,..,8)" no es miembro de la familia (6 no es válido).

El punto 2 designa a un rango. Los siguientes ejemplos ilustran esta representación:

1.- nombre_familia(i:1..5,j:4..20,..)
el índice representado por el símbolo i puede tomar cualquier valor entre 1 y 5, el correspondiente a j puede tomar valores entre 4 y 20, etc..
2.- nombre_familia(i:i1..i2,j:j1..j2,..)
los índices representados por i y j pueden tomar valores entre i1 e i2 ,y j1 y j2 respectivamente. i1, i2, j1, j2 son constantes, por lo que poseen un valor definido, la única restricción en la sintaxis es que i2>i1 y j2>j1, el límite superior debe ser mayor al inferior.

A las definiciones de sintaxis de los procesos se añaden las siguientes restricciones contextuales:

1.- Todo proceso puede referenciar únicamente a sus variables locales.
2.- No es permitida la recursión.

Respecto al primer punto, ningún proceso debe hacer referencia a variables que pertenezcan al cuerpo de cualquier otro proceso. A esta propiedad se le llama disjunción, y los procesos concurrentes deben ser consecuentemente disjuntos. Este punto es consecuencia directa del propio mecanismo de comunicación, el cual no permite el uso de variables compartidas.

En relación al punto 2, todo proceso está identificado por su nombre, que debe ser único.

Se concluye en consecuencia que ningún proceso puede estar definido en función de otro que posea su mismo nombre. Como se estudiará más adelante, no se permite la creación dinámica de procesos. La recursión puede simularse mediante el uso de familias de procesos, en que un proceso de índice i llama al i+1, y éste a su vez llama a i+2, etc.

2.2- Semántica de procesos.

Como se definió en la sección correspondiente a la sintaxis, todo proceso está constituido por una lista de instrucciones, por lo que el proceso cobra vida en el momento que se empieza a procesar su lista correspondiente de instrucciones, y se termina, o finaliza su vida, cuando las ha terminado de procesar.

Durante la vida de todo proceso, éste puede estar en uno de dos estados:

1.- En ejecución.
2.- Bloqueado.

En el primer caso, está en estado de procesamiento de alguna de sus instrucciones y/o compite por uso del procesador (CPU).

En el segundo caso, su ejecución está suspendida como consecuencia de esperar algún mensaje o interacción con otros procesos.

En este último caso, puede existir la posibilidad de que varios procesos que están suspendidos esperen interacción entre ellos, lo cual es síntoma de un abrazo mortal (deadlock), y es consecuencia de una mala programación.

3.- Paralelismo

Existen instrucciones que determinan a todos los procesos que se procesarán en paralelo (concurrentemente), los cuales se conocen en CSP como comandos paralelos.

Conviene hacer la aclaración de que en CSP, los términos comando e instrucción tienen el mismo significado.

3.1- Sintaxis del Paralelismo.

La sintaxis de un comando paralelo es:

<comando paralelo> ::= [<proceso>{||<proceso>}]

El símbolo '||' es indicativo de paralelismo, y todos los procesos encerrados entre los corchetes son procesados concurrentemente.

Cada proceso designado en la sintaxis del comando puede ser un único proceso, o bien ser una familia de procesos. Así por ejemplo, en el comando paralelo:

```
[p1::lista_instruc_p1    ||   p2::lista_instruc_p2    ||...    ||
          pn::lista_instruc_pn]
```

los procesos p1, p2,..pn se procesan en paralelo.

En el caso de una familia de procesos, cada uno de sus integrantes se ejecuta concurrentemente con los demás de la familia. Por lo que X(i:1..límite) :: lista_instruc_i es equivalente a:

```
[X(1)::lista_instruc_1   ||   X(2)::lista_instruc_2    ||..    ||
          X(límite)::lista_instruc_límite]
```

Finalmente, en el siguiente comando se procesa en paralelo a un conjunto de procesos con una familia:

```
[P1::lista_instruc_P1     ||     P2::lista_instruc_P2     ||...||
     Pn::lista_instruc_Pn || X(i:1..límite)::lista_instruc_(i)]
```

y es equivalente a:

```
[P1::lista_instruc_P1     ||     P2::lista_instruc_P2     ||...||
     Pn::lista_instruc_Pn ||     X(1)::lista_instruc_X(1)      ||
     X(2)::lista_instruc_X(2) ||...||
     X(límite)::lista_instruc_X(límite)]
```

Todos los procesos elementos de la familia se procesan en paralelo entre sí, junto con los demás procesos designados entre los corchetes.

Falta por señalar que un comando paralelo puede formar parte de una lista de instrucciones, o también puede suceder que sea la única instrucción en el programa.

3.2- Semántica del Paralelismo.

Con objeto de mostrar la semántica de un comando paralelo existe la necesidad de describir ciertos conceptos auxiliares. La semántica está en función de esos conceptos, por lo que a continuación se hace la descripción correspondiente, a manera de subtítulo.

3.2.1- Conceptos Auxiliares.

En general, la ejecución de una instrucción puede ser:
1.- exitosa o
2.- fallida.

Se considera que la ejecución de una instrucción es fallida cuando en su procesamiento se genera un error.
El tipo de errores clásicos que hacen fallar a una instrucción son, por ejemplo, la división entre cero.

Por el contrario, la ejecución de una instrucción es exitosa cuando no se produce ningún error.

Generalmente, como consecuencia de la ejecución exitosa de una instrucción se produce un cambio en el medio ambiente del proceso del cual forma parte.

Este cambio puede originar una modificación en el estado interno del proceso, como puede ser una asignación a una variable local del mismo.

También puede alterar el medio ambiente exterior; tal es el caso de una instrucción de envío de mensaje a otro proceso.

Se considera que la ejecución de una lista de instrucciones es fallida, cuando al menos la ejecución de una de sus instrucciones componentes es fallida. La ejecución de una lista de instrucciones es exitosa cuando también lo es la de cada una de sus instrucciones constituyentes.

Finalmente, el estado de procesamiento de un proceso está en función de la lista de instrucciones que lo integran. Por lo que el estado de ejecución de un proceso será:

1.- Exitoso, cuando así lo sea el de su correspondiente lista de instrucciones.
2.- Fallido, cuando sea también fallida la ejecución de su lista de instrucciones.

3.2.2- Semántica del comando paralelo.

Al iniciarse el procesamiento de un comando paralelo, se inicia la ejecución simultánea de todos sus procesos constituyentes.
Cada proceso, una vez iniciada su ejecución, se procesa con una velocidad arbitraria e impredecible, lo que da lugar a que algunos finalicen su ejecución (exitosa) antes que otros.

Se considera que un comando paralelo se termina de procesar exitosamente si y sólo si todos sus procesos constituyentes han finalizado su ejecución en forma exitosa. De otro modo, el comando paralelo falla.

3.3- Conclusiones sobre Paralelismo.

1.- Todos los procesos constituyentes de un comando paralelo deben ser disjuntos.
2.- El paralelismo está determinado de una forma estática en el texto del programa.

El primer punto es una consecuencia directa de las características propias de un proceso en CSP; ya que ningún proceso debe referenciar las variables locales de otro.

En relación al segundo punto, el texto del programa determina de una forma estática tanto al número como a la identidad de los procesos que se ejecutarán en paralelo.

La sintaxis del propio comando determina a los procesos a ejecutarse concurrentemente, sin permitir que dinámicamente se creen procesos adicionales a los que el comando especifica.
Lo anterior restringe al programador, quien previamente debe considerar a todos los procesos que se ejecutarán en paralelo.

CSP
4.- Comunicación

CSP es un lenguaje cuya comunicación es mediante el envío de mensajes. Debe existir por lo tanto una forma de comunicación cuyo efecto será la transferencia de un mensaje.

En toda comunicación coexisten cuatro elementos:

1.- El transmisor o fuente.
2.- El receptor o destino.
3.- La información o mensaje a transferir.
4.- El canal o medio de comunicación.

En CSP el transmisor es un proceso llamado 'fuente', cuya función es enviar el mensaje.
El receptor es un proceso denominado 'destino' y su función es recibir el mensaje.

Ni el proceso fuente puede recibir resultados, ni el destino enviarlos, a través del mismo mensaje cuando se establece la comunicación.

El canal o medio de comunicación puede ser la misma red física que interconecta a los procesadores, así como el protocolo de comunicación que usa la red.

No obstante lo mencionado en el párrafo anterior, puede existir comunicación por envío de mensajes entre procesos instalados en un mismo procesador y que comparten memoria, o bien, que únicamente comparten la memoria. En este último caso, el medio de comunicación está directamente soportado por el sistema operativo.

Consecuentemente, es transparente al usuario la forma e implementación del canal de comunicación.

Respecto a los mensajes,existen dos tipos de ellos en CSP:

1.- Los que agrupan a un conjunto de valores de datos.
2.- Los que están formados por el nombre de una función u operación.

En el primer caso se tranfiere a un conjunto de datos en el mensaje. En el segundo punto se tranfiere el nombre de la función, conjuntamente con los valores de sus argumentos.

La sintaxis y la semántica de los mensajes se analizará detalladamente en las subsecciones subsecuentes.

4.2- Mecánica de Comunicación.

Para que se establezca la comunicación entre los procesos fuente y destino, es necesario que ambos participen activamente en la misma mecánica de comunicación. Para el establecimiento de la comunicación se requieren únicamente dos condiciones:

1.- Que el proceso fuente llame o invoque al proceso destino.
2.- Que el proceso destino invoque al fuente.

El orden en que se realicen ambas condiciones es irrelevante, lo importante es que ambas se generen.

A consecuencia de esta mecánica de comunicación, es imprescindible que el proceso fuente conozca la identidad del proceso destino y este último la del fuente.

Una vez que se ha establecido la comunicación, se transfiere el mensaje. Posteriormente, se termina la comunicación entre ambos procesos.

4.3- Sintaxis de la Comunicación.

La invocación o llamada del proceso fuente al destino, y la correspondiente del destino al fuente, se efectúa mediante un par

de comandos o instrucciones. Estos comandos son:

1.- El comando de entrada.
2.- El comando de salida.

El primero lo ejecuta el proceso destino, cuando espera algún mensaje del proceso fuente. El segundo lo ejecuta el proceso fuente, para enviar su mensaje al proceso destino.

Sintaxis de comandos:

1.- Comando de Entrada:

<comando_de_entrada> ::= <proceso_fuente> ? <mensaje>

2.- Comando de Salida:

<comando_de_salida> ::= <proceso_destino> ! <mensaje>

El símbolo de interrogación '?' es utilizado para representar un comando de entrada. El símbolo de admiración '!' representa a un comando de salida. Así por ejemplo, si el proceso fuente 'P' y el proceso destino 'Q' van a establecer una comunicación, entonces su estructura debe ser la siguiente :

P :: ... ; Q ! mensaje ; ...
Q :: ... ; P ? mensaje ; ...

Los puntos suspensivos '...' no son una notación del lenguaje, se le usa en este texto para representar a una lista de instrucciones.

El proceso P envía un mensaje al proceso Q através del comando: "Q ! mensaje". Q recibe el mensaje de P mediante el comando: "P ? mensaje". La transferencia del mensaje es del proceso P al Q, y tanto P como Q deben conocer previamente la identidad del otro con el que se comunicarán.

Haciendo referencia a la introducción, la sintaxis de los dos tipos de mensajes existentes en CSP es la siguiente:

1.- Para los mensajes consistentes únicamente en un conjunto de valores de datos, la sintaxis es:
proceso fuente

<mensaje> ::= <lista_de_expresiones>
<lista_de_expresiones> ::= (<expresión> {,<expresión>})

proceso destino

<mensaje> ::= <lista_de_variables>
<lista_de_variables> ::= (<variable> {,<variable>})

2.- Para los mensajes que están integrados por el nombre de una función:
proceso fuente

<mensaje> ::= <nombre_función> <lista_de_expresiones>

proceso destino

<mensaje> ::= <nombre_función> <lista_de_variables>

Las restricciones que se exigen en ambos tipos de mensajes son:

1.- Compatibilidad en el número de expresiones y de variables.
2.- Compatibilidad en el tipo de datos.

El primer punto se refiere a que el número de expresiones en el proceso fuente debe ser igual al número de variables del proceso destino.

Las listas de expresiones y variables pueden estar constituidas por elementos de distintos tipos de datos. Sin embargo, el segundo punto indica que el tipo de dato de la expresión i-ésima de la lista de expresiones del proceso fuente debe coincidir con el tipo de dato de la variable i-ésima de la lista de variables del proceso destino.

4.4- Semántica de comunicación.

El proceso de comunicación de CSP está caracterizado por las dos siguientes propiedades:

1.- Simetría.
2.- Sincronía.

Respecto a la primera propiedad, el proceso de comunicación entre dos procesos es simétrico cuando ambos participan activamente en él, ya que para establecer comunicación, ambos deben de invocarse (fuente a destino y destino a fuente).

Por el contrario, se considera que el mecanismo de comunicación es asimétrico cuando uno de los dos procesos juega un papel pasivo en él.

El término sincronía deriva del vocablo sincronización. Este término, referido al mecanismo de comunicación, indica sincronización en la comunicación entre dos procesos. Sincronización significa un ordenamiento en la ejecución de los eventos, por lo que sincronía designa un ordenamiento en la comunicación entre dos procesos.

Lo anterior trae implícito un retraso o suspensión en la ejecución de un proceso, para poderse comunicar con el otro.

La sincronía es consecuencia de que en la mecánica de comunicación de CSP no existe almacenamiento temporal de mensajes. El primer proceso que emita su comando de entrada o salida debe esperar (bloqueo o suspensión) a que el otro con el que desea establecer comunicación ejecute también su correspondiente

comando.

Una vez obtenida la comunicación, se tranfiere el mensaje.
Un proceso que está en estado de espera está en realidad suspendido.

Como consecuencia de estas propiedades, ningún proceso puede establecer comunicación con más de un proceso al mismo tiempo.

Finalmente, una vez transferido el mensaje se termina el mecanismo de comunicación, y cada proceso continúa procesando la instrucción subsecuente al comando de entrada o salida que le permitió establecer la comunicación.

En esta sección se ejemplifica la propiedad de sincronía de los procesos. Considere los siguientes procesos P y Q:

```
P :: ...; Q ! mensaje; ...        Q :: ...; P ? mensaje; ...
```

Existen dos casos:
1.- P invoca primero a Q.
2.- Q invoca primero a P.

En el primer caso, P ejecuta el comando: "Q!mensaje" y posteriormente, se suspende en espera de que Q lo invoque.
Cuando Q ejecuta su instrucción: "P?mensaje", se despierta (desbloquea) a P, y se transfiere el mensaje de P a Q. Con lo que termina la comunicación entre ellos.
El segundo caso es lo inverso del primero.

A continuación se analiza la semántica de los mensajes.

En primer lugar se analizará la semántica del primer tipo de mensaje (aquél que agrupa a un conjunto de expresiones o variables) que maneja CSP.

En este tipo de mensaje, la sintaxis del proceso fuente hace referencia a una lista de expresiones; mientras la sintaxis del proceso destino hace referencia a una lista de variables.

La transferencia del mensaje en el proceso de comunicación consiste en la asignación de los valores de las expresiones de la lista en el proceso fuente a las variables de la lista correspondiente del proceso destino.

De esta manera, dos procesos P y Q aparecerían codificados así:

```
P :: ...; Q ! (expr1,expr2,...,exprn); ...
  Q :: ...; P ? (var1,var2,...,varn);...
```

lista de expresiones del proceso fuente 'P':
(expr1,expr2,...,exprn)
lista de variables del proceso destino 'Q':
(var1,var2,...,varn)

En el momento que se establezca la comunicación entre P y Q, el valor de expr1 se asigna a var1, de expr2 a var2, etc..

```
var1 := expr1
var2 := expr2
  .      .
varn := exprn
```

Con objeto de que las asignaciones sean correctas, las variables deben ser del mismo tipo que el de las expresiones a asignárseles. Lo anterior es una justificación de las restricciones impuestas por la sintaxis del mensaje.

Desde el punto de vista semántico, el segundo tipo de mensaje (el que hace referencia al nombre de una función u operación), es una solicitud de algún servicio, por parte del proceso fuente al proceso destino.
El servicio consiste en la ejecución de una función u operación, la consulta de una estructura o base de datos, etc.

Respecto a este segundo tipo de mensaje, la interacción entre los procesos fuente y destino se convierte en una relación cliente/servidor. El proceso fuente es el cliente y el destino es el servidor.

Sin embargo, por las propias características de la mecánica de comunicación, el servidor debe conocer la identidad de los clientes, lo cual no es lo común.

El proceso servidor puede realizar varios servicios o funciones, por ejemplo, si el proceso servidor es el responsable del acceso a una base de datos, entonces sus servicios serán de lectura, escritura, modificación, o eliminación de registros, entre otros varios.

Por lo anterior, el cliente debe invocar al servidor indicándole el nombre del servicio solicitado (lectura, escritura,...), lo cual lo hace a través del símbolo 'nombre de función' que indica la sintaxis de este tipo de mensaje.
Si el tipo de función o servicio requiere argumentos, entonces éstos se transfieren como una lista de expresiones.

El servidor debe también referenciar al nombre de la función a desarrollar, conjuntamente con una lista de variables. En el momento que se establezca la comunicación, se asignan los valores de los argumentos a las variables de la lista del proceso servidor.

En ese momento, el proceso servidor ya posee el valor de los argumentos que requiere para ejecutar la función o servicio correspondiente.

Pueden existir servicios o funciones que carezcan de argumentos, un ejemplo de este caso puede ser la solicitud de un servicio de consulta o lectura.

Es importante hacer notar que en este tipo de mensaje, el cliente solicita un servicio, pero él mismo debe invocar posteriormente al proceso servidor para recibir el resultado de la ejecución del servicio, si es que se generan resultados. En la misma forma, el proceso servidor recibe en principio la solicitud del cliente, ejecuta la operación que el mismo cliente le indica y lo invoca posteriormente, para enviarle el resultado o resultados.

Es necesario por lo tanto generar cuatro comandos, dos de entrada y dos de salida. Para ejemplificar, considere a los procesos P y Q:

```
P :: ...; x := 5; Q ! inserta(2 * x + 1); ...
Q :: ...; P ? inserta(y); ...
```

P solicita de Q que se inserte en alguna estructura de datos al valor de la expresión "2 * x + 1" . Q y P establecen la comunicación (se corresponden), y se asigna a y el valor de 11 (y := 11).

Si Q retornara algún resultado que necesitara P, se añadiría un comando de salida en Q y otro de entrada en P.

4.5- Causas de falla en la comunicación.

En principio, una asignación en el mensaje falla por dos motivos:

1.- Cuando los tipos de datos de la variable y la expresión a asignársele no coinciden.
2.- Cuando la expresión no está definida.

Una falla en la asignación origina un aborto en la instrucción y en el proceso.

Las causas que originan una falla en el mecanismo de comunicación son:

1.- Falla en el comando de entrada.
2.- Falla en el comando de salida.

Ambos tipos de comandos fallan por los dos siguientes motivos:

1.- Falla de asignación en el mensaje.
2.- El proceso invocado ha finalizado su ejecución.

Debido a que una transferencia de mensaje es una asignación de valores de un conjunto de expresiones a variables, entonces pueden fallar las asignaciones por cualquiera de los dos motivos que las originan.

El segundo motivo se refiere a que el comando falla si hace referencia a un proceso que ya culminó su procesamiento.

En ambos casos, la falla de un comando de entrada o salida causa la falla del proceso del cual forma parte.

4.6- Conclusiones.

El mecanismo de comunicación de CSP presenta dos

desventajas, que son interesantes de subrayar como conclusiones.

La primera de ellas está relacionada a la característica de simetría del mecanismo de comunicación.

Cada proceso que desee establecer comunicación con otro debe conocer el nombre o identidad del otro. Esto representa una fuerte desventaja en una interacción de procesos del tipo cliente/servidor.

En esta interacción, un servidor podría ser invocado por varios procesos clientes, aún sin conocer su identidad. Un ejemplo específico es el de una biblioteca de procesos que pueda ser utilizada por cualesquiera que la invoque.

En CSP no es posible implementar una biblioteca, pues los servidores (biblioteca) deben conocer la identidad de sus procesos clientes.

La segunda desventaja está relacionada al flujo de información en los mensajes, ya que en la mecánica de comunicación el flujo de información es en el sentido del proceso fuente al proceso destino. Por tal razón, un comando de salida no puede esperar enviar datos y recibir resultados. Ni un comando de entrada recibir datos y enviar resultados. Si un proceso requiere enviar datos y recibir resultados, existe la necesidad de duplicar tanto el número de los comandos como el de los mensajes, es decir, se debe de invertir en más código.

"CSP"
5.- Sincronización

En el desarrollo de la presente sección se considerarán dos tipos de sincronización:

1.- Sincronización del mecanismo de comunicación.
2.- Mecanismos propios de sincronización.

En todo mecanismo de comunicación existe un orden implícito en las transacciones entre los procesos. Por ejemplo, un proceso no puede recibir un mensaje antes que otro lo envíe.

En el segundo punto se considerará a los comandos custodiados como mecanismo de sincronización.

Un comando custodiado, según C.A.R. Hoare, es una instrucción que en función del estado de una expresión booleana y/o un comando de entrada, ejecuta una lista de instrucciones.

A la parte del comando que agrupa a la expresión booleana y al comando de entrada, o bien a uno de ambos, se le llama guardia booleano.

5.2- Sintaxis de mecanismos de sincronización.

La sintaxis de un guardia booleano es la siguiente:

```
<guardia> ::= <expresión_booleana>
            | <elemento>;<comando_de_entrada>
            | <comando_de_entrada>
<elemento> ::= <expresión_booleana> | <declaración>
```

El símbolo 'declaración' representa la declaración de variables, que se referenciarán en la lista de instrucciones, y que conjuntamente con el guardia, constituyen un comando custodiado.

También representa la declaración de los parámetros formales, que pudieran existir en el comando de entrada. Los parámetros formales son las variables, a las que se asignará el conjunto de valores que integran el mensaje.

Observe que un guardia está constituido por una expresión booleana, o un comando de entrada, o ambos. Es imprescindible que al menos uno de los dos forme parte del guardia.

La sintaxis de un comando custodiado en CSP es la siguiente:

```
<comando_custodiado> ::= <guardia> -> <lista_de_instrucciones>
```

Un comando custodiado puede hacer uso de variables locales a él mismo.

Se define como radio de acción de una variable al conjunto de instrucciones o comandos que pueden referenciarla; fuera de ese conjunto la variable es indefinida. El radio de acción de las variables locales a un comando custodiado comprende desde el punto en que se declaran en el guardia, hasta la última instrucción de la lista que constituye al comando custodiado.

Una restricción que presenta la sintaxis de un comando custodiado es que los parámetros formales de un comando de entrada no pueden ser referenciados en el guardia booleano; únicamente se les puede referenciar en la lista de instrucciones.

Los guardias booleanos y los comandos custodiados también pueden estar indexados y contener rangos. La sintaxis de estos elementos es:

<comando_indexado> ::=
(<rango>{,<rango>})<guardia> -> <lista_de_instrucciones>

5.3- Semántica de los mecanismos de sincronización.

Como primer punto se analizarán las características de sincronización del mecanismo de comunicación, y en segundo lugar se describirá la semántica de los comandos custodiados.

Sin embargo, para la adecuada descripción del primer punto, es necesario definir ciertos conceptos que influyen directamente en la sincronización.

Por la razón anterior, se añade una subsección que define los conceptos, y se tratará en primer lugar.

5.3.1- Definición de conceptos auxiliares.

El estado de un proceso puede evaluarse en cada instrucción que lo constituya. El estado del proceso está representado por el valor que en un momento dado y en un punto determinado de su secuencia de instrucciones tienen todas las variables del mismo, así como sus parámetros.

Al término 'estado' se le conoce mas comúnmente como ambiente (del proceso), y previa a la ejecución de una instrucción existe un ambiente en el proceso.

Como consecuencia del procesamiento de la instrucción, se produce un nuevo ambiente posiblemente diferente al anterior, por los cambios que haya introducido la propia ejecución de la instrucción.

Una condición es una expresión o fórmula lógica de primer orden, con la cual se pretende definir el estado de un proceso. La condición está en función del ambiente del proceso, de tal manera que si hace que la expresión lógica sea verdadera, entonces se dice que se cumple la condición.

En caso contrario, si el ambiente hace que la fórmula sea falsa, entonces no se cumple la condición.

Previa a la ejecución de una instrucción, se puede definir el estado del proceso mediante una condición llamada precondición. De igual forma, posteriormente al procesamiento de la instrucción, se define al estado del proceso mediante una postcondición.

La precondición define al estado inicial y la postcondición al final, como consecuencia de haber ejecutado la instrucción.

Se define una 'aseveración' (assertion) como la secuencia:

{p} s {q}

donde 'p' es la precondición, 'q' es la postcondición y 's' es la

instrucción.

La aseveración es verdadera cuando dado cualquier ambiente en que se cumpla la precondición p (p verdadera), y se termina la ejecución de la instrucción s ,generando un nuevo ambiente, entonces la postcondición q (en función del nuevo ambiente) será verdadera; es falso en cualquier otro caso.

Cuando existe interacción entre dos o más procesos, y alguna aseveración en uno de ellos se invalida como consecuencia de otro proceso, entonces se dice que existe interferencia en la ejecución del primero por parte de este último.

El término invalidar significa que dada la precondición verdadera, y luego de procesar la instrucción, la postcondición resulta ser falsa. En este caso, la postcondición estará en función de valores que otro proceso haya enviado, y que la hagan tener un valor lógico falso.

5.3.2- Semántica del mecanismo de comunicación.

La característica de sincronización o sincronía del mecanismo de comunicación de CSP, permite que los procesos que intervienen en él se sincronicen para la transferencia de un mensaje. Lo anterior implica los siguientes puntos:

1.- Un retraso o suspensión de uno de los dos procesos.
2.- Una vez realizada la comunicación, el proceso fuente sabe que el mensaje ha sido recibido por el proceso destino.

El primer punto ya fue analizado en la sección de comunicación. En relación al segundo punto, el comando de salida del proceso fuente puede considerar como postcondición a la recepción del mensaje por parte del destino.

Consecuentemente, la instrucción subsecuente al comando de salida considerará también como precondición a la recepción del mensaje. Esto trae como consecuencia lo siguiente:

1.- Información de un proceso respecto al nivel de avance del otro, y viceversa.
2.- Se reduce considerablemente la interferencia entre los procesos.

Respecto al segundo punto, la aseveración:

{p} s {q}

donde 'p' es la precondición, 's' es el comando de salida y 'q' es la recepción del mensaje: siempre es verdadera en el proceso fuente.

5.3.3- Semántica de los comandos custodiados.

Como ya se ha indicado, la sincronización establece un orden en la ejecución del proceso para el acceso a algún recurso compartido, o para controlar sus transacciones con otros proce-

sos concurrentes.

El mecanismo de comunicación síncrono de CSP contribuye a este fin como ya se ha analizado. Sin embargo, existe la necesidad de sincronizar a un proceso bajo la ocurrencia de ciertas condiciones, sin que deba establecerse su comunicación con otros.
Para lograr lo anterior se requiere que el proceso se suspenda (espere) hasta que se cumplan tales condiciones; para alcanzar esos objetivos deben tenerse instrucciones que prueben el estado lógico de las condiciones, y en función de éstas provoquen la suspensión del proceso, o procesar una lista de instrucciones.

Estas instrucciones constituyen los denominados comandos custodiados, que están formados por un guardia y una lista de instrucciones. Haciendo referencia a la sintaxis, un guardia puede ser:

1.- Un comando de entrada.
2.- Una expresión booleana.
3.- Una expresión booleana y un comando de entrada.

Si se utilizan los símbolos <expr> y <com_ent>, para representar a la expresión booleana y al comando de entrada, respectivamente, entonces se pueden tener los tres siguientes tipos de comandos custodiados:

1.- <expr> -> <lista_instrucciones>
2.- <com_ent> -> <lista_instrucciones>
3.- <expr>;<com_ent> -> <lista_instrucciones>

En el punto 1, se procesará a la lista de instrucciones si y sólo si <expr> es verdadero.

En el punto 2 se procesará la lista si el comando de entrada es correspondido por el de salida del otro proceso con el que se comunica; 'corresponder' es el indicativo del establecimiento de la comunicación entre los procesos.

El punto 3 es una combinación de los dos anteriores, y se ejecutará la lista de instrucciones si se cumple que:

a).- <expr> es verdadero y
b).- el comando de entrada sea correspondido.

A continuación se analizarán las condiciones bajo las cuales no se procesan las listas de instrucciones.
Cuando en un comando custodiado no se ejecuta su correspondiente lista de instrucciones, se dice que el comando ha fallado.

Respecto a los comandos del primer tipo (punto 1), no se procesará la lista si la expresión <expr> es falsa, en cuyo caso hace fallar al comando; los comandos del segundo tipo fallan si su instrucción de entrada también falla, siendo la causa más común la terminación del proceso con el que esperan comunicación; los del tercer tipo fallan por cualquiera de las dos siguientes condiciones:

a.)- <expr> es falso.
b).- Si <expr> es verdadero y el comando de entrada <com_ent> falla.

Finalmente, falta señalar que los comandos custodiados indexados constituyen un conjunto o arreglo, en el que cada elemento es un comando custodiado que posee el mismo guardia y la misma lista de instrucciones que los demás, pero se diferencian del resto por los distintos valores que toman las variables indexadas, y por el valor de sus índices.

Las variables indexadas pueden formar parte del guardia o de la lista de instrucciones.

Los comandos custodiados de tipo indexado le proporcionan a CSP una ventaja sobre otros lenguajes concurrentes. Considere el siguiente ejemplo:

```
P::(i:1..100)X(i) ? (parámetros) -> ...; X(i) ! (resultados)
```

'P' es un proceso de tipo servidor, el cual recibe de sus clientes un conjunto que procesa, para devolverles los resultados; los clientes son elementos procesos de la familia 'X', X(i) es el proceso i-ésimo de la familia X.

El primer proceso en solicitar servicio de P tiene el acceso exclusivo del proceso P, y hasta que no se le envíen los resultados a este cliente, ningún otro proceso cliente puede ser correspondido por el servidor P.

En otros lenguajes, es necesario añadir variables y comandos para obtener la misma propiedad que CSP presenta mediante el uso de comandos custodiados indexados.

5.3.4- Conclusiones.

Una de las características que mejor contribuye a hacer de CSP un lenguaje concurrente y fácil de programar, es la propiedad síncrona de su mecanismo de comunicación.

Los comandos custodiados son la herramienta que tiene el programador para sincronizar procesos en forma condicional.

CSP
6.- INDETERMINISMO

Existen en CSP dos mecanismos que le proporcionan su característica de indeterminismo:

1.- Comando alternativo.
2.- Comando repetitivo .

Un comando alternativo está constituido por un grupo de comandos custodiados, y su estructura sintáctica es semejante a la del 'case' de Pascal.

Un comando repetitivo consiste en una estructura sintáctica de tipo ciclo o 'loop', cuyo cuerpo es un comando alternativo.

6.2- Sintaxis.

La sintaxis de un comando alternativo es la siguiente:

<comando alternativo> ::=
[<comando custodiado> { [] <comando custodiado>}]

Cada comando se separa de los demás por el símbolo '[]', y todos están agrupados entre dos corchetes ('[', ']').
Recuerde que un comando custodiado en CSP posee la siguiente sintaxis:

<comando custodiado> ::= <guardia> -> <lista_instrucciones>

Por lo que un comando alternativo tiene la siguiente estructura:

```
[  guardia_1 -> lista de instrucciones_1
[] guardia_2 -> lista de instrucciones_2
     .               .
[] guardia_n -> lista de instrucciones_n ]
```

La sintaxis de un comando repetitivo es:

<comando repetitivo> ::= * <comando alternativo>

El asterisco es el símbolo sintáctico que indica repetición. La estructura de un comando repetitivo es la siguiente:

```
* [  guardia_1 -> lista de instrucciones_1
  [] guardia_2 -> lista de instrucciones_2
       .               .
  [] guardia_n -> lista de instrucciones_n ]
```

6.3- Semántica.

En la ejecución de un comando alternativo se pueden tener tres estados:

1.- Falla.
2.- Exito.
3.- Alerta.

Un comando alternativo falla cuando todos los comandos custodiados que lo componen también fallan. En tal caso, se genera la falla o aborto del proceso al cual constituye.

Si sólo uno de los comandos custodiados constituyentes no falla, y su comando de entrada es correspondido (establece comunicación con el otro proceso), entonces se procesa su correspondiente lista de instrucciones, después de lo cual finaliza la ejecución del comando alternativo.

En este último caso, y en el que se mencionará en el siguiente párrafo, la ejecución del comando alternativo es exitosa.

También se considera exitosa la ejecución de un comando alterna-

tivo cuando varios de sus comandos custodiados constituyentes (o todos) no fallan y son correspondidos. En tal caso, se selecciona a uno de ellos indeterminísticamente y se procesa su correspondiente lista de instrucciones, con lo que finaliza exitosamente el procesamiento del comando alternativo.

Un comando alternativo está en estado de alerta si se tienen las siguientes condiciones:

1.- Al menos un comando custodiado no falla y
2.- de estos, ninguno ha sido correspondido.

En este caso, no han terminado su ejecución los procesos que corresponden a los comandos de entrada de los guardias candidatos.

El comando está en espera de que algún comando de entrada sea correspondido, y si más de uno lo llega a estar, se realiza una selección indeterminística y se procesa la lista de instrucciones del elegido.

Si el comando alternativo no falla, entonces finalmente obtendrá el estado de éxito. Una vez obtenido se procesa la instrucción subsecuente al comando alternativo en el proceso.

Como ya se ha indicado, un comando repetitivo es la repetición de un comando alternativo.

La ejecución de un comando repetitivo consiste en el procesamiento continuo de un cuerpo de instrucciones integrado por un comando alternativo. El cuerpo de instrucciones se reprocesará mientras el comando alternativo termine en estado de éxito, finalizará su procesamiento en el momento que el comando alternativo falle, y se suspenderá su ejecución mientras el comando alternativo esté en estado de alerta.

Un comando repetitivo nunca falla, pues termina normalmente su ejecución cuando falla el alternativo que lo constituye.

6.4- Conclusiones.

Los comandos alternativo y repetitivo son los únicos medios con que cuenta CSP para introducir el indeterminismo.

Usando estos comandos las interacciones entre un proceso y los demás con los que mantiene comunicación, són completamente indeterminísticas.

Lo anterior es debido a que no es posible determinar en qué orden y en qué instantes de tiempo intenten los demás procesos comunicarse con él. Un ejemplo clásico es el de un proceso servidor que puede atender a n procesos clientes.

Para dar servicio en forma no determinística se hace uso de un comando repetitivo, en donde cada comando custodiado estará formado por uno de entrada que referencie a cada proceso cliente.

La estructura sería algo semejante a lo siguiente:

```
* [  proceso_cliente_1  ?  servicio -> ....
  [] proceso_cliente_2  ?  servicio -> ....
  [] proceso_cliente_3  ?  servicio -> ....
          .                    .
  ]
```

El comando repetitivo atenderá indeterministicamente a los clientes del proceso servidor del cual forma parte.

Este comando finalizará su ejecución cuando todos los procesos clientes ya hayan terminado su procesamiento.

CSP
7.-CALENDARIZACION (SCHEDULING)

Respecto a este punto, CSP no posee ningún mecanismo que permita calendarizar un recurso compartido a un conjunto de procesos e, incluso, Hoare no indica que los procesos se ejecuten con prioridades.

CSP es por lo tanto pobre en lo que respecta a este punto, sin embargo, la forma en que se puede asignar un recurso a un conjunto de procesos es mediante el uso de familias de procesos.

En el siguiente ejemplo se muestra a un despachador del tipo shortest-job-next scheduler que ejemplifica este punto.

A continuación se presentan dos ejemplos que permiten ejemplificar e ilustrar lo que se ha analizado en el presente capítulo.

EJEMPLOS

Despachador shortest- job -next scheduler

Se trata de un despachador en el cual cada proceso que desee tener acceso al recurso compartido deberá proporcionar su tiempo aproximado de ejecución, y aquél que se registre con el menor tiempo es el que gana el acceso al recurso. Basado en [Welsh, Lister, 1979].

```
SJN::
     cola : (1..n) integer;
     -- es en donde se registran los tiempos de ejecución de
     -- los usuarios
     usuario, candidato : integer;
     -- el  usuario  es  el proceso que  tiene  asignado  el
     -- recurso compartido, y el candidato es el siguiente a
     -- asignársele una vez que quede libre
     j, k: integer;  -- variables auxiliares
     usuario := -1; candidato := -1; k := 0;
     --    -1  significa que no está asignado el candidato
     --    o el usuario, k indica el número de usuarios que
     --    han solicitado acceso, y están en lista de espera

     *[ (i:1..n) usuario(i) ? solicita (tiempo:integer) -->
          cola(i) := tiempo; k:= K + 1;
          candidato := -1;
          -- solicitud del usuario,  en la cola se  registra
          -- el tiempo del usuario i-ésimo y se inicializa
          -- al  candidato  con  objeto de  que  este  nuevo
          -- usuario sea elegible

     []
          (i :1..n) usuario(i) ? libera() --> usuario := -1;
          -- el recurso es liberado por el usuario actual y
          -- deja de existir usuario actual
     []
          candidato = -1; k <> 0 -->  -- selección de
                                      -- candidato nuevo
          j := 1; minimo : integer; minimo := maxinteger;
               *[ j <= n; cola(i) <> 0 -->
                    -- lee todo el arreglo, y sólo
                    -- considera a aquellos elementos
                    -- que existan cola(i) <> 0
                    [
                         cola(i) >= minimo --> skip
                    []
                         cola(i) < minimo -->
                              candidato := i;
                              minimo := cola(i)
                    ]
                    j := j + 1  -- lee siguiente elemento
               ]

               -- si cola(i) = 0 implica que el usuario
               -- i-ésimo  no  ha  sido  registrado  ni  ha
               -- solicitado acceso (no existe)
          []
               usuario = -1; candidato <> -1 -->
                    usuario(candidato) ! ok();
                    -- asigna   el   recurso  al   candidato
                    -- actual,   exclúyelo  de  la  cola   y
                    -- contabiliza
               cola(candidato) := 0;
               k := k - 1;
```

```
                usuario := candidato;
                candidato := -1;
                     -- al hacerse nuevo usuario deja de ser
                     -- candidato
          ]
```

LLamadas de los procesos usuarios:

```
solicitud de registro:          SJN ! solicita(tiempo)
solicitud de acceso:            SJN ? ok()
liberación del recurso:     SJN ! libera()
```

Los cinco filósofos

Otro ejemplo es el conocido como el problema de los cinco filósofos, basado en [Andrews, 1981].
Se trata de cinco filósofos que se pasan su vida comiendo y pensando. Ellos comen en una mesa circular con cinco sillas, y en la que únicamente están cinco tenedores, de tal manera que sólo existe un tenedor entre dos sillas adyacentes. Cada filósofo se sienta en una silla, las sillas están numeradas de 1 a 5 y los filósofos heredan el número de la silla como identificación.
Para que un filósofo pueda comer se necesita que tome los dos tenedores adyacentes a su lugar, de tal manera que mientras unos comen otros deben esperar a que sus vecinos dejen de usar los tenedores.

Los filósofos requieren entrar al comedor para poder comer, y cuando han terminado de comer lo abandonan.

En este programa se hace uso de un proceso COMEDOR y dos familias de procesos: filósofo y tenedor.

El programa principal es:

```
PRINCIPAL::
     [ COMEDOR || filósofo(i:1..5) || tenedor(i:1..5) ]
-- se ejecutan en paralelo todos los procesos
```

Familia de procesos filósofos

```
filósofo (i :1..5) ::
     número_veces_come : integer;  número_veces_come := 0;
     límite : integer; límite := 10000;

     -- las variables número_veces_come y límite se usan
     -- para indicar la vida del filósofo, ya que no puede
     -- comer más que 10000 veces en su vida,  después de lo
     -- cual muere

     *[   -- DESEA COMER
          número_veces_come <= límite; comedor ! entra();
            comedor ? ok() -->
```

```
            -- el filósofo i solicita acceso al comedor
            -- mediante la llamada comedor ! entra(), y
            -- espera a que se le dé acceso a través
            -- del comando comedor ? ok()
        tenedor(i) ! recoge(); tenedor(i) ? otorga() -->
            -- solicita en primer lugar recoger su
            -- tenedor, para lo cual invoca al comando
            -- recoge del proceso tenedor, el que le res-
            -- ponde mediante el comando otorga
        tenedor(i mod 5 + 1) ! recoge();
        tenedor(i mod 5 + 1) ? otorga() -->
            -- solicita el acceso al tenedor anexo a su
            -- lugar, para poder tener ambos tenedores y
            -- poder comer
            -- durante este tiempo el filósofo se dedica
            -- a comer, y ninguno de sus vecinos podrá
            -- quitarle los tenedores
            número_veces_come := número_veces_come + 1
                -- otro dia que come en su vida
        -- termina de comer
        tenedor(i) ! libera();
            -- vuelve a poner ambos tenedores en su
            -- lugar, los libera
        tenedor(i mod 5 + 1) ! libera();
            -- como se puede observar, al liberar los
            -- tenedores ya no existe necesidad de que el
            -- proceso tenedor le de permiso
        comedor ! sale();
            -- abandona el comedor

    []  -- PIENSA
        *[j:integer; j:= 1000000;
          número_veces_come <= límite; j > 0 --> j:=j-1
         ]
        -- la acción de pensar se simula mediante un
        -- comando repetitivo, con variable de iteración j
    ]

        Familia de procesos tenedores

tenedor(i:1..5)::

        -- los únicos filósofos que pueden recoger al
        -- tenedor i son el filósofo i y el (i mod 5) + 1

    *[  -- filósofo i

        filósofo(i) ? recoge() --> filósofo(i) ! otorga();
        -- el filósofo i solicita recoger el tenedor
        filósofo(i) ? libera() --> skip;
        -- deja de usar al tenedor, y vuelve a quedar
        -- libre

    []  -- filósofo i mod 5 + 1 (el vecino)
```

```
            filósofo(i mod 5 + 1) ? recoge() -->
            -- el otro filósofo solicita recoger el tenedor
            -- y se le otorga si está libre
                filósofo(i mod 5 + 1) ! otroga();
            filósofo(i mod 5 + 1) ? libera() --> skip;
            -- deja el tenedor
      ]
```

Proceso comedor

```
COMEDOR::
      j,i: integer; i := 0;
      -- este  proceso es el que se encarga de dejar pasar  a
      los filósofos al comedor, el objetivo es no dejar pasar
      más de 4 porque se puede presentar un abrazo mortal

      *[
         i < 4; filósofo(j) ? entra() -->
           -- el filósofo j-ésimo desea ingresar al comedor
           -- si no hay más de 4 se le otorga el acceso i < 4
           i := i + 1; filósofo(j) ! ok()
           -- se contabiliza su ingreso y se le da permiso
      []
          filósofo(j) ? sale() --> i := i - 1;
           -- termina  de  comer  el  filósofo  j,  abandona  el
           -- comedor y se contabiliza su egreso
       ]
```

SR (Sincronización de Recursos)

1.- Origen y Objetivos

'SR' son las iniciales de "Sincronización de Recursos" (Synchronizing Resources), que es un lenguaje de programación concurrente, que utiliza tanto el envío de mensajes como el uso de variables compartidas como mecanismos de comunicación y sincronización entre procesos.

El diseño del lenguaje se inició en la primavera de 1980, aunque los mecanismos de interacción entre procesos fueron diseñados en 1979.

Este lenguaje fue diseñado por Gregory R. Andrews en el Departamento de Ciencias de la Computación de la Universidad de Arizona en Tucson, [Andrews, 1981, 1982].

Los objetivos fundamentales sobre los que se fundamentó el diseño del lenguaje SR fueron:

1.- Ser un lenguaje de programación de sistemas.
2.- Aplicarse a arquitecturas con múltiples procesadores.
3.- Ser un enlace entre lenguajes diseñados para distintas arquitecturas.
4.- Proveer mecanismos de programación de alto y bajo nivel.

Respecto a los dos primeros puntos, 'SR' fue concebido para la programación de sistemas que controlan arquitecturas con múltiples procesadores y se considera que estos procesadores pueden estar conectados en una variedad de formas, por lo que no es posible clasificar a estas arquitecturas como distribuidas (memoria distribuida), o de memoria compartida.

Por estas razones, es importante que SR sea capaz de manejar la interacción entre procesos mediante el acceso a memoria compartida y también a través del envío de mensajes.

Este último párrafo explica por sí solo el tercer punto, ya que existen lenguajes diseñados específicamente para arquitecturas de memoria compartida y otros confeccionados para implementarse en redes de procesadores, que no comparten memoria.

En relación al cuarto y último punto, SR fue diseñado con la característica de tener mecanismos de bajo y alto nivel; los mecanismos de bajo nivel permiten controlar y programar a los dispositivos periféricos, mientras que los de alto nivel son requeridos para la solución de problemas fundamentales, asociados con programación concurrente en un conjunto de procesos.

Los mecanismos de alto nivel comprenden la comunicación, sincronización, control de acceso, indeterminismo y asignación de recursos, entre otros.

Como dato relevante, es interesante mencionar que el lenguaje SR se utilizó para la implementación del sistema operativo UNIX en versión distribuida.

SR
2.- PROCESOS

En el lenguaje SR se agrupa a los procesos en entidades llamadas Recursos (resources), y un programa está constituido por un conjunto de ellos.

Con objeto de diferenciar a la entidad Recurso de SR de la palabra 'recurso', se representa a lo largo de este capítulo a la entidad 'Recurso' con letra inicial mayúscula.

En esta sección se irá describiendo a detalle cada una de estas entidades. En primer lugar se analizará la estructura y propiedades de la entidad Recurso.

En segundo lugar se estudiará al proceso, ya concebida su estructura como parte del Recurso.

Un Recurso está formado por los siguientes elementos:

1.- Procesos.
2.- Variables permanentes.
3.- Instrucciones de inicialización.

Las variables permanentes son locales al Recurso, pero globales al conjunto de procesos que lo integran. Las instrucciones de inicialización únicamente son ejecutadas cuando se crea el Recurso, y se les utiliza para inicializar a las variables permanentes.

Las variables permanentes pueden ser leídas, referenciadas y asignadas únicamente por los procesos que constituyen al Recurso.

Un proceso está constituido por una lista de instrucciones, y opcionalmente por un conjunto de variables locales a él.

Entre las instrucciones que lo integran existen las de entrada/salida, a las cuales se les denomina 'operaciones' en el lenguaje SR. Un proceso puede tener más de una operación definida en su cuerpo.

Las operaciones son las puertas de comunicación del proceso, cuando ésta se realiza mediante el envío de mensajes; de manera que todo proceso que desee comunicarse a través de mensajes con otro, deberá hacerlo mediante llamadas a sus operaciones.

El conjunto de operaciones definidas en los procesos de un Recurso normalmente tiene por objetivo controlar y operar a un objeto. Por ejemplo, un Recurso cuyas operaciones logran el acceso a una base de datos, otro que maneja al controlador de algún dispositi-

vo periférico (disco, cinta, etc.).

Todos los procesos que constituyen un Recurso pueden compartir las variables permanentes del mismo; sin embargo, la interacción entre procesos de distintos Recursos es únicamente mediante el envío de mensajes.

2.3- Sintaxis.

La sintaxis de un programa es la siguiente:

<programa> ::=
<declaraciones_de_tipos> <Recurso>{<Recurso>}|<Recurso>{<Recurso>}

Un programa puede tener declaraciones de tipos, globales a todos los Recursos que lo componen. Además, posee la declaración del conjunto de todos los Recursos que lo constituyen.

La sintaxis de un Recurso es:

```
<Recurso> ::= resource <nombre> [<declaración_rangos>];
                  <declaraciones>
<declaraciones> ::= <especificaciones_I/O>;[<cuerpo>]
<cuerpo> ::= [<declaraciones>;][<instruc_inicialización>;]
                <proceso>{;<proceso>}
<declaraciones> ::= [<declaración_tipos>;]
                    [<declaración_operaciones>;]
                    [<declaración_var_permanentes>;]
```

El símbolo <nombre> identifica al Recurso, 'declaración_rangos' es optativo e indica la posibilidad de tener familias (arreglos) de Recursos.

Se pueden tener no únicamente vectores, sino también matrices de Recursos. Cada elemento del arreglo, matriz o familia es un Recurso que posee el mismo nombre, las mismas declaraciones, procesos, variables e instrucciones, y se diferencia de los demás de la familia por el valor que tienen sus índices, así como por el valor que poseen sus variables y demás elementos indexados. Todo índice debe tener un valor que corresponda a su rango.

En SR toda instrucción va separada de sus elementos adyacentes por el símbolo ';'.
El símbolo 'especificaciones_I/O' nombra a aquellos elementos que serán visibles e invocables desde otros Recursos, así como aquellos objetos pertenecientes a otros Recursos y que se requiere invocar o referenciar. A los primeros se les llama objetos exportados y a los segundos objetos importados. Las especificaciones son por lo tanto de exportación y de importación. Su sintaxis es la siguiente:

```
<especificaciones_I/O> ::= [<especificaciones_exportación>;]
                           [<especificaciones_importación>]
<especificaciones_exportación> ::=
                  export <lista_objetos> [to <lista_Recursos>]
```

```
<especificaciones_importación> ::=
                    import <lista_objetos> [from <lista_Recursos>]
<lista_Recursos> ::= <nombre_Recurso> {,<nombre_Recurso>}
<lista_objetos> ::= <objeto> {,<objeto>}
<objeto> ::= <constante> | <rango> | <tipo_dato>
           |  <operación>
```

Como puede observarse, los objetos exportables e importables pueden ser constantes, rangos, tipos de datos y operaciones.

El símbolo <cuerpo> representa la estructura propia del Recurso, y está integrado por los siguientes elementos:

1.- Declaraciones (tipos, variables, operaciones, ..).
2.- Instrucciones de inicialización.
3.- Declaraciones de procesos.

En el primer punto se incluyen declaraciones de:

1.- Tipos de datos.
2.- Constantes.
3.- Variables permanentes.
4.- Operaciones.
5.- Rangos.
6.- Procedimientos.

No se incluyeron todos los elementos en la representación de Backus Naur, con objeto de no hacerla más compleja.

Si existe la declaración de variables permanentes, entonces deben incluirse las instrucciones de inicialización de las mismas en el cuerpo del Recurso.

Todas las operaciones que sean implementadas en los procesos que constituyen al Recurso deben de declararse previamente con la siguiente sintaxis:

```
<declaración_operaciones> ::= <declaración_operación>
                                {;<declaración_operación>}
<declaración_operación> ::=
          op <nombre_operación> ( ) {<restricciones>}
        | op <nombre_operación> ( <lista_parámetros>)
          {<restricciones>}
```

Su sintaxis y su semántica se analizarán con más detalle en la sección de comunicación.

Finalmente, en último lugar aparece la declaración de los procesos que constituyen el Recurso.

La sintaxis de un proceso en SR es:

```
<proceso> ::= process <nombre_proc>[<dec_rangos>];<cuerpo_proc>
<cuerpo_proc> ::= [<dec_variables>;]<lista_inst> end
```

El símbolo nombre_proc identifica al proceso; dec_rangos permite tener una familia de procesos. Cada miembro de la familia posee

el mismo nombre que los demás, pero se identifica por el valor de sus índices. Todos los miembros poseen las mismas variables e instrucciones, pero se diferencian por el valor de sus variables o instrucciones indexadas.

La sintaxis de la declaración de rangos es exactamente igual a la de los Recursos. Se permite tener más de un rango, y consecuentemente tener matrices de procesos.

El símbolo dec_variables representa la declaración de variables locales al proceso, que pueden ser accesadas únicamente por las instrucciones que constituyen al proceso.

2.4- Semántica.

Se puede concebir un Recurso como una barrera que encierra a un conjunto de variables y procesos, cuyas únicas puertas de acceso son las instrucciones de comunicación por envío de mensajes, llamadas operaciones e implementables en los procesos.

En el desarrollo de esta subsección se irá describiendo el significado semántico, de los elementos sintácticos que representan Recursos y procesos.

Los elementos sintácticos que representan a un Recurso son:

1.- <dec_variables_permanentes>
2.- <instruc_inicialización>
3.- <especificaciones_I/O>
4.- <procesos>

En relación al punto 1, las variables permanentes, al ser referenciadas por cualquier proceso constituyente del Recurso, sirven como medio indirecto de comunicación entre los mismos procesos.

Lo anterior está en relación directa con la arquitectura del sistema de cómputo, ya que implica el uso de memoria compartible por todos los procesos del Recurso.

Si se habla de una red de procesadores (cada uno con su propia memoria), entonces se requiere que cada Recurso esté cargado en un nodo de la red, por la dificultad que representa el accesar memoria de un nodo a otro de la red (se formaría un cuello de botella).

El punto 2 representa las instrucciones de inicialización de las variables permanentes; estas instrucciones son únicamente de asignación, su función es única y exclusivamente proporcionar un valor inicial a las variables permanentes del Recurso.

El punto 3 representa las especificaciones de exportación y de importación, su función es controlar el acceso al Recurso.
Entre los procesos de un programa existen los siguientes niveles de acceso:

1.- Con todos los privilegios de acceso.

2.- Con privilegios restringidos de acceso.

En el primer caso están todos los procesos que constituyen un Recurso. Todo proceso puede exportar e importar cualquier operación de cualquier otro proceso que pertenezca a su mismo Recurso.

En el segundo caso están todos los procesos externos al Recurso. Para que un proceso P perteneciente a un Recurso Z pueda importar un objeto de un proceso Q, perteneciente a otro recurso W es necesario que:

1.- El objeto esté declarado en la lista de importaciones de su Recurso Z.
2.- El objeto esté declarado en la lista de exportaciones del Recurso W.

Si no se cumplen estos dos requisitos, el objeto no puede ser importado. Si las especificaciones no poseen listas de Recursos, entonces en el caso de exportación, la lista de objetos se exporta a todos los Recursos del programa; en el de importación, los objetos poseen un único identificador en el programa (no hay dos objetos con el mismo nombre).

Finalmente, un proceso cobra vida en el momento que se crea el Recurso al cual constituye (se analizará con mas detalle en la sección de paralelismo). Finaliza su ejecución cuando ha terminado de procesar su lista de instrucciones.

Un proceso puede estar en estado de "listo" (ready) o en estado de bloqueo; en "listo" compite por uso del procesador. Se puede bloquear cuando llama a una operación y espera resultados, o bien cuando es un proceso servidor que implementa varias operaciones y ninguna de ellas ha sido llamada o referenciada.

2.5- Conclusiones.

Como consecuencia de la sintaxis de SR se concluye que:

1.- Ningún Recurso puede ser parte componente de otro.
2.- Ningún proceso puede estar declarado como parte de la declaración de otro.
3.- Ningún Recurso puede ser parte de un proceso.

Existe por lo tanto una jerarquía bien definida en la estructura de SR.

Por otra parte, y como consecuencia del punto 2, ningún proceso puede estar declarado en función de él mismo, lo cual evita que se tenga recursividad.

Una limitación es que todo Recurso debe caber en un nodo de la red, si se implementa en una arquitectura de tipo red.
Si se habla de que dos procesadores de la red comparten memoria, entonces el Recurso debe albergarse en la memoria compartida.

Una gran ventaja que presenta SR es que a través del uso de Recursos, puede implementar operaciones a distintos niveles de aplicación, desde operaciones de bajo nivel que controlan dispositivos de hardware, o implementan primitivas del núcleo de un sistema operativo.

A otro nivel, el programador implementa otro conjunto de Recursos que emplean operaciones de los de más bajo nivel. Se van implementando de esta manera una serie de capas que proporcionan servicios, e implementan operaciones a distintos niveles. Cada capa usa operaciones y objetos de capas inferiores.

Por esta razón, la estructura de SR es idónea para la implementación de sistemas operativos. Como ya se mencionó en la sección de Origen y Objetivos, SR se usó en la implementación de un UNIX distribuido.

SR
3.-PARALELISMO

El lenguaje SR posee un paralelismo implícito, ya que la secuencia en que se crea un programa es la siguiente:

1.- Se cargan los Recursos en memoria.
2.- Se inicia la creación de los Recursos.
3.- Se crean los procesos del Recurso.
4.- Se inicia la asignación compartida del procesador a todos los procesos.

En este caso no se especificó el tipo de arquitectura en que se opera; si se trata de un solo procesador, entonces todos los Recursos se cargan en la misma memoria, y si se trata de una red, entonces cada Recurso se carga en un solo nodo de la red.

Como puede observarse, a partir del punto 4 todos los procesos de todos los Recursos se están ejecutando concurrentemente. No se requiere un comando que explícitamente señale a todos los procesos que van a ejecutarse concurrentemente. Sin embargo, existe una instrucción de invocación de tipo paralelo.

Una instrucción de invocación solicita la ejecución de una operación con un conjunto dado de parámetros actuales. Se trata en realidad de una instrucción de entrada/salida.

La instrucción concurrente de invocación contiene una o más instrucciones de llamada que son invocadas en paralelo (concurrentemente). Y termina cuando todas las llamadas han terminado.

Una instrucción de llamada envía parámetros y espera recibir resultados.

3.2- Sintaxis.

La sintaxis de una instrucción concurrente de invocación es:

<inst_concurrente_invoc> ::= co <lista_instruc_llamada> oc
<lista_instruc_llamada> ::= <instruc_llamada>;
<instruc_llamada>{;<instruc_llamada>}
<instruc_llamada> ::= call <nombre_operación>(<lista_parámetros>)

La instrucción está identificada por las palabras clave 'co' y 'oc'.
El símbolo lista_instruc_llamada representa la lista de 'instrucciones de llamada, de las cuales debe haber al menos dos. Cada instrucción de llamada está identificada por la palabra call.

<nombre_operación> es el nombre de la operación que se invoca, conjuntamente con su lista de parámetros. En esa misma lista se obtienen resultados que genera la operación invocada.

Es importante hacer notar que aún si esta instrucción no se ejecuta, existe concurrencia en el procesamiento de los procesos. Su uso se justifica cuando existe la necesidad de invocar simultáneamente varias operaciones.

3.3- Semántica.

En esta parte de semántica se hará una descripción de la creación de un programa, así como de su terminación. En la misma pueden verse claramente las características de concurrencia de SR.

Como se había mencionado previamente, todos los procesos, una vez creados, se ejecutan concurrentemente con los demás.

Un programa en SR se inicia con la ejecución concurrente de las instrucciones de inicialización de cada Recurso, y la creación de los procesos.

Un programa está formado por un grupo de módulos; en cada procesador de la red (considérese una arquitectura de tipo red) se carga a un módulo, el cual contendrá al conjunto de Recursos que se procesarán concurrentemente en ese procesador.

Existe una cola de operación por cada operación implementada en los Recursos de cada módulo. Recuérdese que una operación es una instrucción de entrada/salida que reside en un proceso, y puede ser invocada por procesos ubicados en el mismo Recurso, en el mismo módulo o en módulos externos.

En la cola de operación se registran todas las invocaciones o llamadas pendientes de ser atendidas. Los parámetros de las llamadas se registran en un área de parámetros.

La creación de los Recursos se realiza en los siguientes puntos:

1.- Inicialización de las variables permanentes.
2.- Creación de los procesos que los constituyen.

Un punto importante que hay que hacer notar es que la referencia a las variables permanentes de un Recurso es indivisible. Es decir, su asignación o escritura no es interrumpida por ningún evento.

Un proceso termina cuando ha ejecutado la lista de instrucciones que lo componen. Un proceso finaliza anormalmente si falla la ejecución de alguna de las instrucciones que lo componen.

La falla de una instrucción trae como consecuencia el aborto del proceso del cual forma parte. Un ejemplo de una instrucción fallida es la división por cero. Un programa en SR termina cuando los procesos que integran a sus Recursos se encuentran en cualquiera de los dos siguientes estados:

1.- Cuando han terminado.
2.- Cuando están bloqueados o suspendidos, y además no existen dispositivos activos.

Un proceso está bloqueado, entre otras causas, por las dos siguientes:

1.- Es proceso cliente, y está esperando la culminación del servicio que solicitó.
2.- Es proceso servidor, y está a la espera de ser solicitado por algún cliente.

La existencia de un dispositivo activo implica la de al menos un proceso activo, ya que todo dispositivo es manejado por un conjunto de procesos.

Un proceso activo puede estar bloqueado, en espera de que el dispositivo termine de efectuar su operación actual.

Cuando todos los Recursos que constituyen un programa han terminado, entonces el sistema reconoce que el programa ha finalizado su ejecución.

3.4- Conclusiones.

Las ventajas que presenta la característica de paralelismo del lenguaje SR son:

1.- La existencia de paralelismo desde la inicialización del programa.
2.- El apoyo de una instrucción explícita que genera concurrencia.
3.- Transparencia de la red.

La inicialización de los Recursos y sus procesos es concurrente, por lo que existe paralelismo desde la inicialización del programa.

En relación al segundo punto, existe la instrucción concurrente de invocación 'co...oc'.
Finalmente, el protocolo de comunicación de la red la hace

transparente al usuario.

Una restricción que presenta SR es que una vez creados los procesos no es posible crear dinámicamente otros, lo cual no es deseable, ya que puede requerirse la creación de procesos según la lógica del propio programa.

SR
4.- Comunicación

En el lenguaje SR, la comunicación entre procesos se hace mediante los siguientes mecanismos:

1.- Envío de mensajes.
2.- Uso de variables compartidas.

Respecto al esquema de un mecanismo por envío de mensajes, el transmisor es un proceso cuya función es iniciar la comunicación con el receptor y enviar un mensaje. El receptor es el elemento pasivo, cuya función es recibir el mensaje y atender la solicitud emitida por el transmisor.

En el lenguaje SR sí se permite que a través del mismo mensaje el receptor envíe información (resultados) y el transmisor los reciba. El mensaje está formado por un identificador y un conjunto de parámetros, que constituyen los datos a transferir.

El canal o medio de comunicación puede ser la misma red de procesadores que constituye la arquitectura del sistema (si la arquitectura es de tipo red). Si se trata de una arquitectura de un solo procesador o de un conjunto de éstos que comparten memoria, entonces el canal está implementado a través de la memoria.

Los procesos que manejan los protocolos de comunicación de la red de procesadores, en el caso de una arquitectura de red, cumplen con los siguientes puntos:

1.- Toda transferencia de mensaje es siempre correcta.
2.- Se conserva el orden de envío de los mensajes.

Respecto al primer punto, el protocolo siempre garantiza que todo mensaje enviado sea siempre recibido sin errores. El punto 2 se refiere a que todos los mensajes enviados por un proceso son recibidos en el mismo orden en que fueron enviados.

El uso de variables compartidas es un mecanismo de interacción entre procesos, que se analizará detalladamente en la siguiente sección.

4.2- Mecanismos de comunicación.

De los mecanismos precursores, únicamente interesa hacer alusión a los que influyeron en forma determinante en el diseño del lenguaje.

Los mecanismos o formas de comunicación que influyeron en el diseño son:

1.- Monitores.
2.- Envío de mensajes con almacenamiento (buffered).

Respecto al primero, por su característica de aislar a un conjunto de datos compartidos y los procedimientos que operan sobre ellos en una barrera cerrada, con unas cuantas puertas de acceso, presenta las siguientes propiedades:

1.- Los procesos generalmente deben esperar.
2.- Su utilización está enfocada a arquitecturas con memoria compartida.
3.- Maneja elementos de bajo nivel, tales como variables de condición.

El primer punto es consecuencia del acceso exclusivo al monitor, ya que éste únicamente puede estar ocupado por un proceso a la vez.

Por otra parte, si en algún procedimiento del monitor llamado por el proceso que lo utiliza en ese momento no se cumple una condición, entonces se bloquea al proceso en la cola de espera de la condición.

Como resultado de lo anterior, si un proceso espera leer uno de los datos compartidos que resguarda el monitor, deberá esperar el resultado de su petición.

Si el proceso escribe o modifica alguno de los objetos compartidos, no existe razón para que espere el acceso al elemento, ya que no espera recibir ningún resultado. Esto representa una seria desventaja en el uso de los monitores.

En relación al segundo punto, su utilización está más bien enfocada al uso de memoria compartida.
Lo anterior es consecuencia de que un monitor comparta datos, y que la interacción entre los procesos sea mediante el acceso a los mismos, razón por la cual deben de ubicarse en una memoria también compartida.

Las características en el diseño de SR que están en función de las ventajas y desventajas de los monitores son:

1.- SR debe implementarse eficientemente en arquitecturas de memoria compartida.
2.- Los procesos deben esperar (si se requiere hacerlo).

El primer punto indica que SR utilice el uso de variables compartidas como medio de comunicación, para una implantación en una arquitectura de memoria compartida.

En el segundo punto, SR hereda como característica de los monitores el uso de llamadas a procedimientos remotos por parte de los procesos, ya que todo proceso que desee comunicarse con el monitor debe hacerlo mediante una llamada a un procedimiento.

Sin embargo, esta característica no es siempre deseable como ya se analizó: para superar esta deficiencia en el mecanismo de comunicación, es necesario tomar las ventajas de la forma de comunicación por envío de mensajes con almacenamiento.

Las características de la comunicación por envío de mensajes con almacenamiento relevantes a SR son:

1.- Los procesos generalmente no tienen que esperar (bloquearse).
2.- Su utilización está más enfocada a arquitecturas distribuidas.

A manera de conclusión, las características respecto a comunicación que se heredaron de los mecanismos precursores y que se consideraron para el diseño del lenguaje SR son:

1.- Fácil implementación en cualquier arquitectura.
2.- Atención a procesos que no esperan recibir resultados, sin que éstos tengan que esperar (bloquearse).
3.- Atención a procesos que sí esperan resultados sin duplicar mensajes.
4.- Manejo de estructuras y mecanismos de concurrencia que no sean de bajo nivel (ejemplo de mecanismos de bajo nivel: variables de condición, semáforos, colas, etc.).

Respecto al primer punto, hace uso de:
a).- Variables compartidas para arquitecturas de memoria compartida.
b).- Envío de mensajes para arquitecturas distribuidas.

En lo concerniente al segundo punto, utiliza envío de mensajes con almacenamiento (buffered).

Finalmente, correspondiente al tercer punto, tiene la facilidad de utilizar llamadas a procedimientos para el manejo de envío de mensajes en forma síncrona, de una manera similar a como interacciona un proceso con un monitor.

SR tiene la característica de manejar más de un solo mecanismo de comunicación, con objeto de poder abarcar todas las propiedades enumeradas en los párrafos anteriores.

Lo anterior da lugar a que existan varios esquemas de interacción entre los procesos.

4.2.1 - Esquemas de interacción entre procesos.

Un esquema de interacción entre procesos es sinónimo de un mecanismo de comunicación, ya que es el esquema bajo el cual se comunican los procesos. Los esquemas de comunicación (interacción) entre procesos en el lenguaje SR son:
1.- Esquema de variables compartidas.
2.- Esquema cliente/servidor.

3.- Esquema de interconexión (tubería).

El esquema de variables compartidas se utiliza para implementarse en una arquitectura de memoria compartida.

Las variables compartidas son las variables permanentes de cada Recurso. Como ya se indicó, estas variables únicamente pueden ser accesadas por los procesos que constituyen el Recurso.

Se convierten de esta forma en un medio indirecto de comunicación entre los procesos que integran al Recurso. En este sentido, un Recurso se asemeja a un monitor.

Las variables permanentes son los datos compartidos; los procesos que las accesan son semejantes a los procedimientos que operan sobre los datos.
Todo proceso externo al Recurso debe interaccionar a través de los procesos del Recurso, y no del acceso a las variables permanentes.

Debido a que se comparten datos, y consecuentemente memoria, entre los procesos de un Recurso, es imprescindible que de implementarse en una arquitectura distribuida, todo Recurso esté cargado en una única memoria. En consecuencia, no existe el esquema de variables compartidas entre Recursos, aún cuando la implementación se realice en una arquitectura no distribuida.

El acceso a una variable compartida es indivisible (no es permitido que el acceso se interrumpa).

En un esquema cliente/servidor, los procesos se comunican mediante envío de mensajes y se tienen las siguientes características:

1.- Un proceso siempre es subordinado del otro.
2.- Generalmente siempre hay resultados que retornar.
3.- Un proceso siempre debe esperar al otro.
4.- La comunicación es síncrona.
5.- La comunicación es asimétrica.

En este esquema, un proceso solicita el servicio de otro proceso, por ejemplo, para el acceso a algún objeto compartido.
Al proceso solicitante se le llama cliente, y al que realiza el servicio se le denomina proceso servidor.
El proceso cliente inicia siempre la comunicación, mientras que el servidor es el ente pasivo que está en espera de ser solicitado.

Por tal razón, el proceso servidor siempre está subordinado al proceso cliente, y éste generalmente espera recibir resultados. Consecuentemente, al realizar la petición de servicio debe esperar (bloquearse) a que el servidor ejecute el servicio, y si existen resultados, entonces espera a que se le envíen.

Una característica del proceso cliente es que no puede continuar

con su ejecución hasta que el proceso servidor haya realizado el servicio.

Por las características expuestas en líneas anteriores, puede suceder cualquiera de los siguientes eventos:

1.- El proceso cliente espera a que el proceso servidor esté disponible para transferir el mensaje.
2.- El proceso servidor está disponible y espera a ser solicitado por el proceso cliente.

En ambos casos, uno de los dos procesos debe esperar para establecer la comunicación. Por esa razón, no se requiere almacenar temporalmente al mensaje.

Este tipo de comunicación es síncrona, ya que ambos procesos deben sincronizarse (esperarse) para establecer la comunicación. Por otra parte, debido a que el proceso servidor no requiere conocer la identidad de los clientes que los solicitan, resulta ser una comunicación con características de asimetría.

En un esquema de interconexión, los procesos se comunican también mediante el envío de mensajes. Las características principales de este tipo de interacción entre procesos son las siguientes:

1.- Ningún proceso tiene que esperar a otro.
2.- No existe retroalimentación de resultados entre los procesos.
3.- La comunicación es asíncrona y asimétrica.

En este esquema, todo proceso que transmite un mensaje continúa con su ejecución, sin ser bloqueado o esperar a que su mensaje sea atendido por el otro proceso receptor.

La razón que justifica el comportamiento anterior del proceso es que éste no espera recibir resultados.
En este esquema, un proceso transmisor puede enviar más de un mensaje al mismo proceso receptor, aún cuando éste último no haya atendido al primero.

Lo anterior implica que existe almacenamiento de los mensajes.
En 'SR' las propiedades de la transferencia de los mensajes en este esquema son:
1.- Todo mensaje enviado es recibido.
2.- Se mantiene el orden en que se envían los mensajes.

Estas propiedades se manejan con los protocolos de comunicación de la red de procesadores (en arquitecturas distribuidas), que posee las mismas propiedades.

En este esquema, el proceso transmisor no se coordina con el receptor para la transferencia del mensaje.
Por lo anterior, este esquema presenta una comunicación asíncrona, por el hecho de que un mensaje (o un conjunto de mensajes)

son enviados por un proceso y recibidos generalmente en un tiempo posterior por el proceso receptor; se crea una especie de tubería através de la cual fluyen los mensajes.

Por esa analogía, se le llamó a este esquema de tipo interconexión (tubería).

Un mensaje es la salida de un proceso y la entrada a otro, y viaja a través de un buffer, que es en realidad la interconexión que enlaza a los procesos.

La comunicación es asimétrica porque el proceso receptor no requiere conocer la identidad del transmisor.

4.3- Sintaxis de la Comunicación en SR.

En relación con los esquemas que utilizan envío de mensajes, las instrucciones que deben ejecutarse para establecer la comunicación son:

proceso:	Instrucción:
Esquema cliente/servidor	
cliente	call
servidor	input (operación)
Esquema de interconexión (pipeline)	
transmisor	send
receptor	input (operación)

Obsérvese que tanto el servidor como el receptor utilizan la misma instrucción en ambos esquemas.
La sintaxis de estas instrucciones es la siguiente:

Sintaxis de la instrucción CALL:

```
<call> ::= CALL <nombre_operación> ([<mensaje>])
<nombre_operación> ::=
      [<designación_Recurso>.]<designación_operación>
<designación_Recurso> ::=
                  <nombre_Recurso>['['<indice_Recurso>']']
<designación_operación> ::=
               <nombre_operación>['['<indice_proceso>']']
```

Nota: Los símbolos '[' y ']' son parte de la sintaxis del lenguaje, y se encerraron entre comillas para diferenciarlos de los símbolos [ ] que se usan para describir la sintaxis.

La sintaxis del mensaje se analizará en una sección subsecuente. El símbolo <nombre_operación> identifica el nombre del servicio u operación que el proceso cliente solicita se ejecute.

Las operaciones se implementan en los procesos, los cuales a su

vez constituyen a los Recursos.

Todas las operaciones implementadas por los procesos de un Recurso se declaran en la parte de declaraciones del mismo Recurso, de tal manera que toda operación es única en el Recurso en el que se implementa, a excepción de que esté definida en una familia de procesos, en cuyo caso se identifica, además del nombre, por el índice del proceso que la implementa.

Los símbolos <nombre_operación> e <índice_proceso> identifican una operación definida en una familia de procesos.

Si el nombre de la operación es único en el programa, entonces no existe necesidad de indicar el nombre del Recurso en el que se declara y se define, a excepción de que se trate de una familia de Recursos, en cuyo caso hay que indicar la instancia del Recurso en el que se implementa, mediante el propio nombre del mismo, <nombre_Recurso>, y el índice o identificador de la instancia, <índice_Recurso>.

La sintaxis de la instrucción SEND es muy semejante a la del CALL.

```
<send> ::= SEND <nombre_operación> ( )
         |  SEND <nombre_operación>(<mensaje>)
```

La sintaxis de <nombre_operación> ya fue descrita en la subsección anterior, por lo que no se incluirá aquí. La única diferencia sintáctica es la palabra clave SEND en lugar de CALL.

La instrucción INPUT es invocada por el proceso servidor o receptor (según el esquema de interacción) e implementa a la operación designada por el correspondiente CALL o SEND.
Esta instrucción puede definir más de una sola operación en su cuerpo. Todas las operaciones definidas en la instrucción INPUT deben estar declaradas en la sección de declaraciones del Recurso.

Esta instrucción es el único medio para definir e implementar operaciones; es la puerta de acceso al Recurso desde el exterior. Su sintaxis es la siguiente:

```
INPUT ::= IN <lista_comandos_operación> NI
<lista_comandos_operación> ::=
          <comando_operación>{[]<comando_operación>}
<comando_operación> ::=
     <designación_operación>[<expresiones_sinc_sched>] -
      ->  <lista_instrucciones>
<designación_operación> ::=
          <nombre_operación>([<mensaje>]) |  else
<expresiones_sinc_sched> ::= [<expresión_sincronización>]
                             [<expresión_scheduling>]
```

El símbolo 'IN' identifica a la instrucción INPUT, y 'NI' limita

el campo de acción de la instrucción. Cada comando de operación representado por el símbolo <comando_operación> define una única operación.

La estructura de un comando de operación consiste en los siguientes elementos:

1.- Identificación de la operación y el mensaje.
2.- Mecanismos de sincronización y calendarización (scheduling).
3.- Lista de instrucciones.

Los mecanismos de sincronización y calendarización se analizarán en las secciones correspondientes; la lista de instrucciones define a la operación, y su ejecución la implementa.

La operación se identifica por su nombre, <nombre_operación>. El mensaje será analizado en la siguiente subsección, y la operación puede no incluir mensaje.

La sintaxis de la declaración de las operaciones se mostró en la sección de procesos, sin embargo, se considera importante volverla a representar con más detalle en esta sección, debido a que no se incluyeron elementos muy importantes que serán descritos en la parte de semántica.

Sintaxis de la declaración de operaciones:

declaración de operación ::=
OP <nombre_operación>([<declaración_mensaje>])[<restricciones>]
<restricciones> ::= '{'<valores_restric.>'}'
<valores_restric> ::= CALL | SEND | CALL,SEND

Nota: Los símbolos '{', '}' son parte de la sintaxis del lenguaje, y se encerraron entre comillas para diferenciarlos de los símbolos {}, que se utilizan para indicar cero o más repeticiones del elemento que encierran.

El nombre de la operación, representado por el símbolo <nombre_operación>, debe ser el mismo que el definido en alguna de las instrucciones INPUT en los procesos del Recurso.

El símbolo <declaración_mensaje> representa la declaración del mensaje, corresponde al mensaje (<mensaje>) de la instrucción INPUT correspondiente.

La sintaxis del mensaje siempre es la misma, independientemente de que la instrucción que la emita sea un CALL o SEND, es decir, es independiente del esquema de interacción que se utilice. Su sintaxis es:

En proceso cliente o transmisor

<mensaje> ::= <lista_de_parámetros_actuales>
<lista_de_parámetros_actuales> ::=

```
                    <parámtero_actual>{,<parámetro_actual>}

                En proceso servidor o receptor

    <mensaje> ::= <lista_de_parámetros_formales>
    <lista_de_parámetros_formales> ::=
                    <parámetro_formal>{,<parámetro_formal>}
```

La sintaxis de la declaración del mensaje de cualquier operación definida en el Recurso es:

```
    <declaración_mensaje> ::=
            <declaración_lista_parámetros_formales>
    <declaración_lista_parámetros_formales> ::=
            <declaración_parámetro_formal>
            {;<declaración_parámetro_formal>}
    <declaración_parámetro_formal> ::=
       <restricción_tipo><nombre_parámetro_formal>:<tipo_dato>
    |  <nombre_parámetro_formal>:<tipo_dato>
    <restricción_tipo> ::= VAR | VAL | RES
```

La declaración del mensaje se incluye en la declaración de la operación, en la parte de declaraciones del Recurso.

En toda instrucción INPUT que referencie y defina una operación, deben coincidir los siguientes puntos con respecto a la declaración de la operación:

1.- Los nombres de las operaciones deben ser iguales.
2.- El número de los parámetros formales debe ser el mismo.
3.- El nombre de los parámetros formales debe ser el mismo.

Observe que en la instrucción INPUT ya no se indica el tipo de dato de los parámetros.

En lo que respecta a las instrucciones de invocación (CALL, SEND) de los procesos clientes o transmisores, debe cumplirse lo siguiente:

1.- El nombre de la operación que invocan debe ser igual al designado por la instrucción de entrada (INPUT) que define e implementa a la operación.
2.- El número de parámetros reales de la instrucción de invocación, debe coincidir con el número de parámetros formales de la instrucción de entrada invocada.
3.- El tipo de dato debe ser el mismo entre los parámetros reales y sus correspondientes parámetros formales.

Si alguna de estas condiciones falla, entonces falla la instrucción de invocación (CALL, SEND) y consecuentemente, la comunicación.

Por último, dos observaciones que requieren atención en relación con las familias de procesos y los rangos son las siguientes:

1.- En la declaración de una familia de procesos, las operaciones que se definen e implementan en ella están relacionadas al índice o instancia del proceso.

2.- En la declaración de un mensaje, es válido tener como tipo de dato un rango de valores.

En relación al primer punto, la operación definida e implementada por un miembro de una familia de procesos es diferente al de otro miembro, aunque su estructura sea la misma.

Sintácticamente, en la designación del nombre de la operación (en la declaración de la instrucción de entrada de la familia de procesos) no se representa ningún rango ni índice. No existe manera sintáctica de diferenciar en esta designación a una operación de una familia de operaciones.

Sin embargo, en la declaración de la operación (en las declaraciones del Recurso), si se representa a la familia, con la siguiente sintaxis:

<declaración_familia_operaciones> ::=
OP <nombre_operación>
[<índice-rango>](<declaración_mensaje>)
<índice_rango> ::= <constante> | <número>
| <constante>:<lím_inf>..<lím_sup>)

En relación al segundo punto, es válido tener en la declaración de tipo, de algún dato de la declaración del mensaje, a un rango.

4.4- Semántica de la comunicación.

Las instrucciones CALL e INPUT constituyen el mecanismo de llamadas a procedimientos, el cual se ejecuta para tener una interacción entre procesos del tipo cliente/servidor.

Sea el proceso P el que ejecuta la instrucción CALL, y Q es el que define al INPUT. En la instrucción INPUT se define la operación OP que invoca la instrucción CALL de P.

Para el establecimiento de la comunicación pueden suceder cualquiera de los siguientes eventos:

1.- La operación OP en la instrucción INPUT de Q, está en espera de ser invocada.
2.- El proceso P ejecuta el CALL e invoca a la operación OP, pero esta ultima aún no está lista para atender el CALL (por estar atendiendo a otro proceso, por no cumplir con los requisitos de sincronización, etc.).

En ambos casos, P o Q deben de esperar a que el otro concrete el establecimiento de la comunicación.

Los pasos requeridos para la ejecución de la instrucción INPUT son:

1.- Debe existir al menos una invocación o llamada (CALL o SEND) de cuando menos una operación (OP_1,..,OP_n).
2.- Si existe más de una operación invocada, se selecciona una de ellas en forma no determinística.
3.- Se prueba la expresión de sincronización, y si es

verdadera, entonces se pasa al punto 4 (Si no existe se considera verdadera).

4.- Se prueba la expresión de calendarización (si existe) y se selecciona alguna de las invocaciones (CALL, SEND). Si no existe se selecciona la primera invocación emitida.

Nota: Los puntos 3 y 4 se analizarán con más detalle en las secciones subsecuentes.

5.- Si se cumplieron los dos puntos anteriores, se procesa la lista de instrucciones de la operación seleccionada con los parámetros de la invocación elegida.

6.- Continúa la ejecución del proceso con la instrucción subsecuente al INPUT.

7.- Si no existen invocaciones o llamadas a ninguna operación en el INPUT, o si de existir algunas, ninguna hace verdadera a la expresión de sincronización, entonces se pasa al punto 8.

8.- Si el INPUT posee un ELSE, entonces se procesa su lista de instrucciones correspondiente, después de la cual finaliza la ejecución del INPUT, de otro modo se pasa al punto 9.

9.- Si no existe el ELSE en la instrucción INPUT y se cumple el punto 7, entonces se suspende (espera) el proceso hasta que exista al menos una invocación; o bien, se prueba periódicamente a las expresiones de sincronización hasta que al menos una sea verdadera, para procesar su correspondiente lista de instrucciones.

En función de la semántica de la instrucción INPUT, tanto el proceso P que emite el CALL como el Q que alberga al INPUT, pueden estar ambos suspendidos en un momento determinado.

El proceso P se mantiene suspendido hasta que se ha terminado de procesar la lista de instrucciones correspondiente a la operación invocada por él a través del CALL, y si existen resultados, entonces se le envían a P.

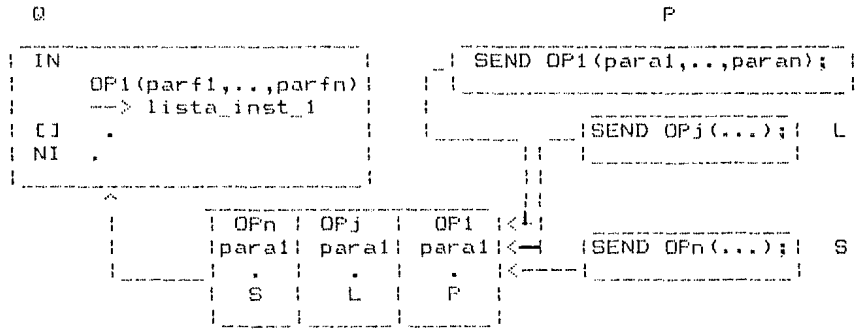
Posteriormente, el proceso P ejecuta la instrucción subsecuente al CALL y Q la subsecuente al INPUT.

Las instrucciones SEND e INPUT constituyen un mecanismo de envío de mensajes con almacenamiento, sirve para implementar una interacción entre procesos del tipo interconexión.

En forma similar al caso anterior, si el proceso P es el que ejecuta la instrucción SEND, Q el que procesa el INPUT, en el que se define la operación 'OP' que invoca a la instrucción SEND de P, entonces, para el establecimiento de la comunicación, a diferencia del CALL, no es necesario que P espere a Q.

Tampoco existen resultados de retorno al proceso P, y si el programador equivocadamente los espera, éstos se pierden.

Inicialmente Q puede estar en espera de ser solicitado, P envía su mensaje y éste se deposita en un almacén temporal como lo muestra la siguiente figura:



En esta figura el proceso S envió primeramente su mensaje, posteriormente L y P. Todos los mensajes están en cola de espera para ser atendidos; Q atiende uno a la vez.

Aún si la cola o almacén está vacio, el mensaje de P es introducido a la cola.

Con lo anterior, todo mensaje será finalmente atendido y los procesos transmisores (S, L, P) pueden continuar su ejecución, sin esperar a que su mensaje sea procesado.

Sin embargo, la cola o almacén no es de tamaño infinito y posee un límite, que si es rebasado ocasiona que los procesos que sigan enviando mensajes sean suspendidos (como en un CALL), hasta que exista suficiente espacio en la cola para almacenar sus mensajes.

4.4.1- Semántica del mensaje.

Tanto la sintaxis como la semántica del mensaje es la misma tanto en una interacción cliente/servidor como en una de tipo interconexión.

En los procesos cliente y transmisor, el mensaje es un conjunto de parámetros reales, como se muestra a continuación:

```
CALL
     op_i  (para1, para2,..., paran)
SEND
para1   -->  parámetro actual '1'
para2   -->  parámetro actual '2'
  .                  .
  .                  .
```

En los procesos servidor y receptor, el mensaje es un conjunto de parámetros formales:

```
OP_i (parf1, parf2,..., parfn)
parf1  --->   parámetro formal '1'
parf2  --->   parámetro formal '2'
```

Una restricción de tipo VAL en el mensaje indica que el parámetro es exclusivamente de entrada, es decir, es un dato que el proceso cliente o transmisor envía al servidor o receptor, según el tipo de interacción.

La semántica de la transferencia de parámetros es por valor (se crea una copia de los parámetros actuales de los procesos cliente o transmisor, y se asigna a los parámetros formales de los procesos servidor o receptor).

Lo anterior es una consecuencia lógica, en la semántica, de una transferencia en una interacción con almacenamiento de mensajes, ya que no existe forma de que el proceso transmisor espere la asignación de sus parámetros reales a los formales del receptor.

Lo que se almacena en el buffer es una copia de los parámetros reales de los procesos transmisores.

Lo que es importante recalcar es que las transferencias de mensajes en interacciones con almacenamiento poseen las cuatro siguientes propiedades:

1.- Los parámetros serán recibidos por los procesos receptores.
2.- Exactamente una copia de los parámetros actuales del proceso transmisor es entregada.
3.- La transferencia es exitosa, y los valores de los parámetros se entregan sin error.
4.- El orden en que se envían los parámetros es el mismo en el que se reciben.

Una restricción de tipo RES en el mensaje indica que el parámetro es exclusivamente de salida, por lo que realmente es un resultado que el proceso servidor envía al cliente.

Esta restricción se aplica exclusivamente a interacciones del tipo cliente/servidor, ya que en una mecánica del tipo interconexión no existe transferencia de mensaje del receptor al transmisor.

Esta semántica es de la categoría de resultado (se crea una copia de los parámetros formales del proceso servidor al ir finalizando la comunicación, cuyos valores son los resultados que requiere el cliente, y se asignan a los parámetros reales del proceso cliente).

Si en el mensaje existe un tipo de restricción VAR, entonces el parámetro es tanto de entrada como de salida.

Por el hecho de ser parámetro de salida, su aplicación es exclusivamente para interacciones del tipo cliente/servidor.

La semántica de la transferencia en estos parámetros es por valor y resultado, y se puede representar en los siguientes puntos:

1.- En primer lugar se crea una copia de los parámetros reales del proceso cliente.
2.- Se asigna la copia a los parámetros formales del proceso servidor.
3.- Se realiza la interacción, a través de una operación en una instrucción INPUT.
4.- Los resultados generados se ubican en los mismos parámetros formales referenciados en el punto 2.
5.- Se crea una copia de los parámetros formales, que se asigna de nueva cuenta a sus correspondientes parámetros reales del proceso cliente.

4.5- Causas bajo las cuales falla la comunicación.

Considérese que un proceso P en el Recurso A invoca una operación OP en el proceso Q del Recurso B. Los puntos que pueden originar una falla en la comunicación, y consecuentemente que ésta no se establezca se enumeran a continuación.

1.- Si la operación invocada por P no existe en el programa, entonces falla la comunicación.
2.- Debe existir compatibilidad en la sintaxis de los mensajes:
    2.1- Mismo número de parámetros reales y formales.
    2.2- Mismos tipos de datos entre los correspondientes parámetros.
3.- Si P y Q no pertenecen al mismo Recurso, para evitar que falle la comunicación se requiere:
    3.1- Que la operación OP sea exportable por el Recurso B (export op;).
    3.2- Que la operación OP sea importable por el Recurso A (import OP;).
4.- Independientemente de que P y Q integren el mismo Recurso, o de que se haya cumplido el punto 3, se debe satisfacer lo siguiente:
    En la declaración de exportación del Recurso, se indica a todas aquellas operaciones exportables, y opcionalmente se especifica como restricción el tipo de invocación que se puede realizar (CALL, SEND) en cada una de ellas.
    Si se invoca a la operación con una instrucción diferente a la que indica su restricción, no se establece la comunicación.
5.- Si se han satisfecho estos puntos, entonces la comunicación es exitosa.

Existen otras situaciones que pueden hacer fallar la comunicación:

1.- Falla en comandos custodiados (se analizará en secciones subsecuentes).
2.- Falla en el hardware (falla de una unidad de procesamiento central, o de un canal de comunicación,

etc.).

4.6- Conclusiones.

Las características de comunicación que hacen del lenguaje 'SR' ser superior a muchos otros lenguajes concurrentes se pueden resumir en los siguientes puntos:

1.- Capacidad de manejar tres mecanismos diferentes de comunicación.
2.- Manejo de control de acceso de operaciones.
3.- Adaptabilidad a diferentes arquitecturas.

Los tres mecanismos a los que se refiere el punto '1' son los que se implementan en los esquemas:

1.- Variables compartidas.
2.- Cliente/servidor.
3.- Interconexión.

La mayoría de los lenguajes concurrentes utilizan uno o dos mecanismos de comunicación, mientras que en SR, el programador en función de sus necesidades tiene la flexibilidad de elegir de entre tres mecanismos, de manera que en SR las deficiencias de un mecanismo se equilibran con los beneficios de los otros, y vice-versa.

En relación al segundo punto, el programador tiene la flexibilidad de controlar el acceso de sus datos y sus operaciones.

El uso de exportaciones e importaciones le permite muy fácilmente lograr su objetivo de controlar el acceso a sus objetos desde otros procesos externos al Recurso.

Finalmente, su adaptabilidad a arquitecturas tanto distribuidas como centralizadas le permite no estar atado exclusivamente a ciertas arquitecturas.

SR
5.-Sincronización

En esta sección se analizarán los siguientes puntos:

1.- Características de sincronización de los mecanismos de comunicación.
2.- Características de sincronización de la instrucción 'INPUT'.
3.- Otros mecanismos de sincronización (guardias), y mecanismos auxiliares.

Previamente al análisis de los puntos anteriores, se detallará la sintaxis de estos mecanismos de sincronización.

5.2- Sintaxis de mecanismos de sincronización.

A continuación se representa la sintaxis de la expresión de sincronización de la instrucción INPUT, así como el de un comando de operación:

```
<comando_operación> ::=
          <guardia_operación> -> <lista_instrucciones>
<guardia_operación> ::= <designación_operación>
          [<expresiones_sincronización_scheduling>]
<expresiones_sincronización_scheduling> ::=
   [<expresión_sincronización>][<expresión_scheduling>]
<expresión_sincronización> ::=
                          AND <expresión_booleana>
```

A la secuencia <designación_operación><expresiones _sincroniza-ción_scheduling> se le llama guardia de operación.

Representado de otra forma, y sin considerar la expresión de calendarización (scheduling) que se analizará en la siguiente sección, un guardia de operación sería así:

```
operación (par_1,...,par_i) AND expresión_booleana
```

ó simplemente

```
operación (par_1,...,par_i)
```

Los otros mecanismos que utiliza SR para efectuar sincronización son los guardias (booleanos). Un guardia es simplemente una expresión booleana.

Un comando custodiado es un mecanismo auxiliar de sincronización, y consiste en un conjunto de instrucciones cuya ejecución está en función de un guardia. Su sintaxis es:

```
<comando_custodiado> ::= <guardia> -> <cuerpo>
<guardia> ::= <expresión_booleana>
<cuerpo> ::= [<lista_declaraciones>] <lista_instrucciones>
<lista_declaraciones> ::= <declaración_variable_local>
                    {;<declaración_variable_local>}
```

El símbolo <declaración_variable_local> representa la declaración de variables cuyo radio de acción (alcance), abarca exclusivamente a la lista de instrucciones que precede (<lista_instrucciones>).

Una característica importante de SR es que la expresión de sincronización de un guardia de operación sí puede referenciar a los parámetros formales de la designación de la operación.

5.3- Semántica de los mecanismos de sincronización.

A continuación se abordarán las características síncronas de los tres mecanismos de comunicación que posee el lenguaje SR.

Respecto al esquema de variables compartidas, el uso de las variables permanentes de un Recurso está restringido únicamente a

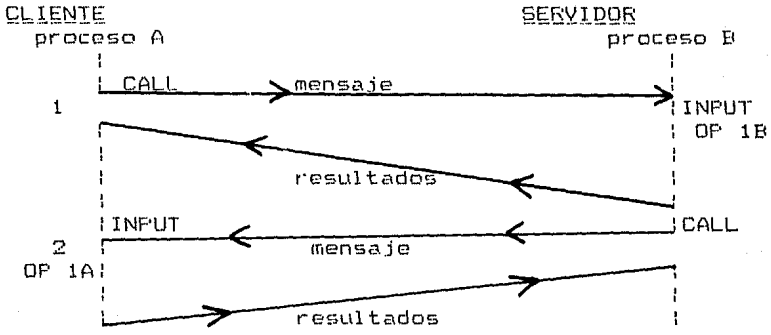
los procesos que lo constituyen. El acceso a esas variables es indivisible, término que también se conoce como atómico.

Lo anterior indica que nada puede interrumpir el acceso a una variable permanente que está referenciando un proceso, lo cual representa una gran ayuda al programador, pues sabe que en un momento determinado sólo un proceso está accesando a una variable permanente, y ésta quedará libre hasta que el proceso la deje de accesar (leer, escribir).

De no existir esta propiedad implementada por SR, el programador tendría que usar candados, semáforos o regiones críticas para garantizar el acceso exclusivo a cada variable, por lo que SR contribuye a establecer la mecánica de sincronización para el acceso concurrente a las variables compartidas de una manera automática.

En lo que concierne al mecanismo de llamadas a procedimientos del esquema cliente/servidor, el servidor no está limitado a proporcionar un único servicio al cliente, sino que puede realizar un conjunto de servicios diferentes.

Si además, el servidor invoca operaciones definidas en el proceso cliente, se establece un esquema de corrutinas como se muestra en la siguiente figura:



En el punto 1 el proceso A invoca la operación OP_1B de B, en el punto 2 el proceso B invoca a OP_1A de A.

Por la propiedad de sincronía de este mecanismo de comunicación (CALL-INPUT, llamada a procedimiento), ambos procesos deben sincronizarse para transferir el mensaje, lo cual trae como consecuencia que uno de los dos procesos se suspenda en espera del otro.

Pero lo más interesante es que una vez realizada la comunicación, cada proceso conoce el estado de avance del otro, y al mismo tiempo se reduce la interferencia (refiérase a la sección de

sincronización del capítulo correspondiente a CSP) entre ambos procesos.

La información que cada proceso desea tener del otro, respecto a su estado de avance, lo obtiene en cada punto de comunicación.

Consecuentemente, el programador puede hacer uso de esta información para dirigir el flujo de control de cada proceso.

Finalmente, en lo referente al esquema de interconexión, y debido a que existe almacenamiento de mensajes entre los procesos transmisores y receptores, se presentan las siguientes observaciones:

1.- El proceso transmisor sabe que sus mensajes serán recibidos por el receptor, pero no sabe en qué momento.
2.- El proceso receptor, al atender un mensaje, no tiene idea de quién lo envió ni en que instante de tiempo lo hizo.

Estas características no permiten que exista sincronización entre procesos transmisores y receptores, lo único que se puede concluir es que el transmisor envía su mensaje previamente a que el receptor lo reciba y atienda. Por lo tanto, este mecanismo de comunicación (SEND-INPUT) es muy pobre en lo que respecta a sincronización.

5.3.1- Características de sincronización de la instrucción INPUT.

SR tiene como característica fundamental el que los procesos y las instrucciones de entrada garantizan la exclusión mutua en la ejecución de las operaciones.

De hecho, los procesos son el único mecanismo necesario para obtener exclusión mutua.
Consecuentemente, la operación definida en una instrucción INPUT, en el momento de ser invocada, únicamente se asigna a un proceso y sólo hasta que ha terminado de ejecutar su lista de instrucciones podrá ser asignada a otro proceso (si se ubica en un ciclo).

En lo que respecta a sus guardias de operación, éstos pueden ser falsos o verdaderos; un guardia de operación de una instrucción INPUT es verdadero si:

1.- Al menos existe una invocación de la operación que designa.
2.- De existir una expresión de sincronización, ésta es verdadera.

En relación al primer punto, toda operación especificada en un guardia puede estar invocada en un momento determinado por más de un proceso.

Si la operación no ha sido invocada, entonces el punto 1 no se

cumple.

Si no existe la expresión de sincronización y el punto 1 se cumple, el guardia de operación es verdadero.

Si la expresión booleana de sincronización existe y es falsa, entonces aunque se cumpla el punto 1, el guardia de operación es falso.

De existir la expresión de sincronización, debe ser verdadera para que cumpliendo con el punto 1 el guardia sea verdadero.

Si el guardia de operación es verdadero, y además ha sido seleccionado para ejecución, entonces se procesa la lista de instrucciones que definen la operación.

Las expresiones de sincronización permiten al programador sincronizar la ejecución de sus operaciones en función de condiciones.

5.3.2- Características de otros mecanismos de sincronización.

Otro de los mecanismos de sincronización que se utilizan son los comandos custodiados.

Si en un comando custodiado el guardia es verdadero, entonces se ejecuta la lista de instrucciones correspondiente, por lo que el efecto de un comando custodiado es el de la ejecución de una lista de instrucciones en función de un grupo de condiciones representados mediante una expresión booleana.

Además de estos mecanismos, existen funciones en 'SR' que se utilizan para auxiliar en la mecánica de sincronización, y se enumeran a continuación:

1.- Función que obtiene número de invocaciones de una operación.
2.- Función que obtiene el índice del proceso en una familia de procesos.
3.- Función que obtiene el índice del Recurso en una familia de Recursos.

La primera función tiene como argumento el nombre de una operación, y genera el número de invocaciones (llamadas) que tiene esa operación, su sintaxis es:

Función_inv ::= ? <nombre_operación>

Por ejemplo, el siguiente fragmento de programa da prioridad a la operación OP1 sobre la OP2 en el sentido de que la operación OP2 es ejecutada únicamente si OP1 no ha sido invocada (en el momento que la expresión de sincronización es evaluada):

```
IN OP1( )  ->  lista_intrucciones_1
[] OP2( ) AND ? OP1 = 0  ->  lista_instrucciones_2
NI
```

La segunda función no posee argumentos y proporciona el índice del proceso que la llamó. Cuando se trata de una familia de procesos, su sintaxis es:

Función_ind_proceso ::= myprocess (mi proceso)

La tercera función es idéntica a la segunda, pero referida a Recursos y familias de Recursos, su sintaxis es:

Función_ind_Recurso ::= myresource (mi Recurso)

5.3.3- Conclusiones.

Las características que mayor ventaja dan a SR respecto a sincronismo, y que proporcionan un gran auxilio al programador son:

1.- La característica de atomicidad en el acceso a las variables permanentes.
2.- La propiedad de sincronía de su mecanismo de llamada a procedimientos.
3.- La ventaja de referenciar a los parámetros formales de la designación de la operación en la expresión de sincronización.
4.- La exclusión mutua en la invocación de los procesos a las operaciones.
5.- El apoyo de guardias y funciones auxiliares.

SR
6.- INDETERMINISMO

Las instrucciones del lenguaje SR que proporcionan características de indeterminismo son las siguientes:

1.- Instrucción INPUT.
2.- Instrucción alternativa.
3.- Instrucción iterativa o repetitiva.

Las instrucciones alternativa e iterativa están integradas a base de comandos custodiados.

6.2- Sintaxis.

La sintaxis de la instrucción INPUT se expuso en secciones anteriores.

La sintaxis de la instrucción alternativa es:

```
<instruc_alternativa> ::= IF <lista_comandos> FI
<lista__comandos> ::= <comando>{[]<comando>}
<comando> ::= <comando_custodiado>
    |  <comando_custodiado> [] ELSE -> <lista_instrucciones>
<comando_custodiado> ::= <guardia> -> <lista_instrucciones>
<guardia> ::= <expresión_booleana>
```

La sintaxis de la instrucción iterativa (repetitiva) es:

<instruc_iterativa> ::= DO <lista_comandos> OD

La sintaxis de <lista_comandos> es la mostrada en la sintaxis de la instrucción alternativa.

6.3- Semántica.

A continuación se describe la semántica de la instrucción INPUT en relación a su propiedad de indeterminismo.

Si una instrucción INPUT define a más de una operación, y más de un guardia de operación es verdadero, entonces se selecciona a uno de ellos indeterminísticamente (al azar) y se procesa su correspondiente lista de instrucciones, después de lo cual finaliza la ejecución de la instrucción INPUT.

Este indeterminismo generado en la instrucción INPUT permite ofrecer la misma oportunidad de atención a los procesos que invocan a las diferentes operaciones (no confundir con procesos que invocan la misma operación) que define, sin que existan prioridades o privilegios de atención de unos con respecto a otros.

Sin embargo, en la implantación física de SR se intenta simular el indeterminismo con una mecánica determinística.

La semántica de una instrucción alternativa en SR es similar a la de la instrucción alternativa de CSP, excepto por los dos siguientes puntos:

1.- La diferencia radica en los guardias, pues en CSP un guardia puede incluir a una instrucción de entrada, mientras que en SR un guardia es simplemente una expresión booleana y no se permite la inclusión de ninguna designación de operación, la cual es el equivalente a la instrucción de entrada de CSP.

2.- La instrucción alternativa de SR permite que si no es verdadero ninguno de los guardias que lo constituyen, entonces se pueda procesar a través de la opción ELSE a una lista de instrucciones, y en el caso de que no se incluya, entonces la instrucción alternativa falla cuando todos sus guardias son falsos.

Finalmente, una instrucción iterativa es la repetición de una alternativa. La instrucción alternativa se repite en un ciclo (loop) hasta que la instrucción alternativa falla.

Si una instrucción iterativa posee un ELSE, entonces ésta nunca finalizará.

Después de que se presenta la falla (de existir), finaliza la ejecución de la instrucción iterativa.

6.4- Conclusiones.

Hay que hacer la observación de que una instrucción INPUT sí puede retrasar (suspender) al proceso del cual forma parte, sin embargo, ninguno de los otros dos que introducen indeterminismo puede suspender o bloquear al proceso que integran. La referencia es con respecto a las instrucciones alternativa e iterativa. Por otra parte, el único que mantiene interacción con procesos es la instrucción INPUT.

Tanto las instrucciones alternativa como iterativa se asemejan a un conjunto de regiones críticas condicionales.

Por último, puede ser necesario tener una instrucción iterativa siempre en ejecución (sin finalización). Este punto es aplicable a sistemas operativos, donde existen procesos que nunca finalizan su ejecución y su estado oscila entre listo (uso CPU) o bloqueado (suspendido).

SR
7.-SCHEDULING

La traducción de este término es 'catalogar, inventariar, calendarizar, incluir en lista, fijar tiempos, etc.', y no refleja el significado que se le da en ciencias de la computación, por tal motivo se usa a veces el término en inglés, o bien, la palabra calendarizar.

Existen diferentes políticas de asignación, como por ejemplo:

1.- Asignar el recurso al primero que lo solicitó.
2.- La asignación según un número conocido como prioridad.
3.- Asignación siguiendo una cola circular.

Estas son unas cuantas del total que existen.
Al proceso o entidad que registra a los procesos que solicitan uso del recurso, que escoge al siguiente a asignársele el recurso y se lo asigna, se le conoce como despachador (scheduler).

Existen lenguajes en los que para programar un proceso 'scheduler' hay que implementar:

1.- La estructura en donde se registran los procesos competidores.
2.- El algoritmo de selección según la política de asignación elegida.
3.- Los algoritmos de actualización de las estructuras.
4.- Los algoritmos de asignación.

En los lenguajes concurrentes se hace uso de guardias y comandos custodiados para la implantación de los algoritmos, sin embargo, se tienen que implementar las estructuras y su acceso.

En el lenguaje SR se quita al programador todo este trabajo, y es el propio lenguaje el que lo implementa, lo cual representa una gran ventaja y facilidad de manipulación.

SR utiliza al guardia de operación para generar el manejo de scheduling, como se mostrará en las siguientes subsecciones.

7.2- Sintaxis.

La expresión de calendarización (scheduling) del símbolo <expresiones_sincronización_scheduling> del guardia de operación de una instrucción INPUT tiene la siguiente sintaxis:

<expresiones_sincronización_scheduling> ::=
[<expresión_sincronización>][<expresión_scheduling>]
<expresión_scheduling> ::= by <expresión_aritmética>

La expresión aritmética es entera, y puede hacer referencia a los parámetros formales de la designación de la operación.

7.3- Semántica.

Como puede observar de la sintaxis, la expresión de scheduling está asociada con una operación.

Por lo tanto, su función es elegir de entre todas las invocaciones de procesos de la misma operación y que satisfagan su expresión de sincronización (verdadera), a aquella que minimice la expresión aritmética.

Ninguna invocación que no satisfaga su expresión de sincronización será considerada en el cálculo de la de calendarización (scheduling), por lo que no será elegible de asignársele la ejecución de la operación. El siguiente ejemplo permitirá entender el efecto de la expresión de calendarización (scheduling).

```
IN SOLICITA(CANTIDAD:INTEGER) BY CANTIDAD -> SKIP
NI
```

Observe que no existe expresión de sincronización, por lo que se considera como verdadera.

Todos los procesos que han invocado a SOLICITA son candidatos a asignársele. Aquella invocación con el menor valor numérico de su parámetro CANTIDAD, es a la que se le asigna la operación. La expresión aritmética es simplemente el valor numérico del parámetro formal CANTIDAD.

Si se considera en este ejemplo a CANTIDAD como el tiempo estimado de ejecución del proceso que invoca a la operación SOLICITA, entonces aquellos con menor tiempo de trabajo ganarán el acceso al recurso, por lo que esta simple instrucción genera un despachador del tipo: el de menor tiempo es el siguiente (shortest job next scheduler).

Todas las invocaciones a una operación se registran en la cola de operación de la misma, y en el momento que es revisada se detectan a aquellas invocaciones que satisfacen su expresión de sincronización.

Posteriormente, se aplica a los candidatos la expresión aritmética y aquella cuyo valor resultante sea menor es la que gana el acceso a la operación.

7.4- Conclusiones.

Esta característica de SR permite que se instrumenten con mucha facilidad y con muy poco código distintos tipos de despachadores (schedulers), según las políticas de asignación, las cuales están representadas mediante la expresión aritmética.

Por último, si no existe expresión de calendarización, entonces el primer proceso en invocar la operación es el primero en otorgársele. Se maneja una cola tipo 'FIFO': primero en invocar es primero en ser servido.

A continuación se presentan algunos ejemplos que ilustran lo analizado en este capítulo.

EJEMPLOS

shortest- job- next scheduler

En primer lugar se describirá un despachador del tipo shortest-job-next scheduler, como el que se ejemplificó en la sección correspondiente de CSP. Ejemplo basado en [Andrews, 1981].

```
resource SJN;

define solicita (call);

process calendariza;

do true -->
     IN solicita (tiempo: integer) by tiempo -->
          -- accesa el recurso compartido, instrucciones de
          -- acceso
     NI
od
end calendariza
end SJN
```

Llamada del proceso usuario: CALL solicita (tiempo)

Los cinco filósofos

Ejemplo basado en [Andrews, 1981].
La descripción es exactamente igual a la del ejemplo de los cinco

filósofos del capítulo correspondiente a CSP, razón por la que ya no se repetirá, refiérase a la página 31.

```
resource cinco_filósofos
     op entra_comedor (CALL);
     op sale_comedor (CALL);
     op levanta_tenedor (var i:1..5) (CALL);
     op deja_tenedor (var i:1..5) (CALL);

     var filósofo_come[1..5];
--   las operaciones que se manejan en el recurso se definen
--   en los procesos tenedores y comedor, los filósofos
--   están representados también mediante una familia de
--   procesos llamada filósofo
--        Cuando un filósofo desea comer debe solicitar
--   acceso al comedor, posteriormente se sienta a la mesa y
--   levanta los dos tenedores anexos a su silla
--   Las llamadas de solicitud y aviso a los otros procesos
--   se realizan mediante las operaciones declaradas en el
--   recurso

-- Familia de procesos filósofos

Process filósofo[1..5];  -- familia de 5 procesos

     var límite, número_veces_come: integer;
     límite := 10000000;  -- edad del filósofo

     -- estas variables indican la edad del filósofo a
--   través del número de veces que come, la edad límite
--   está dada por la variable límite

     DO número_veces_come <= límite -->
          -- mientras viva y coma

          -- COME:
          CALL entra_comedor;
               -- solicita acceso al comedor
               -- y se le otorga, por lo que puede
               -- continuar:
          CALL levanta_tenedor(myprocess);
               -- levanta el tenedor que le corresponde, si
--        es que no está siendo usado por su vecino,
--        recuérdese que myprocess proporciona el
--        índice del proceso en su familia
          CALL levanta_tenedor(myprocess mod 5 + 1);
--        intenta levantar el otro tenedor, y si lo
--        logra entonces come, para lo cual se indica
--        mediante el vector filósofo_come

          -- come:
               filósofo_come[myprocess] := true
          -- deja de comer

          CALL deja_tenedor(myprocess mod 5 + 1);
```

```
            -- deja en primer lugar al último tenedor que
                -- tomó, y en segundo lugar al primero:
            CALL deja_tenedor(myprocess);

            -- y abandona el comedor
            CALL sale_comedor;
            número_veces_come := número_veces_come + 1
            filósofo_come[myprocess] := false;
            --  se contabiliza su edad y además se pone en
            --  falso a filósofo_come para indicar que el
            --  filósofo dejó de comer, después de lo cual se
            --  pone a pensar

            -- PIENSA:
            L := 1
            DO L <= 10000 --> L := L - 1
            OD
            -- se simula la acción de pensar
        OD
    END filósofo;

-- Descripción de la familia de procesos tenedores

    Process tenedores[1..5];

    var tenedor[1..5]:integer;
    tenedor[myprocess] := 0;

    -- un tenedor cuando está desocupado hace que el vector
    -- tenedor en su índice valga 0 y 1 cuando se ocupa

    DO true -->

        IN
            levanta_tenedor(i:1..5) AND tenedor[myprocess] = 0
             AND NOT filósof_come(i)
             AND NOT filósofo_come(i mod 5 + 1) -->
                 tenedor(myprocess) = 1
             --  en primer lugar verifica que exista algún
                 filósofo que desea tomar el tenedor,
                 posteriormente prueba si el tenedor no está
                 ocupado, así como también checa que los
                 filósofos adyacentes no estén comiendo,
                 posterior a lo cual declara al tenedor como
                 ocupado
        []
             deja_tenedor(i:1..5) --> tenedor(myprocess) = 0
        NI
    OD
    end tenedores;

    -- descipción de COMEDOR

    Process comedor;
```

```
var j:integer; j := 0;

--   este proceso es el que otorga el acceso al comedor, con
     objeto  de no permitir que ingresen más de 4  filósofos
     y evitar un abrazo mortal
--   j  es la variable que indica el número de filósofos  en
-- el comedor en un momento dado

DO true -->
     IN  -- ingreso al comedor
          entra_comedor() and j < 4 --> j := j + 1
     []
          -- avisa que abandona el comedor
          sale_comedor() --> j := j - 1
          -- se contabiliza
     NI
OD
end comedor;

end cinco_filósofos; -- fin recurso
```

ADA
INTRODUCCION

ADA es un lenguaje cuyo nombre se asignó en honor de Augusta Ada Lovelace, hija del poeta inglés Lord Byron, quien fue pionera de la computación en el siglo diecinueve.

ADA fue diseñado por un equipo de trabajo dirigido por Jean Ichbiah [Ichbiah, 1980]; desarrollado por iniciativa del Departamento de Defensa de los Estados Unidos, con el objetivo de desarrollar un lenguaje de programación confeccionado a las necesidades del Ejército, la Armada y la Fuerza Aérea.

ADA es un lenguaje de programación cuyo diseño se enfocó básicamente a programación en tiempo real, así como al control de dispositivos físicos. No obstante lo anterior, se puede aplicar a análisis numérico y programación de sistemas.

El mecanismo de comunicación que utiliza es sincrono, y los dos lenguajes (modelos) concurrentes que lo antecedieron, y en cuya filosofía se fundamentó su diseño, fueron CSP y DP [Brinch Hansen, 1978].

A continuación se analizan exclusivamente las características de concurrencia relacionadas con el esquema de análisis propuesto.

ADA
2.- PROCESOS

En el lenguaje 'ADA' se conoce a los procesos con el nombre de tareas.

Un programa en 'ADA' está compuesto por elementos llamados 'unidades del programa', las cuales pueden ser de una de las cuatro siguientes formas:

1.- Subprogramas.
2.- Paquetes.
3.- Unidades genéricas.
4.- Tareas.

Una unidad puede estar constituida por otras, las que a su vez pueden contener otras más, quedando bien establecida la propiedad de anidamiento de unidades en 'ADA'.

Toda unidad está constituida por dos partes: en la primera se declaran y especifican objetos, tipos, funciones, procedimientos, tareas, etc., y la segunda corresponde al cuerpo propio de la unidad, incluyendo declaraciones locales.

Al constituirse como proceso, una tarea alberga a un conjunto de instrucciones y declaraciones, las cuales forman una única entidad con vida propia e independiente a las demás tareas, subprogramas o paquetes.

2.1 - Sintaxis de Tareas.

A continuación se presenta la sintaxis de las tareas:

```
<especificación_tarea> ::=
      task [type] <nombre_tarea> [is
      <lista_declaraciones_puntos_acceso>
end [<nombre_tarea>]]
<lista_declaraciones_puntos_acceso> ::=
      <declaración_punto_acceso>{;<declaración_punto_acceso>}
<declaración_punto_acceso> ::=
      entry <nombre_punto_acceso>[(<parámetros_formales>)]
```

La palabra reservada entry declara el punto de acceso, cuyo nombre está representado por el símbolo <nombre_punto_acceso>, y cuyo mensaje tiene el formato dado por <parámetros_formales>, y que puede no existir. La sintaxis del cuerpo de la tarea es:

```
<cuerpo_de_la_tarea> ::= task body <nombre_tarea> is <descrip>
<descrip> ::= [<parte_declarativa>] <cuerpo>
<cuerpo> ::= begin <lista_instrucciones> [<excepciones>]
             end [<nombre_tarea>];
<excepciones> ::= exception <manejador_excepción>
                  {;<manejador_excepción>}
```

El símbolo <nombre_tarea> es el mismo que el de la parte de especificación. La parte declarativa es local al cuerpo de la tarea, y puede incluirse cualquier tipo de objeto o unidad de programa, con excepción de los paquetes.

En la lista de instrucciones que componen al cuerpo se incluyen los accepts correspondientes a los puntos de acceso declarados en la parte de especificaciones. La sintaxis y semántica de esta instrucción se abordará en la sección de comunicación.

Por último, la palabra exception, así como el símbolo <manejador_excepción>, se describirán en la parte de semántica.

2.2 - Semántica de tareas.

Las tareas interaccionan entre sí a través de los denominados puntos de acceso (entries), que son puntos o instrucciones dentro del cuerpo de una tarea donde ésta puede ser llamada por cualquier otra, estableciéndose de esta forma una comunicación entre ellas.

En ninguna otra instrucción del cuerpo de una tarea que no sea un

punto o puerta de acceso puede haber interacción con otras tareas.

La comunicación establecida con este modo de interacción entre las tareas es del tipo de envío de mensajes.

Una tarea puede tener varios puntos de acceso, cada uno de ellos identificado con un nombre único dentro del cuerpo de la tarea que lo diferencia de los demás.

El punto de acceso está identificado por la palabra reservada accept, seguida del nombre propio del punto. En realidad, el accept es una instrucción que podríamos calificar como de entrada o recepción de mensajes, la cual puede estar integrada por un conjunto de instrucciones.

Mientras se procesa el bloque de instrucciones que componen a la instrucción 'accept', la tarea transmisora que inició la comunicación se mantendrá suspendida.

El accept puede ir acompañado de un grupo de parámetros que pueden ser tanto de entrada como de salida (datos de lectura y resultados), y los cuales constituyen el mensaje. Cualquier tarea puede estar suspendida en uno o más de sus puntos de acceso, esperando a que otras tareas se comuniquen con ella.

Las características de comunicación entre las tareas se analizarán con profundidad en la sección correspondiente a comunicación.

Toda tarea está constituida por una parte de especificaciones y por el cuerpo propio de la misma. En la parte de especificaciones se declaran los puntos de acceso de la tarea, incluyendo la forma de los mensajes.

Debido a que toda tarea que desee interaccionar con otras debe conocer además de los nombres de las tareas, el de los puntos de acceso de las mismas, así como el formato de los mensajes (número y tipo de los parámetros) es imprescindible que tenga acceso a las secciones de especificación de las otras tareas. Por lo tanto, la parte visible de toda tarea es su parte de especificaciones <especificación_tarea>.

Sin embargo, la ejecución de la tarea está definida por su cuerpo, en el que se implementan las operaciones que otras tareas solicitan, y se evalúan los resultados o mensajes de retorno (si los hay) a las tareas transmisoras.

Por lo anterior, el cuerpo de una tarea siempre se mantiene oculto a las demás tareas y unidades de programas.

Dentro del cuerpo de una tarea se pueden declarar tipos, subtipos, variables, subprogramas e incluso tareas, que serán locales al mismo cuerpo. También, la tarea puede formar parte de la declaración de un paquete, un subprograma, otra tarea o una

unidad genérica.

El símbolo type representado en la sintaxis de la parte de especificación de una tarea, es indicativo de que en ADA existe el concepto de tipos de tareas.

El término tipo en tareas tiene el mismo significado que para datos, y se puede concebir al tipo tarea como un modelo o patrón del cual pueden copiarse o reproducirse tareas, a las copias se les llama instanciaciones.

Cada vez que se haga necesario utilizar nuevas tareas de un tipo determinado, simplemente se declaran. La declaración de una tarea es sintácticamente igual que la de cualquier otro dato o variable. Por ejemplo, considere el siguiente tipo de tarea:

```
task type almacén is
     entry deposita;
     entry extrae;
end almacén;
```

y la siguiente declaración:

```
bodega, guarda_ropa:almacén
```

Almacén es un tipo de tarea, mientras que bodega y guarda_ropa son dos tareas cuya especificación y cuerpo son idénticos a la del modelo (almacén), cada una de estas tareas se ejecuta concurrentemente con la otra.

En este ejemplo se consideraron únicamente dos tareas, sin embargo, es ilimitado el número de las que puedan declararse, e incluso se permite la declaración de arreglos (familias) de tareas.

Los arreglos constituidos por elementos de tipo tarea son declarados y accesados de igual forma que si los elementos constituyentes fueran entidades de otros tipos (caracter, entero, real,...). Por ejemplo, cada elemento del arreglo red_almacenamiento es una tarea:

```
red_almacenamiento : array(1..50) of almacén;
```

red_almacenamiento(i) se refiere a la i-ésima tarea del arreglo.

Cada elemento de un arreglo de tareas se procesa concurrentemente con los demás; sin embargo, hay situaciones en que no es muy eficiente manejar arreglos de tareas, ya que un arreglo predefine un número fijo de ellas.

También existe la creación dinámica de tareas, es decir, las tareas se crean a tiempo de ejecución según la propia lógica y requerimientos del programa que las controla.

La declaración de tareas de creación dinámica se hace a través del uso de tipos de acceso. Los objetos cuyo tipo es de acceso,

se utilizan para referenciar objetos creados dinámicamente (tareas, estructuras, etc.), debido a que a diferencia de los estáticos (cuya creación se realiza mediante su especificación en una declaración), no poseen un nombre explícito.

Los objetos de tipo acceso son apuntadores a los nuevos objetos creados, y la instrucción utilizada para crear tareas y objetos dinámicamente es NEW, que tiene como función asignar memoria, y en el caso de tareas, crear una copia con el modelo del tipo con que se declara la tarea, para depositarla en el área asignada, retornando en un objeto de tipo acceso (apuntador) la dirección de la tarea creada.

La sintaxis de las instrucciones con que se crea dinámicamente la tarea son:

```
<declaración_de_tipo_acceso> ::=
     type <tipo_de_acceso> is access <tipo_de_tarea>;
<declaración_de_objeto_de_tipo_acceso> ::=
     <objeto_de_tipo_acceso>:<tipo_de_acceso>;
<instrucción_de_creación_objeto> ::=
     <objeto_de_tipo_acceso> := new <tipo_de_tarea>;
```

Por ejemplo, desea crearse una tarea de tipo 'almacén':

```
1     type tipo_acceso_al is access almacén;
2          apunt_nueva_tarea : tipo_acceso_al;
3          apunt_nueva_tarea := new almacén;
```

En la primera línea se declara el tipo de acceso, en la segunda al objeto de tipo acceso, y en la tercera se crea la tarea.

2.3 - Excepciones.

En la descripción de la sintaxis del objeto tarea se identifica la palabra reservada exception, excepción en español.

Se da el nombre de excepción a un error o a una situación excepcional que pueda suceder durante la ejecución de una unidad de programa. Por ejemplo, la ejecución de una división por cero, la interacción de una tarea con otra que no existe, o que ya había terminado su procesamiento.

Los errores pueden incluso ser originados por fallas o condiciones adversas en el hardware.

La presencia de una excepción consiste en abandonar la ejecución normal de la tarea, de tal manera que pueda prestarse atención al error o situación que se ha presentado.

La forma de prestar atención a la excepción es mediante un conjunto de instrucciones asociadas a ella, el cual recibe el nombre de manejador de la excepción (exception handler).

En la ejecución de una tarea pueden presentarse varias

excepciones diferentes, y puede existir un manejador diferente para cada una, o bien, se puede compartir uno para manejar dos o más excepciones.

Todos los manejadores se agrupan en un bloque que se ubica al final del cuerpo de la tarea.

Cuando se presenta una excepción en el procesamiento de una tarea, la ejecución salta inmediatamente hacia el manejador correspondiente, se procesan las instrucciones que lo constituyen y termina de ejecutarse el bloque o tarea que lo contiene.

Bajo ninguna circunstancia se retorna el control de la ejecución al punto, en el cuerpo de la tarea, donde se presentó la excepción, ni en los puntos consecutivos.

Si se llega a presentar una excepción para la cual no está definido un manejador, entonces la tarea termina su ejecución inmediatamente.

2.4 - Activación, terminación y estados de tareas.

Se había indicado que el cuerpo de una tarea está constituido por un área de declaraciones locales y una lista de instrucciones.

A la elaboración (procesamiento) de la lista de declaraciones que constituyen el área de declaraciones locales se le llama activación de la tarea. A partir de ese momento, la tarea compite ya por uso de la unidad de procesamiento central (CPU), y ocupa un espacio en memoria.

Posteriormente se inicia la ejecución de la lista de instrucciones que componen el cuerpo de la tarea.

Las tareas declaradas en el área de declaraciones de una unidad de programa se activan previamente a la ejecución de la primera instrucción que sigue a la parte declarativa.

La activación de más de una tarea declarada se procesa en paralelo, y el orden en que cada una de ellas termina de ser activada es arbitrario. Por ejemplo, considere el siguiente procedimiento 'P':

```
          Procedure P is
               i: integer range 1..n := 1;
1               A,B : Recurso;
2               C   : Recurso;
3          begin
4               i   := i + 1;
                    .
          end;
```

En los puntos 1 y 2 se elaboran las tareas A, B y C, de tal manera que se inicia su activación y en el punto 4 las tres tareas ya están activas.

Si en el punto 3 algunas de las tareas no han sido aún activadas, no se puede ejecutar la instrucción de la línea 4.

Si la tarea es creada dinámicamente mediante la instrucción NEW, entonces la activación de la misma se realiza después de que la copia del tipo del cual está declarada la tarea se deposita en el espacio asignado de memoria, y se inicializa el objeto creado; y únicamente se retorna el valor del objeto de tipo acceso que apunta a la tarea creada cuando ésta ya ha sido activada.

Consecuentemente, la tarea ya está activa en el punto en que se procesa la instrucción inmediata y subsecuente al NEW.

Se dice que una tarea ha completado su ejecución cuando ha terminado de procesar la lista de instrucciones que constituyen su cuerpo.

Si en la ejecución de las instrucciones del cuerpo de una tarea se evoca una excepción, ya sea en forma explícita o implícita, y no está definido un manejador correspondiente, se abandona la ejecución de la lista de instrucciones, y finaliza (se completa) el procesamiento de la tarea.

Si está definido un manejador para la excepción, entonces se procesa, y posteriormente se puede completar o finalizar la ejecución de la tarea, según el algoritmo que la constituya.

Se dice que una tarea está terminada cuando finaliza su ejecución.

Existe una instrucción que hace terminar la ejecución de una tarea, denominada TERMINATE (termina), la cual es una alternativa de otra instrucción llamada SELECT, cuya sintaxis y semántica se analizará en la sección de indeterminismo.

En el estudio de la activación y terminación de tareas en esta sección se consideró cada tarea con independencia de las demás, es decir, se hizo el análisis del caso de tener únicamente una tarea. En la siguiente sección (paralelismo) se analiza la dependencia entre tareas, respecto a su activación y terminación.

En ADA toda tarea se puede encontrar únicamente en dos estados:

1.- En forma activa.
2.- Suspendida (en espera).

Toda tarea activa compite por uso del CPU, y se ejecuta la lista de instrucciones que componen su cuerpo, mientras que una tarea suspendida no compite por recurso del CPU, y se dice además que está bloqueada, en espera de ser llamada, o bien, si se trata de la tarea transmisora, puede aguardar el retorno de su mensaje.

En otros casos, existe una instrucción que suspende una tarea por un periódo determinado de tiempo, delay, y se puede utilizar para sincronismo entre tareas.

2.5- Conclusiones.

Se puede concluir del análisis realizado a programas y procesos en ADA, los siguientes puntos:

1.- Todas las tareas están divididas en una parte de especificaciones y un cuerpo de instrucciones. La parte de especificaciones es la interfase con el mundo exterior.
2.- Se maneja el concepto de tipo tarea así como el de creación dinámica de tareas, los cuales dan mucha flexibilidad a la programación, y permiten hacer un uso más racional de los recursos del sistema de cómputo.
3.- El manejo de excepciones en tareas da al programador mucha mayor flexibilidad en aplicaciones de tiempo real, y le permiten detectar errores de una manera más fácil.

ADA
3.-PARALELISMO

Como consecuencia del mecanismo de envío de mensajes que usan las tareas para interaccionar entre sí, existe una amplia variedad en las arquitecturas de equipos de cómputo en que puede implementarse el uso de tareas y unidades de programas de ADA.

El uso concurrente de tareas puede implantarse en las siguientes arquitecturas:

1.- Redes de computadoras.
2.- Arquitecturas de multiprocesamiento.
3.- Arquitecturas de un solo procesador con multiprogramación.

En el primer punto, se utiliza una red de computadoras en que los mensajes se transportan a través de los canales físicos de comunicación. De esta forma, pueden tenerse conjuntos de tareas en cada nodo de la red, o bien una tarea por nodo.

Puede haber interacción entre paquetes de distintos nodos, mediante la comunicación de las tareas que los componen.
El uso de la red, así como los protocolos de comunicación, son transparentes al programador.

En las arquitecturas de multiprocesamiento se tiene operando a más de una unidad de procesamiento central (CPU), compartiendo la misma memoria.
En estas dos primeras arquitecturas descritas existen tareas que se están procesando simultáneamente en varios procesadores físicos (CPU).

Sin embargo, en arquitecturas de un solo procesador no es posible que dos o más tareas estén ejecutándose físicamente en el mismo instante de tiempo.
En este último caso, se simula la simultaneidad con el concepto de multiprogramación.

En el lenguaje ADA el paralelismo es implícito, ya que no existe ninguna instrucción que específicamente mande procesar a un conjunto de procesos concurrentes, como el comando paralelo de CSP.

Por tal motivo, en esta sección no se presenta un análisis de la sintaxis, ya que ésta se presenta cuando existe paralelismo explícito.

En esta subsección se describirá la activación y terminación concurrente de tareas, sin embargo, es necesario introducir el concepto auxiliar de dependencia entre tareas, ya que en él se fundamenta el paralelismo.

3.1 - Dependencia entre tareas.

Cada tarea depende de al menos otra tarea, o bien de un subprograma, un paquete (de biblioteca), o un bloque de instrucciones que se esté ejecutando.

A las mencionadas entidades de las que depende una tarea se les llama maestros.

El maestro es el elemento responsable de la creación de la tarea que depende de él; una tarea es creada cuando es activada.

En función de lo expuesto en la sección de procesos, respecto a la creación o activación de tareas, se puede concluir que la creación de las tareas puede ser estática o dinámica.

La diferencia entre estas dos formas de creación radica en que a las tareas creadas estáticamente se les asigna espacio de memoria cuando se les declara, mientras que en las creadas dinámicamente el espacio se asigna en el momento de ejecución de la instrucción NEW.

Una tarea creada estáticamente depende del maestro en cuya parte declarativa está declarada; por ejemplo, considere la siguiente tarea:

```
task body T is
     -- declaraciones
     task T1 is
          --parte visible de T1
     end T1;
     task body T1 is
          -- cuerpo de T1
     end T1;
     task T2 is
```

Sin embargo, en arquitecturas de un solo procesador no es posible que dos o más tareas estén ejecutándose físicamente en el mismo instante de tiempo.
En este último caso, se simula la simultaneidad con el concepto de multiprogramación.

En el lenguaje ADA el paralelismo es implícito, ya que no existe ninguna instrucción que específicamente mande procesar a un conjunto de procesos concurrentes, como el comando paralelo de CSP.

Por tal motivo, en esta sección no se presenta un análisis de la sintaxis, ya que ésta se presenta cuando existe paralelismo explícito.

En esta subsección se describirá la activación y terminación concurrente de tareas, sin embargo, es necesario introducir el concepto auxiliar de dependencia entre tareas, ya que en él se fundamenta el paralelismo.

3.1 - Dependencia entre tareas.

Cada tarea depende de al menos otra tarea, o bien de un subprograma, un paquete (de biblioteca), o un bloque de instrucciones que se esté ejecutando.

A las mencionadas entidades de las que depende una tarea se les llama maestros.

El maestro es el elemento responsable de la creación de la tarea que depende de él; una tarea es creada cuando es activada.

En función de lo expuesto en la sección de procesos, respecto a la creación o activación de tareas, se puede concluir que la creación de las tareas puede ser estática o dinámica.

La diferencia entre estas dos formas de creación radica en que a las tareas creadas estáticamente se les asigna espacio de memoria cuando se les declara, mientras que en las creadas dinámicamente el espacio se asigna en el momento de ejecución de la instrucción NEW.

Una tarea creada estáticamente depende del maestro en cuya parte declarativa está declarada; por ejemplo, considere la siguiente tarea:

```
task body T is
     -- declaraciones
     task T1 is
          --parte visible de T1
     end T1;
     task body T1 is
          -- cuerpo de T1
     end T1;
     task T2 is
```

```
        --parte visible  de T2
end T2;
task body T2 is
        -- cuerpo de T2
end T2;
begin
        -- lista de instrucciones de T
end T;
```

La parte declarativa del cuerpo de la tarea T está constituida, entre otros elementos, por las declaraciones de las tareas T1 y T2, por lo que el maestro de T1 y T2 es la tarea T.

Una tarea creada dinámicamente depende del maestro que elabora la definición correspondiente de su tipo de acceso. Por ejemplo, considere el siguiente conjunto de bloques anidados de instrucciones:

```
bloque 1 { declare
             type GLOBAL is access RESOURCE;
           begin
             bloque 2 { declare
                          type LOCAL is access RESOURCE;
                          X:GLOBAL := new RESOURCE;
                          L:LOCAL  := new RESOURCE;
                        begin
                          -- lista de instrucciones
                        end;
                      -- lista de instrucciones
           end;
```

El tipo de acceso de la tarea designada por X es GLOBAL, por lo que depende del maestro bloque 1, ya que és en el bloque 1 donde se elabora la definición del tipo de acceso GLOBAL.

El maestro del cual depende directamente la tarea designada por L es el bloque 2, ya que en él se elabora la definición de LOCAL.

Sin embargo, la tarea apuntada por L también depende del bloque 1, aunque en forma indirecta, por lo que bloque 1 también es su maestro (no inmediato).

Como se pudo observar de este último ejemplo, una tarea puede depender de más de un maestro, y de un maestro pueden depender varias tareas.

3.2- Activación y terminación de tareas concurrentes.

A continuación se repasará brevemente la estructura de un programa principal, así como el uso de bibliotecas para posteriormente explicar la creación y activación de las tareas dependientes.

Un programa puede ser un paquete, un subprograma, o bien una

instanciación genérica. Cada uno de estos elementos puede albergar a cualesquiera otra unidad, o unidades de programas. Del mismo modo, cada programa puede utilizar unidades de programas ya compilados, que se localicen en una biblioteca.

Al grupo de unidades de programas de biblioteca que referencia y usa un programa principal, se le llama biblioteca del programa, y a las unidades se les denomina unidades de biblioteca.

Una tarea puede contener declaraciones de tareas, las que a su vez también pueden albergar las de otras tareas o subprogramas. No existe un límite en lo que respecta al anidamiento de unidades de programas; además, en cualquier nivel de anidamiento una unidad puede referenciar y usar unidades de biblioteca.

Cuando se ejecuta un programa principal, ya sea un paquete, subprograma o una instanciación genérica, lo primero que se elabora (procesa) es su parte de especificaciones, y posteriormente su cuerpo.

La ejecución de la primera instrucción del cuerpo de un programa principal (después del begin), debe esperarse a que se creen y activen las siguientes tareas:

1.- Todas aquellas de creación estática, que dependen de la unidad que constituye al programa principal (su unidad maestra).
2.- Todas aquellas de creación estática, que dependen de subprogramas, paquetes o instanciaciones declaradas y referenciadas en el programa principal.
3.- Todas las de creación estática, cuyos maestros dependen a su vez de otros maestros, que están declarados y referenciados en el programa principal (el nivel de anidamiento no posee límite definido).
4.- Todas las de creación estática, cuyos maestros son unidades de librería, referenciadas y utilizadas directa o indirectamente por el programa principal (indirectamente a través de subunidades componentes del programa).

Como puede observarse, en la ejecución del programa principal ya existe un grupo de tareas procesándose concurrentemente, sin necesidad de utilizar instrucciones explícitas de inicialización de tareas o ejecución paralela de las mismas; por tal razón, el paralelismo es implícito.

Sin embargo, ADA ofrece al programador las herramientas necesarias para que él cree y active tareas, según su propia lógica y necesidades.
Estas tareas, al ser activadas, se procesarán concurrentemente con las demás, que ya habían sido activadas. Se trata de las tareas de creación dinámica, que se activan en el momento de ejecución de la instrucción NEW.

Se mencionó en la sección de procesos que una tarea finalizaba su ejecución cuando terminaba de ejecutar la lista de

instrucciones que componen su cuerpo, o bien cuando procesaba la instrucción termina (TERMINATE, alternativa del SELECT).

La instrucción termina sólo es elegible de ejecutarse cuando no existen llamadas pendientes (interacción) de otras tareas, lo que ocasiona la terminación de una tarea, si ésta no posee tareas dependientes.

Si una tarea tiene tareas dependientes, no podrá terminar su ejecución hasta que terminen sus tareas dependientes.

En función de lo anterior, una tarea que es maestro, cuyos dependientes no han terminado, puede encontrarse en uno de los dos siguientes estados:

1.- Ha finalizado la ejecución de su lista de instrucciones (cuerpo), pero no puede terminar.
2.- La tarea está en espera (suspendida), como consecuencia de la ejecución de una instrucción TERMINA como alternativa de un SELECT.

En el primer punto, se dice que la tarea ha completado su ejecución. La diferencia entre los términos completar y terminar estriba en que en el primer caso la tarea todavía existe, ocupa un espacio en memoria, y puede competir por uso de recursos del sistema, mientras que una tarea terminada ha dejado de existir.

Una tarea maestro que se encuentre en el primer estado no podrá terminar sino hasta que terminen sus tareas dependientes. Este punto también es válido para subprogramas, paquetes o bloques de instrucciones; es decir, no podrán finalizar su total ejecución hasta que hayan terminado sus tareas dependientes.

Una tarea maestro que se ubique en el segundo estado terminará hasta que sus tareas dependientes hayan terminado, o bien estén en el mismo estado que ella.

Si tanto la tarea maestro como sus dependientes están suspendidas en una instrucción TERMINA, entonces las dependientes junto con la maestra terminan.

En igual forma, el programa principal terminará su ejecución en función de las reglas expuestas anteriormente, y esperará la terminación de cualquier tarea dependiente.

El programador cuenta con otra herramienta que le proporciona ADA, para terminar la ejecución de tareas. Se trata de la instrucción ABORT, la cual hace anormales a las tareas referenciadas en ella.

Una tarea anormal es aquella a la que nó se le permite interaccionar con otras tareas, ni terminar la ejecución de ciertas instrucciones que constituyan su cuerpo, y la tarea completará su procesamiento inmediatamente.

Una vez que la tarea anormal ha completado su ejecución, su

terminación se realizará según las reglas expuestas en líneas anteriores.

Toda tarea que dependa de una anormal se convertirá también en anormal. De esta forma, si una tarea maestra se hace anormal, hace que todas sus dependientes (y dependientes de sus maestras dependientes) también se hagan anormales, es una especie de reacción en cadena. La sintaxis de la instrucción ABORT es:

<instrucción_abort> ::= abort <nombre_tarea>{,<nombre_tarea>};

Toda tarea puede abortar a cualquier otra tarea, inclusive a ella misma.

En ADA es posible desde una tarea o unidad de programa conocer el estado de cualquier otra tarea.

Toda tarea tiene un conjunto de atributos, los cuales indican su estado, y que pueden ser consultados por cualquier unidad. De esta forma, una tarea puede prevenir su interacción con otra, que posiblemente ya terminó, o es anormal.

El atributo <nombre_tarea>'CALLABLE es una variable booleana, cuyo valor es falso si la tarea designada por <nombre_tarea> ha completado o terminado su ejecución, o bien si la tarea es anormal.

El atributo <nombre_tarea>'TERMINATED es también una variable booleana, cuyo valor es verdadero si la tarea designada ha terminado.

3.3- Conclusiones.

1.- Por la estructura de sus programas, ADA presenta mucha versatilidad y flexibilidad para acoplarse a distintas arquitecturas de equipos de cómputo.
2.- ADA presenta una estructura jerarquizada en la dependencia entre tareas, lo cual es beneficioso.
3.- Su esquema de paralelismo está en función de la dependencia entre las tareas.
4.- No posee ninguna instrucción que explícitamente genere paralelismo (es implícito).
5.- El manejo de atributos da mucha flexibilidad a ADA en el manejo e interacción entre tareas.

Respecto al segundo punto, un beneficio de la característica de jerarquización de tareas es el poder agrupar procesos cuyas funciones estén relacionadas lógicamente.

Incluso, un programa puede estar constituido por varios paquetes que agrupen objetos relacionados lógicamente, con tareas también relacionadas entre sí; con la salvedad de que la comunicación entre un paquete y otro sea mediante las tareas que lo constituyen.

En lo que se refiere al cuarto punto, debería de instrumentarse algún mecanismo mediante el cual se permitiera al programador iniciar la ejecución concurrente de varias tareas en forma explícita, lo que proporcionaría mayor libertad en el manejo de concurrencia.

Finalmente, el uso de atributos da mucha flexibilidad en la programación, ya que permite programar situaciones anormales, evitando de esta forma la propagación de errores y eventos indeseables.

ADA
4.-COMUNICACION

En el lenguaje ADA existen los dos siguientes mecanismos de comunicación:

1.- Uso de variables compartidas.
2.- Envío de mensajes.

El primero permite que varias tareas (procesos) accesen simultáneamente un conjunto de variables. Sin embargo, no existe ningún mecanismo implícito para garantizar el acceso exclusivo de una tarea a una variable. Es por lo tanto responsabilidad del programador sincronizar el acceso a las variables.

El segundo mecanismo de comunicación de ADA hace uso del envío de mensajes.

En este caso el transmisor es la tarea activa, que inicia el proceso de comunicación con otra tarea. El receptor es la tarea pasiva, que está a la espera de ser invocada por otra tarea.

El mensaje está constituido por un conjunto de parámetros, tanto de entrada como de salida.

El canal de comunicación puede ser la red física de una red de computadoras; si la arquitectura es de multiprocesamiento, o de un solo procesador con multiprogramación, entonces el canal es la propia memoria del equipo.

A continuación se describe la sintaxis del mecanismo de envío de mensajes, así como su semántica. Sin embargo, para el mecanismo de variables compartidas se analizará exclusivamente su semántica, ya que su sintaxis es la asignación de variables.

4.2- Sintaxis de la mecánica de comunicación.

A las instrucciones de recepción de mensajes de una tarea receptora se les llama puntos de entrada, y a través de éstos se comunican las tareas transmisoras.

A las instrucciones de transmisión de mensajes de una tarea transmisora se les llama puntos de llamada de entrada, ya que invocan al punto de entrada de una tarea receptora.

Tarea receptora:

Las declaraciones de los puntos de entrada se especifican en la parte visible de especificaciones de la tarea receptora, y su cuerpo (si existe) está contenido en la instrucción representada por el accept. La sintaxis de la declaración de un punto de entrada es:

```
<declaración_punto_de_entrada> ::=
     entry <identificador> [(<rango_discreto>)] [<mensaje>];
```

El símbolo <identificador> es el nombre del punto de entrada, <rango_discreto> es un símbolo que representa el rango de valores discretos que pueden tomar los índices de una familia de tareas. Cada tarea de una familia posee puntos de entrada indexados con la tarea correspondiente. La sintaxis del mensaje se estudiará en la siguiente subsección.

La sintaxis de la instrucción ACCEPT (punto de entrada) es la siguiente:

```
<punto_entrada> ::= accept <nombre_punto_entrada>
                       [(<índice_punto_entrada>)] [<mensaje>]
                       [DO <lista_instrucciones>]
                       end [<nombre_punto_entrada>];
```

El símbolo <nombre_punto_entrada> representa el nombre del punto de entrada, y debe ser el mismo que el declarado en la parte de especificaciones. El símbolo <índice_punto_entrada>, representa el valor del índice de la tarea, a la que pertenece el accept, de entre las que constituyen la familia.

La lista de instrucciones representada por <lista_instrucciones> constituye el cuerpo del accept (punto de entrada).

Tarea transmisora:

La sintaxis de la instrucción punto de llamada de entrada que invoca a un punto de entrada es:

```
<punto_llamada_entrada> ::= <nombre_punto_entrada>[<mensaje>];
```

El símbolo <nombre_punto_entrada> es el nombre del punto de llamada de entrada de la tarea invocada.

La sintaxis de un mensaje es idéntica a la de los parámetros de los subprogramas, tanto en su parte real como en su parte formal.
Inclusive, como ha podido observarse, la sintaxis de un punto de llamada de entrada (invocación) es similar a la de una instrucción de llamada a procedimiento.

La sintaxis de la parte formal corresponde a la sintaxis del mensaje en el punto de entrada (instrucción accept) en la tarea receptora, y se muestra a continuación:

Parte formal

```
<mensaje> ::= (<lista_especificación_parámetros>)
<lista_especificación_parámetros> ::=
        <especificación_parámetro>{;<especificación_parámetro}
<especificación_parámetro> ::=
        <lista_identificadores>:<modo_parámetro>[:=<expresión>]
<lista_identificadores> ::= <identificador>{,<identificador>}
<modo_parámetro> ::= [in] | in out | out
```

Los identificadores corresponden a los nombres de los parámetros formales. El modo del parámetro se indica mediante los símbolos in, out (de entrada, salida). En la subsección de semántica se profundizará en el análisis de los modos de los parámetros.

Finalmente, al igual que en los subprogramas, pueden existir valores de default para aquellos parámetros de entrada que no han sido definidos. El valor que adquiere un parámetro de default está expresado en la expresión <expresión>.

La sintaxis de la instrucción de punto de llamada de entrada en la tarea transmisora, que invoca a un punto de entrada de una tarea receptora, es la siguiente:

Parte real

```
<mensaje> ::= (<lista_asociaciones_parámetros>)
<lista_asociaciones_parámetros> ::=
        <asociación_parámetro>{,<asociación_parámetro>}
<asociación_parámetro> ::=
        [<parámetro_formal> =>] <parámetro_real>
```

Como puede observarse, en la sintaxis se puede indicar el nombre del parámetro formal con el que se está asociando el correspondiente parámetro real.

Así, ADA es un lenguaje que permite la asociación por nombre, además de la posicional. En una asociación de parámetros por nombre, explícitamente se indican los nombres de los parámetros formales con los que se asocian los reales, como se muestra en el siguiente ejemplo:

```
BUFFER.LEE(dato => 300, stc => pila)
```

Observe que en una asociación por nombre, el orden es irrelevante, sin embargo, si se utiliza la asociación posicional, entonces cada parámetro real corresponde al formal que guarda la misma posición en la parte formal.

4.3- Semántica.

En esta parte se describirá en primer lugar la semántica del mecanismo de variables compartidas, y posteriormente la de envío de mensajes.

4.3.1- Uso de variables compartidas.

Como se indicó en la parte introductoria, este mecanismo está fundamentado en el acceso simultáneo de variables.
Las tareas pueden comunicarse indirectamente a través de las variables, que son globales al conjunto de las tareas que pueden accesarlas.

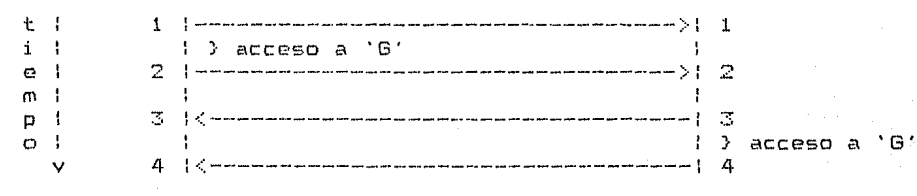
Si cada tarea modifica variables exclusivas de su acceso no existe ninguna restricción, pues no existen dos o más tareas que requieran accesar una misma variable. Sin embargo, si ese no es el caso, y más de una tarea requiere el acceso al mismo conjunto de variables globales, entonces es necesario que el programador garantice el acceso exclusivo a las mismas mediante la comunicación entre ellas.

Si dos o más tareas accesan una variable compartida, entonces ninguna de ellas puede suponer algo respecto al orden en el cual las otras realizan sus operaciones (accesos), excepto en los puntos donde se sincronizan.

Dos tareas se sincronizan en los siguientes eventos:

1.- Al principio y al final de su proceso de comunicación de envío de mensajes.
2.- Al principio y al final de su activación, cuando una depende de la otra.

Respecto al primer punto, se representa a continuación una gráfica que ejemplifica la forma en que el programador debe interaccionar dos procesos, para garantizar el acceso exclusivo a una variable global.



En el punto 1 la tarea A interacciona con B, para indicarle que desea accesar la variable G; entre los puntos de sincronización 1 y 2, A accesa a G, B sabe que no debe accesar a G, hasta que A se lo indique. En el punto 2, A le avisa a B que G está libre, y B puede entonces accesar G.

De igual forma, si la tarea C crea la tarea D, y ambas desean accesar la variable L, entonces C sabe que tiene acceso exclusivo a L en los dos siguientes estados:

1.- Antes de que D se active.
2.- Después de la terminación de D.

Mientras D exista, C no puede suponer nada respecto al orden en que D realice sus accesos a L, a excepción de que exista comunicación entre ellas.

También influye el tipo de acceso que se desee hacer a la variable compartida, y existen dos tipos diferentes de acceso:

1.- Lectura.
2.- Modificación o escritura.

En el primer caso, se puede permitir la lectura simultánea de la variable, ya que se sabe que ninguna tarea la modificará. Sin embargo, si en el segundo caso se sabe que entre dos puntos de sincronización una tarea alterará a la variable, no debe permitírsele a ninguna otra leerla, ni modificarla durante ese período.

En función de lo anterior, las reglas que debe seguir el programador para evitar resultados incorrectos e impredecibles son:

1.- Si una tarea lee una variable compartida entre dos puntos de sincronización, entonces ninguna otra debe modificarla durante ese período.
2.- Si una tarea modifica una variable compartida entre dos puntos de sincronización, entonces no debe permitírsele a ninguna otra tarea leerla, ni modificarla durante ese período.

4.3.2- Semántica del mecanismo de envío de mensajes.

En esta sección se hará una descripción de los siguientes puntos:

1.- Semántica de la comunicación entre dos tareas.
2.- Semántica de la comunicación entre más de dos tareas.
3.- Características del mecanismo.
4.- Semántica del mensaje.
5.- Causas que originan fallas en la comunicación.

4.3.2.1- Semántica de la comunicación entre dos tareas (cita).

Analícese el esquema del mecanismo de comunicación mediante el envío de mensajes, durante las siguientes tres etapas que lo constituyen.

Primera etapa:

La tarea transmisora desea enviar un mensaje a la receptora; para tal efecto invoca (llama) a la tarea receptora. En este punto, puede suceder cualquiera de los dos siguientes eventos:

1.- La tarea receptora estaba esperando ser invocada (suspendida), y al serlo, atiende inmediatamente a la tarea transmisora.

2.- La tarea receptora está ocupada atendiendo quizá a otras llamadas y, consecuentemente, hace esperar a la tarea transmisora.

En esta etapa una tarea comúnmente debe esperar a la otra para efectuar la transmisión del mensaje, es decir, ambas tareas deben sincronizarse.

Segunda etapa:

Después de que ambas tareas se han sincronizado, se realiza la transferencia del mensaje, la cual se puede considerar como un intercambio de información entre ambas tareas.

El mensaje es una lista de parámetros, los cuales pueden ser de entrada, salida, o de entrada y salida simultáneamente.

Como primer paso dentro de esta etapa, se establece la asociación entre los parámetros reales de la tarea transmisora y los formales de entrada de la receptora.

Posteriormente, la tarea receptora puede realizar algunas operaciones con los parámetros de entrada y, tal vez, generar resultados que quedarán registrados en los correspondientes parámetros formales de salida, para después asociarlos con los parámetros reales de la tarea transmisora. Con esto se retorna a la tarea transmisora los resultados que requiere de la receptora.

Los parámetros que son de entrada y salida son aquellos en los cuales se lee un valor de entrada, y en el mismo se retorna un resultado modificando el valor original.

El mensaje puede no contener información (parámetros), y ser simplemente una bandera a utilizar con fines de sincronización entre ambas tareas.
Durante el tiempo que transcurra el intercambio de información, la tarea transmisora estará suspendida (en espera).

Finalmente, una vez que se ha realizado el intercambio de información (si lo hay), ambas tareas continúan con su ejecución en forma independiente; ésta constituye la tercera etapa del esquema de comunicación por envío de mensajes entre tareas.

Este mecanismo de comunicación recibe el nombre de cita o junta de tareas (rendezvous).

Toda cita trae implícita una etapa de sincronización, otra de intercambio (si existe), y finalmente, cada elemento involucrado continúa con sus actividades individuales, al final de la misma.

Como se mencionó en la sección de procesos, las tareas receptoras poseen puntos o puertas (instrucciones accept) dentro de su cuerpo, donde pueden ser invocadas o llamadas.

Estos puntos reciben el nombre de puntos de entrada (entry), y son realmente instrucciones de entrada.

A las instrucciones de llamada en las tareas transmisoras, que invocan a los puntos de entrada de las receptoras, se les llama 'puntos de llamada de entrada' (entry call).

Una tarea receptora puede tener varios puntos de entrada, realizando diferentes operaciones en cada uno de ellos, y se les identifica con un nombre, de tal manera que la tarea transmisora debe, al invocar, referenciar no únicamente al nombre de la tarea receptora, sino al de su correspondiente punto de entrada.

Como ya se ha indicado, el punto de entrada está representado por la instrucción accept, y puede estar integrado por una lista de instrucciones, que precisamente implementan la operación que la tarea transmisora requiere de la receptora.

La cita (rendezvous) se puede resumir en los siguientes eventos:

1.- Sincronización de las tareas transmisora y receptora, con lo que se realiza la invocación al punto de entrada de la receptora.
2.- Se asocian parámetros de entrada (transferencia del mensaje), y mientras se ejecuta la lista de instrucciones (si existe) que constituyen el punto de entrada (accept), se suspende la ejecución de la tarea transmisora.
3.- Al culminar la ejecución de la lista de instrucciones que conforman el punto de entrada, se reactiva la tarea transmisora, y se realiza la asociación de parámetros resultantes (si existen) entre ambas tareas, después de lo cual cada una se sigue procesando en forma independiente.

4.3.2.2- Semántica de la comunicación entre más de dos tareas.

Existe exclusión mutua en el acceso a una tarea, por lo que no es posible que varias estén simultáneamente (en el mismo instante de tiempo) comunicándose con la misma, lo cual permite heredar la exclusión mutua a algún recurso compartido a través de una tarea.

Por lo que, por cada punto de entrada que exista en una tarea,

habrá una cola en donde se registran las solicitudes de comunicación (llamadas) de las tareas invocantes. La primera tarea en invocar será la primera en ser registrada y, consecuentemente, la primera en comunicarse (en establecer la cita), estableciéndose de esta forma un orden FIFO (el primero que invoca es el primero en ser atendido) en la cola.

Sin embargo, ADA sí permite establecer prioridades entre tareas transmisoras, cuando éstas invocan a diferentes puntos de entrada, de tal manera que las tareas con mayor prioridad de procesamiento serán atendidas primero. Para asignar una prioridad a una tarea es necesario indicárselo al compilador mediante la siguiente instrucción:

PRAGMA PRIORITY (<expresión>);

'<expresión>' representa el nivel de prioridad con que deba procesarse la tarea, un mayor valor indica mayor prioridad.

Esta instrucción se incluye en la parte de especificaciones de la tarea, y además la prioridad se mantiene estática, de tal manera que no es posible modificarla a tiempo de ejecución.

Pueden existir anidamientos en las interacciones entre tareas, por lo que una tarea B que mantiene una cita (rendezvous) con la tarea invocada A, puede poseer una instrucción de punto de llamada de entrada, como parte del conjunto de instrucciones que integran su punto de entrada, como por ejemplo, una llamada a la tarea C, la que a su vez, puede invocar a una cuarta tarea D, etc.

Para que pueda finalizar la primera cita establecida (entre A y B), es necesario que termine la segunda (entre B y C) y así sucesivamente, es decir, las citas deben finalizar en el orden inverso en el que se establecieron, iniciando con la última establecida y terminando con la primera. Como se puede observar, se forma una cadena de citas establecidas entre tareas.

Es responsabilidad del programador evitar ciclos en el anidamiento de citas; es decir, que en un nivel de anidamiento determinado se establezca una cita con alguna tarea que ya exista en la cadena de anidamiento, pues se generaría un abrazo mortal, el cual también se produciría si una tarea se invoca a sí misma.

Otra observación que merece atención es la característica de bidireccionalidad de este mecanismo de comunicación, ya que entre las dos tareas que incurren en la mecánica, la transferencia de información puede ser en ambos sentidos (parámetros de entrada, salida).

4.3.2.3- Características del mecanismo de comunicación.

El mecanismo de comunicación por envío de mensajes de ADA posee las siguientes propiedades:

1.- Es síncrono.
2.- Es asimétrico respecto a la invocación.
3.- Es asimétrico respecto a la suspensión.

En lo referente al primer punto, dos tareas deben sincronizarse para la transferencia del mensaje, lo que implica un ordenamiento en la ejecución de las transacciones entre dos tareas; por ejemplo, la recepción del mensaje indica que la tarea transmisora invocó previamente a la receptora.

Para lograr la sincronización, alguna de las tareas debe esperar a la otra, es decir, en un mismo instante de tiempo ambas coinciden en un mismo punto de ejecución; por parte de la transmisora se está en una instrucción de invocación, y por la receptora, en un 'accept' respecto al mismo punto de entrada.

Por tal motivo, la comunicación entre ambas tareas es síncrona. En una comunicación asíncrona, cada tarea se procesaría independientemente, sin que existan puntos dentro de su ejecución en que interaccionen; no existiría el concepto de espera de una tarea por otra.

En lo que respecta al segundo punto, la tarea transmisora invoca a la receptora, y nunca sucede lo inverso; es decir, la tarea transmisora debe conocer la identidad de la receptora, así como de sus puntos de entrada. Sin embargo, la tarea receptora no necesita identificar a las que la invocan.

Esto establece una asimetría, ya que una siempre llama a la otra, y ésta última no invoca a la primera. La tarea transmisora es un proceso siempre activo, mientras que la receptora es la parte pasiva (siempre en espera de ser llamada).

Este tipo de interacción asimétrica entre tareas conforma un esquema de tipo cliente/servidor: el cliente es la tarea activa que solicita un servicio del servidor (receptora).

El cliente debe saber a qué servidor recurrir, pero para el servidor es irrelevante conocer la identidad del cliente. Esta característica permite que existan tareas de biblioteca, que puedan ser invocadas por cualquier tarea usuaria.

De la misma forma, para el acceso a un recurso compartido por parte de varias tareas, se designa a una tarea exclusiva para el control del recurso, y todas las que deseen acceso al mismo deberán invocar a la tarea correspondiente. La tarea que controla al recurso es el servidor, las que la invocan son los clientes.

Finalmente, en lo que respecta al tercer punto, la tarea receptora que procesa la información y recibe el mensaje ocasiona la suspensión de la tarea transmisora, mientras procesa el bloque de instrucciones de su correspondiente punto de entrada. Sin embargo, bajo ninguna circunstancia puede la tarea transmisora suspender la ejecución de la receptora.

En este caso, también se presenta una situación de asimetría, ya que una tarea tiene el privilegio de suspender la ejecución de la otra, mientras que ésta última carece de la misma prerrogativa.

4.3.2.4- Semántica del mensaje.

En el mecanismo de comunicación que utiliza ADA, el mensaje es un conjunto de parámetros, en forma muy semejante a los de un subprograma, o un procedimiento de Pascal.

Como ya se ha indicado, existen parámetros de entrada, salida, y entrada y salida, lo cual da lugar a la existencia de tres modos diferentes de parámetros, los cuales se representan en la sintaxis de la parte formal con los símbolos: in, out e in out.

Los parámetros formales de modo in se consideran constantes, y la asociación es por valor.

Los parámetros formales de modo out son variables, la asociación es por resultado (asignación de valores a los parámetros reales correspondientes al final de la comunicación).

La asociación de los parámetros formales de modo in out es por valor y resultado. Para datos de tipo escalar existen los tres modos de parámetros descritos anteriormente.

Si al finalizar la cita existen parámetros formales de salida que no fueron definidos durante la misma, permanecerán indefinidos los parámetros reales asociados.

Para datos de tipo arreglo, registro, o datos de tipo acceso a tareas, no está determinado el modo de asociación de los parámetros, y dependerá de la implementación, pues este modo puede ser de referencia o de valor (datos de tipo escalar).

Si un parámetro real designa a una tarea, entonces para los tres modos el parámetro formal designará a la misma tarea.

4.3.2.5- Causas que originan fallas en la comunicación.

La comunicación puede fallar por las siguientes causas:
1.- La terminación de la tarea receptora.
2.- La anormalidad de las tareas transmisora o receptora.

En el primer caso, la tarea transmisora desea comunicarse con una tarea que ya terminó, o que no existe.
En este caso, la instrucción de punto de llamada de entrada de la tarea transmisora genera la excepción: tasking_error.

Sin embargo, la tarea transmisora puede hacer uso de los atributos de la receptora : T'CALLABLE y T'TERMINATED, para evitar la generación de la excepción.

En el segundo caso, se puede aplicar la misma técnica con respecto a la anormalidad de la tarea receptora.

Es importante hacer notar que si una vez establecida la cita entre dos tareas y alguna de ellas se hace anormal, la terminación de esa tarea no concluye hasta que haya terminado la cita.

Por otra parte, si la anormalidad se presenta en la tarea transmisora cuando aún no se ha establecido la cita, entonces se remueve a esta tarea de la cola del punto de entrada invocado en la tarea receptora, y se procede a la terminación de la tarea anormal.

En todos los casos, se genera la excepción tasking_error cuando una tarea ha llamado al punto de entrada de una tarea anormal o terminada.

La tarea receptora puede en todo momento conocer el número de tareas transmisoras que desean comunicarse con cada uno de sus puntos de entrada, mediante el atributo E'COUNT, donde E es el nombre del punto de entrada del cual se desea conocer el número (COUNT) de tareas transmisoras pendientes de ser atendidas.

4.6- Conclusiones.

1.- El mecanismo de variables compartidas está limitado; su acceso debe controlarse por parte del programador, y por tal motivo se recomienda usarlo lo menos posible.
2.- El mecanismo de envío de mensajes permite tener una interacción cliente/servidor, lo cual facilita el uso de biblioteca de rutinas. Incluso, ADA es un lenguaje que permite compilación separada, de manera que se van añadiendo los nuevos módulos (unidades) compilados a la biblioteca. Un módulo puede ser una tarea, un subprograma, paquete, etc.
3.- Al igual que en la sección anterior, el uso de atributos, y específicamente E'COUNT, facilita considerablemente a la tarea receptora el manipular su interacción con las transmisoras.
4.- En algunos casos resultaría conveniente manejar interacciones de tipo interconexión, los cuales son asíncronos; ADA no maneja ningún mecanismo de comunicación de tipo asíncrono, lo cual se puede considerar como una desventaja.
5.- La operación de los mensajes a través de parámetros permite que las instrucciones de punto de llamada de entrada sean equivalentes sintácticamente a llamadas a procedimientos, lo cual simplifica considerablemente su uso.

5.-SINCRONIZACION

En esta sección se estudiarán los siguientes tópicos:

1.- Sincronización entre tareas.
2.- Características síncronas del mecanismo de comunicación.
3.- Mecanismos condicionales de sincronización (sintaxis y semántica).
4.- Sincronización con el tiempo (sintaxis y semántica).

5.2- Sincronización entre tareas.

Resulta interesante analizar los puntos en que dos tareas interaccionan entre sí, es decir, aquella fase de su ejecución en que ambas interaccionan en un instante determinado de tiempo.

Una tarea interacciona con su maestra en sus momentos de activación y terminación. La activación puede ser estática o dinámica (en el instante de procesarse la instrucción NEW).
A esos momentos se les llama puntos de sincronización.

Como ya se había indicado en secciones anteriores, dos tareas se sincronizan en los puntos de inicio y terminación de su mecanismo de comunicación por envío de mensajes, es decir, al iniciar y finalizar la cita (rendezvous); estos puntos también son puntos de sincronización.

En conclusión, existen cuatro puntos de sincronización en la interacción entre tareas:

1.- En la activación de una tarea con su maestra.
2.- En la terminación de una tarea con su maestra.
3.- Al inicio de una cita (rendezvous).
4.- Al final de la cita.

5.3- Características síncronas del mecanismo de comunicación.

Como puede observarse, en la mecánica de comunicación de envío de mensajes en ADA, se establecen puntos de sincronización, constituyéndose así el mecanismo de comunicación, también en mecanismo de sincronización.

Lo anterior es consecuencia de que el efecto de la comunicación entre tareas no necesariamente implica un intercambio de información, sino que puede ser en sí un punto de sincronización entre ellas, gracias a la propiedad síncrona del mecanismo.

Otra característica importante que ya se ha analizado en secciones anteriores, pero cuyo análisis respecto de sincronización resulta ser interesante, es el de exclusión mutua.

Una tarea se asemeja en este aspecto a un monitor, pues se constituye en una barrera dentro de la cual sólo se permite la

interacción con una única tarea. Los puntos de interacción son los de sincronización (inicio y final de la cita).

Si existen varias tareas transmisoras que hayan invocado diferentes puntos de entrada, se selecciona a una de ellas para establecer la cita y las demás deben esperar su turno.

En un momento determinado, únicamente puede estar establecida una cita en una tarea receptora, y ni las tareas transmisoras ni las receptoras pueden efectuar más de una cita simultáneamente en un instante de tiempo.

Por lo anterior, no existe necesidad de sincronizar la interacción entre más de dos tareas, pues la implementación de ADA resuelve este tópico.

Se deduce que la filosofía de ADA consiste en asignar tareas a recursos compartibles, con lo cual se propaga la exclusión mutua al propio recurso, y se garantiza la sincronización en el acceso del mismo, por parte de varias tareas usuarias.

Por ejemplo, se puede dedicar una tarea a operar un driver que maneje algún dispositivo periférico, o bien, puede controlar al manejador de una base de datos, etc., de tal manera que toda tarea externa que requiera hacer uso del recurso deberá comunicarse con la tarea controladora del driver o del manejador de la base de datos.

En lo que se refiere a excepciones, el lector puede pensar que la presencia de una excepción pudiera inteferir en forma no deseable con otras tareas, específicamente con la tarea maestra o con las dependientes.

Una excepción se puede presentar en forma totalmente aleatoria, de tal manera que puede interferir asíncronamente con tareas con las que existe interacción.

Con objeto de evitar este tipo de interferencia en ADA, se prohibe e inhibe la propagación implícita de una excepción fuera del cuerpo de una tarea.

5.4- Mecanismos adicionales de sincronización (análisis sintáctico y semántico).

Otro mecanismo de sincronización que tiene a la mano el usuario es el uso de guardias.

Un guardia permite que se establezca la comunicación entre dos tareas, en función de una expresión lógica (booleana), y su sintaxis es la siguiente:

```
instrucción ::= <guardia> <instrucción_alternativa>
<instrucción_alternativa> ::=
     <instrucción_accept> | <instrucción_delay>
   | terminate
```

<guardia> ::= when <expresión booleana> =>

La instrucción 'delay' se analizará en el siguiente subinciso. 'instrucción' puede ser un punto de entrada, o bien, la alternativa de una instrucción SELECT (se analizará en la siguiente sección).

La instrucción termina (TERMINATE) deja a la tarea en un estado de terminación y, si no posee tareas dependientes, finalizará su ejecución.

A la expresión lógica también se le llama condición, y si esta expresión es verdadera se procesa a la instrucción accept.
Si la condición es falsa, aunque existan varias tareas transmisoras en cola de espera que deseen comunicarse, ninguna será atendida hasta que la condición sea verdadera.

Se verifica que previa a cualquier operación, la cola de espera no esté vacía.

Con el uso de los guardias se restringe el orden en que se realizan las operaciones, sincronizando de esta forma la interacción entre tareas.

5.5- Sincronización con respecto al tiempo (análisis sintáctico y semántico).

En programación en tiempo real, en que los procesos deben responder dentro de los límites de tiempo de los fenómenos del mundo real con los que se mantiene interacción, se hace imprescindible para el programador contar con algún mecanismo que permita obtener sincronización con respecto al tiempo.

ADA proporciona una instrucción que auxilia al programador, y cuya sintaxis es la siguiente:
<instrucción_temporización> ::= delay <expresión>;

La <expresión> debe ser del tipo predefinido de punto fijo 'DURATION', su valor es expresado en segundos. El tipo DURATION está declarado así:

```
type DURATION is range DURATION'SMALL..864000;
```

DURATION'SMALL es una constante que representa la más pequeña duración; depende de la implementación del lenguaje, y su valor no debe ser mayor de 20 milisegundos.

La ejecución de la instrucción delay consiste en los siguientes pasos:

1.- Se evalúa la expresión (si es negativa se considera como cero).
2.- Se suspende la ejecución de la tarea por el número de segundos (o fracciones de segundo) que indica la expresión.

Como ejemplo, suponga que una tarea debe leer un conjunto de datos de temperatura de un proceso físico cada minuto; la tarea debe sincronizarse con el tiempo real en que se sucede el proceso físico. Para lograr lo anterior, debe realizar las siguientes operaciones:

1.- Conocer el tiempo actual.
2.- Definir el intervalo que deberá suspenderse la tarea.
3.- Realizar las lecturas cuando se cumpla el intervalo.

El cuerpo de la tarea es:

```
task body lee is
     use CALENDAR;
     -- usa el paquete calendario
     intervalo:DURATION := 60;
     sig_tiempo:TIME := CLOCK + intervalo;
begin
     loop
          delay sig_tiempo - CLOCK;
          -- lee todos los datos
          sig_tiempo := sig_tiempo + intervalo;
     end loop;
end lee;
```

4.6- Conclusiones.

Las ventajas que presenta ADA en lo concerniente a sincronización son:

1.- Su mecanismo de comunicación por envío de mensajes es síncrono, lo que facilita la sincronización entre tareas.
2.- Toda tarea mantiene en un momento determinado interacción con una única tarea; las demás deben esperar su turno. La exclusión mutua en el código que compone el cuerpo de una aceptación es proporcionado gratuitamente por la implementación del lenguaje ADA.
3.- El usuario cuenta con mecanismos auxiliares de sincronización, como los guardias y el delay, que permite obtener sincronización con respecto al tiempo.

ADA
6.-INDETERMINISMO

En el lenguaje ADA existe sólo una instrucción que imprime el sello de 'indeterminismo': SELECT.
La estructura general de la instrucción 'SELECT' es:

```
SELECT
     lista_instrucciones_1;
OR
     lista_instrucciones_2;
          .
          .
OR
```

```
        lista_instrucciones_N;
    END SELECT;
```

Del conjunto de las N listas de instrucciones, se selecciona al azar alguna de ellas y se procede a su ejecución.

La selección es totalmente aleatoria, y no existe por lo tanto ningún orden o algoritmo que la haya determinado.
La estructura representada es muy general, y se incluyó para mostrar la propiedad no detrminística de la propia instrucción.

A continuación se presenta la sintaxis y se analiza la semántica del SELECT.

6.2- Sintaxis.

La Sintaxis de la instrucción SELECT es la siguiente:

```
<instrucción_select> ::=    <selección_con_espera>
                          |   <selección_condicional>
                          |   <selección_temporal>
```

Existen tres formas o construcciones de la instrucción SELECT :

1.- Selección con espera.
2.- Selección condicional.
3.- Selección temporal.

La selección con espera tiene la siguiente sintaxis:

SELECCION CON ESPERA:

```
<selección_con_espera> ::=
        SELECT <alternativa_de_selección>
               { OR <alternativa_de_selección>}
               [else <lista_de_instrucciones>]
        END SELECT;
<alternativa_de_selección> ::=
        [<guardia>]<alternativa_de_selección_con_espera>
<alternativa_de_selección_con_espera> ::=
     <alternativa_aceptación> | <alternativa_temporal>
   | <alternativa_terminación>
<alternativa_aceptación> ::=
     <instrucción_accept>[<lista_de_instrucciones>]
<alternativa_temporal> ::=
     <instrucción_delay>[<lista_de_instrucciones>]
<alternativa_terminación> ::= terminate
<guardia> ::= when <expresión_booleana> =>
```

La <instrucción_delay> es precisamente el delay de ADA que se estudió en la sección de sincronización.
La instrucción terminate pone en estado de terminación a la tarea.

SELECCION CONDICIONAL
Sintaxis de la selección condicional:

```
<selección_condicional> ::=
      select
        <instrucción_de_llamada>;[<lista_de_instrucciones>]
      else <lista_de_instrucciones>
      end select;
```

<instrucción_de_llamada> es la llamada al punto de entrada de alguna otra tarea.

SELECCION TEMPORAL
Sintaxis de la selección temporal:

```
<selección_temporal> ::=
      select
            <instrucción_de_llamada>;
            [<lista_de_Instrucciones>]
      or
            <instrucción_delay>;[<lista_de_instrucciones>]
      end select;
<instrucción_delay> ::= delay <expresión>
```

6.3- Semántica.

6.3.1- Semántica de la Selección con espera.

Como se observó del análisis sintáctico, este tipo de SELECT está integrado por alternativas de selección.

Cada alternativa de selección consiste en una instrucción de aceptación, de temporización (delay), o de terminación, pudiendo ir precedida de un guardia, y sucedida por un conjunto de instrucciones.

Las alternativas de selección se clasifican en dos tipos:

1.- Abiertas.
2.- Cerradas.

Una alternativa de selección es abierta en cualquiera de los dos siguientes casos:

1.- Cuando no posee guardia.
2.- Cuando la condición que constituye al guardia es verdadera.

Una alternativa de selección es cerrada cuando posee guardia cuya expresión es falsa.
Por el tipo de instrucciones que conforman a las alternativas de selcción, éstas se pueden clasificar en:

1.- Alternativas de entrada (aceptación).
2.- Alternativas temporales (delay).
3.- Alternativas de terminación (terminate).

Y cada una de éstas puede ser una alternativa abierta o cerrada. Existen restricciones en el uso de las alternativas de una instrucción SELECT, las cuales se condensan en los siguientes

puntos:

1.- Las alternativas temporal, de terminación, y el ELSE son mutuamente excluyentes.
2.- Unicamente puede existir una alternativa de terminación en el cuerpo del SELECT.
3.- Unicamente puede haber un ELSE en el cuerpo del SELECT.

En lo referente al primer punto, si una de las tres alternativas (incluyendo el ELSE) está presente en el cuerpo de un SELECT, entonces ninguna de las otras dos debe de coexistir en el mismo SELECT.

Del segundo y tercer punto, se deduce que no existe límite en el número de alternativas temporales y de entrada que coexistan en el cuerpo de un SELECT .

El primer paso en la ejecución de una selección con espera consiste en la determinación de todas las alternativas que estén abiertas.
Para lograr ese objetivo, se evalúan todas las expresiones booleanas de los guardias que estén presentes, para determinar su valor verdadero o falso.

Como resultado de este primer paso se descartan las alternativas que estén cerradas.
Posteriormente, se evalúan las expresiones de todas aquellas alternativas temporales que estén abiertas y se registran en los mecanismos de control de tiempo, para que se inicie el conteo de cada instrucción delay.

Si como resultado del primer punto no existe ninguna alternativa abierta, y la instrucción SELECT no posee un ELSE, entonces se activa la excepción PROGRAM_ERROR, después de lo cual finaliza la ejecución del SELECT.

En caso de que existan alternativas abiertas, se tendrán las cuatro siguientes situaciones:

1.- Exclusivamente alternativas de entrada en el SELECT.
2.- Tanto alternativas de entrada como temporales en el SELECT.
3.- Alternativas de entrada y una de terminación en el SELECT.
4.- Alternativas de entrada y un ELSE en el SELECT.

Lo anterior es consecuencia de las restricciones en el uso de las alternativas de selección; a continuación se hará un análisis de cada una de estas cuatro situaciones.

6.3.1.1- Semántica de un SELECT constituido exclusivamente por alternativas de entrada.

A la instrucción ACCEPT (punto de entrada) se le llamará también instrucción de entrada y, por abreviación, se denominará cola FIFO (first input- first output) a la cola de un punto de entrada de la tarea receptora.

En la sección anterior (sincronización), se indicó que en ningún momento era posible que dos o más tareas transmisoras mantuvieran comunicación (por envío de mensajes) simultáneamente con la tarea receptora; es decir, una tarea sólo puede mantener una cita (rendezvous) a la vez.

Lo anterior se hereda a la propia instrucción SELECT, y se resume en lo siguiente:

No se pueden establecer citas en más de un punto de entrada simultáneamente, en el cuerpo de un SELECT

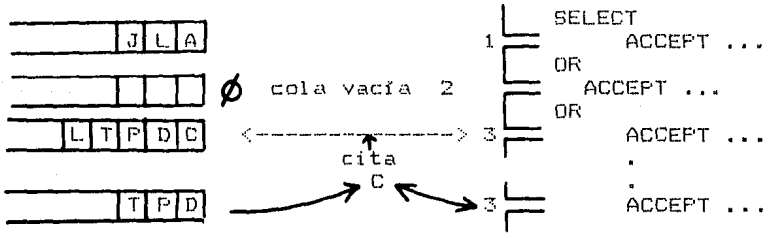
Por parte de las tareas transmisoras sucede la misma situación, ya que no es posible que simultáneamente mantengan comunicación con dos o más tareas receptoras.

Como consecuencia, en el cuerpo de un SELECT, toda tarea transmisora únicamente puede estar registrada en la cola de un solo punto de entrada.

La cola FIFO de algún punto de entrada puede estar vacía, y la instrucción de entrada queda en espera de interaccionar con alguna tarea transmisora que la llegue a invocar.

A continuación se enumeran las fases de ejecución de una instrucción SELECT formada exclusivamente por alternativas de entrada:

1.- Determinación de aquellas alternativas que sean abiertas (se descartan las cerradas).
2.- De las alternativas abiertas cuyos puntos de entrada no posean colas vacías se selecciona indeterminísticamente una.
3.- Se establece la cita entre la primera tarea registrada en la cola y el punto de entrada de la alternativa seleccionada.



4.- Si la instrucción de entrada (accept) va sucedida por una lista de instrucciones, entonces la cita se procesa después de la culminación de la cita.
5.- Finaliza la ejecución de la instrucción SELECT.
6.- Si en el punto 2 todas las alternativas abiertas poseen puntos de entrada con colas vacías, entonces la tarea se espera hasta que alguna otra llame a alguno de sus puntos de entrada.

6.3.1.2- Semántica de un SELECT constituido por alternativas de entrada y temporales.

A continuación se enumeran las etapas de ejecución de esta construcción:

1.- Determinación del estado de las alternativas, y la eliminación de las que estén cerradas.
2.- Cálculo de las expresiones de tipo TIME de las instrucciones contadoras de tiempo (delay), y su registro.

    Existe una cola de relojes (delay queue) donde se registra a las tareas que poseen instrucciones contadoras de tiempo (instrucciones delay). Es una lista ligada en donde las tareas están ordenadas cronológicamente por sus contadores de tiempo.

    Cuando se vence el tiempo de la tarea que encabeza la lista, se genera una interrupción y se vuelve a instalar la tarea en la cola de listos (ready), para que vuelva a procesarse.
    En este punto únicamente se registra la tarea y sus correspondientes delays en la cola de relojes.

3.- Determinación de los puntos de entrada que poseen colas no vacías.
4.- Si existen puntos de entrada con colas no vacías, se selecciona (en caso de haber más de uno) indeterminísticamente uno y se procede al punto 5.
5.- Se establece la cita (rendezvous), y se procesa la lista de instrucciones que integra la alternativa de entrada, así como alguna otra que pudiera sucederla (si existen), con lo cual finaliza la ejecución de la instrucción SELECT.
6.- Si todas las colas de los puntos de entrada están vacías, se suspende la tarea hasta que suceda uno de los dos siguientes eventos:
    a.- Alguna o varias tareas invoquen algún punto de entrada (se registran en las colas correspondientes).
    b.- Haya terminado el conteo del contador de tiempo de alguna instrucción delay.
7.- Si sucede el evento a, se realizan los puntos 4 y 5 anteriores.
8.- Si sucede el evento b, se procede a ejecutar la lista de instrucciones que sucede a la instrucción de temporización delay, y que integra la alternativa temporal, con lo que finaliza el procesamiento del SELECT. Si llegaran a existir varias instrucciones temporales (delay) con el mismo contador de tiempo, se selecciona indeterminísticamente una de ellas.

Finalmente, si todas las alternativas de entrada están cerradas,

entonces se procesa exclusivamente las alternativas temporales.

6.3.1.3- Semántica de un SELECT constituido por alternativas de entrada y una de terminación.

Como repaso de la sección de procesos, y debido a que se hace referencia a estos conceptos, se vuelven a enunciar los eventos bajo las cuales una tarea puede finalizar su ejecución:

a.- La tarea es maestra y todas sus dependientes han concluido su ejecución, o están en estado de espera en una alternativa de terminación.

b.- La tarea no es maestra y todas las tareas dependientes de la maestra asociada con ella han terminado, o se encuentran en estado de espera en una alternativa de terminación.

Si no se cumple ninguno de los puntos a o b, la tarea no podrá terminar (finalizar su ejecución).

A continuación se enumeran las etapas de procesamiento de esta construcción.

1.- Determinación de alternativas abiertas, se descartan las cerradas.

2.- Si en función de los puntos a y b, la tarea puede terminar y todas las alternativas de entrada están cerradas, se selecciona la alternativa de terminación, con lo cual finaliza la ejecución del SELECT y de la tarea misma.

3.- Si existen puntos de entrada con colas no vacías, entonces se selecciona indeterminísticamente alguna alternativa de entrada, se establece la cita y se procesa (de existir) la lista de instrucciones que suceda al punto de entrada. Posteriormente termina el procesamiento del SELECT.

4.- Si todas las alternativas poseen puntos de entrada con colas vacías, entonces pueden suceder las dos siguientes situaciones:

- Si la tarea puede terminar (puntos a , b), se selecciona la alternativa de terminación y finalizan su ejecución tanto el SELECT como la tarea.
- Si la tarea no puede terminar, se queda suspendida en estado de espera hasta que alguna tarea transmisora invoque a algún punto de entrada, o la tarea pueda terminar.

6.3.1.4- Semántica de un SELECT con ELSE y alternativas de entrada.

Este es el caso más simple y se resume en los siguientes puntos:

1.- Se determinan las alternativas que estén abiertas, se descartan las cerradas.

2.- Si todas las alternativas están cerradas, o las colas de sus correspondientes puntos de entrada están vacías, se selecciona el ELSE, se procesa la lista de

instrucciones que lo componen y termina la ejecución del SELECT.

3.- Si alguna o varias de las colas de los puntos de entrada no están vacías, se selecciona indeterminísticamente una de las alternativas, se establece la cita y se procesa la lista de instrucciones que suceda al ACCEPT, después de lo cual finaliza el procesamiento del SELECT.

6.3.2- Semántica de la instrucción de selección condicional.

Como se indicó en la sección de sintaxis, la forma de una instrucción de selección condicional es la siguiente:

```
select
     llamada_punto_entrada
     lista_de_instrucciones
else
     lista_de_instrucciones
end select;
```

A diferencia de la instrucción de selección con espera, en esta construcción la tarea que contiene al SELECT juega el papel de transmisora.

En la instrucción llamada_punto_entrada se invoca un punto de entrada de alguna otra tarea, y puede ir sucedida por una lista de instrucciones.

Cuando la tarea transmisora invoca un punto de entrada de otra receptora, puede suceder cualquiera de los siguientes eventos:

1.- La tarea receptora está lista para aceptar la llamada:
   - En una instrucción de entrada del punto invocado por la tarea transmisora.
   - En una alternativa de entrada de un SELECT, que ha sido seleccionada y cuyo punto de entrada ha sido invocado por la tarea transmisora.

   En ambos casos, la tarea transmisora es la primera en la cola del punto de entrada invocado.

2.- Se cumplen los puntos anteriores, pero la tarea transmisora no encabeza la cola del punto de entrada invocado en la tarea receptora.

3.- La ejecución de la tarea receptora no ha alcanzado aún ni una alternativa de entrada, ni una instrucción de entrada (fuera del cuerpo de un SELECT) cuando la tarea transmisora llama uno de sus puntos de entrada.

4.- Se ha seleccionado una alternativa correspondiente a un punto de entrada diferente al invocado por la tarea transmisora en un SELECT de la receptora.

En la ejecución del SELECT se siguen los pasos mostrados a continuación:

1.- Si al realizar la llamada del punto de entrada de la tarea receptora, se presenta el evento del punto 1 anterior (la tarea receptora está lista), se establece inmediatamente la cita entre ambas tareas.
Posterior a la ejecución de la cita se procesa (de existir) la lista de instrucciones que sucede a la instrucción de llamada, con lo que finaliza el procesamiento del SELECT.

2.- Si al realizarse la instrucción de llamada, la tarea receptora no está lista para atenderla inmediatamente (eventos de los puntos 2, 3 y 4 anteriores), se procede a la ejecución de la lista de instrucciones correspondiente al ELSE, después de lo cual finaliza el procesamiento de la instrucción SELECT.
Si no puede establecerse inmediatamente una cita, se descarta un nuevo intento de ejecución de la instrucción de llamada, y si ésta va sucedida por una lista de instrucciones, tampoco se procesa.

3.- Si la tarea receptora ha finalizado su ejecución, o es anormal en el momento en que se realiza la instrucción de llamada, entonces se activa la excepción TASKING_ERROR en la tarea transmisora.

Observe que la tarea transmisora únicamente se suspende mientras se está realizando la cita con la tarea receptora.

6.3.3- Semántica de la instrucción de selección temporal.

La forma de este tipo de construcción de la instrucción SELECT es la siguiente:

```
select
     llamada_punto_entrada
     lista_de_instrucciones
or
     instrucción_delay
     lista_de_instrucciones
end select;
```

La ejecución de la instrucción se describe en los siguientes puntos:

1.- Se evalúa la expresión de la instrucción temporal 'delay'.

2.- Se registra la tarea en la cola de relojes del sistema y se inicia el conteo de tiempo.
3.- Se ejecuta la instrucción de llamada al punto de entrada invocado en la tarea receptora.
4.- Si la tarea receptora está lista, y puede establecer la cita con la transmisora (refiérase al evento del punto 1 de la sección anterior 6.3.2) antes de que finalice el conteo de tiempo de la instrucción delay, entonces se crea la comunicación y se realiza la cita, al final de lo cual se ejecuta (de existir) la lista de instrucciones que sucede a la instrucción de llamada. Con lo anterior finaliza la ejecución del SELECT.
5.- Si finaliza el contador de tiempo de la instrucción de temporización (delay) antes de que se pueda establecer la cita con la tarea receptora, entonces se procesa la lista de instrucciones que sucede al 'delay', y termina el procesamiento del SELECT.
6.- Si la tarea receptora ha concluido su procesamiento, o es anormal cuando la invoca la tarea transmisora, se genera la excepción TASKING_ERROR en esta última.

6.4- Conclusiones.

Las ventajas que presenta el esquema de indeterminismo del lenguaje ADA se pueden sintetizar en los siguientes puntos:
1.- En una instrucción SELECT no se pueden establecer citas en más de un punto de entrada simultáneamente, lo cual automáticamente proporciona exclusión mutua.
2.- Toda tarea únicamente puede estar registrada en la cola de un solo punto de entrada.
3.- El manejo en la construcción de selección con espera de alternativas de entrada, temporales y de terminación, le da mucha flexibilidad al programador.
4.- La construcción de selección temporal tiene aplicación directa en programación en tiempo real.

El punto 1 hace del cuerpo de un SELECT un área de exclusión mutua.
El punto 2 es consecuencia de la característica síncrona del mecanismo de comunicación por envío de mensajes, ya que una tarea transmisora únicamente puede mantener interacción con un punto de entrada de un SELECT en un momento dado.

ADA
7.-CALENDARIZACION (SCHEDULING)

El despachador (scheduler) es un programa que tiene como función ordenar, limitar en tiempo y, en general, organizar el acceso a un recurso compartido por parte de varios procesos.

Cada proceso puede tener asignada una prioridad, la que le da oportunidad de acceso antes que aquellos otros cuya prioridad sea

2.- Se registra la tarea en la cola de relojes del sistema y se inicia el conteo de tiempo.
3.- Se ejecuta la instrucción de llamada al punto de entrada invocado en la tarea receptora.
4.- Si la tarea receptora está lista, y puede establecer la cita con la transmisora (refiérase al evento del punto 1 de la sección anterior 6.3.2) antes de que finalice el conteo de tiempo de la instrucción delay, entonces se crea la comunicación y se realiza la cita, al final de lo cual se ejecuta (de existir) la lista de instrucciones que sucede a la instrucción de llamada. Con lo anterior finaliza la ejecución del SELECT.
5.- Si finaliza el contador de tiempo de la instrucción de temporización (delay) antes de que se pueda establecer la cita con la tarea receptora, entonces se procesa la lista de instrucciones que sucede al 'delay', y termina el procesamiento del SELECT.
6.- Si la tarea receptora ha concluido su procesamiento, o es anormal cuando la invoca la tarea transmisora, se genera la excepcion TASKING_ERROR en esta última.

6.4- Conclusiones.

Las ventajas que presenta el esquema de indeterminismo del lenguaje ADA se pueden sintetizar en los siguientes puntos:

1.- En una instrucción SELECT no se pueden establecer citas en más de un punto de entrada simultáneamente, lo cual automáticamente proporciona exclusión mutua.
2.- Toda tarea únicamente puede estar registrada en la cola de un solo punto de entrada.
3.- El manejo en la construcción de selección con espera de alternativas de entrada, temporales y de terminación, le da mucha flexibilidad al programador.
4.- La construcción de selección temporal tiene aplicación directa en programación en tiempo real.

El punto 1 hace del cuerpo de un SELECT un área de exclusión mutua.
El punto 2 es consecuencia de la característica síncrona del mecanismo de comunicación por envío de mensajes, ya que una tarea transmisora únicamente puede mantener interacción con un punto de entrada de un SELECT en un momento dado.

ADA
7.-CALENDARIZACION (SCHEDULING)

El despachador (scheduler) es un programa que tiene como función ordenar, limitar en tiempo y, en general, organizar el acceso a un recurso compartido por parte de varios procesos.

Cada proceso puede tener asignada una prioridad, la que le da oportunidad de acceso antes que aquellos otros cuya prioridad sea

menor.

7.2- Asignación de prioridades en forma estática.

En la sección de comunicación se indicó que la forma de asignar prioridades a las tareas en lenguaje ADA es estática, y mediante la instrucción:

PRAGMA PRIORITY <expresión>;

el valor de la expresión es el valor con el que se procesará la tarea.

Si existen tareas con la misma prioridad, no se define el algoritmo de calendarización (scheduling), y éste dependerá directamente de la implementación del lenguaje.

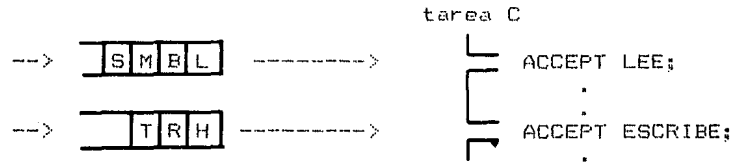
En el proceso de comunicación por envío de mensajes entre tareas, el esquema de calendarización (scheduling) está limitado, y se puede describir en los siguientes puntos:

1.- El orden en que se registran las tareas en la cola de un punto de entrada es según el orden cronológico de sus invocaciones.
2.- Entre dos puntos de entrada de una instrucción SELECT, se selecciona a aquél cuya tarea transmisora (registrada) sea de mayor prioridad.
3.- Una vez establecida la cita, ésta se procesará con una prioridad que estará en función de las prioridades de las tareas involucradas.

Con respecto al primer punto, y como ya se ha mostrado, no existe forma de que se pueda alterar ese orden.

Como el lector puede observar, este esquema (que es invisible en su implementación al usuario) es poco flexible, pues si la tarea receptora controla a algún recurso compartido, en forma implícita no es posible alterar el orden en el acceso al mismo.

Respecto al segundo punto, si se presenta la situación descrita en la siguiente figura:



Si la tarea H es de mayor prioridad que L, entonces se establece la cita entre la tarea receptora C y la transmisora H, antes que con la tarea L, como se explicó en la semántica del mecanismo de comunicación.

Sin embargo, si ambas tareas L y H son de la misma prioridad, o no tienen asignada prioridad explícitamente, el orden en que se atienden es totalmente indeterminístico.

Si no se asigna prioridades a las tareas en forma explícita, mediante la instrucción PRAGMA PRIORITY, ADA les asigna implícitamente una misma prioridad cuyo valor depende de la instalación.

Con respecto al tercer punto, la ejecución del conjunto de instrucciones que componen el cuerpo de la instrucción de entrada (accept) durante la cita, es la única sección de código en que la prioridad de procesamiento puede modificarse a tiempo de ejecución, y de acuerdo con las siguientes reglas:

a.- Si tanto la tarea transmisora como la receptora tienen asignada prioridad explícitamente, el cuerpo de instrucciones se procesará con la prioridad que sea mayor.

b.- Si sólo una de ambas tareas tiene asignada prioridad explícitamente, entonces el código se procesará con esa prioridad.

c.- Si ninguna de las dos tareas posee prioridad asignada en forma explícita, entonces no es determinable la prioridad con que se ejecutará el cuerpo de instrucciones de la instrucción de aceptación (accept).

El uso de prioridades en forma estática y explícita por parte del programador debe limitarse a la importancia o urgencia de procesamiento de las tareas, y no darle uso de control de sincronización.

Como el lector ha podido analizar, el manejo de prioridades y calendarización en ADA es muy restringido, ya que no permite que el programador establezca algoritmos de scheduling en el acceso a recursos compartidos.

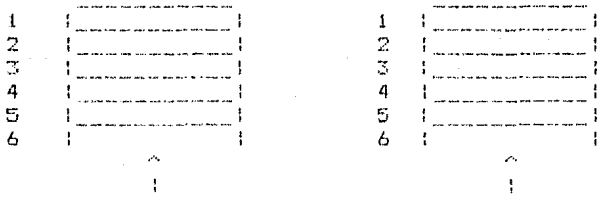
Sin embargo, es posible darle la vuelta a estas restricciones, a costa de utilización de estructuras, algoritmos y haciendo uso de una familia de tareas.

7.3- Esquema de calendarización (scheduling) en forma explícita.

Lo que se desea es que en el acceso a algún recurso, a través de alguna tarea que lo controle, exista un esquema de calendarización en el que las llamadas (invocaciones) a esa tarea se realicen en función de prioridades dinámicas y no del orden de solicitud cronológico.

Supongamos que se desea calendarizar el acceso al recurso, a través de una prioridad (no confundir con la prioridad estática declarada explícitamente en la instrucción PRAGMA) que proporciona la propia tarea transmisora.

Para obtener la calendarización, o scheduling, es necesario tener unas estructuras de datos sobre las cuales calcular la tarea de más alta prioridad, y se proponen las siguientes:

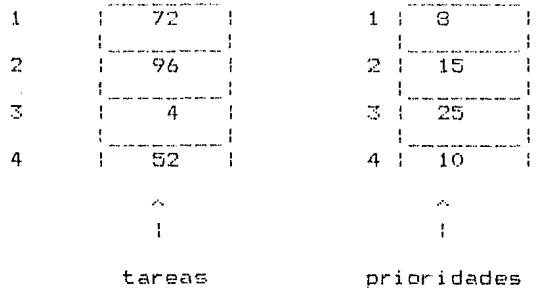


cola de tareas cola de prioridades

La tarea proporciona su propia prioridad de acceso al recurso, la cual se registra en la cola de prioridades y se van registrando en la cola según el orden en que solicitan acceso.

Además de este dato es necesario identificar la tarea, para lo cual se le asigna internamente un número de indentificación, que se registra en la cola de tareas y en la misma posición donde se registró su prioridad en la cola de prioridades.

Por ejemplo, en la siguiente figura se identificó a la tarea J con el número 52, y con prioridad 10:



tareas prioridades

Ambos datos están en la localidad 4 de ambos arreglos.
Posteriormente, el algoritmo de calendarización observa la cola de prioridades y detecta al elemento con mayor prioridad (el 3); con ese dato lee de la cola de tareas al número identificador de la tarea candidata a accesar el recurso.

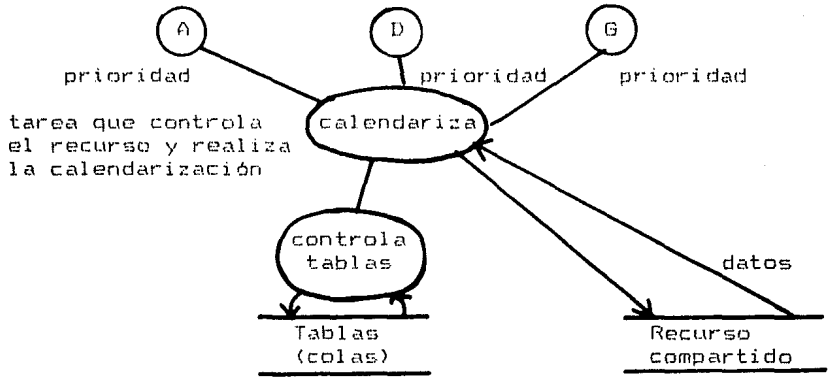
El acceso a las tablas (colas) debe ser sincronizado, ya que son áreas de exclusión mutua.

Las tareas transmisoras que desean accesar el recurso deben solicitar en el siguiente orden:

1.- Se les asigne un número identificador con objeto de que el despachador los pueda reconocer y manipular.
2.- Ya identificados, solicitar su registro en las tablas de identificación y de prioridades.
3.- Finalmente invocar el acceso al recurso.

Para evitar que cada tarea transmisora elabore dos o tres llamadas, éstas se agrupan en un procedimiento, de manera que la tarea transmisora únicamente realiza una llamada.

En la siguiente figura se muestra la arquitectura del esquema planteado:



La tarea controla_tablas registra, y según el algoritmo de scheduling, obtiene la identificación de la tarea candidata. La tarea calendariza asigna un número de identificación a cada tarea transmisora, y le asigna el recurso compartido a la de mayor prioridad . En el mismo procedimiento se invoca nuevamente a calendariza, que es una familia de tareas cuyo índice representa la identificación asignada por ella misma a las tareas solicitantes.

A continuación se representan las tareas.

Ejemplo basado en [Ichbiah, 1979].

```
Procedure accesa_recurso(pri: in integer; d: in out dato) is
     begin
     calendariza.dame_identif(I,pri);
     -- se le asigna identificación 'I'
     -- se registra su prioridad en tabla
     calendariza.dame_recurso(I)(d);
     -- espera a que se le asigne el recurso, según su
     --  prioridad 'pri'
end accesa_recurso;
```

Las tareas A, D y G únicamente hacen la invocación:
"accesa_recurso(prioridad,dato)"

```
task calendariza is
     entry dame_identif(pri:in integer; ID: out integer);
     entry dame_recurso(1..1000)(d: in out dato);
end calendariza;
```

```
task body calendariza is
     G : boolean;
     KD, I : 1..1000;
begin
     I := 0;
     loop
          select
            accept dame_identif(pri:in integer;
                                ID : out integer) do
               ID := I
            end dame_identif;
            controla_tablas.registra(I,pri);
            I := I + 1;
            -- la identificación es un número que se
               -- incrementa
          else
               null;
        end select;
     controla_tablas.vacia(G);
     If G then
          controla_tablas.siguiente(KD);
          accept dame_recurso(KD)(d:in out dato) do
               -- procesa dato en recurso (accesa recurso)
               -- prporcionalo  a la tarea solicitante  cuya
               -- identificación está dada por 'KD'
          end dame_recurso;
     end if;
end loop;
end calendariza;
```

'controla_tablas.vacia' indica si las tablas no están vacias.

Finalmente, la tarea controla_tablas tiene el siguiente cuerpo:

```
task body controla_tablas is
     k : integer := 0;
     total : integer := 0;
     -- K es un índice de las tablas
     -- total indica el número de tareas registradas
begin
     loop
          select
               accept registra (I: in 1..1000;
                                pri: in integer) do
                    cola_tareas(K) := I;
                    cola_prioridades(K) := pri;
               end registra;
               k := K + 1; total := total + 1;
          or
               accept vacia(estado: out boolean) do
                    If total = 0 then estado := false;
                    else
```

```
                        estado := true;
                    end if;
                end vacia;
            or
                accept siguiente (ID:out 1..1000)
                    -- calcula    la    prioridad    máxima
                    -- registrada en la cola de prioridades,
                    --   leé el índice correspondiente y con
                    --   él  lee  la cola  de  tareas,  para
                    --   conocer el número de identificación
                    --   de     la     tarea     candidata.
                    --   Posteriormente actualiza y compacta
                    --   colas.
                end siguiente;
            end select;
        end loop;
    end controla_tablas;
```

7.4- Conclusiones.

El lector ya se ha podido percatar de la poca flexibilidad que ADA proporciona respecto al manejo de scheduling para el acceso a recursos.

El ejemplo muestra el grado de complejidad que se debe invertir para establecer una política de asignación de un recurso a un conjunto de procesos, ya que debe manipularse una familia de tareas.

Para la aplicación de otras políticas de asignación se modifican los algoritmos y las estructuras, sin embargo, el grado de laboriosidad en su programación sigue siendo elevado.

COMPARACION DE LOS MECANISMOS DE CONTROL DE CONCURRENCIA DE LOS LENGUAJES: CSP, SR Y ADA

4.- Comparación de mecanismos de control de concurrencia

En el presente capítulo se hará una comparación de los mecanismos de CSP, SR y ADA, los cuales fueron analizados en detalle en los tres capítulos anteriores.

La estructura de este análisis comparativo se fundamentará en la que se ha venido utilizando como modelo en el estudio de los capítulos anteriores, y que consiste en las siguientes secciones:

1.- Procesos.
2.- Paralelismo.
3.- Comunicación.
4.- Sincronización.
5.- Indeterminismo.
6.- Scheduling.

La razón de esta estructura es mantener la uniformidad con lo analizado en los capítulos anteriores.

A continuación se inicia la comparación entre los tres lenguajes, empezando con el estudio comparativo de la estructura de sus programas y procesos.

1.- PROGRAMAS Y PROCESOS

Esta sección se subdivide en los siguientes puntos:

1.- Adaptabilidad a diferentes tipos de arquitecturas.
2.- Estructura de programas.
3.- Interfaz con el medio ambiente, control de acceso, abstracción y parametrización.
4.- Familias de procesos.
5.- Creación y activación de procesos.
6.- Estados y terminación de procesos.

Es necesario hacer la observación de que CSP está considerado más como un modelo que como un lenguaje.
Por lo anterior, Hoare no describió características que competen exclusivamente al diseño e implementación de un lenguaje, razón por la cual habrá algunos puntos en los que 'CSP' tendrá poca influencia en el análisis comparativo.

1.1- Adaptabilidad a diferentes tipos de arquitecturas.

Del análisis realizado a los tres modelos se puede inferir que el enfoque de su conceptualización es hacia arquitecturas distribuidas (redes de procesadores), aunque los tres se pueden aplicar a arquitecturas convencionales de multiprocesamiento con memoria compartida, o bien, uniprocesamiento (una sola unidad de procesamiento central) con multiprogramación.

Debido a que en SR un conjunto de procesos puede compartir las variables permanentes del 'Recurso' al que integran, existe la restricción de que en una arquitectura de tipo red, cada Recurso del programa esté residente en un solo nodo de la red, y no se distribuya en varios nodos.

ADA también permite que se compartan variables entre procesos, por lo que si se desea que las tareas se comuniquen mediante este esquema de interacción, es recomendable que todas las unidades de programas y procesos que las compartan, se instalen en un único nodo de la red, de tal manera que la comunicación entre procesos de distintos nodos sea exclusivamente por envío de mensajes.

Por lo anterior, se concluye que tanto CSP que no comparte variables, como SR y ADA que poseen esquemas de interacción que manejan variables compartidas y envío de mensajes, se pueden adaptar a cualquier tipo de arquitectura, a diferencia de aquellos lenguajes en los que la interacción entre los procesos es únicamente a través de recursos compartidos, tales como variables, semáforos y monitores, ya que su instalación en una red sería poco factible, pues en algún nodo se instalaría al recurso compartido, y su acceso por parte de los demás crearía un cuello de botella.

En lo referente a su aplicabilidad, tanto CSP como SR se conceptualizaron para aplicarse a la instrumentación de sistemas operativos (centralizados y distribuidos). ADA también puede usarse para ese fín, sin embargo, varios de sus mecanismos de control están diseñados para aplicarse directamente a programación de eventos sincronizados con sucesos, que están en función de procesos que suceden en el mundo real (tiempo real).

1.2- Estructuras de los programas.

En CSP, SR y ADA se conceptualiza de la misma forma a la entidad proceso, es decir, como un conjunto de instrucciones que se procesan como una sola entidad con vida propia e independiente, y en paralelo con las demás. En ADA se les llama tareas en lugar de procesos.

Sin embargo, la estructura de un programa es muy diferente en los tres casos.

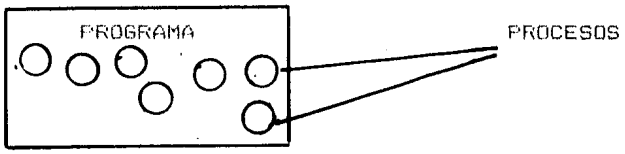
En CSP un programa está constituido exclusivamente por una única clase de entidad, que es el proceso. En realidad, un programa se puede considerar como integrado por un conjunto de procesos que se están *ejecutando* concurrentemente.

Sintácticamente, un programa en CSP puede ser lo siguiente:

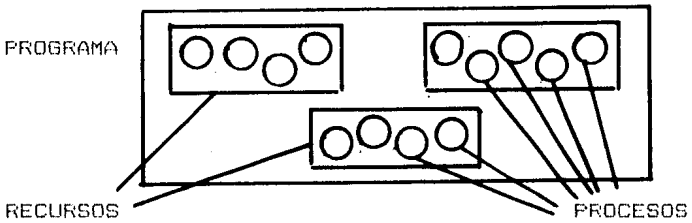
1.- El código de un proceso.
2.- Un comando paralelo.

Un proceso puede tener anidados otros procesos o comandos paralelos. Su estructura se puede mostrar gráficamente en la

siguiente figura:



En el lenguaje SR un programa está constituido por Recursos, que a su vez están integrados por procesos.



Un programa puede tener varios Recursos, y un Recurso uno o más procesos.
En este lenguaje existen por lo tanto dos clases de entidades que componen la estructura de un programa: Recursos y procesos. Sin embargo, un Recurso no puede tener a otro Recurso como parte de su cuerpo, ni un proceso a otro proceso o Recurso. La estructura es jerárquica, y no permite el anidamiento de sus entidades.

Por último, ADA maneja cuatro clases de entidades:

1.- Subprogramas (funciones y procedimientos).
2.- Paquetes.
3.- Unidades genéricas.
4.- Procesos (tareas).

Un programa puede estar formado por los cuatro tipos de entidades. Además, cada una de estas entidades puede tener como parte de su cuerpo a cualquiera de las otras, a excepción de los procesos, los cuales no pueden contener paquetes en su declaración.

No existe límite en el grado de anidamiento, pues un paquete puede estar formado por procesos (tareas), los que a su vez poseén subprogramas, y éstos a su vez otros procesos, subprogramas, etc.

Esta estructura no es jerárquica; sin embargo, se puede concluir lo siguiente:

1.- La entidad Recurso de SR es equivalente al paquete de ADA, ya que ambos permiten agrupar objetos relacionados lógicamente (variables compartidas, etc.), cuyo acceso es a través de los procesos que integran la entidad.

2.- En SR, un proceso se ve invalidado para declarar otros procesos, lo cual impide que un proceso se pueda crear

a consecuencia de otro (dependa del otro); por otra parte, ni CSP ni SR manejan subprogramas. Esto trae como consecuencia el que no puedan manejar recursión, a diferencia de ADA que posee subprogramas recursivos. Para que tanto CSP como SR sean recursivos se necesitaría:

- Que un proceso pueda declarar (crear) a otro con el mismo nombre, lo que no se permite en CSP, ni en SR o
- Manejar funciones o procedimientos (subprogramas) recursivos, que es el caso de ADA.

3.- Tanto SR como ADA utilizan variables compartidas como medio de interacción entre entidades, sin embargo, en ADA pueden existir varias trayectorias de acceso a las mismas por el alto grado de anidamiento de sus entidades, lo cual tiene las siguientes consecuencias:

- Mayor complejidad en su visibilidad.
- Mayor complejidad de su acceso, cuando éste es concurrente.

Mientras que en SR, la trayectoria es de dos niveles: Recurso- proceso.

En la siguiente tabla se resume lo analizado en estos puntos:

	ADAPTABILIDAD ARQUITECTURAS	APLICABILIDAD	ESTRUCTURA PROGRAMA		RECURSION	ACCESO VARIABLES COMPARTID.
			TIPO	NUM. ENTID.		
CSP	*	sistemas operativos	con anid.	1	NO	NO
SR	*	sistemas operativos	jerárquica sin anid.	2	NO	SI sencillo y estructurado
ADA	*	sist. op. prog.tiempo real	con anid.	4	SI	SI complejo

* : Distribuidas, memoria compartida o uniprocesamiento con multiprogramación.

RELACION ENTIDADES:

	SR	CSP	ADA
proceso:	proceso	proceso	tarea
paquete:	..	Recurso	paquete
subpro-gramas:	..	..	funciones procedimientos

1.3- Interfaz con el medio ambiente, control de acceso, abstracción y parametrización.

Tanto SR como ADA presentan un área de especificación de objetos (constantes, variables, tipos, operaciones en SR, puntos de acceso en ADA, etc.), los cuales son visibles al medio ambiente, es decir, a otros 'Recursos' en SR, o a otras unidades de programas en ADA, para su respectivo acceso y/o manipulación. El resto del cuerpo del Recurso o el paquete (o unidad genérica), se mantiene oculto al medio exterior.

Esa área de especificaciones constituye la interfaz con el medio ambiente.

CSP carece de interfaz, pues todo el proceso es visible al medio ambiente y no existe una división explícita entre el cuerpo (oculto) y las especificaciones o declaraciones.

En SR existen restricciones de acceso (control de acceso) a los objetos que resguarda un Recurso, a través de las declaraciones de exportación.

Para que un Recurso externo pueda accesar a un objeto, éste debe estar declarado en la lista de Recursos a los que se exporta, y el objeto en la correspondiente lista de objetos; igual sucede al importar un objeto. Un objeto puede ser la instrucción de entrada u operación de un proceso que integre al Recurso.

En ADA cualquier unidad de programa del medio ambiente puede accesar a los objetos especificados en la interfaz de un paquete, usando la instrucción USE. No existe por lo tanto descriminación en el acceso, y por lo tanto, el control de acceso es más pobre que el de SR.

La abstracción es la capacidad que tiene un lenguaje de poder agrupar y ocultar un conjunto de entidades (datos, tipos, etc.) en una sola entidad, a la cual se le asigna un nombre y que suministra operaciones para que otros procesos externos puedan manipular las entidades agrupadas. Tanto SR como ADA permiten abstraer no únicamente datos, sino también tipos (estructuras). Ya que tanto un Recurso como un paquete pueden ocultar un objeto y/o su tipo (exportable), ofreciendo al medio ambiente externo un conjunto de operaciones o procedimientos con los cuales

manipularlo.

En ADA un tipo de dato puede abstraerse (ocultarse) totalmente, o bien, hacerlo visible para que el medio externo pueda accesar los elementos componentes de su estructura interna en un dato declarado con él.

Sin embargo, en SR existe otra modalidad, en la cual se permite leer el contenido de los componentes internos de un dato declarado con un tipo (exportado) estructurado, pero no se permite modificarlos más que por las operaciones que el propio Recurso exportador ofrezca, es decir, existe una abstracción parcial del tipo, lo cual puede ser de gran ayuda.

Lo anterior ocasiona que SR tenga mejor control en la abstracción de tipos que ADA, ya que en un tipo abstracto de ADA no se puede leer en forma directa la estructura del tipo (los componentes de algún dato declarado con ese tipo) como sucede con SR, sino a través del conjunto de operaciones que se proporcionan para el manejo de las entidades.

La característica de parametrización es exclusiva de ADA, pues ni SR ni CSP la tienen. Esta propiedad contribuye al ahorro de código en la programación de las entidades que integran un programa, pues independiza a los algoritmos de los tipos de los datos.

En el siguiente cuadro se resumen las características analizadas en este punto:

	INTERFAZ	CONTROL DE ACCESO	ABSTRACCION		PARAMETRIZACION
			DATOS	TIPOS	
CSP	NO	No posee	NO	NO	NO
SR	SI	permite discriminar Recursos en el acceso a objetos	SI	total y parcial	NO
ADA	SI	NO descrimina	SI	sólo total	SI

En general, no se puede decir que SR sea mejor lenguaje que ADA o viceversa; uno aventaja al otro en unas características y el otro al primero en otras. CSP resultó ser pobre en las propiedades bajo estudio de este punto.

1.4- Familias de procesos.

Tanto en CSP y SR como en ADA se permite la declaración y creación de familias de procesos.

Una familia puede ser un vector o una matriz n-dimensional de procesos. Cada elemento de la familia es un proceso independiente que ocupa espacio en memoria y utiliza al procesador (CPU), constituido por el mismo cuerpo de instrucciones que los demás de su familia, pero cuyas variables indexadas (por el índice de la familia) toman valores independientes a los del resto de los demás procesos de la familia.

En ADA se indexa a las declaraciones de los puntos de entrada de la familia de tareas. La familia se declara como un arreglo:

A: array (1..50,1..100.1..100) of <tipo tarea>

En SR se permite tener no únicamente familias de procesos, sino de Recursos. Cada Recurso de la familia posee la misma estructura que los demás.

1.5- Creación y activación de procesos.

De lo descrito en los capítulos anteriores, la creación de procesos se puede clasificar en:

1.- Estática.
2.- Dinámica.

En la primera categoría se requiere que todos los procesos estén ya definidos al iniciarse la ejecución de un programa, mientras que en la segunda los procesos se pueden crear durante el procesamiento del programa, y en función de sus propios algoritmos.

La creación de procesos en CSP y SR pertenecen a la primera categoría. En CSP los procesos ya están definidos previamente a la ejecución del comando paralelo en el que se activan.

En SR los procesos son creados en la etapa de inicialización del programa, por los Recursos a los que pertenecen, después de lo cual son activados. No pueden crearse Recursos ni procesos fuera de esta etapa de inicialización, y únicamente los procesos especificados en el texto del programa son creados y activados.

En ADA coexisten las dos categorías: en la primera los procesos se van creando y activando a medida que se van procesando las áreas de especificaciones de las unidades de programas en que se han declarado, mientras que en la segunda los procesos se crean a tiempo de ejecución de ciertas instrucciones, y según la propia lógica del programa.

La creación dinámica de procesos en ADA se puede subclasificar a su vez en otras dos categorías:

1.- Dinámica.
2.- Lexicográfica.

La primera coincide con la definición de creación dinámica expuesta al inicio de este punto, y la segunda se refiere a la creación de procesos como consecuencia de la ejecución recursiva

de los subprogramas en los que se definen o crean.

ADA maneja la creación dinámica de tareas a través del uso de datos de tipo acceso, y de la instrucción NEW. También se crean dinámicamente tareas en la categoría lexicográfica, pues ADA maneja procedimientos y funciones recursivas, las que por la característica estructural de anidamiento de unidades de programas, pueden crear (NEW) o tener declaradas tareas en su cuerpo. Se crea una copia de la tarea por cada nivel de recursividad ejecutado.

En relación a lo anterior, existen muchas situaciones del mundo real a programar en que se maneja un número no determinado de objetos iguales, como por ejemplo:

- Los aviones en vuelo en una región y aeropuertos definidos.
- Los misiles en trayectoria de vuelo, en la simulación de un ataque.
- Los vehículos en una determinada zona, para control de tráfico.

En los tres ejemplos no es determinable el número de aviones, misiles o vehículos en movimiento o en vuelo, aunque se pueden establecer cotas o límites.

Si estos ejemplos se programaran con familias de procesos, asignando cada proceso a un avión o misil en vuelo, a un vehículo en movimiento, sucederá que la mayor parte de las veces existan procesos ociosos, no asignados, pero que están consumiendo recursos del sistema de cómputo.

Sin embargo, el uso de procesos de creación dinámica creará nuevos procesos en función de las necesidades del sistema que controlan o simulan (un avión, misil o vehículo que ingrese a la zona de control), y que terminará su ejecución cuando así también se requiera (el aterrizaje de un avión, la salida de un vehículo de la zona de control, etc.), que para el caso de la familia de procesos, aún que no se requiera su existencia, siguen estando presentes, y consumiendo recursos de memoria y procesador.

Por lo que la creación dinámica de recursos permite optimizar el uso de recursos de equipo de cómputo (memoria, CPU, etc.).

1.6- Estados y terminación de procesos.

Los tres únicos estados en que se encuentra un proceso en CSP, SR y ADA son:

1.- Activo.
2.- Suspendido (bloqueado).
3.- Terminado.

En el primer caso están todos los procesos que se están procesando, y aquellos que están compitiendo por uso del CPU en las colas de listos (ready).

En el segundo caso están los procesos que esperan la comunicación con alguno otro que en ese momento no puede atenderlos, o los que están a la espera de ser invocados, y aquellos que han sido bloqueados por otras causas.

Todos los procesos terminan normalmente cuando terminan de ejecutar la lista de instrucciones que integra su cuerpo, después de lo cual dejan de existir, y por lo tanto, de consumir recursos del equipo de cómputo, lo que constituye el tercer estado del proceso.

Sin embargo, a causa de la falla en la comunicación con otro proceso o alguna otra causa, un proceso puede terminar su ejecución en forma anormal, además de dejar inconclusa la ejecución total de su lista de instrucciones.

En CSP y en SR no existen mecanismos de protección que el mismo programador pueda usar para tomar una determinada acción en el caso de que se presente una situación anormal (excepción). En SR existe un mecanismo de protección únicamente para situaciones de falla en la comunicación entre procesos.

En ADA se tiene la facilidad de tener instrucciones a las cuales se dirige el proceso de ejecución de una tarea cuando se presenta alguna anormalidad, a las que se llama manejadores de excepciones.

Por otra parte, en ADA cualquier tarea puede saber si otra ha terminado su ejecución, haciendo uso del atributo tarea_a_probar'TERMINATED, e incluso, puede detectar a través del atributo tarea_a_probar'CALLABLE si la tarea a probar no está anormal (no ha sido abortada).

En CSP y SR no existe forma de probar el estado de otros procesos, con objeto de evitar interaccionar con ellos, además de que no existe la posibilidad de detectar si un proceso ha terminado anormalmente.

Un Recurso en SR finaliza su ejecución normal cuando todos sus procesos constituyentes también han terminado; sin embargo, no puede detectar si alguno lo hizo en forma anormal.

Lo anterior limita considerablemente a CSP y SR para aplicaciones de programación en tiempo real, en las que una eventualidad del medio físico externo se puede propagar a los procesos, los que en consecuencia pueden terminar anormalmente, sin permitirles el tomar acciones inmediatas y correctivas.

En el siguiente cuadro se resumen las características comparativas analizadas en los últimos puntos:

	Manejan familias de procesos	Creación Activación	Manejo de situaciones anormales
CSP	SI	estática	NO
SR	SI	estática	NO *
ADA	SI	dinámica y lexicográfica	SI

* : únicamente las relacionadas con fallas en la comunicación.

2.- PARALELISMO

El análisis comparativo a efectuar en CSP, SR y ADA respecto a su característica de paralelismo será en función de los siguientes puntos:

1.- Tipo de paralelismo.
2.- Dependencia entre procesos.
3.- Activación concurrente de procesos.
4.- Terminación concurrente de procesos.

2.1- Tipo de paralelismo.

El paralelismo se puede clasificar en función de su generación y sintaxis, en las dos siguientes categorías:

1.- Explícito.
2.- Implícito.

En la primera categoría se indica textualmente a través de una instrucción los nombres de los procesos que se ejecutarán en concurrencia. Al empezarse a ejecutar la instrucción se activan los procesos, y sólo hasta que éstos hayan terminado finalizará la ejecución de la instrucción.

En la segunda categoría los procesos se van activando en forma implícita, a medida que el programa ejecuta sus declaraciones o instrucciones de creación, sin que exista una instrucción explícita de ejecución concurrente que los nombre.

CSP está en la primera categoría, ADA se ubica en la segunda, y SR se puede considerar que abarca las dos, como se mostrará después.

El paralelismo explícito permite combinar fácilmente procesamiento secuencial y concurrente, aparte de que el programador tiene un mejor control de la concurrencia, como se muestra en el si-

guiente programa escrito en CSP:

```
programa::
     [A:: x, y: integer;
          y := 100; x := 1;
          *[ x <= y -> x := x + 1]
          [ P1 || P2 || P3]          --> comando paralelo
          [ y > x ->  ...
          []
             x <= y  -> ...
          ]
     ]
```

El procesamiento es secuencial antes y después de la ejecución del comando paralelo.

En una estructura de programa con tanto anidamiento como en ADA, puede resultar complejo visualizar la concurrencia.

En SR y ADA el paralelismo es implícito, ya que se realiza sin que exista de por medio una instrucción específica de paralelismo, como el comando paralelo de CSP.

Sin embargo, en SR se puede tener como contenido de una operación (instrucción de entrada) OP1 todo el código del proceso P1 del ejemplo anterior, y de igual forma OP2 y OP3 pueden ser otras operaciones en otros procesos, que posean en su área de exclusión mutua el contenido de los procesos P2 y P3, respectivamente.

Finalmente, se hace uso de la instrucción concurrente de invocación de SR:

```
CO call OP1 ; call OP2 ; call OP3 OC
```

lo cual es semánticamente equivalente al comando paralelo de CSP.

En conclusión, mediante la instrucción concurrente de invocación de SR se puede generar paralelismo explícito y, por lo tanto, SR a diferencia de CSP y ADA tiene la característica muy útil de poder manipular ambos tipos de paralelismo.

2.2- Dependencia entre procesos.

En este punto se hace referencia al término dependencia como la relación entre procesos, en función de su activación y terminación (no relacionarla con comunicación).

En función del análisis de los capítulos anteriores, se concluyen los siguientes puntos:

1.- La dependencia entre procesos concurrentes es similar en CSP y ADA, siendo ésta de tipo jerárquico.
2.- En SR no existe dependencia entre procesos.

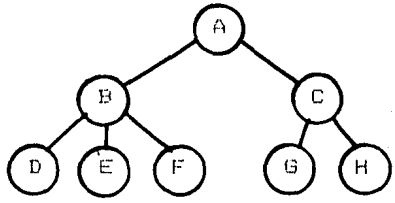
En CSP se considera que el texto o código comprendido entre los símbolos ||, [y ||, o || y] de un comando paralelo, es un proceso en todo el sentido de la palabra.

Además, ese proceso puede tener uno o varios comandos paralelos como parte de su lista de instrucciones.

Los procesos que constituyen un comando paralelo se activan como consecuencia de la ejecución de esa instrucción, en el contexto del proceso al cual pertenece. Por lo que la activación de los procesos depende del proceso al cual pertenece el comando paralelo.

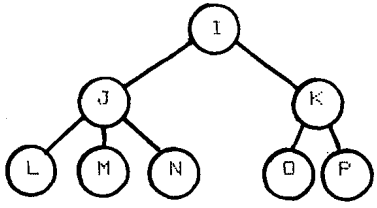
En ADA los procesos dependen de la unidad de programa en donde se declaren o se creen (NEW), como ya se estudió.

Existe por lo tanto una semejanza considerable entre CSP y ADA en lo referente a la dependencia de sus entidades. Por ejemplo, si en CSP el proceso A posee al comando paralelo [B || C], B posee a [D || E || F], y C posee a su vez [G || H], entonces la relación de dependencia se puede representar gráficamente así:



En CSP D, E y F dependen de B; G y H de C; B y C de A. La representación gráfica de la dependencia es un árbol.

En ADA sucede algo semejante. Si la unidad (de programa) I tiene declaradas las unidades J y K, en la que J crea a L, M, N; y K declara y/o crea a O y P, se tiene la siguiente estructura jerárquica de dependencia:

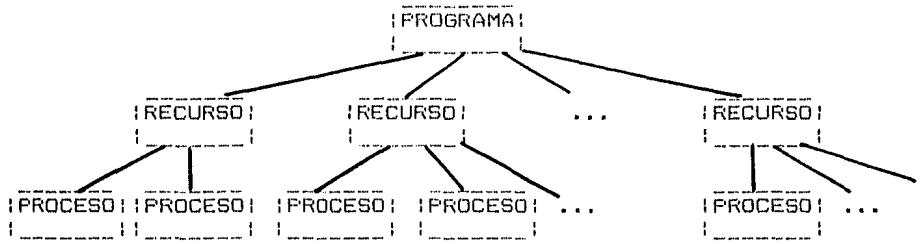


El maestro inmediato de L, M y N es J; O y P dependen directamente de K, e indirectamente de I. Observe la similitud en ambos casos (CSP y ADA).

En SR un proceso no puede ser parte de la declaración de otro, ni se pueden crear dinámicamente, como ya se describió en la sección anterior.

No existe por lo tanto dependencia entre procesos; la única

dependencia es entre los procesos y el Recurso del cual forman parte. Un Recurso depende a su vez del programa principal, como se muestra a continuación:



Los procesos se crean como consecuencia de la ejecución del Recurso al que constituyen, y la terminación de éste depende de la terminación de sus procesos constituyentes.

2.3- Activación y terminación concurrente de procesos.

La activación de procesos está en función de su dependencia. Sin embargo, haciendo referencia a la estructura jerárquica de dependencias de CSP y ADA se concluye lo siguiente:

1.- En CSP la concurrencia únicamente existe en los procesos que constituyen las hojas del árbol.
2.- En ADA la concurrencia es en todos los nodos del árbol.
3.- En SR la concurrencia existe a nivel de proceso y de Recurso.

Respecto al primer punto, recuérdese que el comando paralelo es una instrucción de una lista secuencial de instrucciones que constituyen un proceso.

Mientras el comando paralelo se esté ejecutando, el proceso del cual forma parte está en espera de su terminación, para después continuar con la ejecución de sus demás instrucciones, y se puede considerar que el proceso principal se bloquea.

Respecto al segundo punto, si una tarea en ADA crea en forma estática o dinámica a otras tareas, entonces se ejecuta concurrentemente con ellas, de tal manera que en un momento determinado un maestro se está procesando en paralelo con todas sus tareas dependientes, y si alguna de éstas es otro maestro, entonces todas sus dependientes se estarán procesando también en paralelo.

En SR todos los procesos de un Recurso se ejecutan en paralelo, y todos los Recursos de un programa también se procesan en concurrencia. Consecuentemente, todos los procesos de todos los Recursos que constituyen un programa también se ejecutan concurrentemente.

En CSP y ADA la implantación de la concurrencia es de la raíz del árbol hacia sus hojas, a medida que se van ejecutando los progra-

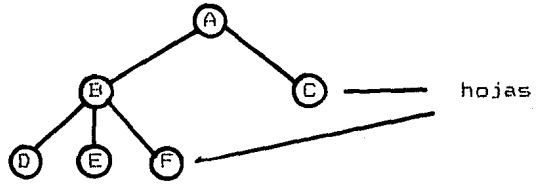
mas.

En SR primero se crean los Recursos, y posteriormente los procesos, después de lo cual se activan y se establece la ejecución de la concurrencia.

Las características de terminación de los procesos concurrentes se asientan en los siguientes puntos:

1.- En CSP y ADA la terminación es en el sentido de las hojas hacia la raíz de su árbol de dependencia.
2.- En SR la terminación en la ejecución de un Recurso está en función de la terminación de todos sus procesos constituyentes.

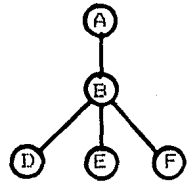
Tomando nuevamente como ejemplo el árbol de dependencia de CSP, cuando finaliza la ejecución del comando paralelo: [G || H], C (que había estado bloqueado) puede continuar su procesamiento, además de que se convierte en hoja del árbol (G y H ya terminaron) y, por lo tanto, su ejecución es concurrente con los procesos D, E y F:



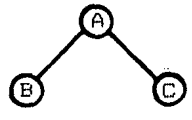
De ahí pueden suceder cualquiera de los siguientes eventos:

- Termina C.
- Terminan D, E y F.

En el primer caso queda el árbol:



En el segundo caso, B se convierte en hoja y su procesamiento continúa concurrentemente con C:



En ambos casos, la terminación empieza por las hojas y termina en

la raíz.

En ADA sucede algo similar, pues debe recordarse que la condición para que una unidad o tarea maestra pueda terminar, es que sus dependientes ya hayan terminado o deseen terminar (alternativa TERMINATE de un SELECT).

Así que haciendo referencia al árbol que sirvió como ejemplo en la subsección de dependencia, para que el maestro K pueda terminar, lo deben de hacer sus dependientes O y P; igual sucede con J. La terminación va de las hojas hacia la raíz.

En SR un Recurso no podrá finalizar su ejecución hasta que todos sus procesos constituyentes hayan terminado. Un programa en SR no culminará su procesamineto hasta que todos sus Recursos constituyentes hayan finalizado su ejecución.

En la siguiente tabla se resumen todas las características de paralelismo de CSP, SR y ADA:

PARALELISMO

	Tipo sintáctico	Dependencia entre procesos	Activación concurrencia	Terminación concurrencia
CSP	explícito	jerárquica (árbol)	De la raíz a las hojas, sólo existe en las hojas	De las hojas a la raíz
SR	implícito y explícito	no tiene	Programa a Recursos y Recursos a procesos	Procesos a Recursos y Recursos a programas
ADA	implícito	jerárquica (árbol)	De la raíz a las hojas, existe en todos los nodos	De las hojas a la raíz

3.-COMUNICACION

En esta sección se hará un análisis comparativo de CSP, SR y ADA, enfocado a los siguientes puntos:

1.- Esquema de interacción entre procesos.
2.- Clasificación del mecanismo de comunicación en función de su característica de sincronización.
3.- Simetría con respecto a la invocación.
4.- Simetría con respecto a la suspensión.
5.- Almacenamiento de mensajes.

6.- Modos de transferencia de mensajes.
7.- Características de asociación de mensajes.
8.- Control de la comunicación.
9.- Control de fallas en la comunicación.

3.1- Esquema de interacción entre procesos.

Del análisis de los capítulos anteriores, se determinó la existencia de los siguientes esquemas de interacción entre procesos:

1.- Esquema de variables compartidas.
2.- Esquema de participación equitativa.
3.- Esquema cliente/servidor.
4.- Esquema de interconexión.

Las características del esquema de participacvión equitativa son:

a.- Los dos procesos participan activamente en la comunicación.
b.- Ambos deben referenciarse y sincronizarse mutuamente.
c.- Ninguno es subordinado del otro.

Del punto 'b' se observa que la comunicación es síncrona.

CSP utiliza en su mecánica de comunicación, única y exclusivamente el esquema de participación equitativa, ya que ambos procesos involucrados en la comunicación deben invocarse mutuamente, sincronizarse y participar activamente (ninguno se subordina).

Los esquemas de interacción que utiliza SR son:
1.- Variables compartidas.
2.- Cliente/servidor.
3.- Interconexión.

Y los que tiene implementado ADA son:
1.- Variables compartidas.
2.- Cliente/servidor.

El más rico en su variedad de esquemas de interacción o comunicación entre procesos es SR, y el más pobre es CSP.

SR proporciona al programador tres distintos esquemas, a aplicarse también a tres diferentes necesidades. No a todos los sistemas o eventos a programar se les va a restringir a un esquema cliente/servidor, o de participación equitativa, cuando por las propias características del evento sea más adecuado utilizar otro esquema.

Proporcionar al programador más de donde elegir le da mucha mayor flexibilidad, lo que como consecuencia permitirá optimizar código y recursos de cómputo.

SR aventaja a ADA y a CSP en lo concerniente a este punto.

3.2- Clasificación del mecanismo de comunicación en función de su característica de sincronización.

En relación a la sincronización, los mecanismos de comunicación analizados en los capítulos anteriores se clasifican en las dos siguientes categorías:

a.- Mecanismos síncronos.
b.- Mecanismos asíncronos.

La segunda se puede a su vez subclasificar en otras dos:

a.- Los que usan envío de mensajes.
b.- Los que no usan envío de mensajes (por ejemplo, variables compartidas).

Los mecanismos de comunicación de los lenguajes (modelos) están en función directa de sus esquemas de interacción e, incluso, se puede pensar que un esquema es realmente un mecanismo de comunicación.

Cada esquema pertenece a una de las categorías de sincronismo de este punto, y consecuentemente del lenguaje que lo opere.

La siguiente tabla relaciona los esquemas con sus características de sincronismo:

Esquema	Categoría sincronismo
variables compartidas	asíncrono (no usa envío de mensajes)
participación equitativa	síncrono
cliente/servidor	síncrono
interconexión	asíncrono (con envío de de mensajes)

Estas características se heredan a los lenguajes (modelos), y el resultado final se puede visualizar en la siguiente tabla:

Mecanismos de comunicación

	Esquemas interacción	Características sincronismo
CSP	Participación equitativa	síncrono
SR	variables compartidas, cliente/servidor e int.	síncrono y asíncrono con ambas modalidades
ADA	variables compartidas y cliente/servidor	síncrono y asíncrono sin envío de mensajes

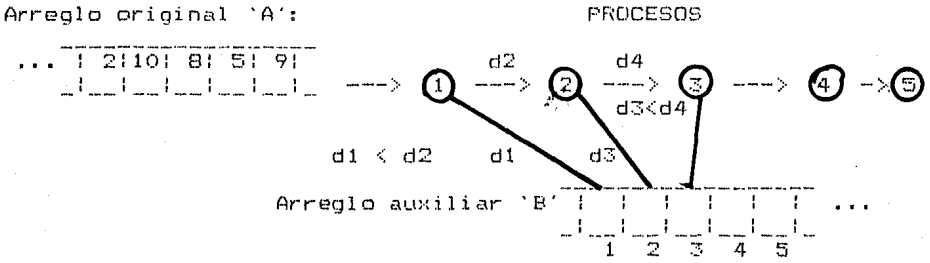
SR es el único de los tres que posee un mecanismo de comunicación de envío de mensajes y de tipo asíncrono.

Se considera que el tamaño del almacén de mensajes (buffer) para el esquema de interconexión es infinito, sin embargo, en la implementación se debe considerar un límite a su tamaño, que de ser rebasado ocasiona la suspensión de los procesos transmisores hasta que vuelva a existir espacio para sus mensajes.

A continuación se presenta un ejemplo clásico en el que toma ventaja el uso de asincronismo en el mecanismo de envío de mensajes.

El ejemplo [Andrews, 1981] consiste en ordenar concurrentemente un arreglo de enteros en orden ascendente, usando tantos procesos como elementos tenga el arreglo.

El algoritmo se puede describir auxiliándose de la siguiente figura:



Se utiliza un arreglo auxiliar B que es compartido por todos los procesos en el 'Recurso' (SR).

El proceso 1 lee directamente del arreglo original A los datos, lee el primero y lo guarda en B[1], lee el siguiente y lo compara contra B[1], si es mayor, se lo envía al proceso 2, si es menor, lo intercambia con B[1], y el que ocupaba el lugar de B[1] lo envía al proceso sucesor 2; las lecturas subsecuentes las va comparando contra B[1] y según su valor las intercambia, o las envía al proceso sucesor, hasta terminar de leer todo el arreglo.

Los procesos 2, 3,..., realizan actividades similares, considere al proceso i-ésimo:

1.- El primer dato que se le envía lo almacena en B, y en su respectivo casillero (mismo índice del arreglo B con el del proceso, considérese al proceso i-ésimo).

2. - Los posteriores envíos los compara contra B[i], si son menores los intercambia con B[i] y el valor anterior de B[i] lo envía al proceso sucesor (i+1); si es mayor lo envía directamente al sucesor, hasta que ya no existan más datos.

3.- El último proceso cuyo índice es N (tamaño del arreglo), después de almacenar en B[n] el último dato del arreglo, informa al proceso 1 de la terminación del ordenamiento (existen tantos procesos como elementos existan en el arreglo).

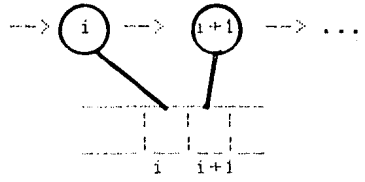
A continuación se presenta el código del programa en SR. Recuerde que la función myprocess indica el índice del proceso que la llamó.

```
RESOURCE ORDENA_ARREGLO;
     DEFINE ordena (CALL); -- llamada externa del usuario
     VAR B : array of INTEGER; -- Arreglo auxiliar B
         N : INTEGER; -- tamaño del arreglo
     PROCESS EMPIEZA; -- primer proceso
          VAR temp, i : INTEGER;
          DO true -->
               IN
                    ordena (VAR A: array of INTEGER;
                            tamaño: INTEGER);
                    N := tamaño; -- A:arreglo a ordenar que pro-
                         -- porciona el usuario, junto con su
                         -- tamaño
                    B[1] := A[1]; i := 2;
                         -- almacena el primer dato en B[1]
                    DO i <= N -->
                         IF B[1] <= A[i] -->
                              SEND SUCESOR[2] (A[i])
                                   -- lo envía directamente
                         []
                            B[1] <= A[i] -->
                              -- intercambia y envía
                              temp := B[1];
                              B[1] := A[i];
                              SEND SUCESOR[2] (temp)
                         FI
                         i := i + 1
                         -- siguiente dato
```

```
                    OD
                    IN termino() --> A := B
                         -- Copia el arreglo B a A, con objeto de
                         -- dejar el resultado nuevamente en A
                    NI
                    -- 'termino' es la señal que le manda el
                    -- proceso 'sucesor[n]' cuando ya concluyó el
                    -- ordenamiento
               NI
          OD
     END EMPIEZA;

-- A continuación se representan los procesos sucesores

     PROCESS ordena [ 2..tamaño];
          VAR i, temp: INTEGER; -- variables locales a 'ordena'

          DO true -->
               IN SUCESOR(v:INTEGER) --> B[myprocess] := v
               NI;
               i := myprocess;
               -- el primer dato lo almacena en B
               DO i < N -->
                    IN SUCESOR(v:INTEGER) -->
                         IF B[myprocess] <= v -->
                              SEND SUCESOR[myprocess + 1] (v)
                              --lo envía directamente al sucesor
                         []
                            B[myprocess] > v -->
                                   -- intercambia y envía
                                   temp := B[myprocess];
                                   B[myprocess] := v;
                                  SEND SUCESOR[myprocess+1](temp)
                         FI
                    NI;
                    i := i + 1 -- siguiente dato
               OD;
               IF myprocess = N --> SEND termino()
               [] myprocess <> N --> SKIP
               FI
                    -- envía la señal de terminación a
                    -- EMPIEZA si se trata del proceso 'N'

          OD
     END ordena;
END ORDENA_ARREGLO;
```

Observe que en un momento determinado varios procesos se están ejecutando en paralelo, lo cual no sucedería si en lugar de SEND se usara CALL (mecanismo síncrono).

Veamos qué sucedería si se utilizara un mecanismo síncrono:



El proceso i recibe su primer dato y lo almacena en B[i] con la siguiente instrucción:

```
                    IN SUCESOR (v: INTEGER) ---> B[myprocess] := v
                    NI
```

el dato subsecuente lo lee, y según su valor ejecuta:

```
                    CALL SUCESOR[myprocess + 1](v)
```

o

```
                    CALL SUCESOR[myprocess + 1](temp)
```

ambas instrucciones forman parte del cuerpo de IN.

El proceso i tendría que esperar a que el proceso i+1 lo atendiera, sin embargo, el i+1 debe esperar a que el i+2 lo atienda para poder continuar, y así sucesivamente.

Hasta que se alcance un nuevo proceso, se descongela la ejecución de los procesos predecesores de la cadena.

Esto multiplica el tiempo de ejecución del algoritmo, disminuye la concurrencia y hace más fácil llegar a un estado de abrazo mortal (deadlock) si algún proceso que no sea el último llama a uno de sus predecesores.

3.3- Simetría con respecto a la invocación.

Este punto se refiere a la simetría que dos procesos que desean comunicarse por envío de mensajes, guardan respecto a su invocación.

Si ambos procesos deben invocarse e identificarse mutuamente para poderse comunicar, entonces la comunicación es simétrica; de otro modo es asimétrica.

La simetría requiere que ambos procesos jueguen un papel activo para el establecimiento de la comunicación. CSP es el único de los tres lenguajes cuya comunicación es simétrica, ya que tanto el proceso transmisor como el receptor deben llamarse mutuamente.

El esquema de participación equitativa es el único donde existe simetría.

El esquema cliente/servidor de ADA y SR es asimétrico por las siguientes razones:

1.- El cliente siempre invoca al servidor, éste último siempre juega un papel pasivo.
2.- El servidor nunca reconoce la identidad del cliente, salvo que este último se lo indique en el mensaje.

El esquema de interconexión de SR es obviamente asimétrico.

En general, son mayores las desventajas que las ventajas de un mecanismo simétrico (por invocación) de comunicación, ya que la simetría no permite que sean fácilmente implementables las bibliotecas de procesos.

Una biblioteca ofrece a los usuarios (procesos clientes) un conjunto de servicios a través de sus procesos servidores, los cuales no tienen porqué conocer la identidad de los clientes, debido a que no es determinable saber cuales procesos usuarios podrán hacer uso de la biblioteca.

La simetría puede ser útil en aquellas aplicaciones en que el servidor requiera conocer la identidad del cliente, como se muestra a continuación.

Considérese el problema de compartir un recurso (no confundir con el Recurso de SR) entre n procesos usuarios. En CSP se puede programar en las tres siguientes líneas:

```
RECURSO_COMPARTE::
     * [ (i : 1..n) usuario(i) ? solicita() -->
                    usuario(i) ? libera() ]
```

y se programa con una familia de procesos usuarios; solicita y libera permiten solicitar y liberar el acceso al recurso compartido, respectivamente.

Observe que el recurso se otorga al usuario 'i', y sólo él puede liberarlo.

Para programar esta misma aplicación en ADA se requieren las siguientes 15 líneas de código:

```
1    TYPE ident is new INTEGER RANGE 1..n;
2    TASK RECURSO_COMPARTE IS
3         ENTRY solicita(id : IN ident);
4         ENTRY libera(ident'first..ident'last);
5    END;
6    TASK BODY RECURSO_COMPARTE IS
7         usuario : ident;
8    BEGIN
9         LOOP       -- 'id' es identificación del cliente
10             ACCEPT solicita(id: IN ident) DO
11                  usuario := id;  -- solicita acceso
12             END;
13             ACCEPT libera(usuario: IN ident); -- libera al
                          -- usuario que solicitó
14        END LOOP;
15   END RECURSO_COMPARTE;
```

Las ventajas de CSP sobre ADA se pueden resumir en los siguientes puntos:

1.- Reducción de código.
2.- Ahorro en mensajes.
3.- Mayor seguridad de acceso.

El segundo punto es consecuencia de que en ADA se inviertan más mensajes para indicar la identidad del proceso usuario (cliente).

El último punto es indicativo de que en la tarea descrita de ADA, cualquier proceso puede enviar un mensaje con una identificación diferente a la suya, para accesar el recurso compartido.

El programa escrito en SR sería muy similar, y adolecería de los mismos puntos.

3.4- Simetría con respecto a la suspensión.

Este punto se refiere a la simetría que dos procesos, que se comunican por envío de mensajes, guardan respecto a su estado de suspensión o bloqueo.

La mecánica de comunicación es, en este sentido, simétrica, si una vez establecida la comunicación entre los procesos transmisor y receptor, tanto el transmisor tiene la capacidad de suspender la ejecución del receptor, como este último la del transmisor.

En un esquema de participación equitativa, los procesos involucrados se sincronizan y se transfiere el mensaje (si existe), pero en ninguno de los dos se suspende la ejecución una vez establecida la comunicación.

En el esquema cliente/servidor, el servidor mantiene suspendido al cliente mientras realiza el servicio solicitado (punto de entrada en ADA, operación en SR), pero bajo ningún motivo el cliente puede suspender o alterar la ejecución del servidor. La comunicación entre ambos es asimétrica.

En el esquema de interconexión ninguno de los procesos (transmisor o receptor) suspende su procesamiento.

El único esquema al que puede aplicarse este concepto es el de cliente/servidor. En consecuencia, los únicos lenguajes en que llega a presentarse suspensión de procesos como efecto del mecanismo de comunicación son SR y ADA, y además, la mecánica es asimétrica.

Resulta poco justificable tener una relación simétrica en la que el transmisor (cliente) también pudiera suspender la ejecución del servidor, además de que la implementación del lenguaje sería mucho más compleja.

En la siguiente tabla se sintetizan las propiedades de simetría de los mecanismos de CSP, SR y ADA.

SIMETRIA

	por invocación	por suspensión
CSP	simétrica	no hay suspensión
SR	asimétrica	asimétrica
ADA	asimétrica	asimétrica

3.5- Almacenamiento de mensajes.

En este punto se hace un análisis del nivel de almacenamiento de mensajes en cada uno de los tres lenguajes.

En lugar de hacer referencia a los lenguajes, el análisis se realizará en los esquemas de interacción, ya que SR tiene más de uno por envío de mensajes.

En un esquema de participación equitativa como el de CSP, los dos procesos involucrados en la mecánica de comunicación deben sincronizarse para poderla establecer, después de lo cual se efectúa la transferencia.

Generalmente uno de los dos procesos debe suspenderse (en espera del otro), y si se trata del que posee el comando de salida con el mensaje, entonces se bloquea conjuntamente con el mensaje, hasta que el otro proceso esté listo, y a partir de ese momento se realiza la asociación de los parámetros que constituyen el mensaje.

Observe que no existe necesidad de tener un almacén que guarde temporalmente los mensajes.

En una interacción cliente/servidor, en la que el cliente se suspende, el bloque de control del proceso cliente (process control block) tiene espacio para guardar el mensaje o bien, un apuntador a él, razón por la que no existe necesidad de manipular almacenamiento de mensajes.

Sin embargo, en la interacción de tipo interconexión sí existe necesidad de almacenar mensajes, por su característica de asincronismo. Para ejemplificarlo, observe la siguiente gráfica:

TIEMPO:	procesos transmisores: mensajes:			proceso receptor: mensajes:
	A	B	C	D
1	A1	.	.	.
2	.	B1	.	A1
3	A2	.	C1	B1
4	.	B2	.	A2
5	.	.	.	.

D empieza a atender mensajes en el tiempo 2, A1 es el primer mensaje que recibe; pero mientras D lo procesa, B le envía otro mensaje, recuerde que ni A ni B se bloquean, y por lo tanto existe necesidad de almacenar temporalmente B1; en el tiempo 3 se le envían los dos mensajes A2 y C1, los que en ese momento no puede recibir, y por lo tanto se tienen que almacenar. El tamaño del almacén debe ser lo suficientemente grande para dar cabida a varios mensajes.

En SR cada mensaje almacenado es una copia de los parámetros reales del proceso transmisor que los envió.

De los tres lenguajes, el único en que se opera el almacenamiento de mensajes es SR, por su esquema de interconexión.

3.6- Modos de transferencia de mensajes.

Para el análisis de este punto se ha estimado conveniente hacer una analogía con conceptos de transmisión de datos, e incluso asignar los mismos calificativos.

La transmisión de datos a través de un canal de comunicación físico se puede clasificar en las tres siguientes categorías:

1.- Simplex.
2.- Half-duplex.
3.- Full-duplex.

En la primera la transmisión es unidireccional, es decir, la información únicamente puede transferirse en un sentido.

En la segunda la transmisión es bidireccional, y los mensajes pueden transferirse en los dos sentidos entre transmisor y receptor, a excepción de que el canal solamente puede ser ocupado por un proceso a la vez; es decir, no es permitido que simultáneamente los dos procesos utilizen el canal (uno envía primero y el otro después).

Finalmente, una transmisión Full-duplex es bidireccional, y sí se permite que ambos procesos usen el canal simultáneamente.

En CSP la transferencia de mensajes es unidireccional, ya que siempre es en el sentido del proceso que posee el comando de salida al que tiene el de entrada, y su transferencia se puede clasificar como SIMPLEX.

En el esquema de interconexión de SR la tranferencia también es SIMPLEX. La diferencia respecto a CSP es su capacidad de almacenamiento de mensajes: el canal contiene un almacén (buffer).

En el esquema cliente/servidor de ADA y SR la transferencia es bidireccional, sin embargo, se puede considerar que en primer lugar el cliente envía el mensaje y consecuentemente es el primero en utilizar el canal (puede ser el canal físico de una red de procesadores, o implementarse en memoria), posteriormente, y con el mismo mensaje (los mismos parámetros) el servidor puede enviarle al cliente información (resultados); en este caso, el que usa el canal es el servidor.

Observe que ambos utilizan el canal, pero en distintos tiempos, por lo que la transferencia está en la categoría de half-duplex.

Ninguno de los esquemas de interacción maneja transferencias del tipo full-duplex.

El modo de transferencia de CSP restringe al lenguaje, y favorece la duplicación de mensajes. Considere el siguiente ejemplo.

Se trata de un proceso que opera sobre una tabla (recurso compartido): se le proporciona un dato, lo busca en la tabla y regresa el valor del dato asociado. El ejemplo programado en ADA es:

```
TASK TABLAS IS
     ENTRY BUSCA(dato: IN INTEGER; resultado: OUT INTEGER);
END TABLAS;
TASK BODY TABLAS IS
     TYPE tab IS dato1, dato2 : INTEGER; END RECORD;
     tabla : array (1..N) of tab;
     BEGIN
          LOOP
               ACCEPT  BUSCA(dato:  IN  INTEGER;
                             resultado:  OUT INTEGER) DO
                    FOR I IN tabla LOOP
                         EXIT WHEN tabla(i).dato1 = dato
                    END LOOP;
                    resultado = tabla(i).dato2
               END BUSCA;
          END LOOP;
     END TABLAS;
```

El ejemplo programado en CSP, considerando tabla como una matriz bidimensional es:

```
tablas::
     tabla:(1..100,1..100) integer; i: integer;
     i := 1;
     * [
          cliente ? busca(dato1) -->
               * [
                    i <= 100; tabla(i,1) <> dato1 -->
                         i := i + 1
                 ] cliente ! tabla(i,2)
       ]
```

En estos ejemplos no se consideró el caso de que el dato no existiera en las tablas, con objeto de no complicarlo, pues lo que se desea mostrar es lo correspondiente a las tranferencias.

En ADA el proceso cliente llama a tablas así:

```
TABLAS.BUSCA(dato_a_buscar,dato_asociado);
```

En CSP se requieren las siguientes invocaciones, por parte del usuario:

```
TABLAS ! busca(dato_a_buscar)     Y
TABLAS ? dato_asociado
```

Observe que en ADA sólo hubo necesidad de hacer una invocación y utilizar un mensaje, mientras que en CSP se requirieron dos invocaciones y dos mensajes.

3.7- Características de asociación de mensajes.

En función de lo analizado en los lenguajes CSP, SR y ADA respecto a este punto, se concluye lo siguiente:

1.- En CSP la asociación de parámetros es únicamente por valor, por ser la transferencia de tipo SIMPLEX.

2.- En el esquema cliente/servidor de ADA y SR la asociación de parámetros está dada en la siguiente tabla:

Parámetro:	Tipo asociación:
Entrada	valor
Salida	resultado
entrada/salida	valor y resultado

en ADA és exclusivamente para datos de tipo escalar.

En ADA no está definida la asociación para datos que no sean de tipo escalar.

Como el lector puede observar, la asociación entre parámetros se realiza en los tres casos mediante la copia de un conjunto de parámetros a otro conjunto, y viceversa.

No se hace uso de la asociación por referencia, muy comúnmente usada en otros varios lenguajes, y la razón está en la característica de distribución del sistema, pues resulta poco factible el que dos procesos localizados en distintos nodos de

una red de procesadores accesen un área común de referencia, que se tendría que localizar en la memoria de uno de los dos procesadores.

La siguiente tabla sintetiza los puntos 4, 5 y 6 de esta sección:

MENSAJES

	Almacenamiento mensajes	Modo de transferencia	Asociación de parámetros		
			entrada	salida	ent/sal
CSP	NO	SIMPLEX	valor	-	-
SR	SI(esq. inter-conexión)	HALF-DUPLEX	valor	resul	valor y resul
ADA	NO	HALF-DUPLEX	valor *	resul*	valor* y resul

* Aplicable únicamente a datos de tipo escalar.

3.8- Control de la comunicación.

Este punto está dirigido a analizar el control que sobre la comunicación puede ejercer un proceso cuando interacciona con más de uno.

CSP no es muy específico en relación a este punto, debido a que no ha sido implementado como lenguaje, sino que se trata de un modelo. Por esta razón no se considera para el presente análisis.

Tanto en SR como en ADA, por cada una de las instrucciones de entrada (puntos de entrada) de un proceso servidor existe una estructura de tipo cola en la que están registrados todos los procesos clientes que la han invocado y que están suspendidos en espera de ser atendidos.

En ADA el orden en que se registran es en el que se atienden, y de esta manera se establece una cola de tipo FIFO (el primero en invocar es el priemro en atender).

ADA no posee control sobre estas estructuras, por que no puede alterar el orden de atención de las tareas registradas, aunque éstas posean prioridad asignada (estática).

En SR el orden en que se registran los procesos clientes también es de tipo FIFO: el primero en invocar es el primero en la cola.

Pero a diferencia de ADA, SR sí tiene control sobre la cola, y

puede seleccionar de entre las invocaciones que tiene registradas a la que va a atender primero.

La selección se hace de la siguiente forma:

1.- Una primera selección de aquellos procesos registrados cuyo mensaje satisfaga la expresión de sincronización de la instrucción de entrada (designación de la operación), si la expresión está en función de los parámetros que constituyen el mensaje.

2.- Si está también definida la expresión de scheduling en el guardia, entonces se escoge de entre los procesos previamente seleccionados al que minimice la expresión, de otro modo se toma de entre el subconjunto previamente seleccionado de procesos al que más cerca esté de la cabeza de la cola.

3.- Si no existen expresiones de sincronización ni de scheduling, entonces se atiende al primero de la cola.

Lo anterior se debe a la facilidad que presenta SR de poder utilizar, en las expresiones de sincronización y scheduling del guardia, los parámetros formales de la instrucción de entrada (mensaje).

Como se puede observar, SR aventaja considerablemente a ADA en el control de la comunicación, respecto a la selección de los procesos registrados en la cola de una instrucción de entrada (punto de entrada en ADA).

Sin embargo, ADA ofrece al programador una herramienta de control que permite seleccionar una de varias colas de operación, en función de las prioridades de los procesos que tienen registrados en sus respectivas cabezas (los primeros procesos registrados en las colas).

Se seleccionará de entre los procesos registrados en las cabezas de las colas de operación, de aquellos puntos de entrada elegibles de competir en la instrucción SELECT (alternativas abiertas de aceptación) al que sea de mayor prioridad, si es que el programador asignó prioridades a procesos en forma explícita (pragma priority), de otro modo la selección es totalmente indeterminística.

A este respecto SR no posee ningún control, y la selección es totalmente no determinística.

En la siguiente tabla se resume lo analizado en este punto.

CONTROL DE COMUNICACION
A NIVEL DE

	Selección de procesos en una sola cola de instrucción (punto) de entrada	Selección de procesos cabeceras en varias colas de instrucciones (puntos) de entrada
SR	en función de expresiones booleanas y aritméticas	no tiene control
ADA	no tiene control	en función de prioridades especificadas en forma explícita a los procesos por el programador

3.9- Control de fallas en la comunicación.

El análisis se enfoca a la capacidad que tiene algún mecanismo o lenguaje de recuperarse de alguna falla en la mecánica de comunicación, independientemente de la causa que la haya producido.

Las fallas de comunicación en CSP son irreversibles, ya que originan la falla y terminación de los procesos involucrados en ella, sin que se pueda realizar ninguna acción preventiva o correctiva.

Las fallas de los comandos de entrada y salida se pueden originar por la terminación (exitosa o fallida) de sus correspondientes procesos fuente y destino, lo que a su vez origina la falla de los procesos a los cuales constituyen.

La falla se puede salir de control y propagarse en cadena, sin posibilidad de prevenir aún más su propagación y efectos.

En SR el tratamiento que se da a las fallas de comunicación consiste en agrupar un conjunto de manejadores de excepciones (de diferentes operaciones) en un Recurso.

Cuando se presenta la falla se suspende el proceso, y el manejador de excepción relacionado es invocado por el núcleo del sistema operativo, el cual intenta corregir la situación y si es posible reactivar al proceso suspendido.

En ADA se genera la excepción TASKING_ERROR cuando falla la comunicación, y se propaga a los dos procesos involucrados.

Como ya se ha indicado, se puede programar dentro de cada proceso un manejador relacionado con la excepción por falla de comunica-

ción, que intente corregir la situación.

En ADA existen manejadores de excepciones con los que se puede inhibir y/o controlar la propagación de la misma, además de que mediante el uso de los atributos T'CALLABLE y T'TERMINATED cualquier tarea puede saber el estado de otras con las que desea comunicarse, y de esa forma evitar la generación de una excepción, como consecuencia de una mala comunicación.

En ADA se termina de realizar una cita cuando en su procesamiento se presenta una falla.

En SR no se define la acción que se toma cuando dos procesos están comunicándose.

En la siguiente tabla se sintetiza el análisis de este punto:

CONTROL DE FALLAS EN COMUNICACION

	Consecuencia falla	Modo de atención
CSP	falla procesos: irreversible	-
SR	Posibilidad se reactiven procesos involucrados	Recurso independiente de manejadores de excepción
ADA	El control está manejado por los mismos procesos	Los mismo procesos lo controlan, através de sus manejadores de excepciones

4.- SINCRONIZACION

La presente sección se ha estimado subdividir para su análisis comparativo en los siguientes puntos:

1.- Sincronización en el acceso a variables compartidas.
2.- Caracterísitcas síncronas del mecanismo de comunicación.
3.- Mecanismos condicionales y auxiliares de sincronización.
4.- Sincronización en función del tiempo.

4.1- Sincronización en el acceso a variables compartidas.

De los tres lenguajes, únicamente ADA y SR hacen uso del esquema de variables compartidas. Del análisis realizado a cada uno de ellos se concluyen los siguientes puntos:

1.- El acceso a las variables compartidas es más restringido en SR que en ADA.
2.- SR proporciona en forma automática atomicidad en el acceso a las variables compartidas.

Con respecto al primer punto, cualquier tarea de ADA que tenga acceso a otros paquetes a través de la instrucción USE, también lo tendrá a las variables globales declaradas en la parte visible de ellos, así como también tendrá acceso a variables globales de unidades de programas que formen parte del cuerpo de la unidad de la cual forma parte.

Sin embargo, en SR los procesos de un Recurso no tienen acceso a las variables permanentes de otros Recursos, aunque en forma indirecta pueden accesarlos a través de operaciones e instrucciones de entrada programadas en procesos.

Lo anterior es consecuencia de la filosofía distribuida de SR, pues la idea central es la de tener un Recurso por nodo en una red de procesadores, por lo que el medio de comunicación más adecuado es por envío de mensajes.

Respecto al segundo punto, en SR el acceso a las variables permanentes del Recurso es indivisible (atómico), es decir, el acceso es ininterrumpible, lo cual es sumamente ventajoso, pues en todo momento se sabe que sólo un proceso está accesando la variable, y hasta que él la desocupe otro podrá accesarla.

En ADA no existe atomicidad en el acceso a las variables, y es responsabilidad del programador asegurar el acceso exclusivo a las mismas.

En la siguiente tabla se resumen estas características:

VARIABLES COMPARTIDAS

	Alcance del acceso	Tipo de acceso
SR	restringido al Recurso	atómico (indivisible)
ADA	poca restricción	interrumpible

4.2- Características síncronas del mecanismo de comunicación.

Respecto a este punto se concluye lo siguiente:

1.- El mecanismo de comunicación de CSP posee exclusivamente un punto de sincronización.
2.- Los mecanismos de ADA y SR poseen dos puntos de sincronización.

3.- Cuando se ha establecido la comunicación con el proceso servidor, éste se convierte en un área de exclusión mutua.
4.- En los esquemas de participación equitativa de CSP, y el cliente/servidor de SR y ADA, se reduce considerablemente la interferencia entre los procesos.
5.- En los esquemas citados en el punto 4, todo proceso puede saber el estado de avance del otro con el que ha mantenido comunicación.
6.- La comunicación puede utilizarse en CSP, SR y ADA como una herramienta de sincronización entre los procesos, sin que exista la transferencia de un mensaje.
7.- El mecanismo de comunicación de interconexión de SR es totalmente asíncrono, y tiene las siguientes características:
    a.- No posee puntos de sincronización.
    b.- Incrementa la posibilidad de existencia de interferencia entre ambos procesos involucrados.
    c.- Ninguno de los dos procesos puede saber el estado de avance del otro.

Recuerde que un punto de sincronización entre procesos es aquél en el que ambos interaccionan en un instante.

En CSP ambos procesos se sincronizan para el envío del mensaje, y en ningún otro punto de su existencia vuelven a sincronizarse (salvo que existan otros comandos de entrada/salida en los que se invoquen mutuamente).

El esquema cliente/servidor de SR es semánticamente idéntico al de ADA, ambos están implementados con el mecanismo de llamadas a procedimientos remotos, y en ambos casos se realiza una cita (rendezvous) entre los procesos.

Sus puntos de sincronización son el inicio y el final de su cita, como ya se indicó en el capítulo 3.

Con respecto al tercer punto, y haciendo referencia a capítulos anteriores, el mecanismo de llamadas a procedimientos remotos es una herencia del mecanismo monitor, cuyo uso permite sincronizar el acceso a variables compartidas. Una operación de SR o un punto de entrada de ADA, al ser invocado y establecer la cita con el proceso cliente, hacen que el proceso servidor inhiba el acceso hacia él por parte de otros procesos clientes, aunque invoquen otras operaciones (puntos de entrada) declaradas en su cuerpo. Ofrece exclusión mutua sobre sus recursos (variables) y código mientras exista la cita, por lo que tanto en SR como en ADA se pueden asignar procesos al control de entidades compartidas, tales como archivos, dispositivos periféricos, etc., heredándose así la exclusión mutua a los mismos.

Con respecto al punto 6, el objetivo de que se establezca la comunicación entre dos procesos sin que exista una transferencia de mensaje es crear un punto de sincronización entre ambos, con lo que la ejecución de uno está en función directa de la del

otro, y viceversa.

En la siguiente tabla se sintetiza lo referente a este punto:

MECANISMOS DE COMUNICACION
CARACTERISTICAS DE SINCRONIZACION

	¿Puntos de sincronización	¿Ofrece exclusión mutua ?	¿Posibilidad interferencia ?	¿Sirve como mecanismo de sincronización entre procesos ?
CSP	1	-	poca	SI
SR-cliente/serv.	2	SI	poca	SI
SR-interconexión	0	NO	mayor	NO
ADA	2	SI	poca	SI

4.3- Mecanismos condicionales y auxiliares de sincronización.

Los mecanismos condicionales que sincronizan la ejecución y comunicación de los procesos en función de condiciones establecidas por el programador son:

Lenguaje	Mecanismo
CSP:	Guardia booleano
SR:	Guardia booleano
	Guardia de operación
ADA:	Guardia booleano

Cada uno de ellos tiene relacionada una lista de instrucciones que se procesará si y sólo si el guardia es verdadero.

En CSP y SR a la combinación del guardia con su lista de instrucciones asociada se le llama comando custodiado:
<guardia> --> <lista_instrucciones>

Existen ciertas aclaraciones al respecto de los guardias, que se indican en los siguientes puntos:

1.- En CSP el guardia puede ser:
    a.- Una expresión booleana.
    b.- Una expresión booleana y un comando de entrada.
    En ambos casos se pueden añadir declaraciones de variables locales al comando custodiado.
2.- En SR y en ADA un guardia booleano es simplemente una expresión booleana:
    SR: <expresión_booleana> --->
    ADA: WHEN <expresión_booleana> =>
3.- En SR un guardia de operación está constituido por los tres siguientes elementos:
    a.- expresión booleana (guardia booleano).
    b.- designación de operación (instrucción de entrada).
    c.- expresión aritmética de scheduling (opcional).

En ADA se puede tener, como primera instrucción de la lista de instrucciones asociada al guardia, un ACCEPT (punto de entrada) y el efecto es el mismo que el guardia de CSP o el guardia de operación de SR (eliminando la expresión de scheduling).

En todos los casos, el efecto y la semántica de este mecanismo de sincronización es el mismo, y es el único mecanismo de tipo condicional que usan los tres lenguajes.

Sin embargo, SR supera considerablemente a CSP y ADA en que el guardia puede hacer referencia a los parámetros formales de la operación o instrucción de entrada.

Lo anterior permite al proceso servidor controlar la comunicación en función del contenido del mensaje.

De todos los procesos clientes que desean comunicarse con el servidor, únicamente atenderá a aquellos cuyos parámetros posean un valor tal que hagan verdadera la expresión booleana del guardia (que está en función de ellos).

Ni ADA ni CSP pueden controlar la comunicación en función del contenido del mensaje.

Entre los mecanismos auxiliares de sincronización están las siguientes funciones:

SR	ADA
? <nombre_operación>	<nombre_punto_entrada>'COUNT

que pueden ser utilizadas en la expresión booleana del guardia. Ambas indican el número de tareas registradas en la cola de operación de la instrucción (punto) de entrada cuyo nombre está dado por el símbolo <nombre....>.

Por ejemplo, si se desea que una operación OP2 se ejecute si y sólo si OP1 no ha sido invocada, se hace uso de la función en el siguiente fragmento de programa:

SR:

```
IN    OP1() --> lista_instrucciones_1
[]    OP2() AND ? OP1 = 0  --> lista_instrucciones_2
NI
```

ADA:

```
SELECT
     ACCEPT OP1 DO
          lista_instrucciones_1
     END OP1;
OR
     WHEN OP1'COUNT = 0 =>
          ACCEPT OP2 DO
               lista_instrucciones_2
          END OP2;
END SELECT;
```

Se debe tener cuidado en el uso de esta función, pues se pueden generar efectos colaterales o interferencia por un mal uso.

Síntesis del análisis de mecanismos condicionales y auxiliares de sincronización:

	condicional	auxiliar	control de la comunicación en función del mensaje
CSP	guardia	no tiene	NO
SR	guardia booleano y de operación	?	SI
ADA	guardia	COUNT	NO

4.4- Sincronización en función del tiempo.

ADA es el único lenguaje que ofrece al programador una instrucción, que permite controlar y sincronizar eventos en función del tiempo real.

La instrucción delay permite que un proceso se suspenda un determinado número de segundos.

delay <expresión>

SR y CSP carecen de una instrucción similar, lo que los pone en franca desventaja con respecto a ADA en aplicaciones de tiempo real.

Para ejemplificar lo anterior, considérese la necesidad de programar un despertador o alarma. A continuación se presenta el texto en ADA y en SR, los ejemplos están basados en [Welsh Lister, 1979, 1981]:

SR

```
RESOURCE DESPERTADOR;
     DEFINE despierta_a_las (CALL);
     DEFINE reloj (CALL);

PROCESS ALARMA;
hora_despertar, tiempo : INTEGER;
DO true -->
     IN
          -- hora en segundos: 0 < hora < 86400
          IF hora > tiempo -->
               intervalo := hora - tiempo;
          []
               -- horario del siguiente día
               hora < tiempo -->
                    intervalo := hora + (86400 - tiempo)
          []
               hora = tiempo -->
                    intervalo := 0
          FI
          hora_despertar := tiempo + intervalo
     []
          reloj() --> tiempo := tiempo + 1
          -- función atada a un reloj real
     []
          hora_despertar = tiempo --> CALL despierta()
          -- ya es la hora, !despierta
     NI
OD
END ALARMA;
END DESPERTADOR;
```

ADA:

```
TASK despertador IS
     ENTRY despierta_a_las (hora: IN TIME)
END despertador;
TASK BODY despertador IS
     USE CALENDAR;
     intervalo : DURATION;
begin
     LOOP
          ACCEPT despierta_a_las (hora: IN TIME) DO
               intervalo := hora - CLOCK
          END despierta;
          DELAY intervalo;
          DESPIERTA  -- manda sonar la alarma
     END LOOP
END DESPERTADOR;
```

Las diferencias entre ambos programas se sintetizan en los siguientes puntos:

1.- ADA maneja directamente tiempos (tipos TIME y DURATION), a través de su paquete CALENDAR, mientras que SR tiene que mapear los tiempos a enteros.

2.- SR tiene que apoyarse en el proceso reloj, el cual debe estar conectado directamente a una interrupción de un reloj del equipo, elaborado en hardware.
ADA únicamente hace uso de la instrucción delay, y no requiere procesos externos atados al hardware del equipo de cómputo.

Por las anteriores observaciones, ADA está más adaptado a aplicaciones de tiempo real que SR y CSP.

El programa en CSP sería muy semejante al de SR, pues requeriría el uso de algún proceso externo conectado directamente a un reloj físico.

5.- INDETERMINISMO

Para el análisis comparativo de esta sección, se estimó que era más conveniente desglosarlo en función de una clasificación propia de los mecanismos de indeterminismo estudiados en los capítulos anteriores, en la que se observan las características favorables y desfavorables de unos con respecto a otros.

La semántica de un mecanismo de indeterminismo está definida por la de todas aquellas instrucciones indeterminísticas que poseen la misma semántica, aunque tengan diferente sintaxis.

La clasificación de los mecanismos de indeterminismo se muestra en el siguiente cuadro sinóptico:

- Mecanismos de Indeterminismo
  - con interacción de procesos
    - receptivos
      - pasivos de atención a procesos
      - activos de atención a procesos
      - de temporización
      - de terminación
    - transmisores
  - sin interacción de procesos

Esta sección se analizará en relación con los siguientes incisos:

1.- Mecanismos sin interacción de procesos.
2.- Mecanismos transmisores con interacción de procesos.
3.- Mecanismos receptivos con interacción de procesos.

5.1- Mecanismos sin interacción de procesos.

Estos mecanismos no interaccionan con procesos externos al que pertenecen. En su estructura no existe ninguna instrucción de comunicación y están constituidos exclusivamente por guardias booleanos, es decir, expresiones lógicas y listas de instrucciones.

Con estos mecanismos se puede implementar un IF o un DO de FORTRAN. Su estructura es la siguiente:

```
SR
     IF
          <guardia1> --> <lista_inst_1>
     []
          <guardia2> --> <lista_inst_2>
     []
                .
                .
     FI
```

Las únicas instrucciones que constituyen este mecanismo son:

1.- Comando alternativo de SR.
2.- Comando repetitivo de SR.

SR las usa para elaborar su instrucción alternativa IF, y su instrucción iterativa DO, en lugar de usar un IF-THEN-ELSE, o un DO I=1,N END DO, con objeto de uniformizar la sintaxis con el uso del símbolo [].

5.2- Mecanismos transmisores con interacción de procesos.

Estos mecanismos incluyen instrucciones de comunicación por envío de mensajes con otros procesos.

Según la tabla de clasificación, de estos mecanismos existen los receptivos y los transmisores.

La diferencia se fundamenta en el tipo de instrucción de comunicación con que están implementados; si la instrucción es una invocación o llamada (llamada de punto de entrada), entonces el papel que juega el proceso que componen es el de transmisor o cliente. Por esta razón se les llama mecanismos transmisores.

Los receptivos están implementados exclusivamente por instrucciones de entrada (operaciones en SR, puntos de entrada en ADA), y el papel que juega el proceso del cual forman parte es de receptor o servidor.

Las únicas instrucciones que constituyen al mecanismo transmisor con interacción de procesos son:

1.- La selección condicional de ADA.
2.- La selección temporal de ADA.

En estas instrucciones el indeterminismo está en realidad

restringido, pues siempre se intenta ejecutar primero la instrucción de llamada (mayor prioridad) que la otra alternativa (ELSE o delay). Sin embargo, debido a su interacción indeterminística con otros procesos, no es determinable saber qué alternativa será la que se procesará.

Este mecanismo fundamenta su utilidad en la repetición (ciclo) de las instrucciones que lo constituyen.

Así por ejemplo, una selección condicional que se repite en un ciclo puede aprovecharse en el procesamiento de cálculos locales del proceso, mientras periódicamente intenta establecer comunicación con otros, evitando de esta forma un estado de total ociosidad, que sería el caso de tener exclusivamente a la única instrucción de llamada.

Si la lista de instrucciones correspondiente al ELSE de una selección condicional en un ciclo es una instrucción nula (null), entonces se implementa un mecanismo de comunicación de busy waiting.

Otras aplicaciones muy importantes de estas instrucciones son las siguientes:

Instrucción	Aplicación
1.- Selección condicional -----	'polling'
2.- Selección temporal --------	supervisión

Respecto al 'polling', considérese un proceso P1 que interacciona con P2, P3, P4 y P5. El orden en que interacciona es de P2 a P5, de manera que primero ve si P2 está listo; si no lo está entonces intenta interaccionar con P3, y así sucesivamente, hasta que vuelve a intentarlo con P2, P3, etc.. en un cilo repetitivo. El código de P1 se muestra a continuación:

```
TASK BODY P1 IS
BEGIN
     LOOP
          SELECT
               P2.listo
               lista_inst_2  -- Procesa a P2
          ELSE
               SELECT
                    P3.listo
                    lista_inst_3  -- Procesa a P3
               ELSE
                    SELECT
                         P4.listo
                         lista_inst_4 -- Procesa a P4
                    ELSE
                         SELECT
                              P5.listo  -- Procesa a P5
                              lista_inst_5
                         ELSE
                              NULL
                         END SELECT;
```

```
                    END SELECT;
                END SELECT;
            END SELECT;
        END LOOP;
    END P1;
```

Como se puede observar, el 'polling' se obtiene en función del anidamiento de instrucciones de selección condicional.

La aplicación de supervisión de la selección temporal permite supervisar eventos físicos, tales como sensores, o simplemente procesos reales.

Su aplicación está enfocada directamente a programación en tiempo real. Por ejemplo considere el siguiente fragmento de tarea:

```
-- supervisor de la presión de una caldera
LOOP
    SELECT
        -- verifica y procesa la presión de la caldera
        CALDERA.PRESION (dato);
        -- Procesa al dato
    OR
        -- si en .2 segundos no hay lectura, entonces el
        -- sensor de presión ha fallado
        RAISE ALARMA_PRESION_CALDERA
        -- suena la alarma
    END SELECT;
END LOOP;
```

Observe que ADA supera considerablemente a CSP y SR, debido a que estos últimos carecen de mecanismos transmisores y con interacción de procesos y están más limitados para aplicaciones de programación en tiempo real.

5.3- Mecanismos receptivos con interacción de procesos.

Por su estructura y características, se pensó en subclasificar a estos mecanismos en las siguientes categorías:

1.- Mecanismos pasivos de atención a procesos.
2.- Mecanismos activos de atención a procesos.
3.- Mecanismos de temporización.
4.- Mecanismos de terminación.

5.3.1- Mecanismo pasivo de atención a procesos.

Las características del mecanismo pasivo de atención a procesos son:

1.- Su estructura está formada exclusivamente por alternativas (guardias) constituidas por instrucciones de entrada (operaciones, ACCEPT).
2.- Es un mecanismo pasivo, ya que siempre está a la espera de que alguna de sus instrucciones de entrada sea invocada, para atender a algún proceso.

Consecuentemente, puede ocasionar la suspensión del proceso del cual forma parte.

3.- Su utilización está enfocada a atender indiscriminadamente a cualquiera de los procesos con los que puede establecer comunicación (fairness), dando la misma oportunidad a todos.

4.- Es el mecanismo que hace del proceso al que pertenece el clásico proceso servidor, por ser un mecanismo pasivo y receptivo, características de un proceso servidor.

Las instrucciones que comprenden este mecanismo son:

1.- Comando alternativo de CSP.
2.- Comando iterativo de CSP.
3.- Instrucción INPUT (sin else) de SR.
4.- Selección con espera de ADA, integrada exclusivamente por alternativas de aceptación.

Existen ligeras diferencias entre estas instrucciones, las que se enumeran a continuación:

1.- El comando alternativo de CSP puede fallar, y causar la terminación anormal del proceso, lo recomendable es evitar la falla del comando.
2.- El comando iterativo de CSP nunca falla.
3.- El mecanismo de ADA genera la excepción PROGRAM_ERROR cuando falla (cuando todas las alternativas de la instrucción SELECT están cerradas), para la cual se puede programar un manejador de excepción.
4.- El mecanismo de SR nunca falla, pues se suspende la ejecución hasta que existan invocaciones o alguno de los guardias se haga verdadero, lo cual puede ocasionar un abrazo mortal (deadlock).

En ADA el programador puede tener un mejor control en la falla del mecanismo (a través del uso de manejadores de excepción) que en los lenguajes CSP y SR.

5.3.2- Mecanismo activo de atención a procesos.

Las características de este mecanismo son:

1.- Su estructura es igual a la del mecanismo pasivo, añadiendo la alternativa ELSE.
2.- Este mecanismo nunca falla, pues siempre se ejcutará la lista de instrucciones del ELSE cuando las otras alternativas no sean elegibles de procesarse.
3.- Es un mecanismo activo en el sentido de que nunca hace suspender la ejecución del proceso que compone, y además se puede aprovechar en el procesamiento de cálculos locales si se repite en un ciclo, mientras no existan llamadas de procesos transmisores (clientes).
4.- También es un mecanismo que da la misma oportunidad de atención a todos los procesos que intenten comunicarse (fairness).
5.- Hace del proceso que constituye también un proceso servidor.

Las instrucciones que pertenecen a esta categoría de mecanismo son:

1.- Instrucción INPUT con ELSE, de SR.
2.- Selección con espera de ADA, integrada por un ELSE y alternativas de aceptación.

Observe que CSP carece de instrucciones pertenecientes a esta categoría de mecanismo de indeterminismo.

5.3.3- Mecanismos de temporización.

Las características de este mecanismo son las siguientes:

1.- Su estructura está constituida por alternativas temporales y de aceptación (ADA es el único de los tres lenguajes que posee este mecanismo).
2.- Este mecanismo permite controlar la mecánica de comunicación en función del tiempo, ya que proporciona intervalos de tiempo a sus procesos clientes para que lo invoquen.
3.- Su utilización se aplica directamente a la programación en tiempo real.

La única instrucción en esta categoría es:

1.- Selección con espera de ADA, formada con alternativas temporales y de terminación.

La instrucción delay de ADA es la piedra angular en que está fundamentado este mecanismo, razón por la cual ni CSP, ni SR proveen instrucciones indeterminísticas que pertenezcan a este tipo de mecanismo.

5.3.4- Mecanismo de terminación.

Las características fundamentales de este mecanismo son:

1.- Su función principal es supervisar el estado de terminación de todos los procesos con los cuales mantiene (o puede potencialmente tener) comunicación.
2.- La terminación de los procesos con los que interacciona no origina falla en este mecanismo, ni generación de excepciones.

Como consecuencia de la ejecución de este mecanismo, se puede realizar la terminación del proceso al cual constituye, por lo que permite de una manera limpia concluir la ejecución del proceso cuando ya han culminado su procesamiento todos los procesos con los que potencialmente interacciona (o puede interaccionar).

Las instrucciones que componen este mecanismo son:

1.- Comando repetitivo de CSP.
2.- Selección con espera de ADA, constituida por una alternativa de terminación y alternativas de aceptación.

La diferencia entre estas instrucciones radica en que a

consecuencia de la terminación de todos los procesos con los que este mecanismo puede tener comunicación, en CSP el comando finaliza su ejecución, y el proceso al cual constituye sigue activo; mientras que en ADA finaliza también su ejecución el proceso del cual forma parte el mecanismo.

SR no proporciona un mecanismo similar, ni define la acción que se toma cuando en una instrucción INPUT (sin else), todos los procesos clientes terminaron su procesamiento.

En la siguiente tabla se relacionan las instrucciones con los mecanismos a los que pertenecen:

	MSP	MT	MRP	MRA	MRT	MTE
CSP:						
- comando alternativo			X			
- comando repetitivo			X			X
SR:						
- comando alternativo	X					
- comando repetitivo	X					
- INPUT (sin ELSE)			X			
- INPUT (con ELSE)				X		
ADA:						
- selección condicional		X				
- selección temporal		X				
selección con espera y:						
- exclusivamente alternativas de aceptación			X			
- alternativas aceptación y ELSE				X		
- alternativas aceptación y temporales					X	
- alternativas aceptación y terminación						X

Referencias:

MSP	------>	Mecanismos sin interacción de procesos.
MT	------>	Mecanismos transmisores con interacción.
MRP	------>	Mecanismos receptivos pasivos.
MRA	------>	Mecanismos receptivos activos.
MRT	------>	Mecanismos receptivos de temporización.
MTE	------>	Mecanismos receptivos de terminación.

Obsérvese que ADA posee seis instrucciones indeterminísticas ubicadas en cinco mecanismos diferentes, mientras que SR tiene cuatro instrucciones en tres mecanismos y CSP dos instrucciones en dos mecanismos.

Consecuentemente, el más rico en manipulación de indeterminismo es ADA, además de que es el único que opera mecanismos de indeterminismo aplicables a la programación en tiempo real. Por otra parte, las instrucciones IF y FOR de ADA complementadas con las demás instrucciones de indeterminismo abarcan al mecanismo de indeterminismo del tipo sin interacción con procesos.

La única instrucción que pertenece a dos categorías diferentes de mecanismos de indeterminismo es el comando iterativo de CSP, debido a que, mientras los procesos con los que interacciona no fallen o no terminen su ejecución, su comportamiento es el de un mecanismo pasivo, pero cuando todos los procesos terminan, entonces es cuando culmina el procesamiento de este comando.

En función de este análisis comparativo, se concluye que ADA aventaja considerablemente a CSP y SR.

6.-CALENDARIZACION(SCHEDULING)

El único de los tres lenguajes que proporciona un mecanismo explícito para el control y manejo de scheduling es SR.

ADA y CSP carecen totalmente de mecanismos para control de scheduling que apoyen al programador.

El mecanismo de SR está constituido por la expresión de calendarización (scheduling) de su guardia de operación, que puede estar en función de los parámetros formales de la designación de la operación, es decir, en función del contenido del mensaje.

De todos los procesos registrados en la cola de operación, selecciona a aquél cuyos parámetros minimicen a la expresión de calendarización (scheduling).

Para la programación de schedulers en ADA y CSP se tiene que recurrir al uso de familias de procesos, lo cual complica e incrementa considerablemente el texto de los programas y algoritmos.

A continuación se presentan ejemplos que ilustran la gran poten-

cialidad de este mecanismo de scheduling de SR, y la considerable diferencia que se tiene que invertir en código y estructuras para la implementación en ADA y CSP.

Por lo anterior, SR aventaja a ADA y a CSP en lo referente a manipulación de scheduling.

Mecanismos de scheduling

CSP	SR	ADA
NO tiene	expresión de scheduling de su guardia de operación	NO tiene

Ejemplo [Welsh Lister, 1979]:

Considere la programación de un scheduler del tipo 'shortest-job-next', en el que cada proceso que requiera hacer uso del procesador debe proporcionar su tiempo estimado de ejecución, y el scheduler seleccionará a aquél que posea el menor tiempo.

CSP

```
SJN::
     cola: SET(n) INTEGER; rango:(1..n) INTEGER;
     usuario, siguiente: INTEGER;
     cola := (); usuario := NIL; siguiente := NIL;

     *[
               -- El  usuario realiza el comando:
               --  SJN  !  solicita(tiempo)
          (i:1..n)users(i) ? solicita(tiempo : INTEGER)
               --> cola.incluye(i); rango(i) := tiempo;
                    siguiente := NIL;
                    -- hace siguiente igual a nulo, con objeto de
                    -- que el algoritmo considere esta petición
     []   (i:1..n) users(i) ? libera() --> usuario := NIL
          -- El usuario dá aviso de que ya liberó al recurso
          -- razón por la cual ya no existe usuario (= NIL)
     []   siguiente = NIL; -! cola.vacia -->
               -- min : INTEGER; min := MAXINT;
               -- selección del candidato a usar el recurso:
               *[  i  in cola -->    -- 'in' no es parte  de  CSP,
                                     -- pero se usa para abreviar
                    [ rango(i) >= min --> SKIP
                    []
                      rango(i) < min --> siguiente := i;
                                         min := rango(i)
                    ]
     []
```

```
        -- Ya existe candidato, y el recurso está libre,
            -- entonces se asigna al candidato
        usuario = NIL; siguiente <> NIL -->

            users(siguiente) ! OK;
            -- el usuario debe además de pedir el acceso
                -- al recurso: SJN ? OK(), el que se le
                -- otorga hasta que es elegido
            usuario := siguiente;
            cola.excluye(usuario);
            siguiente := NIL; -- ya no hay candidato
    ]
```

Observe que los usuarios es una familia (users), donde la identificación de cada usuario es su índice en el grupo.
Cada usuario debe realizar dos comandos para poder accesar al recurso: uno para que se le registre en la lista o cola de atención, y otro para accesar al recurso compartido. En la estructura rango se registra el tiempo estimado de procesamiento que proporciona el usuario.

ADA

Ejemplo (la implementación se hizo tomando como base el ejemplo anterior):

```
TASK SJN IS
    PROCEDURE solicita (ident: IN identificador;
                        tiempo : IN TIME);
        -- En este caso, el usuario únicamente realiza una
        -- llamada 'solicita', sin embargo, debe proporcionar
        -- su identicación
    ENTRY libera  -- señal de liberación del recurso ,por
                  -- parte del usuario
END;
TASK BODY SJN IS
    NIL : CONSTANT INTEGER := 0;
    usuario: INTEGER RANGE 0..n := NIL;
    siguiente: INTEGER RANGE 0..n := NIL;
    min : INTEGER;
    cola:   array  (identificador'first..identificador'last)  of
              identificador;
    rango : array  (identificador'first..identificador'last)  of
              INTEGER;

    ENTRY registra(ident: IN identificador; tiempo: IN TIME);
    ENTRY accesa(identificador'first..identificador'last);
        -- familia de tareas usuarias
    PROCEDURE solicita (ident: IN identificador;
                        tiempo : IN TIME);
    BEGIN
        registra(ident,tiempo);
        -- se registran en la estructura 'cola' los dos datos
```

```
        accesa(ident);
        -- únicamente   la   tarea   'ident-ésima'   accesará   al
        -- recurso, familia de tareas
    END solicita;


    BEGIN
        LOOP
            SELECT
                ACCEPT registra(ident: IN identificador;
                                tiempo: IN INTEGER) DO
                    cola.incluye(ident);
                    rango(ident) := tiempo;
                    -- en RANGO se registra el tiempo
                END registra;
                siguiente := NIL; -- considera a esta
                          -- solicitud en el cálculo del
                          -- nuevo candidato
            OR
                WHEN usuario = NIL AND siguiente != NIL =>
                    -- Ya hay candidato a accesar el recur
                    -- so, y éste está libre
                    ACCEPT accesa(siguiente);
                    usuario := siguiente; siguiente := NIL;
                    cola.excluye(usuario);
            OR
                ACCEPT libera;
                usuario := NIL; -- recurso libre, asignalo
            ELSE
                -- cálculo del nuevo candidato
                IF siguiente = NIL AND NOT cola.vacia THEN
                    min := MAXINT;
                    FOR t IN cola LOOP  --no es exactamente
                        -- ADA, se hace así por abreviar
                        IF rango(t) < min THEN
                            siguiente := t;
                            min := rango(t);
                        END IF;
                    END LOOP;
                END IF;
            END SELECT;
        END LOOP;
    END SJN;
```

Observe que en estos dos programas se utilizan dos estructuras de datos, una para registrar la identificación de la tarea usuaria, y otra para registrar el tiempo. No se incluyeron a detalle el manejo de las estructuras, pues para lo que se desea analizar no se consideró necesario incluirlas.

Ejemplo [Andrews, 1981]:

```
PROCESS CALENDARIZA;
     DO --> true
          IN
               solicita(tiempo:INTEGER) by tiempo -->
                 'instrucciones de acceso al recurso'
          NI
     OD
END CALENDARIZA;
```

A continuación se enumeran las diferencias entre los programas SR, ADA y CSP:

1.- Reducción considerable del número de líneas de SR con respecto a ADA y CSP.
2.- SR proporciona implícitamente las estructuras de datos, que tanto CSP como ADA tienen que elaborar.
3.- El propio proceso CALENDARIZA controla al recurso y proporciona exclusión mutua en el acceso.
4.- No se requiere utilizar familias de procesos en SR.
5.- En SR, a diferencia de CSP y ADA, los procesos usuarios únicamente efectúan una llamada, y solamente proporcionan el tiempo estimado de ejecución.

CONCLUSIONES

En el presente trabajo se consideró el análisis de CSP, SR y ADA, que manejan el mecanismo de envío de mensajes.

Como se indicó en la introducción, CSP se consideró por ser un modelo precursor a otros que se desarrollaron en la presente década, y que hicieron uso del envío síncrono de mensajes.

ADA es un lenguaje muy rico en mecanismos de control de comunicación, sincronización, indeterminismo, etc., cuyo uso se está extendiendo por todo el mundo y cuyo diseño fue resultado de un exhaustivo trabajo en el que el Departamento de Defensa de los Estados Unidos de Norteamérica no escatimó recursos. Por tal razón, era determinante su inclusión en este trabajo.

SR es un lenguaje no comercial, y cuyo enfoque en su diseño fue dirigido hacia el desarrollo de sistemas distribuidos.

Por su aparición, SR es más reciente que ADA. Andrews tomó como uno de sus puntos de partida a ADA para el diseño de SR, con lo que el resultado fue un lenguaje enriquecido fundamentalmente en aspectos de comunicación, sincronización y manejo de scheduling. Por lo anterior, se decidió tomarlo también como elemento de análisis y comparación en este trabajo.

Sin embargo, la decisión en la selección de estos tres lenguajes no fué fácil, ya que existen muchos lenguajes y modelos concurrentes en la actualidad.

A continuación se describirán brevemente algunos de ellos.

Un lenguaje que sirvió como pilar para el desarrollo de mecanismos utilizados por otros lenguajes más contemporáneos es el de Procesos Distribuidos de Brinch Hansen [Brinch Hansen, 1978].

Este lenguaje introdujo el uso de llamadas a procedimientos remotos tan utilizado por ADA y SR.

Su mecanismo de comunicación es por envío de mensajes, y es síncrono. Hace uso de guardias y comandos custodiados para el manejo de sincronización e indeterminismo.

Procesos Distribuidos (DP) es un lenguaje cuyo diseño estuvo muy influenciado por los monitores, y su componente principal es el proceso.

Sin embargo, el proceso de DP es diferente al de ADA, CSP y SR, pues en DP un proceso está constituido por un cuerpo principal y por procedimientos, que son las puertas de acceso de procesos externos. La diferencia fundamental estriba en que se procesan concurrentemente el cuerpo principal y los procedimientos, como si fueran entidades independientes.

Los procedimientos son las operaciones o las instrucciones de entrada.

Otro lenguaje que hace uso exclusivo del mecanismo de envío de mensajes de tipo asíncrono es PLITS [Feldman, 1979], cuyas iniciales significan Lenguaje de Programación en el cielo.

PLITS fue diseñado por Jerome Feldman, y está constituido por procesos y mensajes. A los procesos se les llama módulos, y su creación es de tipo dinámico.

Para cada módulo existe una cola de mensajes, lo cual es lógico por su característica asíncrona de comunicación. Los módulos tienen absoluto control sobre sus colas de mensajes, y pueden seleccionar libremente el mensaje a procesar.

Los mensajes están formados por asociación de nombres y valores, y no existen dos mensajes con el mismo nombre, que se manipula como una cadena de caracteres.

Otro lenguaje contemporáneo es C concurrente [Tsujino, Ando, 1984], el cual está teniendo cada vez mayor aceptación.

Fue concebido por Y. Tsujino, M. Ando , T. Araki y N. Tokura de la Universidad de Osaka en Japón.

C concurrente es una extensión del lenguaje C, pero añadiendo características de concurrencia. Su diseño está enfocado a aplicarse a una arquitectura distribuida, como una red de procesadores.

Su entidad principal es el proceso, el cual es de creación dinámica. Utiliza un mecanismo síncrono de envío de mensajes muy similar al de ADA y SR, y tiene un buen control de acceso, pues maneja listas de exportación e importación.

Una característica adicional es que un proceso cliente puede opcionalmente enviar un mensaje con una bandera de estado, la cual usará el proceso servidor para la selección de atención a procesos clientes; todos aquellos mensajes cuya bandera sea igual a un valor determinado serán atendidos primero.

Este mecanismo es mejor que el de ADA, pero no es tan potente como el de SR.

Hace uso también del mecanismo de variables compartidas para comunicación entre procesos.

Al igual que ADA, SR y CSP, utiliza guardias booleanos para sincronización condicional, y poseé una instrucción de indeterminismo llamada SELECT semejante a la de ADA, pues maneja alternativas temporales y de aceptación.

Otros lenguajes que hacen uso del mecanismo de envío de mensajes son GYPSY [Good et al., 1979], que maneja almacenamiento de

mensajes, y ARGUS [Liskov and Schaifler, 1982], que utiliza ideas de Procesos distribuidos y que además maneja excepciones.

Como consecuencia de la lectura del presente trabajo el lector debió haber obtenido una visión amplia sobre los diferentes tipos de mecanismos, tanto de paralelismo, comunicación, sincronización e indeterminismo como calendarización, que le permita identificarlos en los diferentes modelos y lenguajes que se vayan generando en programación concurrente.

BIBLIOGRAFIA

1.- Communicating Sequential Processes
C.A.R. Hoare, ACM Communications, Agosto 1978, Vol. 21, N. 8

2.- Synchronizing Resources
Gregory R. Andrews
ACM Transactions on Programming Languages and Systems
Octubre 1981, 405-430

3.- The Distributed Programming Language SR, Mechanisms, Design and Implementation
Gregory R. Andrews
Software- Practice and Experience, Vol. 12, 719-753

4.- Reference Manual for the ADA Programming Language
ANSI/MIL-STD-1815A-1983
United States Department of Defense

5.- Rationale for the Design of the ADA Programming Language
J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, B.A. Wichmann

6.- ADA for Experienced Programmers
A. Nico Habermann, Dewayne E. Perry
Addison-Wesley Publishing Company

7.- ADA Concurrent Programming
Narain Gehani
Prentice Hall, Inc.

8.- Programming with ADA: an introduction by means of Graduated Examples
Peter Wegner
Prentice Hall, Inc.

9.- Coordinated Computing Tools and Techniques for Distributed Software
Robert E. Filman, Daniel P. Friedman
McGraw-Hill Book Co.

10.- A Comparison of two notations for Process Communication
Jim Welsh, Andrew Lister y Eric J. Salzman
Proceedings of the Symposium on Language Design and Programming Methodology, Sydney, 10-11 September 1979.

11.- A Comparative Study of Task Communication in ADA
Jim Welsh, Andrew Lister
Software- Practice and Experience, Vol. 11, 1981

REFERENCIAS

Andrews, 1981 — Andrews, G.R. "Synchronizing resources" ACM Trans. Prog. Lang. Syst. 3, 4 (Oct. 1981), 405 - 430.

Andrews, 1982 — "The distributed programming language SR - Mechanisms, design and implementation" Softw. Pract. Exper. 12, 8 (Agost. 1982), 719 - 754.

Brinch Hansen, 1972 — "Structured multiprogramming" Commun. ACM 15, 7 (Julio 1972), 574, 578.

Brinch Hansen, 1973 — "Concurrent programming concepts" ACM Comput. Surv. 5,4 (Dic. 1973), 223-245.

Brinch Hansen, 1975 — "The programming language Concurrent Pascal" IEEE Trans. Softw. Eng. SE-1, 2 (Junio 1975), 199 - 206.

Brinch Hansen, 1978 — "Distributed processes: A concurrent programming concept" Commun. ACM 21, 11 (Nov. 1978), 934 - 941.

Cook, 1980 — Cook R.P. "MOD - A language for distributed programming" IEEE Trans. Soft. Eng. SE-6, 6 (Nov. 1980), 563 - 571.

Filman, Feldman, 1984 — Robert E. Filman, Daniel P. Friedman "Coordinated Computing Tools and Techniques for Distributed Software", McGraw-Hill Book Co., 1984

Feldman, 1979 — Feldman, J.A. "High level programming for distributed computing" Commun. ACM 22, 6 (Junio 1979), 353 - 368.

Good et al., 1979 — Good, D.L.., Cohen, R.M. y Keeton Williams "Principles of proving concurrent programs in Gypsy" In Proc. 6th ACM Symp. Principles of Programming Languages (San Antonio, Texas, Enero 29 - 31, 1979). ACM, New York, 1979, pp. 42 - 52.

Halstead, 1977 — "Elements of Software Science", Elsevier North-Holland Inc., N.Y.

Hoare, 1972

Hoare, C.A.R. "Towards a theory of parallel programming" In C.A.R. Hoare y R.H. Perrot (Eds.) Operating Systems Techniques. Academic Press, New York, 1972, pp 61 - 71.

Hoare, 1974

Hoare, C.A.R. "Monitors: an operating system structuring concept" Commun. ACM 17,10 (Oct. 1974), 549-557.

Hoare, 1978

Hoare, C.A.R. "Communicating sequential processes" Commun. ACM 21, 8 (Agosto 1978), 666-677.

Ichbiah, 1979

Ichbiah J.D., Barnes J.G.P., Heliard J.C., Roubine O., Wichmann B.A., Krieg-Brueckner B. "Rationale for the Design of the ADA Programming Language", 1979

Ichbiah, 1980

Ichbiah, J. "In proceeding of the ADA debut ", Dfense Advanced Research Projects Agency, Arlington, Sept. 1980

Liskov and Scheifler, 1982

Liskov,B.L. and Scheifler, R. "Guardians and actions:Linguistic support for robust distributed programs" In Proc. 9th ACM Symp. Principles of Programming Languages (Albuquerque, New Mexico, Enero 25 - 27,1982). ACM, New York, 1982,pp. 7-19.

Tsujino, Ando, 1984

Y. Tsujino, M. Ando, T. Araki y N. Tokura, "Concurrent C: A Programming Language for Distributed Multiprocessor Systems", Osaka University, Japón, Softw. Pract. and Exp., Vol. 14 (11), 1061 - 1078 (Nov. 1984).

U.S. Department of Defense, 1983

"Reference Manual for the ADA Programming Language", ANSI/MIL-STD-1815A-1983, United States Department of Defense, 1983.

Welsh Lister, 1979

Welsh, J. and Lister, A. "A comparison of two notations for process communication", Proc. of the Symp. on Lang. Design and Prog. Methodology, Sydney, 10-11 Sept. 1979

Welsh Lister, 1981 — Welsh, J. and Lister, A. "A comparative study of task communication in ADA", Softw. Pract. Expr. 11 (1981), 257 - 290.

Wirth, 1977 — Wirth, N. "Design and Implemetation of Module" Softw. Pract. Expr. 7 (1977), 67 - 84.