Programación Lógica Concurrente y Programación con Restricciones.

Carlos Jesús Gálvez Fernández: Alumno del Departamento LCC, Universidad de Málaga, España.
E-mail ***sharlythebest@bigfoot.com***

F. Javier Román Navarrete: Alumno del Departamento LCC, Universidad de Málaga, España.
E-mail ***knuria@yahoo.com***

Juan José Domínguez Duarte: Alumno del Departamento LCC, Universidad de Málaga, España.
E-mail

1. INTRODUCCIÓN

El paradigma lógico concurrente surge de los muchos intentos encaminados a paralelizar programas lógicos con la intención de mejorar su eficiencia .

ALGORITMO = LÓGICA + CONTROL

Los sistemas de programación lógica pueden verse como ***cajas negras*** generan vinculaciones de variables, y que a partir de una entrada calculan una salida.

La idea que se persigue al proponer un nuevo paradigma es mantener la lógica (semántica declarativa) y modificar el comportamiento del algoritmo (semántica operacional).

En 1974 Kowalski mostró que la lógica de las cláusulas de Horn tiene una ***interpretación procedimental***:

A <- B1, B2, ... , Bn (Cláusula de Horn)

- CLÁUSULAS (A <- B1, B2, ... , Bn) <=> definiciones de procedimiento donde:

  A <=> cabecera.

  B1, ... , Bn <=> cuerpo del procedimiento.

- OBJETIVOS (<- B1, B2, ... , Bn) <=> llamadas a procedimientos.

- ALGORITMO DE DEMOSTRACIÓN <=> intérprete del lenguaje.

- ALGORITMO DE UNIFICACIÓN <=> asignación de variables, paso de parámetros, selección y construcción de datos.

La mayoría de los lenguajes lógicos concurrentes (LLCs) son un intento de mejorar la eficiencia de los lenguajes lógicos mediante la utilización del paralelismo AND de flujo. Responden, por lo tanto, a un modelo REACTIVO y tienen una ***interpretación basada en procesos*** [van Emden y Lucena, 1982] :

- OBJETIVO ATÓMICO (Bi) <=> proceso donde si:

  Bi = p(t1, ... , tn)

  entonces:

  p <=> estado del proceso.

  t1, ... ,tn <=> estado de los datos.

- CONJUNTO DE OBJETIVOS (B1, ... , Bn) <=> red de procesos interconectados mediante sus variables lógicas compartidas.

- INSTANCIACIÓN DE VARIBLES <=> comunicación de procesos.

El comportamiento de los procesos se puede interpretar de la siguiente forma:

- El proceso en estado A se transforma en un nuevo proceso:
  A :- G | B.

  ```
  p(x):- X=f(a,Y) | q(Y).
  ```

- El proceso en estado A se transforma en k procesos concurrentes:

A :- G | B1, ... , Bk.

```
p(X):- X=f(A,B,C)| q(A),r(B),s(C).
```

- Fin del Proceso en estado A (halt):
  A :- G | true.

```
p(X):- X=f(a)|.
```

- Cambio del estado de los datos:
  A :- G | ... , A.

```
p(X):- X=f(a,Y)| Y=f(b,Z),p(Z).
```

2. TIPOS DE PARALELISMOS

Vamos a estudiar dos potentes herramientas fundamentales para implementar el paralelismo en los programas lógicos, como son el AND y OR Concurrentes. Se puede utilizar una técnica o ambas. Los lenguajes Concurrent Prolog, Parlog y GHC (Guarded Horn Clauses) implementan este tipo de concurrencia.

2.1 AND-CONCURRENTE

Es una operación que nos permite la ejecución concurrente de cláusulas. Dado un programa cualquiera:

A ← L1, L2, L3, ..., Ln.

En un lenguaje lógico se ejecutaría secuencialmente cada uno de las cláusulas. ¿Por qué no ejecutamos concurrentemente cada una de los objetivos de tal forma que L1, L2, .., Ln actúen como procesos independientes?

¿Cómo funciona el operador AND-concurrente?

Cuando un proceso termina con éxito su ejecución, seguimos ejecutando el resto, si és el último entonces el conjunto de cláusulas

termina con éxito. Si por el contrario una de los objetivos falla, entonces el resto de los procesos terminan su ejecución dándonos como resultado fallo.

La evaluación paralela de los objetivos de una conjunción tiene la ventaja de que todo el trabajo realizado es "útil", ya que para resolver una cláusula es necesario resolver todos sus subobjetivos.

2.1.1 PROBLEMAS DEL AND-CONCURRENTE

Uno de los problemas de este operador es la compartición de variables (dependencia de datos) que limita la concurrencia. Los principales modelos de implementación son:

-**No restringido**: Todos los subobjetivos se intentan en resolver en paralelo sin tener en cuenta la dependencia de variables. Requiere otro nivel más de unificación.

-**Restringido**: Se utiliza un grafo de dependencia para determinar qué subobjetivos se intentará resolver concurrentemente. Este grafo de dependencia se puede crear en tiempo de compilación (***estático***) o en tiempo de ejecución (***dinámico***). La primera opción no aprovecha el paralelismo del programa al máximo, mientras que la segunda opción sí lo hace, pero a cambio de una mayor sobrecarga del sistema.

-**De flujo (Stream)**: Los procesos con variables compartidas siguen un modelo productor-consumidor mediante la "instanciación parcial" de variables. Este modelo es el que ha tenido más éxito dentro del AND paralelismo y en el que vamos a hacer más hincapié.

Sea el siguiente programa, que calcula si una ciudad pertenece a un determinado país:

```
Ciudad(España, Málaga).
Ciudad(España, Valencia).
Ciudad(España, Murcia).
Ciudad(Francia, París).
```

Si ejecuto las cláusulas ***Ciudad(España, X), Ciudad(Francia, X)***. secuencialmente, fallaría la ejecución, puesto que no existe una ciudad que pertenezca a España y Francia al mismo tiempo. Sin embargo si la ejecución fuera concurrente, puede terminar con éxito, ya que podría ocurrir que el primer proceso instanciara X con el valor Málaga y el segundo proceso lo hiciera con el valor París. Cuando existen variables compartidas por varios objetivos este operador puede fallar.

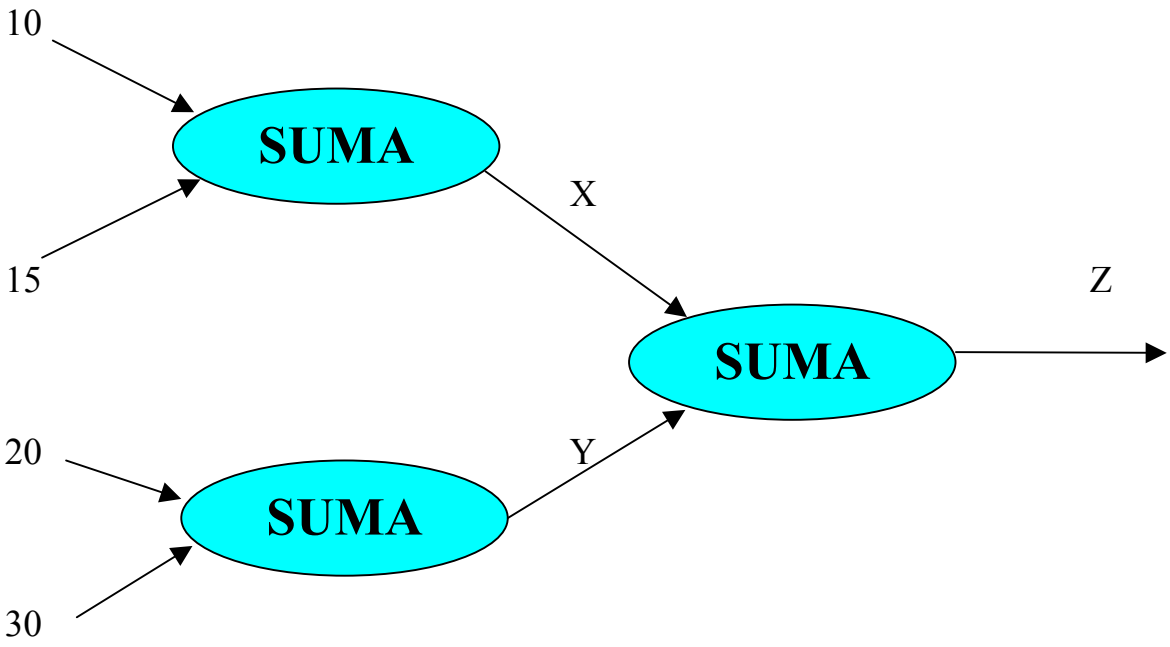
¿ Cuándo podemos utilizar AND concurrente?

1. Cuando no existen variables compartidas.
2. Cuando existen variables compartidas y sólo una cláusula instancia un valor a la variable.

Para poder implementar el concepto de variable solamente instanciada por una cláusula, utilizamos variables de sólo lectura. Se denota por el cierre de una interrogación. **X?** Representa que X es una variable de sólo lectura y se interpreta de la siguiente forma: X suspende su unificación temporalmente hasta que otra cláusula instancia un valor a X.

Ejemplo:

```
Suma(10,15,X), Suma(30,20,Y), Suma(X? ,Y? ,Z).
```



Las cláusulas siguen siendo procesos, pero ahora las variables compartidas funcionan como canales de comunicación. Un proceso con variables de sólo lectura no comienza a ejecutarse hasta que todas las variables no estén unificadas.

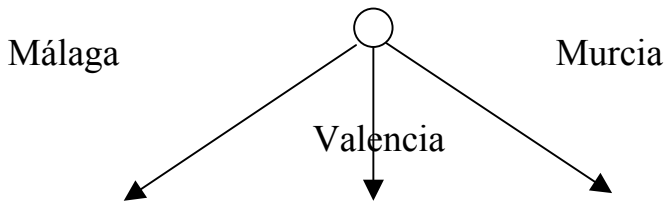
2.2 OR CONCURRENTE

Es una técnica de búsqueda dentro de un árbol de resolución, de forma que se busca concurrentemente cada una de las ramas del árbol. Cuando una de las ramificaciones (búsqueda) obtiene una solución, esa va a ser la solución del sistema y se detienen el resto de búsquedas.

Ejemplo:

Dado el problema de las ciudades y países explicado anteriormente y teniendo el siguiente objetivo:

```
Ciudad(España, Murcia).
```



Cada una de las búsquedas (ramas) se van a realizar concurrentemente. Si hiciéramos una ejecución secuencial realizaríamos una búsqueda en la primera y segunda rama, obteniendo como resultado fallo. Luego realizaría una exploración en la 3º rama obteniendo éxito. Al realizar una ejecución concurrente puedo encontrar la solución antes y no tengo que realizar búsquedas en profundidad, sino por niveles. Además tiene la ventaja de la independencia de cómputos.

2.2.1 PROBLEMAS DE OR-CONCURRENTE

El OR-Concurrente es una herramienta muy poderosa, pero no suele usarse porque es muy ineficiente. Una ejecución puede generar un árbol de resolución muy grande, por lo que puede saturar cualquier ordenador ya que no hay espacio suficiente para guardar los entornos (ligaduras de las variables). A esta complicación se le denomina explosión combinatoria. Además, si sólo se necesita una solución del problema, el trabajo realizado por la mayoría de las ramas es "inútil" ya que no concluirá en una solución (sólo una tendrá éxito). Se han propuesto varios modelos que intentan resolver el problema de la explosión combinatoria:

-**Entorno Compartido**: Las distintas ramas comparten un único entorno.
-**Entorno no Compartido**: Para cada rama se mantiene un entorno independiente.
-**Recomputación:** Se repite el cómputo de la rama para así no tener que guardar toda la información del entorno.

Se suele utilizar una variante de esta herramienta, lo que se denomina OR concurrente limitado. Para implementar esta variante vamos a utilizar un nuevo operado, el operador ***commit***, que se denota con el símbolo "|".

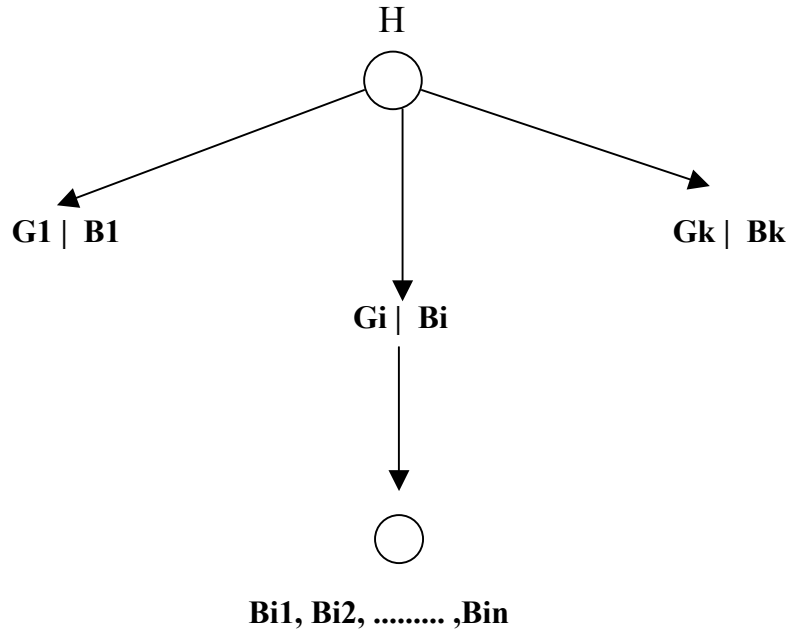
La idea del operador ***commit*** es comprometerse con una determinada cláusula tan pronto como sea posible. Ahora los programas se van a representar de la forma:

$$H \leftarrow G1, G2, G3, ..., Gm \mid B1, B2, ... ,Bn.$$

La interpretación del operador commit es que H puede reducirse a B1, B2, ..., Bn si el conjunto de guardas G1, ..., Gm termina conéxito su ejecución.

Ejemplo:

H1 ← G1 | B1.
Hi ← Gi | Bi.
Hk ← Gk | Bk.



En el momento que una de las búsquedas cumpla todas las guardas, elegimos esa cláusula y continuamos la búsqueda, olvidándonos del resto. Por lo tanto el operador commint funciona como un corte.

2.3 OTRAS TÉCNICAS:

Aparte de estas dos formas de paralelizar programas lógicos, hay otras técnicas para conseguir concurrencia en los programas lógicos, como son paralelizar el acceso a la base de datos (hechos) y paralelizar el algoritmo de unificación.

3. TIPOS DE INDETERMINISMO. SOLUCIONES AL INDETERMINISMO:

Existen dos tipos fundamentales, el ***indeterminismo don´t care*** y el ***indeterminismo don´t know***, el cuál se divide en indeterminismo AND y OR.

-Indeterminismo don´t care: Al utilizar el operador commit, intentamos comprometernos lo antes posible con una cláusula, esto nos va a generar un indeterminismo, ya que para un conjunto de entradas podemos tener varias salidas distintas. Asumimos que independientemente de la cláusula que elija voy a obtener la misma solución correcta. Por lo tanto sé que existe el indeterminismo, pero no me preocupo (don´t care).

-Indetermismo don´t know: el programador no necesita conocer cuál de las posibilidades indicadas en el programa es la correcta; es responsabilidad de la ejecución del programa elegir la transición adecuada (la clave: backtracking). Existen dos subtipos:

-Indeterminismo AND: Posibilidad de elegir un objetivo u otro de la conjunción de objetivos para ser reducido.

-Indeterminismo OR: Posibilidad de elegir una cláusula u otra entre todas la que pueden usarse en la reducción de un determinado objetivo.

4. INTRODUCCIÓN A LA PROGRAMACIÓN LÓGICA CON RESTRICCIONES

Prolog tiene varias limitaciones derivadas del principio de unificación síntáctica, tales como la perdida de simetría en los parámetros de entrada salida y la imposibilidad de dar respuestas infinitas.

Supongamos ahora, para ilustrar esto, un programa Prolog que se satisface si la lista que se le argumenta esta ordenada:

```
Ord([]).
Ord([X]).
Ord([X,Y|Z])← X <= Y, ord([Y|Z]).
```

nos encontramos con un problema si hacemos una consulta del tipo:

```
?- ord([X,Y,Z]).
```

siendo X, Y, Z variables sin instanciar, puesto que daría un fallo al intentar comprar las variables X e Y.

Para solucionar este problema podemos insertar infinitos predicados de la forma 0<1, 1<2, 2<3, 3<4, ...pero nos conduciría a infinitas soluciones. Otra forma de encontrar solución consistiría en añadirle una nueva propiedad, que será la que adoptemos para nuestro ejemplo, al interprete del lenguaje:

> **Suposición:** el interprete del lenguaje es capaz de reconocer que la expresión $X \leq Y$ tiene al menos una solución.

Ya podemos resolver la consulta y siendo $\exists(X \leq Y \wedge Y \leq Z)$ la suposición que tenemos que hacer para encontrar una solución, podemos considerar como respuesta la propia suposición.

Tenemos ahora dos extensiones deseables de la programación lógica clásica:

-Símbolos de predicado con semántica en el interprete
-Respuestas que no son ecuaciones.

5. LENGUAJES LÓGICOS CON RESTRICCIONES

Englobando estas dos extensiones tenemos la programación lógica con restricciones (Constraint Logic Programing o simplemente CLP) que extiende la Programación Lógica clásica con nuevos elementos que el interprete del lenguaje sería capaz de reconocer. Su semántica está implementada en el interprete y no dada por el usuario programador:

- Símbolos de constantes.
- Símbolos de funciones.
- Símbolos de predicados.

Así todo lenguaje lógico con restricciones está parametrizado por una interpretación de esos símbolos. Como ejemplo tenemos el lenguaje CLP ***CLP(Z)***, asociado al cual tenemos una interpretación *Z* cuyas constantes son los enteros y los predicados y funciones son los definidos

para estos (+, ×, =, <, ...). Así un programa lógico con restricciones tiene la forma:

$A \leftarrow R_1, ..., R_N, A_1, ..., A_M.$

Donde las R_i son fórmulas interpretadas (restricción) por el lenguaje y las A_i son predicados cuya semántica se la da el programador.

La semántica operacional de los programas lógicos está basada en la tradicional, pero incluye nuevos conceptos, el más importante de los cuales es el almacén de restricciones (***constraint store***), que es una restricción global formada por restricciones y renombramientos que evoluciona en la forma TRUE $\rightarrow$ TRUE $\wedge$ R_1 $\rightarrow$ TRUE $\wedge$ R_1 $\wedge$ $R_2 \rightarrow$..., y cuyo contenido será la salida del algoritmo. La interacción con esta restricción global se da por medio de instrucciones de consulta y modificación (***ask y tell***).

6. PROGRAMACIÓN CONCURRENTE CON RESTRICCIONES

La integración del paradigma lógico concurrente y con restricciones tiene una mayor importancia desde la tesis de Saraswat, en la que introducía los lenguajes ***cc*** (***Concurrent Constraint***), donde la comunicación y sincronización de procesos se realiza a través de una restricción global y mediante expresiones ***ask-tell***. Notamos algunas características de la comunicación basada en restricciones:

- Las restricciones especifican información parcial.
- La comunicación es aditiva.
- Los canales son genéricos.

También se añaden símbolos que representan técnicas del paralelismo en programación lógica, tales como $\rightarrow$, que representa el indeterminismo ***don't care***, o $\Rightarrow$, que representa el ***don't know***. También incorpora operadores ***commit***, que permiten proteger acciones básicas.

Entre los distintos lenguajes ***cc*** encontramos JANUS, LUCY, PROCOL, AKL, GOFFIN, y para la implementación de éstos encontramos diversas propuestas de máquinas abstractas, tales como la K-machine, BERTRAND o CLAM.