UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

Facultad de Ingeniería

DIAGNOSTICADOR DE PROGRAMAS PROLOG PURO CON NEGACION POR FALLA

T E S I S

Que para obtener el Título de:

INGENIERO EN COMPUTACION

P r e s e n t a

ALEJANDRO MANCILLA ROSALES

DIRECTOR DE TESIS:

DR. DAVID A. ROSENBLUETH LAGUETTE

MEXICO, D.F. 2000

A mis padres: Lucina y Benito.
A mis hermanos: Emilio, Teresa, Benito, Ana Lilia,
José Luis, Félix y Gabriel.
A mis cuñados: Ana Teresa, Arnulfo y Teresa.
A mis sobrinos: Analuz, Rodrigo, María del Mar y Gosbinda.
A Zamanta, por su cariño y motivación.

Índice General

Capítulo 1

Introducción

En esta tesis se presentará una herrramienta computacional interactiva que facilite la diagnosis de errores en un programa de computadora realizado en el lenguaje de programación Prolog (programa Prolog).

Serán dos los tipos de programas Prolog que podrán ser diagnosticados por medio de dicha herramienta: programas Prolog puro y programas Prolog puro con negación por falla.

Los tipos de errores que permitirá diagnosticar esta herramienta son tres. Programas que terminan con solución incorrecta, programas que terminan y que no encuentran solución, y programas que no terminan, dado un límite superior de recursión establecido por el programador.

Por otra parte, se desarrollará un programa Prolog puro con negación por falla que permita calendarizar actividades para procesos químicos con recursos limitados. Se hará uso del diagnosticador para revisar y corregir los errores que se encuentren en el desarrollo del calendarizador.

1.1 Descripción del problema

Cuando se escribe un programa de computadora, la información que se tiene acerca del problema que se quiere resolver suele ser vaga e incompleta. Por tal motivo, el problema se comienza a resolver a partir de suposiciones mediante las cuales permitan abstraerlo y así poder obtener un modelo del mismo. De esta manera, y con base en el modelo obtenido se inicia la construcción del programa de computadora que resuelva dicho problema.

Con fundamento en lo antes expuesto, se puede decir que un programa

de computadora se construye a partir de un conjunto de suposiciones sobre un problema dado, donde dicho conjunto puede llegar a ser arbitrariamente complejo. Una consecuencia de tales suposiciones es el comportamiento del programa. En general, no podemos anticipar con precisión todos los comportamientos de un programa dado. Esto se manifiesta, principalmente, en los resultados de indecidibilidad que cubren los aspectos más interesantes del comportamiento de programas para cualquier sistema de programación no trivial [Jr.67].

La localización de un error (bug) en un programa de computadora es una actividad que puede llegar a demandar una cantidad considerable de tiempo. Esto se debe, principalmente, a que las suposiciones acerca del problema planteado están incorrectas o incompletas. Además, es práctica común que la diagnosis de un programa sea realizada por la misma persona que lo escribió, por lo que localizar un error es aún más complejo ya que resulta complicado adoptar un enfoque diferente al que se tomó para obtener una o varias de esas suposiciones.

Por todo esto, se puede decir que el problema de la diagnosis de programas está presente en cualquier lenguaje de programación usado para comunicarnos con la computadora, y de aquí que se debiera resolver a un nivel abstracto, con ayuda de una herramienta computacional interactiva que facilite localizar e incluso ayudar a corregir el error en el programa.

1.2 Objetivo de la tesis

El objetivo de la presente tesis es el estudio de la diagnosis de programas escritos en Prolog puro con negación por falla. Como resultado de dicho estudio se desarrollará un programa en Prolog, mediante el cual se pueda diagnosticar un programa Prolog puro con negación por falla. Con esto queremos indicar que el lenguaje Prolog se usará tanto para escribir el diagnosticador como los programas a diagnosticar.

Para lograr el objetivo propuesto intentamos dar respuesta a la siguiente pregunta:

¿Cómo se identifica un error en un programa que se comporta incorrectamente?

Para ello se utilizarán los algoritmos proporcionados por Ehud Y. Shapiro [Sha82]. Estos algoritmos están diseñados para identificar tres tipos de errores

en un programa Prolog puro que se comporta incorrectamente:

- terminación con solución incorrecta.
- terminación con solución no encontrada.
- no terminación.

Dichos algoritmos tienen una característica adicional, que son interactivos, ya que cuestionan al programador (o a cualquiera que conozca el comportamiento deseado del programa a diagnosticar) por la correctez de resultados intermedios de llamadas a procedimientos presentes en el programa, utilizando esas respuestas para diagnosticar el programa y determinar el error.

A continuación se muestran algunos ejemplos de preguntas que estos algoritmos realizan durante el proceso de la diagnosis:

- ¿Es `[a,b,c]` una salida correcta para append con entrada `[a,b]` y `[c,d]`?
- ¿Cúal es la salida correcta de `partition` con entradas `[2,5,3,1]` y 4?
- ¿Es válido para `sort`, con entrada `[1,2,3]`, llamarse recursivamente con entrada `[1,1,2,3]`?

El primer tipo de preguntas las realiza el algoritmo que diagnostica la terminación con solución incorrecta; preguntas del segundo tipo las realiza el algoritmo que diagnostica la terminación con solución no encontrada; y el tercer tipo de preguntas las realiza el algoritmo que diagnostica la no terminación de programas.

Por otra parte, estos algoritmos pueden ser aplicados a lenguajes de programación cuyo mecanismo básico de cómputo es una llamada a procedimiento (o función), pero que son insensibles a los trabajos internos del procedimiento. Es decir, los algoritmos abstraen todos los detalles de cómputo, excepto las llamadas realizadas a procedimientos, sus entradas y sus salidas. Ejemplos de lenguajes de programación con esta característica son: Prolog, Lisp, Haskell.

En esta tesis se extiende el uso de estos algoritmos a programas Prolog puro con *negación por falla*. La negación por falla es un caso especial de la suposición del mundo cerrado de las bases de datos [Rei81]. La suposición del

mundo cerrado se basa en la idea de que el conjunto de información verdadera está contenida en la base de datos, de tal manera que si buscamos alguna información y no se encuentra en dicha base de datos podemos inferir que esa información es falsa. Para hacer esta suposición efectiva, en Prolog se utiliza la regla de negación por falla.

Finalmente, se hace uso de la característica interactiva de los algoritmos para almacenar las respuestas proporcionadas por el programador. Existirá una base de datos única para cada programa diagnosticado. Dicha base de datos contiene un conjunto de hechos con un valor de verdad (`true` o `false`) asociado que se utiliza para evitar repetir preguntas realizadas con anterioridad, así como para probar que todas las entradas previas sigan obteniendo la salida deseada, ya que de lo contrario, el programa podría seguir comportándose incorrectamente.

1.3 Calendarizador de procesos químicos

Dado que el diagnosticador recibe como entrada un programa escrito en Prolog puro con negación por falla, se propuso realizar un programa con tales características que resolviera un problema práctico y con el cuál, además, se mostrara el uso potencial de esta herramienta.

El problema práctico que se propuso es el problema de la calendarización de procesos químicos. Este problema tiene características propias que lo hacen de interés general, ya que es un problema que se presenta al momento de querer maximizar el uso de un conjunto finito de recursos que permiten realizar ciertas actividades y, a la vez, realizar dichas actividades en el menor tiempo posible.

Además, este problema también es de interés computacional debido a que es un problema de los llamados NP-completos. Los problemas NP-completos poseen un espacio de búsqueda de solución extremadamente grande. Por esta razón, para tratar con este tipo de problemas, desde el principio, es necesario restringir el espacio de búsqueda y así asegurar la mejor solución posible dentro de ese espacio, más no la solución óptima del problema.

Como resultado de este trabajo se desarrolló un programa Prolog puro con negación por falla que resolviera el problema de la calendarización de procesos químicos. Para comprobar el correcto funcionamiento del programa utilizamos el diagnosticador para hacer pruebas y detectar errores en el calendarizador. Este tipo de pruebas se realizaron durante el desarrollo del

calendarizador y al final del mismo.

1.4 Estructura de la tesis

En este capítulo se presentaron los objetivos y la definición del problema a resolver en esta tesis.

En el siguiente capítulo se presentan los fundamentos básicos de la programación lógica. Además, se describen las características principales del lenguaje de programación Prolog.

Los algoritmos de la diagnosis y su implementación en Prolog, se presentan en el capítulo 3. Se explican cada uno de los algoritmos, junto con ejemplos para su mejor comprensión. En el capítulo 4, se muestra la implementación de los algoritmos de la diagnosis en Prolog puro con negación por falla.

El análisis y la solución del problema de la calendarización de procesos químicos se tratan en los capítulos 5 y 6. Ahí se explican tanto las características del problema como los métodos analíticos y heurísticos que se utilizaron para resolverlo.

En el capítulo 7 se presentan tanto los resultados como las conclusiones de este trabajo.

Se anexa el código tanto del diagnosticador de programas Prolog puro con negación por falla, como del calendarizador de procesos químicos.

Este trabajo ha sido beneficiado enormemente por el apoyo contínuo, entusiasmo y crítica constructiva de parte de mi asesor el Dr. David Rosenblueth, a quien le doy mi más sincero agradecimiento. Finalmente, también deseo agradecer al Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas (IIMAS) de la UNAM, por permitirme hacer uso de sus instalaciones y equipo de cómputo en el desarrollo de la presente tesis.

Capítulo 2

Conceptos de programación lógica y Prolog

En este capítulo se presentarán los conceptos de programación lógica y del lenguaje Prolog necesarios para hacer esta tesis autocontenida. Sin embargo, no se pretende sustituir una adecuada introducción al tema por autores tales como Apt [Apt97], Clocksin y Mellish [CM84], Lloyd [Llo87], y Sterling y Shapiro [SS94]. Los conceptos de programación lógica aquí presentados fueron tomados de *Foundations of Logic Programming* [Llo87], mientras que la descripción del lenguaje de programación Prolog está basada en *From Logic Programming to Prolog* [Apt97] y *The Art of Prolog* [SS94].

2.1 Introducción

La programación lógica es un formalismo simple, pero poderoso, útil para la programación de computadoras y para la representación del conocimiento. Este formalismo fue introducido en 1974 por Robert Kowalski. La programación lógica se desarrolló a partir de trabajos sobre la prueba automatizada de teoremas basada en el método de resolución. La mayor diferencia entre la programación lógica y dichos trabajos de prueba de teoremas es que la programación lógica puede ser utilizada tanto para probar como también para hacer computación. De hecho, la programación lógica ofrece un nuevo paradigma de la programación, cuyo primer lenguaje de programación es Prolog, desarrollado a mediados de la dcada de los 70s por un grupo dirigido por A. Colmerauer.

La programación lógica tiene como principal fundamento la lógica de primer orden. Ésta proporciona los medios para deducir consecuencias lógicas a partir de premisas. La lógica, además, nos permite expresar en un lenguaje preciso nuestro conocimiento sobre un problema.

La lógica formal ha resultado ser una herramienta útil para el hombre en la formación de su desarrollo intelectual y científico desde tiempos remotos. A pesar de que ha sido utilizada como una herramienta en el diseño, el razonamiento sobre y la programación de computadoras, el uso de la lógica como lenguaje de programación, llamada *programación lógica*, tiene apenas 25 años.

El paradigma de la programación lógica parte de un punto distinto del de la mayoría de los lenguajes de programación. En lugar de ser derivado por una serie de abstracciones y reorganizaciones del modelo de von Neumann y el conjunto de instrucciones de la computadora, se deriva de un modelo abstracto, el cual no tiene relación o dependencia con ningún modelo de máquina.

Una de las ideas principales de la programación lógica, debida a Robert Kowalski, es la que nos dice que un algoritmo consta de dos componentes independientes, la lógica y el control [Kow79]. La *lógica del algoritmo* es la declaración de *qué* problema se va a resolver, mientras que el *control del algoritmo* es la declaración de *cómo* se va a resolver. Esta separación le da la posibilidad al programador que sólo tenga que proporcionar los componentes lógicos del algoritmo y deje el control al intérprete del lenguaje. Con esto se obtienen dos lecturas de un programa lógico, una declarativa y otra procedural. La primera lectura dice *qué* hace el programa lógico, mientras que la segunda dice *cómo* lo realiza.

La programación lógica permite resolver los problemas de computación de una manera más natural, ya que se basa en la creencia de que la computadora debería ejecutar instrucciones que son fáciles de proporcionar por los humanos,en lugar de que el aprendizaje humano se exprese en términos de las operaciones de una computadora, como en algún momento de la historia se pensó que era mejor.

La programación lógica sugiere que ni siquiera instrucciones explícitas sean dadas, sino que el conocimiento sobre el problema y las suposiciones suficientes para resolverlo sean declarados explícitamente, como axiomas lógicos. Tal conjunto de axiomas constituyen una alternativa para la programación convencional. El programa puede ser ejecutado al proporcionarle un problema a la computadora, formalizado como un enunciado lógico a ser demostra-

do, llamado un enunciado *meta*. La ejecución del programa es un intento de resolver el problema, esto es, demostrar que la meta es una consecuencia lógica del conjunto de axiomas.

Desde el punto de vista de la demostración de teoremas, el único interés es establecer si una fórmula dada es consecuencia lógica de un conjunto de axiomas o no, mientras que desde el punto de vista de la programación, el interés está en los valores que toman las variables, pues éstas proporcionan la *salida* de la ejecución del programa. Esto quiere decir que en la programación lógica se pueden utilizar variables en las metas para que posteriormente reciban un valor, con lo cual se obtiene una solución al problema planteado.

2.2 Programación Lógica

A continuación se presentarán los conceptos básicos de programación lógica. La comprensión de estos conceptos ayudará a entender tanto los fundamentos de la programación lógica como el trabajo realizado en esta tesis.

- En programación lógica, la estructura básica es un término. Un *término* es una constante, una variable o un término compuesto.

- Una *constante* denota un individuo particular tal como un número o una palabra. Una constante se escribe como una secuencia ya sea de dígitos o de caracteres alfanuméricos; en el segundo caso inicia con una letra minúscula.

- Una *variable* denota un objeto definido, pero sin identificarlo. Una variable se escribe como una secuencia de caracteres alfanuméricos que se distinguen por comenzar con una letra mayúscula o el carácter "_". Las variables en programas lógicos se comportan de manera distinta de las variables en lenguajes de programación convencionales. En programación lógica, las variables representan valores desconocidos, muy parecido a las matemáticas. Los valores asignados a las variables son términos (expresiones). Estos valores son asignados por medio de ciertas sustituciones llamadas unificadores más generales (mgu).

- Un término compuesto consiste en un functor (llamado *functor principal* del término) y una secuencia de uno o más términos, llamados *argumentos*. Un *functor* se caracteriza por su *nombre*, el cual es un

símbolo de functor, y su *aridad* o número de argumentos. Las constantes, son un caso particular, se consideran functores de aridad cero.

Sintácticamente, los términos compuestos tienen la forma

$$f(t_1, t_2, \ldots, t_n), n \geq 0$$

donde el functor tiene nombre f, aridad n, y cada t_i es un *argumento* de f. Cuando $n = 0$, los paréntesis se omiten. Un functor f de aridad n se denota como f/n. Los functores que tienen el mismo nombre, pero diferente aridad son distintos. Los términos son *aterrizados* (ground) si no contienen variables; de otra manera son *no aterrizados* (nonground).

- Un *átomo* (o *fórmula atómica*) es una expresión de la forma

$$p(t_1, t_2, \ldots, t_n), n \geq 0$$

donde p es un símbolo de predicado y $t_1, t_2, \ldots, t_n$ son términos. Si $n = 0$, entonces los paréntesis se omiten. Un átomo se interpreta como la afirmación de que la relación p existe entre los individuos llamados $t_1, t_2, \ldots, t_n$. El símbolo para un átomo puede ser cualquier secuencia de caracteres, la cual va entre comillas si existe posibilidad de confusión con otros símbolos, como variables o enteros.

- Una *sustitución* θ es un conjunto finito (posiblemente vacío) de elementos de la forma $\{X_1/t_1, X_2/t_2, \ldots, X_n/t_n\}$, donde cada X_i es una variable, cada t_i es un término distinto de X_i y las variables $X_1, X_2, \ldots, X_n$ son distintas. A cada elemento X_i/t_i se le llama una *ligadura* (binding) para X_i. θ se llama *sustitución aterrizada* si todos los t_i son términos aterrizados.

  Las sustituciones ligan variables a términos. Éstas son el único medio de asignación de valores a variables que existe dentro de la estructura de la programación lógica. Las sustituciones son generadas durante el proceso de cómputo, de esta manera la asignación de valores a variables se da implícitamente.

- Una *expresión* es un término, una literal, una conjunción o una disyunción de literales. Una *expresión simple* es un término o un átomo.

- Sea $\theta = \{X_1/t_1, X_2/t_2, \ldots, X_n/t_n\}$ una sustitución y E una expresión. Entonces $E\theta$, la *instancia* de E por θ, es la expresión obtenida a partir de E al reemplazar simultáneamente cada ocurrencia de la variable X_i

en E por el término $t_i (i = 1, \ldots, n)$. Si $E\theta$ es aterrizada, entonces $E\theta$ se llama una *instancia aterrizada* de E.

- θ es llamada un *unificador* de s y t si $s\theta = t\theta$. Si un unificador de s y t existen, decimos que s y t son *unificables*.

  Informalmente, la unificación es el proceso de hacer idénticos a términos por medio de ciertas sustituciones. Para poder unificar, dos átomos deben tener la misma estructura, si ignoramos constantes y variables.

- Sean θ y τ sustituciones. Decimos que θ es *más general que* τ si para alguna sustitución η tenemos que $\tau = \theta\eta$.

- θ es el *unificador más general* (mgu) de s y t si es un unificador de s y t y es más general que todos los unificadores de s y t.

  Intuitivamente, un mgu es una sustitución que hace dos términos iguales, pero lo hace de la "forma más general", sin ligaduras innecesarias.

- Una *ecuación* es una pareja de términos que se denota $s = t$. Una sustitución θ es una *solución* (o un *unificador*) de un conjunto de ecuaciones $\{s_1 = t_1, s_2 = t_2, \ldots, s_n = t_n\}$ si y sólo si $s_i\theta = t_i\theta$ para toda $i = 1, \ldots, n$. Dos conjuntos de ecuaciones son *equivalentes* si y sólo si ambos admiten todas y sólo las mismas soluciones.

  A continuación se definen dos transformaciones sobre conjuntos de ecuaciones - *descomposición de términos* y *eliminación de variables*. Ambas transformaciones preservan soluciones de conjuntos de ecuaciones.

- *Descomposición de términos.* Si un conjunto E de ecuaciones contiene una ecuación de la forma $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$, donde f es un símbolo de función de aridad n, entonces el conjunto

  $E' = E - \{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\} \cup \{s_1 = t_1, s_2 = t_2, \ldots, s_n = t_n\}$

  es equivalente a E.

- *Eliminación de variables.* Si un conjunto E de ecuaciones contiene una ecuación de la forma $X = t$ donde $t \neq X$, entonces el conjunto

  $E' = (E - \{X = t\})\theta \cup \{X = t\}$

donde $\theta = \{X/t\}$, es equivalente a E.

Un conjunto de ecuaciones E se particiona en dos subconjuntos: su parte *soluble* y su parte *insoluble*. La parte soluble es su máximo subconjunto de ecuaciones de la forma $X = t$, tal que X no ocurre en ninguna parte del conjunto completo de ecuaciones, excepto como lado izquierdo de esta sola ecuación. La parte insoluble es el complemento de la parte soluble. Un conjunto de ecuaciones se dice que es *completamente soluble* si y sólo si su parte insoluble es vacía.

A continuación se muestra un algoritmo de unificación. Es un procedimiento de normalización no determinística para un conjunto de ecuaciones E dado, que repetidamente escoge y ejecuta una de las siguientes transformaciones hasta que ninguna aplica o se encuentra el fallo.

- Seleccione una ecuación de la forma $t = X$, donde t no es una variable y reescríbala como $X = t$.
- Seleccione cualquier ecuación de la forma $X = X$ y bórrela.
- Seleccione cualquier ecuación de la forma
  $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$
  donde f y g son símbolos de función de aridad n y m, respectivamente; si $f \neq g$ o $n \neq m$, termine con fallo; en otro caso, si $n = 0$ borre la ecuación, si no $(n \geq 1)$ reemplácela con n ecuaciones de la forma $s_i = t_i, (i = 1, \ldots, n)$.
- Seleccione cualquier ecuación de la forma $X = t$ donde X es una variable que ocurre en cualquier otra parte mas en el conjunto de ecuaciones y además $t \neq X$. Si t es de la forma $f(t_1, \ldots, t_n)$, donde f es un símbolo de función de aridad n, y si X ocurre en t, entonces pare con fallo; en otro caso, sea $\theta = \{X/t\}$ y reemplace cualquier otra ecuación $l = r$ por $l\theta = r\theta$.

Si este procedimiento termina con éxito, el conjunto de ecuaciones que obtenemos de salida es completamente soluble. Su parte soluble define una sustitución llamada el *unificador más general* de todos los términos participantes como lados de las ecuaciones en E. Si termina con fallo, el conjunto de ecuaciones E no se satisface y, por lo tanto, no existe ningún unificador para él.

- Una *cláusula definida* o *regla* es un enunciado lógico cuantificado universalmente en todas sus variables, y tiene la siguiente forma:

$$A \leftarrow B_1, B_2, \ldots, B_k$$

donde, $k \geq 0$, y tanto A como B_i son átomos.

Declarativamente, el enunciado anterios se lee como: "A se implica por la conjunción de las B_i" ; y su interpretación procedural es: "Para resolver el problema A, primero resolver los subproblemas $B_1, B_2, \ldots, B_k$". A es llamada la *cabeza* de la cláusula, y la conjunción de las B_i, el *cuerpo* de la cláusula. Si $k = 0$, la cláusula se conoce como un *hecho* y se escribe A, omitiendo el símbolo $\leftarrow$. Un hecho se lee "A es cierta" en forma declarativa, y "el problema A ya está resuelto" en forma procedural.

- Una *submeta* es un átomo que aparece del lado derecho del símbolo $\leftarrow$.

- Una *consulta* o *meta* es una conjunción de la forma

$$\leftarrow A_1, A_2, \ldots, A_n \text{ con } n \geq 0$$

donde las A_i son submetas. En una consulta, las variables que en ella ocurren están cuantificadas existencialmente.

Si $n = 0$, la consulta se llama la *consulta vacía* y se denota con $\square$. Declarativamente, $\square$ representa una contradicción y, proceduralmente, indica que el problema $\square$ se resuelve sin hacer nada.

- Un *programa lógico* es un conjunto finito de cláusulas definidas. Un ejemplo de un programa lógico para añadir una lista a otra, es el siguiente:

```
append([],X,X)

append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs)
```

Programa 1. Programa lógico que añade una lista a otra.

El término `[X|Xs]` denota una *lista* cuya cabeza es `X`, y cuya cola es `Xs`. El resultado de unificar el término `[A,B|Y]` con el término (lista) `[1|[2|[3|[4|[]]]]]` [1] es `A=1, B=2, Y=[3,4]`. Al unificar las listas `[X|Xs]` y `[a]` se obtiene que `X=a` y `Xs=[]`. El símbolo `[]` representa la *lista vacía*.

[1] La lista posee una forma abreviada, para este caso particular la forma abreviada es: [1,2,3,4]

- La *resolución SLD* es una regla de deducción no determinística por medio de la cual las consultas son transformadas. Su origen se debe a J.A. Robinson en un artículo publicado en 1965 y fue propuesto como regla de computación por R.A. Kowalski en 1979. Técnicamente, se caracteriza como una resolución lineal sobre cláusulas definidas, usando una regla de selección. La resolución lineal es una restricción particular de la aplicación no determinística de la regla de deducción general definida en [Rob65] de manera tal que una sola cláusula fija se transforma al resolverla contra otras cláusulas en un conjunto dado.

  La resolución SLD es una restricción adicional de la resolución lineal donde (1) la cláusula fija es una consulta, (2) las cláusulas en el conjunto son definidas, y (3) una función oracular selecciona qué átomo en la consulta se resuelve y contra qué cláusula definida en el conjunto se resuelve. Así, las letras "SLD" significan, respectivamente, "Selección", "Lineal" y "Definida".

  La resolución SLD consiste en escoger un átomo $(q_i(s_i))$ en el cuerpo de la consulta

  $\leftarrow q_1(s_1), \ldots, q_n(s_n).$

  y una cláusula definida

  $h(t) : -b_1(t_1), \ldots, b_n(t_m)$

  en el conjunto dado, cuya cabeza unifique con $q_i(s_i)$ gracias a una sustitución de variables θ(i.e.,$q_i(s_i)\theta = h(t)\theta$), entonces se reemplaza el átomo por el cuerpo de la cláusula definida, aplicando la sustitución θ a toda la nueva consulta. Esto es,

  $\leftarrow (q_1(s_1), \ldots, q_{i-1}(s_{i-1}), b_1(t_1), \ldots, b_n(t_m), q_{i+1}(s_{i+1}), \ldots, q_n(s_n))\theta.$

  El proceso se repite y se detiene si el cuerpo de la consulta está vacío (éxito), o ninguna cabeza de las cláusulas definidas en el conjunto unifica con el átomo seleccionado (fallo). Existen dos opciones no deterministicas realizadas en el proceso: escoger el átomo que se reescribe en la consulta y seleccionar una cláusula de las tantas que potencialmente unifican con ese átomo. En cualquiera de los casos, la resolución SLD es consistente (i.e., no deriva soluciones incorrectas) y, ya que las decisiones son tomadas por una función de selección no determinística, también es completa (i.e., deriva todas las soluciones).

- Un *cómputo* o *cálculo* de un programa lógico P se puede describir informalmente de la siguiente manera. El cálculo comienza con alguna meta inicial A (posiblemente conjuntiva). Esta meta puede tener dos posibles resultados: éxito o fallo. Si un cálculo tiene éxito, entonces los valores finales de las variables en A son la salida del cálculo. Una meta dada puede tener varios cálculos exitosos, cada uno dando como resultado una salida diferente.

  A continuación se define el cómputo de un programa lógico. Sean
  $$N = \leftarrow A_1, A_2, \ldots, A_m, m > 0$$
  una meta (conjuntiva) y
  $$C = A \leftarrow B_1, B_2, \ldots, B_k, k \geq 0$$
  una cláusula tal que A y A_1 sean unificables con una sustitución θ. Entonces
  $$N' = \leftarrow (B_1, \ldots, B_k, A_2, \ldots, A_m)\theta$$
  se dice que es *derivada* a partir de N y de C con sustitución θ.

- Sea P un programa lógico y N una meta. Una *derivación* de N a partir de P es una secuencia (posiblemente infinita) de tercias $\langle N_i, C_i, \theta_i \rangle$, $i \geq 0$;tal que N_i es una meta, C_i es una cláusula en P con nuevos símbolos de variables que no ocurren previamente en la derivación, θ_i es una sustitución, $N_0 = N$, y N_{i+1} se deriva a partir de N_i y C_i con sustitucón θ_i para $i \geq 0$.

- Una derivación de N a partir de P es llamada una *refutación de N a partir de P* si $N_l = \square$ (la consulta vacía) para alguna $l \geq 0$. Tal derivación es finita y de longitud l, y suponemos por convención que en tal caso $C_l = \square$ y $\theta_l = \{\}$. Si existe una refutación de la meta A a partir de un programa P también decimos que P *tiene éxito* sobre A.

- Una manera más intuitiva, aunque menos completa de describir refutaciones (cómputos exitosos) de programas lógicos es mediante el *árbol de derivación*. En un árbol de derivación, los nodos son metas que ocurren en el cómputo, con sus variables instanciadas a sus valores finales, y los arcos representan la relación de invocación de metas. La refutación para la consulta ←append([a,b],[c,d],X) se muestra en el siguiente árbol de derivación.

```
←append([a,b],[c,d],[a,b,c,d])
        ←append([b],[c,d],[b,c,d])
                ←append([],[c,d],[c,d])
                        □
```

El método utilizado para realizar la refutación se conoce como reducción al absurdo. En este, el cálculo progresa vía reducción de una meta. En cada paso, tenemos alguna meta actual $\leftarrow A_1, A_2, \ldots, A_n$. Entonces, se escoge la cláusula $A' \leftarrow B_1, B_2, \ldots, B_k$ en P de manera no determinística. La cabeza de la cláusula A' unifica con A_1 con sustitución θ, y la meta reducida es $(B_1, B_2, \ldots, B_k, A_2, \ldots, A_n)\theta$. El cálculo termina cuando la meta actual es la meta vacía (que denota una contradicción).

- La *interpretación de Herbrand* para un programa P es un conjunto de átomos aterrizados construibles a partir de símbolos de P.

- El *modelo de Herbrand* para un programa P es una interpretación I tal que todas las cláusulas de P son ciertas en I.

- El *significado* de un programa P se define como el modelo mínimo de Herbrand para P. El modelo mínimo de Herbrand es muy importante, ya que los algoritmos de la diagnosis lo utilizan para encontrar un error en un programa P.

- Un *significado deseado* M de un programa Prolog es el conjunto de metas aterrizadas (libres de variables) sobre las cuales el programa debería tener éxito (modelo mínimo de Herbrand).

- Decimos que A_1 es una *solución* a una meta A si el programa regresa sobre una meta A su instancia A_1. Además, decimos que una solución A es cierta en un significado deseado M si cualquier instancia de A está en M. De otra forma, es falsa en M.

  Para obtener más información sobre la teoria de modelos de Herbrand referimos al lector a [Llo87] y [vEK76].

2.3 Prolog

Prolog es un lenguaje de programación basado en la teoría de la programación lógica. Todo programa lógico, cuando es visto como una secuencia y no como una conjunto de cláusulas, es un programa Prolog puro. El cálculo, en un programa Prolog puro, se obtiene al imponer ciertas restricciones en el proceso de cómputo de programación lógica para hacerlo más eficiente.

Considerando los objetivos de esta tesis, los aspectos más importantes de programas escritos en Prolog son la facilidad con que podemos manipularlos, razonar acerca de ellos y ejecutar otros programas escritos en Prolog.

A continuación explicaremos varios aspectos de Prolog puro, incluyendo su sintaxis, el cálculo de una meta en Prolog, y la interacción con un intérprete de Prolog, SICtus Prolog.

2.3.1 Sintaxis

La sintaxis que los programas Prolog puro utilizan tiene la siguiente convención. Tanto las cláusulas como las consultas terminan con un punto "." y en los hechos el símbolo ":-" se omite. Cabe recordar que el símbolo "$\leftarrow$" en programación lógica, se sustituye, en Prolog, por el símbolo ":-".

Por *definición* de un predicado con símbolo p en un programa dado P entendemos al conjunto de todas las cláusulas de P que usen a p en sus cabezas.

Las cuerdas que comienzan con una letra minúscula se reservan para constantes, nombres de functores o símbolos de relación. Cualquier cuerda que inicia con una letra mayúscula o "_" se identifica como una variable. Los comentarios comienzan con el símbolo "%" y terminan con un fin de línea.

Por otra parte, existen dos diferencias importantes entre la sintaxis de programas lógicos y la de programas Prolog que es necesario mencionar.

En la lógica de primer orden, y consecuentemente, en programación lógica, se supone que los símbolos de functor y de relación de diferente aridad forman clases mutuamente disjuntas de símbolos. Por el contrario, en Prolog el *mismo* nombre puede ser usado por símbolos de functor o relación con diferente aridad. Mas aún, el mismo nombre con la misma aridad puede ser utilizado por símbolos de functores y de relación. A lo anterior se le conoce con el nombre de *sintaxis ambivalente.*

El lenguaje Prolog, además, permite las llamadas *variables anónimas*, escritas como "_". Estas variables tienen una interpretación particular, ya que

cada ocurrencia de "_" en una consulta o en una cláusula se interpreta como un variable diferente. Así, por definición, una variable anónima ocurre en una consulta o en una cláusula sólo una vez. Las variables anónimas forman un elemento simple y elegante que algunas veces incrementa la legibilidad de programas de manera notoria.

La forma más simple de un enunciado es llamada un *hecho*. Los hechos son una forma de establecer que una relación existe entre objetos. Un ejemplo es:

```
append([],X,X).
```

Este hecho nos dice que si a cualquier lista `X` le añadimos la lista vacía `[]` a la izquierda, obtenemos lista `X`; o que la relación append existente entre los individuos `[]` y `X`, es `X`. Es importante mencionar que si un predicado aparece en una secuencia de hechos, éste también puede ser la cabeza de una cláusula.

El poder real de cualquier lenguaje de programación es su mecanismo de abstracción. En Prolog, la cláusula o regla proporciona el mecanismo de abstracción. Un hecho sólo puede especificar que una tupla de valores satisface un predicado, mientras que una regla puede especificar bajo qué condiciones una tupla de valores satisface un predicado.

Este mecanismo lo apreciamos mejor en la recursión. Con la recursión es posible observar la relación existente entre objetos de manera clara y concisa. En el predicado anterior, append, se observa que si el resultado de añadir `Xs` a `Ys` es `Zs`, entonces el resultado de añadir la lista `[X|Xs]` a `Ys` es la lista `[X|Zs]`. Con la recursión expresamos información general sobre la relación establecida entre los objetos.

La siguiente regla define el caso recursivo de nuestra relación append.

```
append([X|Xs], Ys, [X|Zs]):-
      append(Xs, Ys, Zs).
```

Finalmente, el programa Prolog que describe el predicado append/3 entre listas es:

```
append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]):-
      append(Xs,Ys,Zs).
```

2.3.2 El mecanismo de ejecución y retroceso

Los desarrolladores del lenguaje Prolog describen el mecanismo de ejecución y retroceso de la siguiente manera [BBP+81]:

> Para *ejecutar* una meta, el sistema busca a partir del inicio del programa la primera cláusula cuya cabeza *se relacione* o *unifique* con la meta. El proceso de unificación encuentra la instancia común más general de los dos términos, la cual, si existe, es única. Si una relación se encuentra, entonces la instancia de la cláusula relacionada es *activada* para ejecutar una por una, de izquierda a derecha, las metas (si las hay) que se encuentran en el cuerpo de la misma. Si en algún momento el sistema no puede encontrar una relación para una meta, entonces retrocede, es decir, rechaza la cláusula activada más recientemente, deshaciendo cualquier sustitución hecha por la relación de la cabeza de la cláusula. A continuación, reconsidera la meta original que activó la cláusula rechazada, e intenta encontrar una cláusula subsecuente que también se relacione con la meta.

Este mecanismo permite obtener el comportamiento no-determinístico característico de Prolog, ésta es una de las razones que justifican el hecho de realizar un programa como el calendarizador de procesos químicos.

Por otro lado, con ayuda del retroceso podemos encontrar otras posibles soluciones del programa con una entrada dada, aún cuando ya se haya encontrado una, pues la solución en la mayoría de las veces no es única. Esto también permite hacer una evaluación de las soluciones obtenidas para seleccionar la que mejor resultado proporcione.

2.3.3 Control y negación por falla

Cuando se escriben programas Prolog surgen problemas de tipo sintáctico que son irrelevantes para los programas lógicos. Tanto el orden de las submetas como el de las cláusulas debe estar determinado. El orden de las submetas en la cláusula determina el orden en que son resueltas, mientras que el orden de las cláusulas en el programa determina el orden en que son utilizadas.

Las consecuencias de estas decisiones pueden ser la diferencia en la eficiencia del desempeño de programas Prolog. Esto se debe a que si se cambia

el orden de las cláusulas en un predicado, entonces se intercambian las ramas del árbol de solución para la submeta de ese predicado, por lo tanto, se intercambia el orden en el que se encuentran las soluciones.

Además de la ordenación de submetas y cláusulas, Prolog proporciona otra manera para controlar el flujo del programa, ésta es el *corte* cuyo símbolo es "!". Un corte se inserta en un programa del mismo modo que una submeta, y la consecuencia es la eliminación de algunas ramas del árbol de solución que de otra manera estarían a su alcance. Es decir, al momento en que el intérprete encuentra un corte, esta submeta tiene éxito y restringe el árbol de solución a todas las opciones tomadas desde que la meta unificó con la cabeza de la cláusula en donde ocurrió el corte. En otras palabras, Prolog descarta cualquier otra alternativa a la cláusula cuya cabeza unificó con la meta a solucionar, por lo que al realizar un retroceso y volver encontrar el corte no es posible encontrar la solución que se localiza en otra rama del árbol.

Un predicado que puede ejemplificar el uso del corte es el siguiente:

```
member([X], [X|_]):- !.
member(X,[_|Xs]):- member(X,Xs).
```

En este caso el predicado `member/2`, solo encuentra una solución.

```
:-? member(X, [mario, juan, ricardo]).
X = mario ;
no
```

Sin embargo,

```
:-? member(mario, [mario, juan, ricardo]).
yes

:-? member(juan, [mario, juan, ricardo]).
yes

:-? member(ricardo, [mario, juan, ricardo]).
yes
```

siguen cumpliéndose.

El corte puede ser usado para implementar una versión de negación por falla. El programa siguiente define un predicado `not(X)`, que tiene éxito si la meta `X` falla.

```
not(X):- X, !, fail.
not(X).
```

donde `fail` es una submeta que falla.

La terminación de la meta `not(X)` depende de la terminación de `X`. Si `X` termina, tambien `not(X)` termina. Si `X` no termina, entonces `not(X)` puede o no terminar, dependiendo de que un nodo exitoso se encuentre antes de encontrar una rama infinita.

La implementación de negación por falla no garantiza que trabaje correctamente para metas no aterrizadas. En la mayoría de las implementaciones de Prolog, es responsabilidad del programador asegurar que las metas negadas estén aterrizadas antes de ser resueltas.

El predicado `not` es muy útil ya que nos permite definir conceptos interesantes. Por ejemplo, consideremos un predicado `disjoint(Xs,Ys)`, cierto si dos listas `Xs` y `Ys` no tienen elementos en común. Este puede ser definido como

```
disjoint(Xs,Ys):- not(member(Z,Xs), member(Z,Ys)).
```

Una propiedad interesante del predicado `not(X)` es que nunca instancía los argumentos en la meta `X`. Esto se debe al fallo explícito después de que la llamada a `X` tiene éxito, lo cual deshace todas las ligaduras hechas anteriormente por `X`.

Debido a que el predicado `not(X)` de Prolog, no es exactamente una negación, entonces se ha cambiado por el símbolo de predicado `\+`, de tal forma que no se asocie la negación del lenguaje natural con la negación por falla de Prolog.

Capítulo 3

Diagnosticador de programas Prolog puro

En este capítulo se desarrollará la descripción de un programa en Prolog que tiene la finalidad de encontrar errores en programas escritos en Prolog puro. El diagnosticador podrá analizar cualquier programa Prolog puro [Sha82]. La extensión de los algoritmos para tratar con programas que usan negación por falla se describirá en el siguiente capítulo.

3.1 Introducción

Los diagnosticadores de programas son una clase especial de intérpretes de programas denominados meta-intérpretes, los cuales son intérpretes de programas escritos en el mismo lenguaje de implementación. La capacidad de escribir fácilmente meta-intérpretes es una característica muy poderosa de un lenguaje de programación, Prolog posee esta característica. Con esta característica se obtiene acceso al proceso de computación del intérprete del lenguaje de programación y permite la construcción de un ambiente de programación integrado.

La depuración es un aspecto esencial de la programación. El propósito de los lenguajes de programación de alto nivel no está en la capacidad de escribir programas sin errores, sino en el poder de las herramientas computarizadas para soportar el proceso de desarrollo de programas. Por razones de arranque y elegancia, estas herramientas se implementan mejor en el mismo lenguaje. Tales herramientas son programas para manipular, analizar, y simular otros

programas; en otras palabras, meta-programas.

Para depurar un programa, se debe suponer que el programador tiene en mente algún comportamiento deseado del programa, y un dominio deseado de aplicación en el cual el programa debiera exhibir este comportamiento. Dados estos, comportamiento y dominio deseados, la depuración consiste en encontrar discrepancias entre el comportamiento real del programa y el comportamiento deseado por el programador.

En este diagnosticador se distinguen tres tipos de errores en el comportamiento de un programa:

1. programas que no terminan
2. programas que encuentran una solución incorrecta
3. programas que no encuentran una solución

Cada tipo de error recibe, para su diagnosis, el siguiente trato uniforme. Se define una propiedad de comportamiento erróneo, para el cual un error en el programa implica que tiene un comportamiento de tal propiedad. Este es el comportamiento que tiene que ser modificado para eliminar el error en el programa.

3.2 Diagnosis de no terminación

En general, no es posible detectar si un programa, en cualquier lenguaje, no termina; la pregunta es indecidible. Lo mejor es asignar, *a priori*, algún límite de tiempo de ejecución o alguna profundidad máxima de recursión del programa, y cancelar la ejecución si el límite es excedido. Es deseable almacenar parte de la computación, es decir, almacenar información de la pila de ejecución del programa para poder analizarlo y determinar la causa de la no terminación.

Para poder obtener información de la ejecución del programa es necesario utilizar un meta-intérprete con un límite de profundidad en el árbol de refutación que se construye al interpretar el programa Prolog.

El meta-intérprete desarrollado usa un predicado principal llamado `solve/3`, el cual, en su primera llamada, recibe como argumentos la meta a analizar, el límite de la profundidad del árbol de refutación a construir y el tercer argumento nos devuelve información acerca del árbol de refutación construido para la meta recibida como primer argumento.

3.2.1 Descripción del algoritmo

A continuación describiremos el algoritmo utilizado para la detección de ciclos (posiblemente) infinitos en un programa escrito en Prolog puro.

Con la llamada al predicado `solve(A,D,Overflow)` se inicia la detección de ciclos infinitos, donde `A` es la meta inicial, y `D` el límite superior en la profundidad de recursión. La llamada tiene éxito si una solución es encontrada sin exceder la profundidad de recursión predefinida, con `Overflow` instanciado a `true`. La llamada también tiene éxito si la profundidad de recursión se excede, pero en este caso `Overflow` contiene la pila de metas, es decir, la rama del árbol de computación que excede el límite de profundidad `D`.

La implementación de este algoritmo se muestra a continuación.

```
% solve(A,D,Overflow):-
% A tiene un arbol de computo de profundidad menor que D y
% Overflow  igual a true, o A tiene una rama en el \'arbol de
% refutacion mas grande que D, y Overflow contiene una lista
% de sus primeros D elementos.

solve(true,_,true):- !.
solve(_,0,overflow([])):- !.
solve((A,B),D,Overflow):-
     D > 0, !,
     solve(A,D,OverflowA),
     solve_conjunction(OverflowA,B,D,Overflow).
solve(A,D,true):-
     D > 0,
     predicate_property(A,built_in), !, A.
solve(A,D,Overflow):-
     D > 0,
     clause(A,B),
     D1 is D-1,
     solve(B,D1,OverflowB),
     return_overflow(OverflowB,A,B,Overflow).

solve_conjunction(overflow(S),_,_,overflow(S)).
solve_conjunction(true,B,D,Overflow):-
     solve(B,D,Overflow).
```

```
return_overflow(true,_,_,true).
return_overflow(overflow(S),A,B,overflow([(A:-B)|S])).
```

Existe un predicado llamado `solve/2`,el cual hace una llamada a `solve/3`. En caso de que `Overflow` no este instanciado a `true`, `solve/2` hace una llamada al predicado `stack_overflow/2` que realiza el análisis correspondiente.

El predicado `stack_overflow/2` recibe como entrada la meta original P y la pila S devuelta por `solve/3`. `stack_overflow/2` requiere de un individuo que le proporcione información sobre el significado deseado del programa a analizar y responda a preguntas acerca de la recursión de los predicados implicados en la saturación de la pila preestablecida. En caso de que no se encuentren ciclos, entonces se busca de manera lineal en toda la pila.

```
%%% Sobreflujo del Stack

stack_overflow(P,S):-
      writeln([' Diagnosticando ...', P]), nl,
      check_loop(S,Sloop).

check_loop(S,Sloop):-
      find_loop(S,Sloop), !, check_segment(Sloop).
check_loop(S,_):-
      check_segment(S).

find_loop([(P:-Q)|S],Sloop):-
      check_looping((P:-Q),S,Sloop).

check_looping((P:-Q),S,[(P:-Q)|S1]):-
      looping_segment((P:-Q),S,S1), !.
check_looping(_,S,Sloop):-
      find_loop(S,Sloop).

looping_segment((P:-Q),[(P1:-Q1)|S],[(P1:-Q1)|S1]):-
      check_same((P:-Q),(P1:-Q1),S,S1).

check_same((P:-Q),(P1:-Q1),S,[]):-
```

```
      same_goal(P,P1), !,
      writeln([P, ' esta divergiendo']), nl.
check_same((P:-Q),_,S,S1):-
      looping_segment((P:-Q),S,S1).

check_segment([(P:-Q),(P1:-Q1)|S]):-
      query(legal_call,(P, P1),true), !,
      check_segment([(P1:-Q1)|S]).
check_segment([(P:-Q),(P1:-Q1)|S]):-
      false_subgoal(P,Q,P1,C), !, false_solution(C).
check_segment([(P:-Q),(P1:-Q1)|S]):-
      write('diverging clause '), nl,
      write((P:-Q)).

false_subgoal(P,(Q1, Q2),P1,Q):-
      Q1 \== P1,
      check_q1(Q1,Q2,P,P1,Q).

check_q1(Q1,_,_,_,Q1):-
      query(forall, Q1, false), !.
check_q1(_,Q2,P,P1,Q):-
      false_subgoal(P,Q2,P1,Q).
```

3.2.2 Ejemplo de la diagnosis

A continuación mostramos un programa con errores que ordena números utilizando el algoritmo de ordenamiento por inserción.

```
%%%%  Programa objeto
isort([],[]).
isort([X|Xs],Ys):-
     isort(Xs,Zs), insert(X,Zs,Ys).

insert(X,[],[X]).
insert(X,[Y|Ys],[X,Y|Ys]):-
      X < Y.
insert(X,[Y|Ys],[Y|Zs]):-
```

```
insert(Y,[X|Ys],Zs).
```

Al realizar la consulta `isort([2,2],X)` observamos que el programa no termina. Si utilizamos `solve/3` para analizarlo, con una profundidad de 6, obtenemos:

```
| ?- solve(isort([2,2],X),6,S).

S =
overflow([
 (isort([2,2],[2,2,2,2,2,2]):-
       isort([2],[2]),insert(2,[2],[2,2,2,2,2,2])),
 (insert(2,[2],[2,2,2,2,2,2]):-
        insert(2,[2],[2,2,2,2,2])),
 (insert(2,[2],[2,2,2,2,2]):-
        insert(2,[2],[2,2,2,2])),
 (insert(2,[2],[2,2,2,2]):-
        insert(2,[2],[2,2,2])),
 (insert(2,[2],[2,2,2]):-
        insert(2,[2],[2,2])),
 (insert(2,[2],[2,2]):-
        2<2)
]),

X = [2,2,2,2,2,2] ?
```

Podemos observar que el programa entra en un ciclo infinito. Debido a que el algoritmo necesita revisar los resultados de llamadas realizadas "a la izquierda" de la llamada divergente antes de concluir que una cláusula está divergiendo, es más eficiente almacenar estos resultados durante el cómputo, y devolverlos cuando ocurra un sobreflujo, en lugar de recalcularlo. Por esta razón, cuando detectamos una rama con una profundidad mayor a la preestablecida la asociamos al argumento de salida del predicado `solve/3`.

Si rebasamos el límite de profundidad, entonces hacemos una llamada al predicado `stack_overflow/2`, el cual realiza la siguiente interacción con el programador.

```
Diagnosticando ...
```

```
isort([2,2],[2,2,2,2,2,2])

Es
  isort([2,2],[2,2,2,2,2,2]),insert(2,[2],[2,2,2,2,2,2])
una llamada legal? si.

Es
  insert(2,[2],[2,2,2,2,2,2]),insert(2,[2],[2,2,2,2,2])
una llamada legal? no.

error diagnosticado:
   insert(2,[2],[2,2,2,2,2,2]):-insert(2,[2],[2,2,2,2,2])
   esta divergiendo
```

Se dice que un predicado se llama cíclicamente si se llama a sí mismo con la misma entrada con que fue llamado originalmente. Por ello, podemos observar que el error en nuestro programa de ejemplo se encuentra en la llamada cíclica a:

`insert(2,[2],[2,2,2]):-insert(2,[2],[2,2]).`

Este error se corrige al intercambiar la tercer cláusula del predicado `insert/3` por la siguiente:

```
insert(X,[Y|Ys],[Y|Zs]):-
       insert(X,Ys,Zs).
```

Al ejecutar nuevamente `isort([2,2], X)` se obtiene el resultado deseado `X=[2,2]`. Hasta aquí dejamos el problema de la diagnosis de programas que no terminan y continuamos con el problema de la diagnosis de programas que terminan con solución incorrecta.

3.3 Diagnosis de terminación con solución incorrecta

El segundo tipo de error que los algoritmos permiten diagnosticar es el regreso de una solución incorrecta. Un programa puede regresar una solución falsa sólo si tiene una cláusula falsa.

Una cláusula C es falsa con respecto a un significado deseado M si tiene una instancia cuyo cuerpo es cierto en M y cuya cabeza es falsa en M. Tal instancia es llamada un contra ejemplo de C.

3.3.1 Descripción del algoritmo

Dado un árbol de prueba sin variables correspondiente a una solución incorrecta, podemos encontrar una instancia falsa de una cláusula recorriendo el árbol de prueba en modo post-orden.

Al hacer el recorrido, probamos si cada uno de los nodos en el árbol de prueba es cierto. Si encontramos un nodo falso, entonces la cláusula cuya cabeza es el nodo falso y cuyo cuerpo es la conjunción de los hijos de dicho nodo, es un contra ejemplo de la cláusula en el programa. Dicha cláusula del programa es falsa y se debe modificar o eliminar.

En este algoritmo se supone que existe un *oráculo* que puede responder preguntas acerca del significado deseado del programa. El oráculo es una entidad externa al algoritmo de diagnosis. Puede ser el programador quien puede responder a las preguntas acerca del significado deseado del programa, u otro programa que ha mostrado tener el mismo significado que el significado deseado. La segunda situación puede ocurrir en el desarrollo de nuevas versiones de un programa mientras se utiliza la versión anterior como oráculo.

Una de las principales preocupaciones con algoritmos de diagnosis es la mejora de la complejidad de las preguntas, esto quiere decir, reducir el número de preguntas que requerimos para diagnosticar el error.

El algoritmo de diagnosis para soluciones falsas que se utiliza en esta implementación se conoce como divide-y-pregunta (*divide-and-query*). El algoritmo progresa al dividir el árbol de prueba en dos partes, aproximadamente iguales, y preguntando por el nodo que se encuentra en el punto de división. Si el nodo es falso, el algoritmo se aplica recursivamente al subárbol cuya raíz es este nodo. Si el nodo es cierto, su subárbol se elimina del árbol de prueba y se reemplaza por *true*, y un nuevo punto medio se calcula.

El programa Prolog que implementa este algoritmo es el siguiente.

```
%%% Interprete que calcula el punto medio del arbol de prueba
%% fpm - encuentra el punto medio
fpm(((A,B),Wab), M, W):- !,
   fpm((A,Wa), (Ma,Wma), W), fpm((B,Wb), (Mb,Wmb), W);
   Wab is Wa + Wb,
```

```
   check_wma(Wma, Wmb, Ma, Mb, M).
fpm((A,0), (true,0), _W):-
   predicate_property(A, built_in), !, A.
fpm((A,0), (true,0), _W):-
   fact(A,true), !.
fpm((A,Wa), M, W):-
   clause(A,B), fpm((B,Wb), Mb, W),
   Wa is Wb +1,
   check_wa(Wa, W, A, B, Mb, M).

%% check_wma - Revisa el peso medio de un nodo
check_wma(Wma, Wmb, Ma, _Mb, (Ma, Wma)):-
   Wma >= Wmb, !.
check_wma(_Wma, Wmb, _Ma, Mb, (Mb, Wmb)).

%% check_wa - Revisa el peso de un nodo
check_wa(Wa, W, _A, _B, Mb, Mb):-
   Wa > (W+1)/2, !.
check_wa(Wa, _W, A, B, _Mb, ((A:-B), Wa)).
```

La primera cláusula del predicado `fpm/3` calcula el nodo más pesado regresado por las llamadas recursivas de los hijos, y el peso total del nodo. La segunda cláusula elimina las metas que son predicados del sistema (*built-in*). La tercera cláusula corta del árbol las metas que están en $M' \subseteq M$, esto es, que están en el significado del programa. La cuarta cláusula trata con metas unitarias (*hechos*) al activar una cláusula, y también decide cuándo está en la mitad superior o en la mitad inferior del árbol de refutación y regresa su salida correspondiente.

3.3.2 Ejemplo de la diagnosis

Analicemos el siguiente programa con errores de ordenamiento por inserción.

```
isort([], []).
isort([X|Xs], Zs):-
     isort(Xs, Ys), insert(X, Ys, Zs).

insert(X, [], [X]).
```

```
insert(X, [Y|Ys], [Y|Zs]):-
      Y > X, insert(X, Ys, Zs).
insert(X, [Y|Ys], [X,Y|Zs]):-
       X =< Y.
```

Primero probamos isort con la entrada [2,1,3],

```
|?- isort([2,1,3], Xs).

 Xs = [2,3,1]
```

y obtenemos un resultado erróneo.
Ahora revisamos el programa con el diagnosticador.

```
|?- isort([2,1,3],X).
Resolviendo ...
Solucion: isort([2,1,3],[3,1]) ;
ok? n.

Error: solucion incorrecta isort([2,1,3],[3,1]).

diagnosticando ...

Consulta: isort([3],[3])? s.

Consulta: isort([1,3],[3,1])? n.

Consulta: insert(1,[3],[3,1])? n.

Consulta: insert(1,[],[1])? s.

Error diagnosticado:
      insert(1,[3],[3,1]):- 1=<3, insert(1,[],[1])
      es falsa.
```

Después de analizarlo se percibe que el error se encuentra en la cláusula

```
insert(1,[3],[3,1]) :-
      3 > 1, insert(1, [], [1])
```

Así, se observa que los argumentos de > están intercambiados. Se corrige el error, y se prueba de nuevo el predicado `isort/2`,

```
|?- isort([2,1,3], Xs).

Xs = [1,2,3]
```

Ahora se obtiene la salida correcta.

Después de la descripción de la implementación del algoritmo que permite encontrar errores en un programa que devuelve soluciones incorrectas, se pasará a describir el último de los algoritmos y su implementación en Prolog.

3.4 Diagnosis de terminación con solución no encontrada

El tercer tipo de error identificable es una solución no encontrada. Diagnosticar una solución no encontrada es más difícil que los dos errores anteriores.

Para un lenguaje de programación determinístico, si y es una salida correcta para un predicado p con entrada x, pero el cómputo de p con x termina y regresa una salida diferente a y, entonces esta salida es incorrecta y los algoritmos para la diagnosis de terminación con salida incorrecta son aplicables. Para un lenguaje de programación no determinístico, sin embargo, puede suceder que cualquier cálculo en p con x termine y regrese una salida correcta en M, pero que ningún cálculo regrese y. Tal programa se dice que *falla de manera finita* sobre el predicado p con entrada x y salida y. Este comportamiento necesita un tratamiento especial para lenguajes de programación no determinísticos, como es el caso de Prolog.

Se dice que una cláusula cubre una meta A con respecto a un significado deseado M si tiene una instancia cuya cabeza es una instancia de A y cuyo cuerpo está en M. Se puede mostrar que si un programa P no encuentra una solución con respecto a un significado deseado M, entonces hay una meta A en M que no se cubre por ninguna cláusula en P.

3.4.1 Descripción del algoritmo

Diagnosticar una solución no encontrada impone una responsabilidad mayor en el oráculo, pues no sólo tiene que saber cuándo una meta tiene una solución

sino también debe proporcionar esa solución, si es que existe. Usando tal oráculo, una meta no cubierta puede ser encontrada como sigue.

Al programa se le proporciona una solución no cubierta, es decir, una meta en el significado deseado M del programa P, para la cual P falla. El algoritmo comienza con la solución inicial no cubierta. Para cada cláusula cuya cabeza unifica con ella, verifica, usando el oráculo, si el cuerpo de la cláusula tiene una instancia en M. Si no hay tal cláusula, la meta no se cubre, y el programa termina. De otra forma, el algoritmo encuentra una meta en el cuerpo que falla. Al menos una de ellas debe fallar,si no el programa hubiera resuelto el cuerpo y de ahí la meta, en contraste a nuestra suposición. Este algoritmo se aplica recursivamente a dicha meta.

Una implementación en Prolog de este algoritmo se muestra a continuación.

```
% Diagnosis de un procedimiento incompleto

missing_solution(A) :-
 writel(['Error. Solucion no encontrada ',A,'.']),
 nl, write('Diagnosticando ..'), query(exists,A,true),
 missing(A,X).

missing(A,X):-
       \+solve(A,true), !, ip(A,X),
       handle_error('atomo no cubierto',X).
missing(A,X):-
       write('Llamada invalida a ip'), nl.

ip(A, X) :-
  clause(A, B), check_ip1(B,X).

check_ip1(B, X):-
         ip1(B, X), !, true.
check_ip1(B, A).

ip1((A,B), X) :- !,
   query(exists, A, true), check_a(A, B, X).
ip1(A, X) :-
   query(exists, A, true), A , !,
```

```
   break( ip1(A, X) ).
ip1(A, X):-
   query(exists, A, true), ip(A, X) ).

check_a(A, B, X):-
       A, ip1(B,X), !.
check_a(A, _B, X):-
       \+A, ip(A, X).
```

3.4.2 Ejemplo de la diagnosis

Este tipo de comportamiento, terminación con solución no encontrada, lo presenta el siguiente programa.

```
:- dynamic isort/2, insert/3.
%%%%%  Programa objeto

isort([],[]).
isort([X|Xs], Zs):-
     isort(Xs, Ys), insert(X, Ys, Zs).

insert(X, [Y|Ys], [X,Y|Ys]):-
      X =< Y.
insert(X, [Y|Ys], [Y|Zs]):-
      X > Y, insert(X, Ys, Zs).
```

Si ejecutamos el programa observamos el siguiente comportamiento.

```
| ?- isort([3,2,1],X).

no
```

Ahora, si ejecutamos el diagnosticador a este programa, observamos:

```
|?- isort([3,2,1],[1,2,3]).
Resolviendo ...
no hay (mas) solucion(es)
```

```
ok? no.

Consulta: isort([3,2,1],[1,2,3])? si.
Error: solucion no encontrada isort([3,2,1],[1,2,3]).

Diagnosticando ...

Resolviendo ...

Consulta: isort([2,1],A)? si.
Cual A? [1,2].

Consulta: isort([1],A)? si.
Cual A? [1].

Consulta: isort([],A)? si.
Cual A? [].

Consulta: insert(1,[],[1])? si.


error diagnosticado: insert(1,[],[1]) no es cubierto
```

A partir de esta información uno se puede percatar que al predicado `insert/3` le hace falta el caso base:

```
insert(X,[],[X]).
```

Después de corregir este error y volver a ejecutar el programa se obtiene la salida correcta.

```
| ?- isort([3,2,1],X).

X = [1,2,3] ?

yes
```

Capítulo 4

Diagnosticador de programas Prolog puro con negación por falla

4.1 Introducción

En el capítulo anterior se mostraron los algoritmos que permiten identificar los tres tipos de errores mencionados en programas Prolog puro. En este capítulo se extenderá el uso de estos algoritmos a programas Prolog puro con negación por falla.

La posibilidad de extender las capacidades de los algoritmos para realizar la diagnosis de programas Prolog puro con negación por falla permitirá extender también el uso potencial del diagnosticador, ya que la mayoría de los programas Prolog que se realizan hacen uso de elementos tales como el corte (!) para aumentar la rapidez de ejecución de los mismos.

4.2 Extensión a negación por falla

Los algoritmos hasta ahora descritos solo diagnostican programas Prolog puro. Para que los algoritmos anteriores puedan tratar con la negación por falla hace falta lo siguiente.

Después de analizar los algoritmos que diagnostican terminación con solución incorrecta y terminación con solución no encontrada, se observó que los algoritmos son duales y que si agregamos las siguientes líneas de código a los

predicados `ip` y `fp` es suficiente para tratar con la negación por falla.

```
fp(\+(A),X):-  ip(A,X).

ip(\+(A),X):-  fpm((A,W),_,0), fp(A,W,X).
```

Una meta `\+(A)` tiene éxito si falla de manera finita; de aquí que si la meta `\+(A)` tiene éxito al encontrar una solución incorrecta, entonces indica que `A` falla al encontrar una solución incorrecta. Por lo tanto, esto justifica la cláusula `fp(\+(A),X):- ip(A,X)`

De manera similar, si `\+(A)` falla al no encontrar solución, entonces indica que `A` tiene éxito al no encontrar solución. Por lo tanto, la cláusula `ip(\+(A),X):- fpm((A,W),_,0), fp(A,W,X)` es correcta. Es necesario llamar al predicado `fpm` para calcular el peso `W` del árbol de prueba, pues es un argumento necesario para el predicado `fp`.

4.2.1 Ejemplo de diagnosis con negación por falla

Consideremos el siguiente programa que encuentra un elemento en la diferencia simétrica de dos listas.

```
difference(X, Ys, Zs):-
          miembro(X, Ys), \+(miembro(X, Zs)).
difference(X, Ys, Zs):-
          miembro(X, Zs), \+(miembro(X, Ys)).

miembro(X, [Y|Ys]).
miembro(X, [Y|Ys]):-
       miembro(X, Ys).
```

Para probar que el comportamiento del predicado es correcto, intentemos con un ejemplo:

```
| ?- difference(1,[1,2],[2,3]).

no
```

El programa no encuentra ningún elemento diferente en las dos listas. Para facilitar la diagnosis del programa se ejecuta el diagnosticador de programas Prolog puro con negación por falla, con una de las posibles soluciones.

```
|?- difference(1,[1,2],[2,3]).
Resolviendo ...
no hay solucion(es)
ok? no.

Pregunta: difference(1,[1,2],[2,3])? si.
Error: solucion no encontrada difference(1,[1,2],[2,3]).
Diagnosticando ...
Resolviendo ...

Pregunta: miembro(1,[1,2])? si.

Pregunta: miembro(1,[2,3])? no.


Error diagnosticado:
      miembro(1,[2,3]):-true es un atomo no cubierto
```

Con esto, uno se da cuenta que el caso base del predicado `miembro` está incorrecto. Por tal motivo, se modifica por el hecho `miembro(X,[X|Xs]).` y con ello se obtiene la solución correcta.

```
| ?- difference(X,[1,2],[2,3]).

X = 1 ? ;

X = 3 ? ;

no
```

Capítulo 5

Calendarizador de procesos químicos

En este capítulo se describen las características del problema de la calendarización de procesos químicos, además de la implementación en Prolog puro con negación por falla. También se definen algunos conceptos necesarios para la mejor comprensión del problema de la calendarizacón de procesos. Las definiciones incluídas en este capítulo fueron tomadas del libro *Introduction to Sequencing and Scheduling* [Bak74]

La *calendarización* o *programación de actividades* es la asignación de recursos en el tiempo para realizar una colección de tareas. La calendarización nos proporciona elementos tales como principios, modelos, y técnicas que nos facilitan el procedimiento para determinar un calendario.

5.1 Introducción

El problema de la calendarización de actividades se presenta cuando la naturaleza de las tareas a calendarizar ha sido descrita y la disponibilidad de los recursos ha sido determinada. Es decir, la tarea de la calendarización inicia cuando han sido respondidas preguntas tales como:

- ¿Qué producto o servicio será proporcionado?
- ¿En qué cantidad será producido?
- ¿Qué recursos estarán disponibles?

La respuesta dada a cada una de ellas determina los límites dentro de los cuales la calendarización se debe realizar sobre un periodo de tiempo establecido.

En la calendarización se transforman las metas y restricciones definidas en la toma de decisiones a una función objetivo y restricciones explícitas, respectivamente, en la descripción del problema planteado.

Idealmente, la función objetivo debe constar de todos los costos involucrados en el sistema que dependen de las decisiones de calendarización. Sin embargo, en la práctica tales costos a menudo son difíciles de cuantificar o incluso de identificar plenamente. Son tres los tipos de metas en la toma de decisiones que parecen ser relevantes en la calendarización:

- El uso eficiente de los recursos
- La rápida respuesta a las demandas, y
- La finalización en una fecha cercana a la establecida por la fecha límite

Frecuentemente, una medida importante relacionada al costo de la ejecución del sistema, tales como el tiempo ocioso de las máquinas, el tiempo de espera de los trabajos o el tiempo necesario para la culminación del trabajo se utilizan como sustitutos del costo total del sistema.

5.2 Restricciones del problema

Comúnmente, se encuentran dos tipos de restricciones factibles en los problemas de calendarización. Primero, la existencia de límites en la capacidad de los recursos disponibles y, segundo, las restricciones tecnológicas en el orden que deben ser realizadas las tareas. Una solución a un problema de calendarización es cualquier solución factible de estos dos tipos de restricciones, así "resolver" un problema de calendarización consiste en responder a dos tipos de preguntas:

- *¿Qué recursos serán asignados para ejecutar cada tarea?*
- *¿En qué momento será realizada cada tarea?*

En otras palabras, la esencia de los problemas de calendarización consta tanto de decisiones de asignación como de secuenciación.

La *secuenciación* consiste en establecer el orden en el cual las actividades se realizarán. Cuando hay un solo recurso, la asignación del mismo está determinada totalmente por el orden en el que se realizan las actividades. Por esta razón, la parte de secuenciación se considera como un caso particular del problema de calendarización donde un ordenamiento de los trabajos determina completamente un calendario. La *asignación de recursos*, por otra parte, consiste en determinar qué recurso será asignado a cada tarea para su realización.

La mayoría de los resultados analíticos han sido dirigidos a problemas de longitud de tiempo total (*makespan*) requerido para completar todos los trabajos, por su relativa sencillez. Desde el punto de vista práctico, el énfasis en esta longitud es bastante razonable, ya que para resolver un procedimiento heurístico consistente en tareas ininterrumpibles se resuelve primero el problema de asignación de recursos y después el problema de la secuencia.

5.3 Elementos del problema

Los *elementos* de un problema de calendarización de actividades son un conjunto de recursos y una colección de trabajos o procesos a ser calendarizados. Cada proceso consiste en varias operaciones o etapas.

La formulacón más común del problema de calendarización especifica que cada proceso tiene exactamente m etapas, una para cada recurso, aunque es posible tener un número arbitrario de etapas en un proceso dado. No existe dificultad para describir, con esta formulación, casos generales en los cuales un proceso pueda requerir más de una vez el mismo recurso en su secuencia de etapas.

En el caso de este problema particular, la calendarización de procesos químicos, se tiene un conjunto de *recursos* tales como personal y piezas de equipo para compartir entre varios procesos. Un *proceso* está formado por una serie de etapas dependientes entre sí. Cada *etapa* tiene una precedencia explícita para su realización dentro de un proceso dado. Además, se tiene una restricción que no ocurre en problemas de calendarización de otros tipos de actividades: una vez comenzado un proceso, no puede existir ningún retraso en la realización de la secuencia de etapas de dicho proceso, ya que esto puede provocar una reacción química no deseada.

A menudo, es necesario un modelo formal para ayudar en la toma de decisiones de problemas de calendarización. Uno de los modelos más simples

y más ampliamente usado es el *diagrama de Gantt*, el cual es una representación gráfica de relaciones entre procesos de un mismo calendario. En su forma básica, el diagrama de Gantt es un gráfica donde se muestra la asignación de recursos en el tiempo. La descripción gráfica consiste en una colección de bloques, cada uno de los cuales muestra los tiempos requeridos de cada recurso utilizado en cada etapa del proceso. Generalmente, los recursos específicos se muestran a lo largo del eje vertical y la escala del tiempo se muestra a lo largo del eje horizontal. Un ejemplo de ello se muestra en la Figura 1, donde tenemos dos procesos, A y B, con tres etapas cada uno.

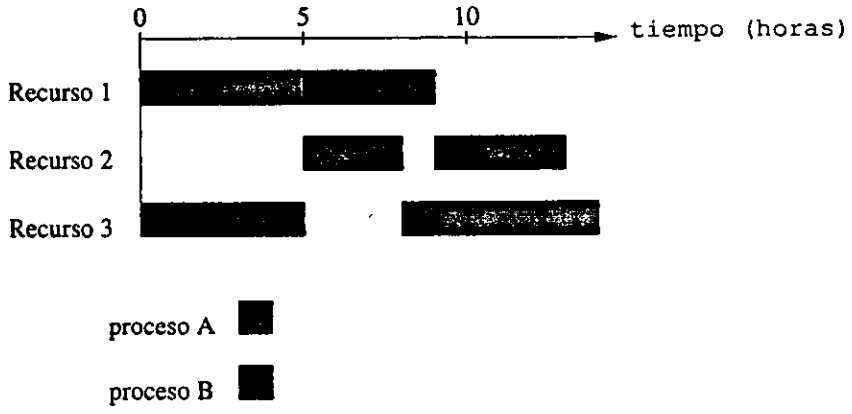

Figura 1: *Diagrama de Gantt*

Un *calendario factible* es una solución tanto de las restricciones de los recursos como de las restricciones lógicas. Las *restricciones de los recursos* se cumplen cuando dos etapas nunca ocupan el mismo recurso simultáneamente. Las *restricciones lógicas* se satisfacen cuando todas las etapas de cada proceso pueden ser calendarizadas en el orden de precedencia previamente especificado y sin traslaparse.

Existe, en principio, un número infinito de calendarios factibles para cualquier problema de calendarización de actividades, ya que un lapso arbitrario de tiempo ocioso puede insertarse en cualquier recurso entre parejas adyacentes de etapas de distintos procesos. Una vez que la secuencia de etapas para cada recurso se especifica, este tipo de tiempo ocioso no puede ser

útil para ninguna función objetivo razonable. Por el contrario, es deseable que las etapas de los distintos procesos estén lo más cercanas posible para evitar tiempos ociosos.

Se dice que existe un *tiempo ocioso superfluo* si alguna etapa puede empezarse antes sin alterar las secuencias de etapas en cada recurso. Ajustar el tiempo de inicio de alguna etapa en esta forma es equivalente a mover un bloque de operación a la izquierda en el diagrama de Gantt preservando la secuencia de etapas. A este tipo de ajuste se le conoce como *desplazamiento local a la izquierda* (DLI). Dada una secuencia de etapas para cada recurso, sólo hay un calendario en el cual el DLI no puede ser realizado. Al conjunto de todos los calendarios en los cuales ningún DLI puede ser realizado se le llama *conjunto de calendarios semiactivos* y es equivalente al conjunto de todos los calendarios que no contienen el tiempo ocioso superfluo mencionado anteriormente. Este conjunto "domina" al conjunto de todos los calendarios, lo cual indica que es suficiente con considerar sólo los calendarios semiactivos para optimizar cualquier función objetivo razonable.

El tiempo de inicio de una etapa en un calendario semiactivo está restringido ya sea por el tiempo de procesamiento de un trabajo diferente que utiliza el mismo recurso o por el procesamiento de la etapa precedente, del mismo proceso, en un recurso diferente. En el caso donde la terminación de una etapa anterior en el mismo recurso está restringiendo la calendarización, es posible encontrar una mejora. Esta mejora es posible sólo cuando se puede empezar antes una etapa de un proceso sin retardar cualquier otra etapa del mismo o de otro proceso. A este tipo de ajuste se le llama *desplazamiento global a la izquierda* o simplemente *desplazamiento a la izquierda* (DI). Un ejemplo de este tipo de ajuste se puede observar en la Figura 2. El conjunto de todos los calendarios en los cuales ningún DI puede ser realizado se llama *conjunto de calendarios activos*, y es claramente un subconjunto de los calendarios semiactivos.

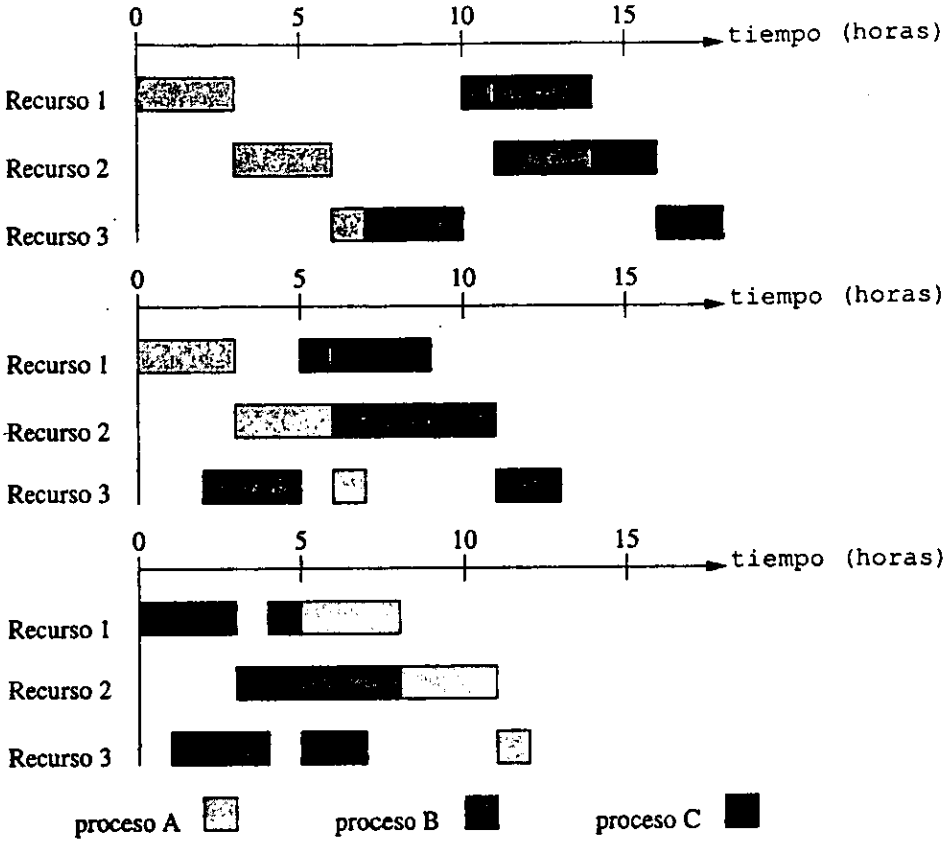

Figura 2: *Efecto de los DI al alterar un calendario semiactivo y hacerlo más compacto*

De la misma forma en que el conjunto de calendarios semiactivos domina al conjunto de todos los calendarios, también el conjunto de calendarios activos domina al conjunto de calendarios semiactivos. En otras palabras, para optimizar cualquier función objetivo razonable basta con considerar sólo calendarios activos.

Una *ruta* (routing) es el conjunto de asignaciones de recursos de un proceso dado. El número de calendarios activos es una función de las rutas y de los tiempos de procesamiento de un problema dado, mientras que el número

de calendarios semiactivos es sólo función de las rutas.

Regresando al problema de la calendarización de procesos químicos. Se tomarán en cuenta los calendarios activos para determinar los posibles calendarios que resuelvan el problema, ya que proporcionan un conjunto de calendarios factibles para optimizar cualquier función objetivo razonable.

5.4 Implementación del calendarizador

El problema de la calendarización de procesos químicos tiene varias características que lo hacen apropiado para ser resuelto automáticamente con una computadora, ya que se pueden formalizar las restricciones como un problema de optimización. Sin embargo, existen dos dificultades. Primero, los problemas de calendarización son NP-completos, por ejemplo, tener el tamaño de problemas prácticos típicos, nos imposibilitan el enumerar todos los posibles calendarios. Segundo, en nuestro problema específico, la función objetivo no es clara. A menudo las personas que realizan los calendarios prefieren un calendario de otro y no se puede establecer de antemano sus preferencias, disminuyendo así la calidad de los calendarios. Por esta razón, optamos por dejar nuestro sistema sin función objetivo, pero restringiendo los calendarios a sólo aquellos calendarios activos.

Debido a que el tamaño del espacio de búsqueda para la resolución del problema de la calendarización es considerablemente grande, es necesario utilizar técnicas heurísticas para recorrer este espacio y para restringirlo. La técnicas heurísticas que utilizamos es la llamada *first-fail* (primero en fallar).

Esta heurística nos dice que seleccionemos primero la alternativa con mayor posibilidad de fallo, ya que probando primero con las partes más difíciles del espacio de búsqueda, hace que las fallas aparezcan más pronto. Nosotros utilizamos esta heurística para resolver nuestro problema, para lo cual hacemos que los procesos ingresen a la calendarización a partir del proceso que tarda más tiempo en realizarse hasta el que tarda menos.

El procedimiento que se sigue para lograr lo anterior es realizar primero un ordenamiento descendiente de los procesos químicos, tomando en cuenta el tiempo que tardan en realizarse, y finalmente se calendarizan los procesos en ese orden tomando en cuenta el conjunto de calendarios activos posibles.

La manera con que se identifica un proceso en el calendarizador es con el functor `p(nombre, lista_etapas)`, donde `nombre` es el nombre del proceso y `lista_etapas` es una lista de etapas. Cada etapa se identifica por el

functor `e(recurso, inicio, fin)`, donde `recurso` es el nombre del recurso necesario en esa etapa, `inicio` indica el tiempo de inicio dentro del proceso y `fin` denota el tiempo en que finaliza dicha etapa.

Los recursos se identifican por el functor `r(nombre, inicio, fin)`, donde `nombre` denota el nombre del recurso, `inicio` indica el tiempo en que comienza la disponibilidad del recurso y `fin` indica el tiempo en que finaliza dicha disponibilidad.

```
% Este es el programa que calendariza procesos quimicos.

cpq(Process, Res_in, Res_out, Solution):-
    transform_processes(Process, Process_trans),
    schedule(Process_trans, Res_in, Large_Process, Res_out, Solution).

schedule([], Recursos, [], Recursos, []).
schedule(Processes, Resources, [L_proc|Shorts],
                                R_trans, [Sol|Soluciones]):-
    get_longer_process( Processes, L_proc, S_procs),
    program_a_process(L_proc, Resources, Sol, New_Res),
    schedule(S_procs, New_Res, Shorts, R_trans, Soluciones).
```

Capítulo 6

Método de eliminación de variables de Fourier

La calendarización de procesos químicos establece un conjunto de restricciones a satisfacer para generar un calendario factible utilizando únicamente el conjunto de calendarios activos. Es decir, aquellos calendarios donde podamos empezar los procesos lo más pronto posible y llevar a cabo todos los procesos, para tener ociosos a los recursos el menor tiempo posible. La descripción del método fue tomada de [Ros94].

6.1 Introducción

Hemos observado que en nuestro problema de calendarización de procesos químicos existen seis tipos de restricciones: etapas de procesos, disponibilidad de recursos, precedencia, tiempo de inicio, tiempo de terminación y uso exclusivo de recursos. Para poder satisfacer todas estas restricciones, es necesario la transformación de nuestras restricciones a un problema de satisfacción de restricciones.

Las restricciones de las etapas de cada proceso comprenden al conjunto de intervalos de tiempo para cada etapa y al conjunto de recursos alternativos por cada etapa. La disponibilidad de recursos tiene como restricción al conjunto de intervalos de tiempo para cada recurso.

La precedencia indica el orden parcial que existe en las etapas de cada proceso. El tiempo de inicio se refiere al tiempo mas temprano al cual cada proceso puede comenzar. La fecha de término es el retraso permitido al cual

cada proceso puede terminar. El uso exclusivo se refiere aque a lo más una etapa de cada proceso puede utilizar un recurso a la vez.

Un problema de satisfacción de restricciones consiste en un conjunto de variables que varían sobre algunos dominios dados, junto con un conjunto de relaciones sobre las variables. A este conjunto de relaciones sobre variables se le denomina *restricciones.*

Una *solución* es una asignación de valores a las variables satisfaciendo las restricciones. Con esto, logramos traducir nuestro problema a un sistema de desigualdades lineales. Este sistema puede ser resuelto por medio de distintos métodos. Nosotros utilizamos el método denominado de eliminación de variables de Fourier.

6.2 Descripción del método

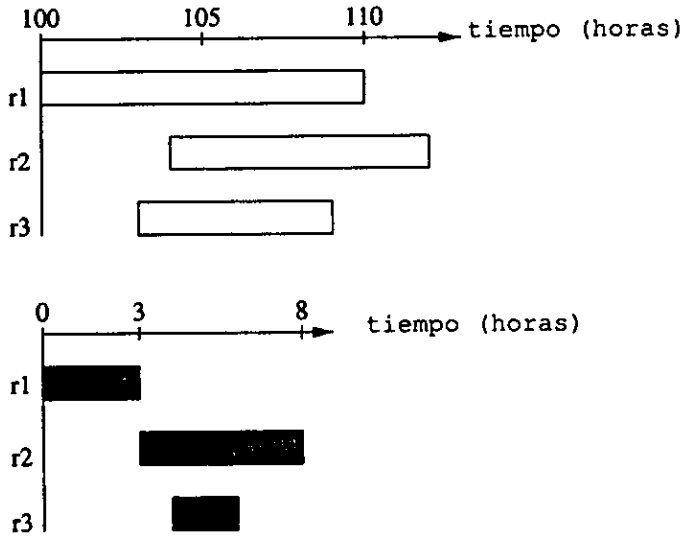

Figura 3: *Intervalos de disponibilidad (arriba) y un proceso (abajo) hipotéticos.*

Para describir el método de eliminación de variables de Fourier vamos a utilizar un ejemplo hipotético de un problema de calendarización de un proceso de tres etapas.En la Figura 3, se muestran los intervalos de tiempo de disponibilidad de los recursos, así como las etapas del proceso y sus requerimientos.

Consideremos la representación del proceso. Denotamos con d_i la duracion de la etapa i, y con b_i, el tiempo de inicio de la etapa i con respecto al inicio del proceso:

$$\begin{aligned} d_1 &= 3 \quad b_1 = 0 \\ d_2 &= 5 \quad b_2 = 3 \\ d_3 &= 2 \quad b_3 = 4 \end{aligned}$$

Usamos los digitos 1, 2 y 3 para hacer referencia a los recursos r1, r2 y r3, respectivamente. Representamos los tiempos de inicio y fin de los intervalos de disponibilidad del recurso i, con r_i y r'_i, respectivamente.

$$\begin{aligned} r_1 &= 100 \quad r'_1 = 110 \\ r_2 &= 104 \quad r'_2 = 112 \\ r_3 &= 103 \quad r'_3 = 109 \end{aligned}$$

Denotando con x_i el tiempo de inicio de la etapa i, podemos establecer las siguientes restricciones:

$$x_2 = x_1 + 3 \tag{6.1}$$
$$x_3 = x_1 + 4 \tag{6.2}$$

además de:

$$100 \leq x_1 \leq (110 - 3) \tag{6.3}$$
$$104 \leq x_2 \leq (112 - 5) \tag{6.4}$$
$$103 \leq x_3 \leq (109 - 2) \tag{6.5}$$

Sustituyendo (5.1) en (5.4), obtenemos:

$$104 \leq x_1 + 3 \leq (112 - 5) \tag{6.6}$$

A continuación restamos 3 a (5.6):

$$(104 - 3) \leq x_1 \leq (112 - 5 - 3) \tag{6.7}$$

Como (5.3) y (5.7) deben satisfacerse, entonces:

$$max(100, 104 - 3) \leq x_1 \leq min(110 - 3, 112 - 5 - 3) \tag{6.8}$$

la cual podemos reescribir como:

$$101 \leq x_1 \leq 104 \tag{6.9}$$

De manera similar, si sustituimos (5.2) en (5.5) obtenemos:

$$103 \leq x_1 + 4 \leq (109 - 2) \tag{6.10}$$

Al despejar x_1, tenemos que

$$(103 - 4) \leq x_1 \leq (109 - 2 - 4) \tag{6.11}$$

De (5.9) y (5.11) obtenemos:

$$max(101, 103 - 4) \leq x_1 \leq min(104, 109 - 2 - 4) \tag{6.12}$$

esto es,

$$101 \leq x_1 \leq 103 \tag{6.13}$$

Por lo tanto, las restricciones se satisfacen si el valor de x_1 está entre 101 y 103. Dado que nosotros tomamos únicamente los calendarios semiactivos, pues deseamos tener ociosos el menor tiempo posible a los recursos. Entonces tomamos $x_1 = 101$. Una interpretación gráfica de la solución obtenida se muestra en la Figura 4.

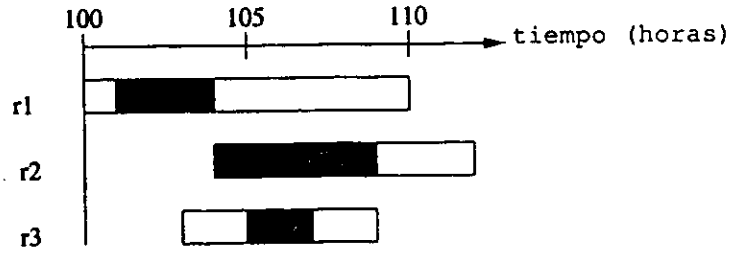

Figura 4: *Calendarización de un proceso hipotético*

6.3 Implementación del método

A continuación se muestra la implementación del método de Fourier en Prolog puro con negación por falla. El predicado principal `fourier_method/3`, recibe como primer parámetro la lista de etapas de un proceso a calendarizar, como segundo parámetro los recursos disponibles para realizar la calendarización, y en el tercer parámetro obtenemos la solución.

```
fourier_method(Etapas, Recursos, Solucion):-
time_list(Etapas, Recursos, Tis, Tfs),
is_max(Tis, Max),
is_min(Tfs, Min),
Max =< Min,
translate(Etapas, Max, Solucion).


time_list([], _, [], []).
time_list([e(R,I,D)|Proceso], [r(R,Ti,Tf)|Rs],
                          [Tie|Tis], [Tfe|Tfs]):-
      Tie is Ti - I,
      D =< (Tf-Tie),
      Tfe is Tf - D - I,
      time_list(Proceso, [r(R,Ti,Tf)|Rs], Tis, Tfs).

time_list([e(RA,I,D)|P], [r(RB,Ti,Tf)|Rs],
                          [Tie|Tis], [Tfe|Tfs]):-
      time_list([e(RA,I,D)],Rs,[Tie],[Tfe]),
      time_list(P,[r(RB,Ti,Tf)|Rs],Tis, Tfs).


is_max([X|Xs], Max):-
      max(Xs,X,Max).

max([], Max, Max).
max([A|As], B, Max):-
      B >= A,
      max(As, B, Max).
max([A|As], B, Max):-
```

```
      \+(B >= A),
      max(As, A, Max).

is_min([X|Xs], Min):-
      min(Xs, X, Min).

min([], Min, Min).
min([X|Xs], Y, Min):-
      Y =< X,
      min(Xs, Y, Min).
min([X|Xs], Y, Min):-
      \+(Y =< X), min(Xs, X, Min).

translate([],_,[]).
translate([e(R,I,D)|P], Tip, [e(R,Tie,Tfe)|PC]):-
      Tie is Tip + I,
      Tfe is Tie + D - 1,
      translate(P, Tip, PC).
```

Capítulo 7

Diagnosis del calendarizador

En el presente capítulo mostraremos un ejemplo de diagnosis del calendarizador que nos ayudó a encontrar errores en el desarrollo del mismo.

Dado el programa siguiente:

```
program_a_process(p(_,Etapas), Recursos, Solucion, Nuevos_Rec):-
          fourier_method(Etapas, Recursos, Solucion),
          update_availability(Solucion, Recursos, Nuevos_Rec).
```

El predicado `fourier_method/3` está definido en el capítulo anterior, sólo que el predicado `time_list/4` estaba definido como sigue:

```
time_list([], _, [], []).
time_list([e(R,I,D)|Etapa], [r(R,Ti,Tf)|Rs],
                          [Tie|Tis], [Tfe|Tfs]):-
     Tie is Ti - I, D =< (Tf-Tie), Tfe is Tf - D - I,
     time_list(Etapa, Rs , Tis, Tfs).

time_list([e(RA,I,D)|Etapa], [r(RB,Ti,Tf)|Rs],
                          [Tie|Tis], [Tfe|Tfs]):-
     time_list([e(RA,I,D)],Rs,[Tie],[Tfe]),
     time_list(Etapa, Rs, Tis, Tfs).
```

Al hacer la consulta

```
:-?program_a_process(p(pA,[e(ra,0,3),e(rb,3,2)]),
           [r(ra,105,120),r(rb,100,120)],
           X, Y).
```

```
X = p(pA,[e(ra,105,107),e(rb,108,109)]),
Y = [r(rb,100,108),r(rb,109,120),r(ra,107,120)]
```

Sin embargo, al hacer la consulta

```
:-?program_a_process(p(pA,[e(ra,0,3),e(rb,3,2)]),
            [r(rb,100,120),r(ra,105,120)],
            X, Y).
no.
```

Es decir, si los recursos no están en el mismo orden en que son utilizados por las etapas, entonces el programa no encuentra la solución. Para encontrar el error en el programa hicimos uso del diagnosticador. Para tal efecto se definió el predicado test/2.

```
test(RecursosModificados, Calendario):-
  program_a_process(p(pA,[e(ra,0,3),e(rb,3,2)]),
            [r(rb,100,120),r(ra,105,120)],
            RecursosModificados, Calendario).
```

y se ejecutó el diagnosticador.

```
Introduce consulta
DS> test(X,Y).
Resolviendo ...
no hay solucion(es)
ok? n.

Pregunta: test(X,Y)? s.
Cul(es) X,Y? p(pA,[e(ra,105,107),e(rb,108,109)]),
              [r(rb,100,108),r(rb,109,120),r(ra,107,120)].

Error: solucion no encontrada
test(p(pA,[e(ra,105,107),e(rb,108,109)]),
   [r(rb,100,108),r(rb,109,120),r(ra,107,120)]).
Diagnosticando ...
Resolviendo ...

Pregunta:
program_a_process(p(pA,[e(ra,0,3),e(rb,3,2)]),
```

```
          [r(rb,100,120),r(ra,105,120)],
          p(pA,[e(ra,105,107),e(rb,108,109)]),
          [r(rb,100,108),r(rb,109,120),r(ra,107,120)])? s.

Pregunta:
fourier_method([e(ra,0,3),e(rb,3,2)],
          [r(rb,100,120),r(ra,105,120)],
          [e(ra,105,107),e(rb,108,109)])? s.

Pregunta:
time_list([e(ra,0,3),e(rb,3,2)],[r(rb,100,120),r(ra,105,120)],
          X,Y)? s.

Cul(es) X,Y? [105,97],[117,115].

Pregunta:
time_list([e(ra,0,3)],[r(ra,105,120)],[105],[117])? s.

Pregunta:
time_list([e(rb,3,2)],[r(ra,105,120)],[97],[115])? n.

Error diagnosticado:
  time_list([e(ra,0,3)],[r(ra,105,120)],[105],[117]),
  time_list([e(rb,3,2)],[r(ra,105,120)],[97],[115])
es un atomo no cubierto
```

Después de analizar el problema, se observó que la lista de recursos disminuía en un elemento cada vez que `time_list/4` hacía recursión. Es decir, el predicado `time_list/4` en sus dos últimas cláusulas, le quitaba un elemento a la lista de recursos.

El predicado `time_list/4` después de la corrección quedó así.

```
time_list([], _, [], []).
time_list([e(R,I,D)|Proceso], [r(R,Ti,Tf)|Rs],
                          [Tie|Tis], [Tfe|Tfs]):-
      Tie is Ti - I, D =< (Tf-Tie), Tfe is Tf - D - I,
      time_list(Proceso, [r(R,Ti,Tf)|Rs], Tis, Tfs).
```

```
time_list([e(RA,I,D)|P], [r(RB,Ti,Tf)|Rs],
                         [Tie|Tis], [Tfe|Tfs]):-
      time_list([e(RA,I,D)],Rs,[Tie],[Tfe]),
      time_list(P,[r(RB,Ti,Tf)|Rs],Tis, Tfs).
```

De tal manera que al ejecutar el predicado `test/2` de nueva cuenta, se obtuvo la solución correcta.

Capítulo 8

Resultados y Conclusiones

El desarrollo de diagnosticadores de programas proporciona conocimientos acerca de los programas que, en principio, los programadores solemos pasar por alto: la interpretación de los programas. El diagnosticador desarrollado en este trabajo es un ejemplo de ello.

8.1 Resultados

He desarrollado un sistema diagnosticador de programas Prolog puro con negación por falla que facilita la revisión de este tipo de programas Prolog. Además, el diagnosticador proporciona herramientas tales como la prueba de entradas previamente ejecutadas con éxito y una base de datos que incrementa la información que se tiene acerca del significado deseado del programa para posteriores diagnosis.

Por otra parte, el diagnosticador de programas nos sirvió para realizar un análisis más objetivo sobre las suposiciones que planteamos originalmente para la calendarización de procesos químicos. Las suposiciones nos pueden llevar a tener problemas que jamás llegamos a contemplar al momento de realizarlas.

La diagnosis del calendarizador se realizó durante y al final del desarrollo del mismo. Los resultados obtenidos fueron satisfactorios, ya que sin la ayuda del diagnosticador la detección y corrección de errores hubiera sido una actividad difícil, si no es que imposible.

8.2 Conclusiones

La necesidad de diagnosticar programas tanto en el desarrollo como en el mantenimiento de los mismos, es indudable. La herramienta desarrollada facilita la localización y corrección de errores en un programa Prolog puro con negación por falla.

Como se mostró en los diversos ejemplos, la interactividad del diagnosticador nos permite observar tanto el recorrido en el cálculo de nuestras metas, así como la localización del error en el programa de manera más eficaz.

El calendarizador de procesos químicos es un sistema que aunque no está completamente terminado, sí resuelve de manera satisfactoria calendarios para este tipo de actividad. Decimos que no está completamente terminado porque, hasta el momento, no calendáriza lotes; además carece de una interfaz gráfica que lo haga amigable al usuario, pues las soluciones se muestran en forma de listas Prolog, las cuales no son de fácil lectura.

Con los resultados obtenidos en el desarrollo del problema planteado podemos concluir que los objetivos propuestos se lograron. Desarrollamos un diagnosticador interactivo y de uso general, y también un calendarizador de procesos químicos. Esperamos que este trabajo sirva de ayuda en el desarrollo de programas Prolog, además de que fomente algunas de las técnicas en la implementación y la depuración de programas de computadora.

Apéndice A

Código fuente del diagnosticador

A.1 Programa principal

Programa: ds.pl

```
%%%%%%%% Un sistema de diagnosis %%%%%%%%%%%%%%%%%%%%%%%%%%
:-[solve].
:-[diagsys].
:-[dsutil].

:- use_module(library(system)).

ds :-
    ds0(Name), ds1(P,Name).

ds0(Name):-
    write('Introduce el nombre de tu archivo (sin .pl)? '), read(N),
    consult(N), previous(N, Name),
    search_file(Name), consult(Name).

search_file(Name):-
    name('ls | grep ', X), name(Name, Y),
    append(X, Y, Z), name(Command, Z), system(Command), !.
search_file(Name):-
```

```
    open(Name, write, Stream),
    write_term(Stream, ':- dynamic solutions/2.', [indented(true)]),
    nl(Stream), close(Stream).

ds1(P, Name):-
    write('Introduce consulta'), nl, write('DS> '), read(P), get_p(P,Name).

previous(N, Name):-
    name(N, X), append(X, [95,100,98], Z), name(Name, Z).

get_p(bye, Name):-
    get_p(exit, Name).
get_p(exit, Name):-
    open(Name,write,Stream),
    write_term(Stream,':- dynamic solutions/2.',[indented(true)]),
    nl(Stream), store_on(Stream).
get_p(P, Name):-
    solve_and_check(P),
    ds1(_P1, Name).

solve_and_check(P):-
    bagof0((P,X), solve(P,X), S),
    check_solutions(P, S).

check_solutions(P, S):-
    check_solution(P, S).

check_solution(_P, S):-
    member((P1, overflow(X)), S),
    !, stack_overflow(P1, X).
check_solution(_P, S):-
    member((P1, true), S),
    fact(P1, false),
    !, false_solution(P1).
check_solution(P, S):-
    fact(P, true),
    \+ member((P, true), S),
    !, missing_solution(P).
```

```
check_solution(P, S):-
    confirm_solutions(P, S).

confirm_solutions(P, [(P1, _X)|S]):-
    write_list(['Solucion: ',P1,' ;']), nl,
    confirm_solution1(P, P1, S).
confirm_solutions(P, []):-
    write('no hay solucion(es)'), nl,
    confirm_solution2(P).

confirm_solution1(P, P1, S):-
    check_p1(P1),
    !, nl,
    confirm_solutions(P, S).
confirm_solution1(P, P1, S):-
    confirm('ok'),
    !, assert_fact(P1, true), confirm_solutions(P, S).
confirm_solution1(_P, P1, _S):-
    assert_fact(P1, false), false_solution(P1).

check_p1(P):-
    predicate_property(P, built_in),!.
check_p1(P):-
    fact(P, true).

confirm_solution2(P):-
    predicate_property(P, built_in), !, nl.
confirm_solution2(_P):-
    confirm('ok'), !, true.
confirm_solution2(P):-
    ask_for_solution(P),
    assert_fact(P, true), missing_solution(P).

handle_error('clausula falsa', X):- !,nl,
    write_list(['Error diagnosticado: ',X,' , es una clausula falsa']),
    nl, nl, plisting(X).
handle_error('atomo no cubierto', X):- !, nl,
    write_list(['Error diagnosticado: ',X,' es un atomo no cubierto']),
```

```
    nl, nl. %plisting(X).
handle_error('clausula divergente', X):- !, nl,
    write_list(['Error diagnosticado: ',X,' esta divergiendo']),
    nl, nl, plisting(X).
```

A.2 Utilerías

Programa:dsutil.pl

```
:- dynamic legal_call/2, new_fact/1, not/1.

query(forall, A, V):- ground(A), !, query(exists, A, V).
query(forall, A, V):-
    break(query(forall, A, V)).

query(legal_call, (P1, P2), false):-
    same_goal(P1, P2), !.
query(legal_call, (P1, P2), V1):-
    legal_call((Q1, Q2), V1), same_goal(P1, Q1), same_goal(P2, Q2), !.
query(legal_call, (P1, P2), V):-
    confirm(['Es ', (P1, P2), ' una llamada legal']), !,
    assert(legal_call((P1, P2), true)), V = true.
query(legal_call, (P1, P2), V):-
    assert(legal_call((P1, P2), false)), V = false.

query(exists, A, V):-
    predicate_property(A, built_in), !, prove_a(A, V).
query(exists, A, V):-
    mgt(A, A1), solutions(A1, S), is_instance(A, A1),!,
    (member(A, S), V = true; \+ member(A, S), V=false).
query(exists, A, true):-
    fact(A, true), !.
query(exists, A, V):-
    ask_for_solutions(A, S), !,
    (S=[], V=false; member(A, S), V=true).

same_goal(P, Q):-
    functor(P, F, N), functor(Q, F, N),
```

```
    copy_term(P, Pi), copy_term(Q, Qi), !, variants(Pi, Qi).

variants(P, Q):-
    verify((numbervars(P,0,N), numbervars(Q,0,N), P=Q)).

prove_a(A, true):-
    A, !, true.
prove_a(_, false).


not(P):- P, !, fail.
not(_).


mgt(P,P0):-
    functor(P, F, N),
    functor(P0, F, N).

is_instance(P1, P2):-
    verify((numbervars(P1, 0, _), P1=P2)).

verify(P):-
    \+(\+(P)).

write_list(L, E, _):-
    var(L), !, write2(E, L).
write_list([], _, _):-
    !, true.
write_list([X], E, _):-
    !, write2(E, X).
write_list([X|L1], E, S):-
    !, write_list(X, E, nil), write2(s, S), write_list(L1, E, S).
write_list(L, E, _):-
    write2(E, L).

write2(w, X):- write(X).
write2(v, X):- writev(X).
write2(s, nil):- !, true.
```

```
write2(s, nl):- !, nl.
write2(s, bl):- !, write(' ').
write2(s, comma):- !, write(', ').
write2(s, S):-
    write(S).

write_list(L):-
    write_list(L, v, nil).

reade(X):-
    read(X1),
    (expand(X1, X), !, true; X= X1).

expand(t, true).
expand(y, true).
expand(s, true).
expand(f, false).
expand(n, false).
expand(a, abort).
expand(b, break).


directive(abort).
directive(trace).
directive(break).
directive(info).
directive(true):- !, fail.
directive(false):- !, fail.
directive(_ =< _):-  !, fail.
directive(_ < _):- !, fail.
directive(_ > _):- !, fail.
directive(_ >= _):- !, fail.
directive(X):-
    predicate_property(X, built_in).

writev(X):-
    lettervars(X), write(X), fail.
writev(_).
```

```
lettervars(X):-
    varlist(X, V1),
    unifyvars(V1,['X', 'Y', 'Z', 'A', 'B', 'C', 'D',
              'E', 'F', 'G', 'H', 'I', 'J']).
lettervars(X, V1):-
    varlist(X, V1),
    unifyvars(V1,['X', 'Y', 'Z', 'A', 'B', 'C', 'D',
                  'E', 'F', 'G', 'H', 'I', 'J']).

unifyvars([X|L1], [X|L2]):-
    unifyvars(L1, L2).
unifyvars([],_).

remove_duplicates([], []).
remove_duplicates([Head|Tail1], [Head|Tail2]) :-
    delete(Tail1, Head, Residue),
    remove_duplicates(Residue, Tail2).

delete([], _, []).
delete([Head|Tail], Element, Rest) :-
     Head==Element, !,
     delete(Tail, Element, Rest).
delete([Head|Tail], Element, [Head|Rest]) :-
     delete(Tail, Element, Rest).

varlist(X, Vlist):-
    variables(X, L, []), remove_duplicates(L,Vlist).
variables(X, [X|L], L):-
    var(X), !.
variables(T, L0, L):-
    T =.. [_F|A], variables1(A, L0, L).

variables1([T|A], L0, L):-
    variables(T, L0, L1), variables1(A, L1, L).
variables1([], L, L).

listtoand([], true):- !.
```

```
listtoand([X], X):- !.
listtoand([X|Xs], (X,Ys)):-
    !, listtoand(Xs, Ys).

varand(P, Vs):-
    varlist(P, Vlist),
    listtoand(Vlist, Vs).

ask_which(A):-
    lettervars(A, V1), listtoand(V1, V2), V2 \== true,
    write_list(['Which ', V2, ' ?']), fail.
ask_which(_).

ask_for_solutions(P, S):-
    bagof0(P, ask_for_solution(P), S),
    assert(solutions(P, S)).

ask_for_solution(P):-
    nl, ask_for(['Pregunta: ', P], V, (V=true; V=false)),
    checkv(P, V).

checkv(_P, false):- !, fail.
checkv(P, _V):-
    ground2(P), !, true.
checkv(P, _V):-
    varand(P, Pvars),
    repeat,
      write_list(['Which ', Pvars, '? ']), ttyflush,
      reade(Answer),
      (Answer = false, !, fail;
       Answer = Pvars, !, true;
       write('does not unify; try again'), nl), !.

ask_for(Request, Answer, Test):-
    repeat,
    ask_for(Request, Answer), Test, !.

ask_for(Request, Answer):-
```

```
    repeat,
    write_list(Request), write('? '), ttyflush,
    reade(X),
    (directive(X), !,
      (X, !; write('? '), nl),
      ask_for(Request, Answer);
      Answer = X), !.
confirm(P):-
    ask_for(P, V),
    confirmv(V, P).

confirmv(true, _):- !, true.
confirmv(false, _):- !, fail.
confirmv(_, P):- confirm(P).

member(X, [X|_]).
member(X, [Y|Xs]):-
    member(X, Xs).

bagof0(X, P, S):-
    bagof(X, P, S), !, true; S=[].

fact(P, V):-
    var(P), !, (solutions(_, S), member(P, S), V=true;
    solutions(P, []), V=false).
fact(P, V):-
    solutions(P, S),
    (member(P, S), V=true; \+member(P, S), V=false).

assert_fact(P, V):-
    fact(P, V1), !, (V=V1, !, true; break(assert_fact(P, V))).
assert_fact(P, V):-
    \+ ground2(P), !, break(assert_fact(P, V)).
assert_fact(P, true):-
    !, assert(solutions(P,[P])).
assert_fact(P, false):-
    !, assert(solutions(P,[])).
assert_fact(P, V):-
```

```
    break(assert_fact(P, V)).

break(P):-
    portray_clause(P),nl, call(break).

ground2(P):-
    numbervars(P, 0, 0).


check_vars(true, true).
check_vars(Vars, Clause):- numbervars(Clause, 0, 0), !.
check_vars(Vars, Clause):-
    read(Answer),
    !, check_answer(Answer, Vars, Clause).

check_answer(no, _Vars, _A):- !, fail.
check_answer(n, _Vars, _A):- !, fail.
check_answer(true, _Vars, _A):- !.
check_answer(y, _Vars, _A):- !.
check_answer(yes, _Vars, _A):- !.
check_answer(Vars, Vars, A):- !.
check_answer(Answer, _Vars, A):-
    write('Illegal Answer!'),
    !, query(A, true).


plisting([]):- !.
plisting([P|Ps]):- !,
    plisting(P), nl, !, plisting(Ps).
plisting(X):-
    plisting_x(X, P1, Q),
    write_list(['listado de ', P1, ' :']),
    nl, nl, nl,
    (clause(P1, Q), tab(4),
     writev((P1:-Q)), write('.'),
      nl, fail; true), nl.
plisting_x(X, P1, Q):-
    X = (P :- Q), !, mgt(P,P1).
```

```
plisting_x(X, P1, _):-
    mgt(X, P1).

append([],Ys,Ys).
append([X|Xs], Ys, [X|Zs]):-
    append(Xs, Ys, Zs).


run_previous_inputs:-
    solutions(_P,S), S \== [], member(B, S), \+B,
    write('Error: '), write(B), (' no se satisface.'), nl.
run_previous_inputs:-
    write('Entradas previas verificadas.'), nl.

store_on(Stream):-
    retract(solutions(P,S)),
    portray_clause(Stream, solutions(P,S)),
    fail.
store_on(Stream):-
    true, close(Stream).


ask_then_do(Pregunta, Respuestas):-
% muestra la pregunta. Una respuesta es una lista de parejas
% (respuesta, accion) ; verifica que la respuesta que el usuario da esta en
% una pareja; si esta ejecuta la accion asociada a ella
 asf_for(Pregunta, Respuesta),
 member((Respuesta, Accion),
       Respuestas)-> Accion;
 setof(Respuesta, Accion^member(
       (Respuesta, Accion), Respuestas),
        Respuestas),
 writel(['Respuestas validas son ', Respuestas]), nl,
 ask_then_do(Pregunta, Respuestas).
```

Programa:diagsys.pl

```
%% Diagnosticador que detecta tres tipos de errores
```

```
%% fpm((A, Wa), (M, Wm), W):-
%% resulve A, cuyo peso es Wa. Encuentra una submeta M en la
%% computacion cuyo peso, Wm, es menor que W/2
%% y es el hijo mas pesado de un nodo cuyo peso, Wm,
%% es menor que (W+1)/2.

fpm(((A,B),Wab), M, W):- !,
    fpm((A,Wa), (Ma,Wma), W),
    fpm((B,Wb), (Mb,Wmb), W),
    Wab is Wa + Wb,
    check_wma(Wma, Wmb, Ma, Mb, M).
fpm((A,0), (true,0), _W):-
    predicate_property(A, built_in),
    !, A.
fpm((A,0), (true,0), _W):-
    predicate_property(A, built_in), !, A; fact(A,true).
fpm((A,Wa), M, W):-
    clause(A,B),
    fpm((B,Wb), Mb, W),
    Wa is Wb +1,
    check_wa(Wa, W, A, B, Mb, M).

check_wma(Wma, Wmb, Ma, _Mb, M):-
    Wma >= Wmb, !, M = (Ma, Wma).
check_wma(_Wma, Wmb, _Ma, Mb, M):-
    M = (Mb, Wmb).

check_wa(Wa, W, _A, _B, Mb, M):-
    Wa > (W+1)/2, !, M = Mb.
check_wa(Wa, _W, A, B, _Mb, M):-
    M = ((A:-B), Wa).



%%  Traza de un procedimiento incorrecto
%%  por medio de divide-and-query.
```

```
false_solution(A):-
    write_list(['Error: solucion incorrecta ', A, '. Diagnosticando ...']),nl,
    fpm((A,W), _, 0),  %% just to find W, the lenght of the computation
    fph(A, W, X).

fph(A, W, X):-
    fp(A, W, X), !, handle_error('clausula falsa', X).
fph(_, _, _):-
    write('Llamada invalida a fp '), nl.


fp(A, 0, A) :- !.
%%%%%%% clausula nueva
fp(\+(A),_,X):- ip(A, X).
%%%%%%%
fp(A, W, X):-
    fpm((A,Wa), ((P:-Q),Wm), W),
    question_one(A, Wa, Wm, P, Q, X).

question_one(_A, 1, _Wm, P, Q, (P:-Q)):- !, true.
question_one(A, Wa, Wm, P, _Q, X):-
    question_two(A, Wa, Wm, P, X).

question_two(A, Wa, Wm, P, X):-
    query(forall, P, true), !,
    assert(solutions(P,[P])),
    Wa1 is Wa - Wm,
    fp(A, Wa1, X).
question_two(_A, _Wa, Wm, P, X):-
    fp(P, Wm, X).


%% Traza de un procedimiento incompleto
%% Diagnosis del problema de solucion no encontrada

missing_solution(A):-
    write_list(['Error: solucion no encontrada ', A,
'.  Diagnosticando ...']),
```

```
    nl, query(exists, A, true),
    \+ solve(A, true) ->
    ip(A, X), nl, handle_error('atomo no cubierto', X);
   write('Llamada invalida a ip'), nl.

%%%%%%%% clausula nueva
ip(\+(A), X):- fpm((A,W),_,0), fp(A,W, X).
%%%%%%%%
ip(A, X):-
    clause(A, B), ip_aux(B,X).
ip(A, A).

ip_aux(A,X):-
    ip1(A, X), !, true.
ip_aux(A, A).

ip1((A, B), X):- !,
    query(exists, A, true), ip1_A(A, B, X).
%%%%%%%%%%% clausula nueva
ip1(\+(A),X):-
    query(exists,A,false), fpm((A,W),_,0), fp(A,W, X).
%%%%%%%%%%
ip1(A, X):-
    query(exists, A, true), ip1_aux(A, X).

ip1_A(A, B, X):-
    A, ip1(B, X).
ip1_A(A, _B, X):-
    \+ A, ip(A, X).
ip1_aux(A, X):-
    A, !, break(ip1(A,X)).
ip1_aux(A, X):-
    ip(A,X).

%%%%   End of program deb2c.pl  %%%%%%%%%%%%%%%%%%%%%%%%

% Deteccion del sobreflujo de una pila (stack).
```

```
stack_overflow(P, S):-
    write_list([P, ' Diagnosticando ...']), nl,
    check_loop(S, _Sloop).

check_loop(S, Sloop):-
    find_loop(S, Sloop), !, check_segment(Sloop).
check_loop(S, _Sloop):-
    check_segment(S).

find_loop([(P:-Q)|S], Sloop):-
    check_looping((P:-Q), S, Sloop).

check_looping((P:-Q), S, [(P:-Q)|S1]):-
    looping_segment((P:-Q), S, S1), !.
check_looping(_, S, Sloop):-
    find_loop(S, Sloop).

looping_segment((P:-Q), [(P1:-Q1)|S], [(P1:-Q1)|S1]):-
    check_same((P:-Q), (P1:-Q1), S, S1).

check_same((P:-_Q), (P1:-_Q1), _S, []):-
    same_goal(P, P1), !,
    nl, write_list(['" ',P, ' " esta en un ciclo']), nl.
check_same((P:-Q), _, S, S1):-
    looping_segment((P:-Q), S, S1).

check_segment([(P:-Q), (P1:-Q1)|S]):-
    check_segmento([(P:-Q), (P1:-Q1)|S]).
check_segmento([(P:-_Q), (P1:-Q1)|S]):-
    query(legal_call, (P, P1), true), !, check_segment([(P1:-Q1)|S]).
check_segmento([(P:-Q), (P1:-_Q1)|_S]):-
    false_subgoal(P, Q, P1, C), !, false_solution(C).
check_segmento([(P:-Q), (_P1:-_Q1)|_S]):-
    handle_error('clausula divergente',(P:-Q)).

false_subgoal(P, (Q1, Q2), P1,  Q):-
    Q1 \== P1,
    check_q1(Q1, Q2, P, P1, Q).
```

```
check_q1(Q1, _Q2, _P, _P1, Q1):-
    query(forall, Q1, false), !.
check_q1(_Q1, Q2, P, P1, Q):-
    false_subgoal(P, Q2, P1, Q).
```

Apéndice B

Código fuente del calendarizador

B.1 Programa principal

Programa: cpq.pl

```
% Este es el programa que calendariza procesos quimicos.

% Resuelve el problema de asignacion de recursos de un proceso
% de varias etapas, por medio del metodo de Fourier (Sistema de inecuaciones).

:-[fourier].
:-[update].
:-[heavyproc].
% cpq(+P, +R, -R_trans, -Solucion)
cpq( Procesos, Recursos, R_trans, Solucion):-
    transform_processes( Procesos, P_trans),
    schedule( P_trans, Recursos, Proceso_mayor, R_trans, Solucion).


schedule([], Recursos, [], Recursos, []).
schedule(Processes, Resources, [L_proc|Shorts], R_trans, [Sol|Soluciones]):-
    get_longer_process( Processes, L_proc, S_procs),
    program_a_process(L_proc, Resources, Sol, New_Res),
    schedule(S_procs, New_Res, Shorts, R_trans, Soluciones).
```

```
program_a_process(p(_,Etapas), Recursos, Solucion, Nuevos_Rec):-
        fourier_method(Etapas, Recursos, Solucion),
        update_availability(Solucion, Recursos, Nuevos_Rec).
```

B.2 Utilerías

Programa: fourier.pl

```
% Programa que resuelve un sistema de inecuaciones por el metodo de Fourier
% Se tiene una lista de etapas de un proceso y estas deben seguir el orden
% especificado en la lista para su realizacion. Ademas, se tiene una lista de
% recursos utilizables con sus respectivos tiempos de disponibilidad. Y se
% quiere lograr una calendarizacion de un proceso en particular resolviendo
% las inecuaciones planteadas por el tiempo menor en el que debe empezarse el
% proceso y el tiempo mayor en el cual el proceso se deba iniciar o de lo
% contrario, el proceso no puede realizarse en el tiempo de disponibilidad de
% los recursos.

fourier_method(Etapas, Recursos, Solucion):-
    time_list(Etapas, Recursos, Tis, Tfs),
    is_max(Tis, Max),
    is_min(Tfs, Min),
    Max =< Min,
    translate(Etapas, Max, Solucion).



% time_list(+Proceso, +Recursos, -Tiempos)
% Solucion es la lista de las etapas del Proceso, con los tiempos mapeados a
% los tiempos de disponibilidad de los Recursos.

time_list([], _, [], []).
time_list([e(R,I,D)|Proceso], [r(R,Ti,Tf)|Rs], [Tie|Tis], [Tfe|Tfs]):-
    Tie is Ti - I,
    D =< (Tf-Tie),
```

```
    Tfe is Tf - D - I,
    time_list(Proceso, [r(R,Ti,Tf)|Rs], Tis, Tfs).

time_list([e(RA,I,D)|P],[r(RB,Ti,Tf)|Rs],[Tie|Tis],[Tfe|Tfs]):-
    time_list([e(RA,I,D)],Rs,[Tie],[Tfe]),
    time_list(P,[r(RB,Ti,Tf)|Rs],Tis, Tfs).


is_max([X|Xs], Max):-
    max(Xs,X,Max).

max([], Max, Max).
max([A|As], B, Max):-
    B >= A,
    max(As, B, Max).
max([A|As], B, Max):-
    \+(B >= A),
    max(As, A, Max).

is_min([X|Xs], Min):-
    min(Xs, X, Min).

min([], Min, Min).
min([X|Xs], Y, Min):-
    Y =< X,
    min(Xs, Y, Min).
min([X|Xs], Y, Min):-
    \+(Y =< X), min(Xs, X, Min).

translate([],_,[]).
translate([e(R,I,D)|P], Tip, [e(R,Tie,Tfe)|PC]):-
    Tie is Tip + I,
    Tfe is Tie + D - 1,
    translate(P, Tip, PC).
```

Programa: heavyproc.pl

```
% Programa que obtiene el proceso que tiene mas requerimientos
```

```
transform_processes([], []).
transform_processes([X|Xs], [Y|Ys]):-
    modify_stages(X, Y),
    transform_processes(Xs, Ys).

modify_stages([], []).
modify_stages(p(N,Xs), p(N,Ys)):-
    modify(Xs, Ys).

modify([], []).
modify([e(R,I,F)|Rest], [e(R,I,D)|New_Rest]):-
    D is F-I+1,
    modify(Rest, New_Rest).


% get_longer_process(+Lista_de_procesos,?Proceso_mayor,?Lista_nueva)
% Se cumple si Proceso_mayor es el proceso que tarda mas tiempo en realizarse
% y Lista_nueva es la Lista_de_procesos menos Proceso_mayor.

get_longer_process( Xs, X, Ys) :-
    longer_process( Xs, X, Ys).
get_longer_process( Xs, Y, [X|Zs]) :-
    longer_process( Xs, X, Ys),
    get_longer_process( Ys, Y, Zs).

% longer_process(Xs,Max,Rest)
% Xs es una lista de procesos, donde cada proceso es de la forma
% p(Numero,[e(Recurso,Inicio,Fin),...])
% Max es el proceso de Xs con mas requerimientos
% Rest es el Xs menos Max

longer_process( [X|Xs], Max, Rest) :-
    longer( Xs, X, Max, Rest).

% longer(Xs,X,Max,Rest)
% Xs es una lista de procesos, donde cada proceso es de la forma
% p(Numero,[e(R,I,F),e(R,I,F),...])
% X  es el proceso de Xs con mas requerimientos
```

```
% Max es el proceso con mas requerimientos de X y Xs
% Rest es Xs menos Max si Max es un elemento de Xs, o
% Rest es Xs si Max no es un elemento de Xs.

longer( [], X, X, []).
longer( [X|Xs], Y, Max, [Y|Rest]) :-
    more_resources(X,Y),
    longer( Xs, X, Max, Rest).
longer( [X|Xs], Y, Max, [X|Rest]) :-
    \+(more_resources(X,Y)),
    longer( Xs, Y, Max, Rest).


% more_resources(p(P,R),p(M,S)
% Se cumple siempre que el proceso P {p(P,R)}
% tenga mas requerimientos que el proceso M {p(M,S)}

more_resources(p(_,R),p(_,S)):-
    resources(R,LR),
    resources(S,LS),
    LR > LS.

% resources(e(_,S,E)|Xs],M)
% Realiza la suma de requerimientos de cada uno de los elementos
% de la forma e(R,I,D). I de inicio y D  de duracion.

resources([e(_, _, D)|Xs],M):-
    resources_2([e(_, _,D)|Xs],0,M).

resources_2([], M, M).
resources_2([e(_, _, D)|Xs],S,M):-
    S1 is S+D,
    resources_2(Xs, S1, M).
```

Programa: update.pl

```
% update_availability(+Etapas,+Lista_vieja,-Lista_nueva).
% Se cumple siempre que Lista_nueva sea igual a Lista_vieja con el intervalo
% modificado del recurso utilizado. En cualquiera de sus cinco casos.
```

```
%caso 0  Cuando no hay etapas del proceso que ajustar.
update_availability([], Recursos, Recursos).
%caso 1  Cuando hay etapas del proceso para ajustar.
update_availability([e(R,I,D)|Es], Recursos, Disponibilidad):-
    update_an_availability(e(R,I,D), Recursos, Nuevos_recursos),
    update_availability(Es, Nuevos_recursos, Disponibilidad).

% update_an_availability(E,R,Rs)
%caso 1 Cuando la etapa cabe enmedio del tiempo disponible del recurso.
update_an_availability(e(R,I,F), [r(R,A,C)|Recursos],
                                 [r(R,A,I),r(R,B,C)|Recursos]):-
    I > A,
    I < C,
    B is F,
    B < C.
%caso 2 Cuando la etapa cabe al final del tiempo disponible del recurso.
update_an_availability(e(R,I,F), [r(R,A,B)|Recursos], [r(R,A,I)|Recursos]):-
    I > A,
    B =:= F.
%caso 3 Cuando la etapa cabe al inicio del tiempo disponible del recurso.
update_an_availability(e(R,I,F), [r(R,A,C)|Recursos], [r(R,B,C)|Recursos]):-
     I =:= A,
     B is F,
     B < C.
%caso 4 Cuando la etapa cabe justo en el tiempo disponible del recurso.
update_an_availability(e(R,I,F), [r(R,A,B)|Recursos], Recursos):-
     I =:= A,
     B =:= F.
%caso 5 Cuando la etapa no cabe en el tiempo disponible y busca otro tiempo.
update_an_availability(e(R,I,F), [r(R2,A,B)|Recursos],
                                 [r(R2,A,B)|Disponibilidad]):-
    update_an_availability(e(R,I,F), Recursos, Disponibilidad).
```

Bibliografía

[Apt97] Krzysztof R. Apt. *From Logic Programming to Prolog.* Prentice Hall, Europe, 1997.

[Bak74] Kenneth R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley & Sons, 1974.

[BBP+81] D.L. Bowen, L. Byrd, L.M. Pereira, F.C. Pereira, and D.H.D. Warren. PROLOG on the DECSystem-10 user's manual. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1981.

[CM84] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer, Heidelberg, 2 edition, 1984.

[Jr.67] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[Kow79] Robert Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, July 1979.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[Rei81] R. Reiter. On closed world databases. In *Readings in Artificial Intelligence.* Webber and Nilsson, 1981.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computer Machinery*, 12(1):23–41, January 1965.

[Ros94] David A. Rosenblueth. An interactive system for scheduling chemical processes. In *The Second International Conference on the Practical Application of Prolog*, pages 423–440. Royal Society of Arts, London, April 1994.

[Sha82] Ehud Y. Shapiro. *Algorithmic Program Debugging.* PhD thesis, Yale University, May 1982.

[SS94] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Mass., 1994.

[vEK76] M.H. van Emden and R. Kowalski. The Semantics of Predicate Logic as Programming Language. *Journal of ACM*, 23(4):733–743, October 1976.