UNA METODOLOGÍA PARA DESARROLLAR APLICACIONES USANDO PROGRAMACIÓN POR RESTRICCIONES

Daniel Díaz Araya, Francisco S. Ibáñez y Raymundo Q. Forradellas
Laboratorio Integrado de Sistemas Inteligentes (LISI)
Instituto de Informática
Universidad Nacional de San Juan,
e-mail: {ddiaz, fibanez, kike}@iinfo.unsj.edu.ar

Resumen

En los últimos años la programación por restricciones ha automatizado la solución de problemas combinatorios complejos en muchos dominios tan diversos como planificación, asignación de recursos, optimización, etc.

En este trabajo describiremos una metodología general que permite desarrollar programas utilizando la tecnología de restricciones.

En la programación por restricciones un problema se representa involucrando incertidumbre, esto es, sus variables, y las restricciones sobre estas variables. En otras palabras, la parte desconocida del problema se representa con ***variables restringidas.*** De este modo, para cualquier problema dado, su representación consiste en la declaración de las variables y la colocación de las restricciones sobre ellas. La resolución de un problema consiste en encontrar un valor para cada variable de modo que se satisfagan todas las restricciones. Debido a que es posible que exista más de una solución para un problema dado, la elección de una solución, normalmente se determina de acuerdo a un criterio de optimización.

Para capturar la representación del problema, la metodología propuesta, utiliza la programación orientada a objetos [**Booch, 91**], mediante el uso de clases. Algunas de estas clases representan, variables restringidas, valores para los dominios de estas variables restringidas, restricciones, y algoritmos de solución.

De esta manera con la combinación de estos dos paradigmas obtenemos por el lado de la programación por restricciones eficiencia y declaratividad para abordar y manejar problemas industriales complejos en los cuales existe una gran explosión combinatoria [**Hente,89**] [**Jaffa, 87**]. Mientras que por el lado de la programación orientada a objetos obtenemos la facilidad para capturar abstracciones y mecanismos que se encuentran en el dominio del problema.

Este trabajo fue desarrollado en el marco del Proyecto CICITCA "Problemas de Asignación de Recursos utilizando Técnicas de Inteligencia Artificial"

1.1 Introducción

El objetivo del este trabajo es proponer una metodología para desarrollar aplicaciones utilizando la tecnología de restricciones.

La metodología utiliza una arquitectura de clases, la cual provee los componentes necesarios para programar con restricciones. Algunas de estas clases representan, variables restringidas, valores para los dominios de estas variables restringidas, restricciones, y algoritmos de solución, la descripción de los componentes se realiza a un nivel lo suficientemente abstracto que nos permite comprender las actividades que implica la metodología. Por esta razón se necesitan tanto los conceptos de programación por restricciones como de los de la programación orientada a objetos, para poder aplicarla.

Las actividades que componen la metodología son las siguientes:

1. Escribir en lenguaje natural la descripción del problema para clarificar el propósito de la aplicación y las restricciones involucradas.
2. Diseñar una representación para el problema -un modelo para el problema- basado en la descripción. En la representación, declaramos lo desconocido en el problema (variables restringidas) y le colocamos restricciones. Por último se deben establecer los mecanismos de búsqueda.
3. Usar clases y funciones provistas por la arquitectura mencionada para implementar el modelo.
4. Diseñar los mecanismos de búsqueda para encontrar las soluciones.

Es importante destacar que la metodología propuesta contiene sólo pautas generales para abordar un problema, sin embargo en problemas reales éstas se deben extender para resolver la complejidad que los mismos presentan.

También es necesario aclarar que en problemas industriales la ejecución de todas las actividades no produce de inmediato una solución para el problema, sino cada actividad puede producir conocimiento tanto para una la actividad que le anteceda, como para una que le preceda, el flujo de conocimiento entre las actividades no es unidireccional sino bidireccional. Así un modelo que solucione el problema es el resultado de la interacción entre las distintas actividades.

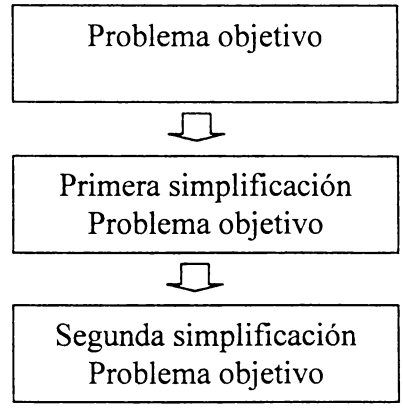
Luego de introducir los conceptos de programación por restricciones y describir la arquitectura de clases antes mencionada, se plantea un ejemplo pedagógico que permitirá describir y explicar los distintos pasos de la metodología planteada.

1.2 Descripción de un Problema - Actividad Nro 1

Para explicar cada paso de la metodología se resuelve un problema conocido. El problema de colorear un mapa, este involucra elegir los colores para los distintos países en el mapa, de tal manera que a lo sumo se usen solo cuatro colores y que ningún país colindante tenga el mismo color. Consideremos los seis países Bolivia, Uruguay, Brasil, Argentina, Chile y

Paraguay. Este problema queda bien definido con esta descripción, pero en los problemas reales necesitan una descripción más elaborada con el objeto de construir posteriormente el modelo que la representara. Ente los aspectos relevantes de esta descripcion detallada se encuentra los objetivos, la descripción general , las consideraciones generales, las consideraciones sobre la complejidad, la entrada detallada y la salida detallada. Para problemas complejos como problemas de scheduling una opción para abordar la complejidad es descomponer el problema en una serie de problemas de menor complejidad

Esta descomposición es un proceso iterativo donde un problema, es el resultado de la evolución de un problema más simple.



1.3 Diseño de la Representación del problema - Actividad Nro 2

Para problemas simples como el propuesto en el ejemplo bastará con lo siguiente:

- Declaración de las variables restringidas (ó variables dominio).
- Colocación las restricciones.
- Búsqueda de soluciones.

En la práctica sobre problemas complejos el diseño de la representación no es tan fácil, no solo se hace necesario aplicar el análisis y diseño orientado a objeto [**Booch, 91**], sino tambien es necesario coordinar el mismo con una arquitectura subyacente que soporta programación por restricciones. Por ejemplo para solucionar un problema de scheduling **[Ddiaz,98]** se deben combinar estos tres pasos en un modelo objeto, este modelo se debe dividir a su vez en dos grandes categorias de clase, una que nos permita representar el problema y otra que articule los medios para solucionar el mismo. Además, el diseñador también debe tener en cuenta con qué artefactos de software cuenta para poder implementar los mismos, esto le permite por un lado, economizar una gran parte del tiempo que consume la creación de los mecanismos de búsqueda. Por otro lado, el beneficio de pensar en la arquitectura subyacente es el de articular las entidades creadas de manera que, la sola colocación de restricciones produzca una gran poda a priori del arbol de búsqueda. Otra necesidad que debe cubrir el diseñador tiene que ver con el tiempo de respuesta para encontrar una solución. Esto es un requerimiento muy solicitado por los usuarios.

1.3.1 Declaración de las variables restringidas

Las variables dominio son el medio para representar lo desconocido en este problema- el

color para los países - como variables dominio. Para cada país necesitaremos una variable dominio, cuyos posibles valores serán 0, 1, 2 y 3, que representan respectivamente los colores azul, blanco, rojo y verde.
Variables dominio:
Bolivia(0,3), Uruguay(0,3), Brasil(0,3),
Argentina(0,3), Chile(0,3), Paraguay(0,3);

1.3.2 Colocando Restricciones

La restricción *"distinta de"* nos permite expresar la idea de que dos variables no pueden asumir el mismo valor En nuestro ejemplo usaremos esta técnica para tratar el hecho que cuando dos países son fronterizos, ellos no pueden tener el mismo color. Para el ejemplo, las siguiente sentencia indica que Brasil y Bolivia no pueden tener el mismo color pues son fronterizos.
(Brasil != Bolivia)

Para colocar restricciones, lo haremos utilizando la palabra Colocar, que se usa de la siguiente manera:

```
// Restricciones
Colocar( Brasil != Bolivia);
Colocar( Brasil != Paraguay);
Colocar( Brasil != Argentina);
Colocar( Paraguay != Argentina);
Colocar( Paraguay != Bolivia);
Colocar( Bolivia != Chile);
Colocar( Argentina != Chile);
Colocar( Argentina != Uruguay);
```

1.3.3 Búsqueda de Soluciones

Generalmente, una simple colocación de restricciones produce una poda del espacio de soluciones, pero esto en general, no es suficiente para obtener soluciones. De esta manera, la simple colocación de restricciones normalmente abre una puerta a todas las posibles soluciones. Cuando queremos encontrar soluciones, necesitamos algo un poco más específico.

Una de las características de programación por restricciones es que la asignación de valores a las variables no se realiza ciegamente, más bien, se produce una propagación de las restricciones sobre los dominios de las variables. De esta manera se explotan los efectos de los cambios en algunas variables sobre otras variables que tiene el problema. Los efectos de la propagación de restricciones es la reducción de los dominios de las variables. De este modo las restricciones se pueden usar activamente para restringir los posibles valores de las variables tan pronto como sea posible. Así, los algoritmos proveen un camino eficiente para encontrar una solución rápidamente.

Para nuestro ejemplo, la búsqueda de soluciones consiste en elegir una variable que no esté instanciada (que no posea un valor asignado) e instanciarla (asignarle un valor), y repetir el proceso hasta agotar las variables. Debido a los efectos de la propagación de restricciones, puede ocurrir que no se necesiten instanciar todas las variables, sólo una pequeña parte de ellas.

1.4 Usar clases y funciones provistas por la arquitectura mencionada para implementar el modelo. - Actividad Nro 3

Para describir esta etapa es necesario describir la arquitectura sobre la cual se implementará el modelo.

1.4.1. Descripción de la arquitectura.

La arquitectura que se describe a continuación está basada en el modelo de objetos de la herramienta de Ilog, denominada Ilog-Solver **[ILOG,Sol-R] [ILOG,Sol-U] [ILOG,Sch-R] [ILOG,Sol-U].**

La arquitectura se fundamenta en dos grandes categorías de clases:

- La categoría de clases para la representación
  Esta contiene las clases que permiten capturar el conocimiento sobre el dominio del problema Las abstracciones más importantes de esta categoría de clases es la denominada variable dominio. Con cada una de estas variables restringidas se asocia un conjunto de posibles valores llamado, el dominio de la variable. Cuando el dominio de una variable contiene un solo valor, se dice que la variable está instanciada.

- La categoría de clases para la solución del problema
  La resolución de un problema consiste en seleccionar un valor perteneciente al dominio de cada variable de modo que se satisfagan todas las restricciones.

Convenciones para nombres

Los nombres para las variables globales como así también los nombres de las clases se escriben concatenadamente con la letra de comienzo de cada palabra en mayúscula. Por ejemplo:

LisiIntVar

Una letra en minúscula comienza los nombres de los argumentos, instancias, y funciones miembros Por ejemplo

aVar
LisiIntVar::getValue

Además suponemos que estas clases están implementadas sobre el lenguaje de programación C++ De esta manera, la sintaxis y semántica utilizada es la el lenguaje C++

1.4.1.1 Clases que representan tipos de datos básicos

LisiInt Este tipo representa enteros con signo que son manejados por las variables
LisiAny Este tipo representa un puntero a cualquier objeto
LisiBool Este tipo representa valores Booleanos ; estos valores son **False** y **True**.

1.4.1.2 Clases que representan variables restringidas y expresiones

Una variable restringida es una variable que posee un dominio que define los valores permitidos para la misma. Una variable restringida (también conocida como variable dominio) no puede instanciarse o ligarse con ningún valor fuera de los que el dominio que contiene. Nuestra arquitectura sólo contiene variables restringidas en el dominio de los enteros.

Una ***variable restringida entera*** es aquella cuyo dominio está compuesto por valores del tipo **LisiInt** (enteros). Su dominio se puede representar por un intervalo cuando los valores son consecutivos y por enumeración de enteros cuando no son consecutivos. La clase para variables restringidas enteras es **LisiIntVar.**

Las variables restringidas enteras se pueden combinar con operadores aritméticos conformando ***expresiones restringidas enteras***. La clase **LisintExp** es la clase raíz para las ***expresiones restringidas enteras***. Una variable restringida entera es en si misma una expresión restringida entera. La clase LisiIntVar es una subclase de LisiIntExp.

Reversibilidad. Todas las funciones y funciones miembros definidas para la clase LisiIntExp y para la clase LisiIntVar son reversibles. En particular, los cambios hechos usando las funciones de colocación de restricciones son llevados a cabo con asignaciones reversibles. Por lo tanto, el valor, dominio, y restricciones colocadas sobre cualquier variable restringida son restablecidos cuando se realiza backtracking.

1.4.1.3 Expresiones Restringidas Enteras LisiIntExp

Variables restringidas enteras y enteros ordinarios (instancias de la clase LisiInt) se pueden combinar conjuntamente para formar expresiones. Todo desarrollador puede construir expresiones aritméticas mediante la combinación de constantes númericas y variables restringidas usando los operadores suma + , resta -, multiplicación *, y división /.
Cada expresión restringida entera tiene un mínimo y un máximo. Decimos que la expresión esta ligada, o equivalentemente, instanciada, si su mínimo es igual a su máximo.

El dominio de una expresión restringida entera se puede reducir hasta el punto de llegar a ser vacio.

Existe más de un constructor para crear expresiones restringidas enteras, pero el operador = (igual) es el mas usado para crearlas. Por ejemplo, sean dos variables dominio x e y, el camino para crear la expresión suma es el siguiente:

```
LisiIntExp sum = x  + y;
```

Un segundo camino es el siguiente:

```
LisiIntExp sum;
sum  = x + y;
```

1.4.1.4 Variables Restringidas Enteras LisiIntVar

Las variables restringidas enteras están representadas por la clase LisiIntVar, el dominio de una variable restringida entera contiene valores del tipo LisiInt. Los dominios se representan por un intervalo cuando los valores son consecutivos, o por enumeración cuando no son consecutivos.

Constructores

LisiIntVar (LisiInt min, LisiInt max , char* name). Contruye una variable dominio cuyo dominio esta comprendido entre los valores min y max inclusive. Ej LisiIntVar prueba (0, 4) - variable dominio con dominio {0, 1, 2, 4}

Funciones de Acceso

LisiInt getMin(). Retorna el menor valor de dominio de una variable restringida (V.R.) Entera. Ej LisiInt domMin = prueba.getMin();

LisiInt getMax(). Retorna el mayor valor de dominio de una V.R. Entera. Ej LisiInt domMax = prueba.getMax();

LisiInt getValue(). Retorna el valor de una V.R. entera

LisiBool isBound(). Retorna LisiTrue si la V.R. entera esta ligada, o lo que es lo mismo, su mínimo es igual su máximo.

Funciones Modificadoras

Una función modificadora reduce el dominio de una expresión restringida entera, si es que puede hacerlo

setValue(LisiInt value). Remueve del dominio de la V.R. todos los valores que son diferentes de value. Esta acción dispará una propagación de restricciones, podado los dominios de las variables que estan relacionados con dicha variable a la que se le aplica SetValue

1.4.1.5 Restricciones

El mecanismo fundamental por el cual esta arquitectura permite encontrar soluciones son las restricciones. Estas son condiciones presentes en todo momento luego de que la restricción es tomada en cuenta por el programa, o sea cuando la restricción es colocada. Una restricción esta ligada a una o más variables. Los dominios de todas las variables restringidas del programa deben ser consistentes, lo cual es equivalente a cumplir todas las restricciones colocadas. Por cada cambio que sufre una variable, se revisan todas sus restricciones asociadas y en post de conservar consistente los valores de su dominio con las restricciones asociadas, se puede producir una reducción en su dominio. Esta reducción puede producir también una reducción en los dominios de las variables asociadas mediante restricciones, a esta acción se la denomina ***propagación de restricciones***, la cual tiene como efecto reducir los dominios de las variables restringidas del programa, dando lugar a una reducción en el espacio de búsqueda del problema.

La colocación de restricciones se realiza mediante la función LisiTell, la misma coloca la restricción y produce una primera propagación sobre las variables que asocia dicha restricción.

Ejemplo:

```
LisiTell ( x != y );
```

Restricciones en variables restringidas enteras

Definir y colocar restricciones sobre variables restringidas enteras es una tarea fácil debido a los operadores de definición de restricciones que brinda LisiIntVar. Por ejemplo supongamos que se tienen dos variables restringidas enteras ***x*** e ***y*** , lo que se desea es que x sea siempre mayor que y, el siguiente código muestra como hacerlo.

```
LisiIntVar x(0,5);
LisiIntVar y(0,10);
```

LisiTell(x>y);
La tercera línea ha definido y colocado una restricción sobre las variables x e y lógicamente los dominios de x e y sufrirán modificaciones luego que la tercera linea se ejecute, dominio de x = { 1, 2, 3, 4, 5}, dominio de y = { 0, 1, 2, 3, 4}

Restricción igual "==". Esta restricción condiciona a que el valor con el cual se liguen cada una de las variables sean iguales. Por ejemplo, dado el siguiente código:
LisiIntVar x(0,5);
LisiIntVar y(0,10);
LisiTell(x == y);
Los dominios de cada variables después de la tercera línea son
x = { 0,1,2,3,4,5}
y = { 0,1,2,3,4,5}
Los valores de *y* comprendidos entre 6 y 10 se han extraído porque nunca sastisfacerán la restricción, cualquiera de los valores restantes pueden sastifacer la restricción. Una vez que una de las variables se lige con algún valor, la otra se ligará con el mismo valor. La siguiente sentencia liga el valor de "x" a 2, entonces por propagación de restricciones "y" queda ligada a 2
x.setValue(2);
Los dominios de cada variables después de ejecutar la sentencia son :
x = { 2 } y = { 2 }
Restricciones existentes para LisiIntVar
Igual a ==
Distinto !=
Menor o igual que <=
Menor que <
Mayor o igual que >=
Mayor que >

1.4.1.6 Arreglos de Variables Restringidas Enteras y Restricciones genéricas

La clase LisiIntVarArray es la clase para un arreglo de instancias de LisiIntVar.

El arreglo hace más fácil la implementación de restricciones genéricas (restricciones que se aplican a un grupo de variables). En este contexto, una restricción genérica es una restricción que se aplica a todas las variables del arreglo. Existen funciones miembros de la clase arreglo, disponibles para la colocación de tales restricciones genéricas. De esta manera una restricción genérica se coloca y se guarda solamente una vez para todas las variables en el arreglo.

Constructores

LisiIntVarArray (LisiInt size, LisiInt min ,LisiInt max). Contruye un arreglo de variables dominio cuyo dominio esta comprendido entre los valores mín y máx inclusive. Ej LisiIntVarArray A(3,0,4), contiene las siguiente variables A[0]={0,1,2,3,4}; A[1]={0,1,2,3,4}; A[2]={0,1,2,3,4}

Funciones de Acceso y asignación

opèrator [] . LisiIntVar& LisiIntVarArray::operator[](LisiInt index) Este operador retorna una variable restringida entera correspondiente a un índice dado en el arreglo de variables restringidas enteras. Por ejemplo LisiIntVar var1=A[1]

Expresiones Restringidas Enteras con Arreglos
Las siguientes funciones miembros devuelven expresiones restringidas enteras las cuales permiten formar restricciones genéricas
LisiSum
LisiIntExp LisiSum(const LisiIntVarArray array). Esta función crea una expresión restringida entera igual a la suma de los elementos del arreglo.
LisiMax Esta función permite crear una expresión restringida entera igual al máximo de los elementos del array.
LisiMin Esta función permite crear una expresión restringida entera igual al mínimo de los elementos del array.

Restricciones sobre Arreglos de Variables Restringidas Enteras.
Para formar restricciones sobre arreglos se utilizan las expresiones anteriores y los operadores para restricciones sobre variables restringidas enteras. Por ejemplo para colocar que la suma del arreglo A sea igual 3, se tiene el siguiente código:

```
LisiIntVarArray A(3, 0,4);
LisiIntExp suma=LisiSum(A);
LisiTell(suma==3);
```

LisiAllDiff Es una restricción sobre un arreglo que hace que las variables restringidas de un arreglo tomen valores diferentes unas de otras, cuando estas son instanciadas.

```
LisiConstraint LisiAllDiff (const LisiIntVarArray array);
```

Ejemplo. Los valores que toman todas las variables del arreglo A, deben ser distintos entre sí.

```
LisiIntVarArray A(3, 0,4);
LisiTell(LisiAllDiff(A));
```

1.4.1.7 La categoría de clases para la solución del problema

Clase Goal (Objetivos).
La búsqueda se basa en la idea de goals. Una **goal** es un objeto cuyo comportamiento lo define el algoritmo de búsqueda que soporta, esta puede tener éxito o fallar. Conocer cómo ha sucedido, o sea, si el algoritmo de búsqueda ha podido o no encontrar una solución, permite tomar decisiones sobre los posibles caminos de búsqueda existentes. Una solución es alcanzada usando el backtracking y disparando goals. De esta manera uno puede usar las goals para guiar la búsqueda aunque no conozca nada acerca de camino exacto por el cual se llega a una solución. Desde luego, si todos los caminos posibles se agotan, una falla se producirá y ninguna solución será encontrada.
Las goals se pueden combinar usando el operador And y el operador Or, los cuales son respectivamente LisiAnd e LisiOr. El operador LisiOr se usa para definir un punto de elección entre dos subgoals para que en caso de que si la ejecución de primera subgoal conduce a inconsistencias, el estado del sistema anterior a la ejecución de esta se restablezca, y la segunda goal sea ejecutada.

La clase LisiInstantiate
Esta clase deriva de la clase base goal y recibe como argumento una variable restringida y le asigna un valor del dominio. Entonces automáticamente se propagan las restricciones colocadas sobre esta variable restringida. Si ningún fallo ocurre durante esta propagación la goal termina satisfactoriamente, caso contrario falla, el valor del dominio es descartado y

prueba con otro hasta que la goal sea satisfecha, o no queden más valores en el dominio de la V.R. en cuyo caso la goal falla.

La clase LisiGenerate

Esta goal recibe como argumento un arreglo de variables restringidas y le asigna a cada variable restringida un valor de su dominio.

LisiGenerate se puede pensar como una goal que iterativamente selecciona una variable restringida (o más precisamente, un índice de un arreglo determinado), llama a LisiInstantiate para que instance esta variable restringida, luego LisiGenerate llamará subsecuentemente a LisiInstantiate para instanciar las variables restringidas restantes. En la selección de las variables restringidas que sé instancian interviene la función **chooseIndex**. Esta tomará el arreglo a instanciar y determinará cual variable instanciar primero. Estas funciones se conocen como criterios de selección de variables y uno de los parámetros de LisiGenerate es el nombre de la función o criterio de selección de variable.

La sintaxis para LisiGenerate es:

LisiGenerate(LisiIntVarArray vars, LisiChooseIntIndex chooseIndex

1.5 Mapeo del modelo diseñado en la arquitectura propuesta - Búsqueda de soluciones - Actividad Nro 4

Para el ejemplo propuesto el mapping entre el modelo y la arquitectura propuesta es de la siguiente manera

```
1)  LisiIntVar Bolivia(0,3), Uruguay(0,3), Brasil(0,3), Argentina(0,3),
            Chile(0,3),   Paraguay(0,3);

2)  LisiIntVarArray coloresParaPaises(6,  Bolivia, Uruguay, Brasil, Argentina,
                                          Chile, Paraguay);
      // Restricciones
3)    LisiTell( Brasil != Bolivia);
      LisiTell( Brasil != Paraguay);
      LisiTell( Brasil != Argentina);
      LisiTell( Paraguay != Argentina);
      LisiTell( Paraguay != Bolivia);
      LisiTell( Bolivia != Chile);
      LisiTell( Argentina != Chile);
      LisiTell( Argentina != Uruguay);

    //Búsqueda de una solución
4)  LisiSolve(LisiGenerate(coloresParaPaises));
    // Imprimir
    print( "Bolivia:", Bolivia);
    print ( "Uruguay:", Uruguay);
    print ("Brasil:", Brasil);
    print ("Argentina:", Argentina);
    print ("Chile:", Chile);
    print ("Paraguay:", Paraguay);
```

Explicación
1) Creo las variables dominio
2) Se construye un arreglo de variables dominio, el cual contendrá todas las variables dominio que representan los países, este arreglo constituye una ayuda a la hora de buscar soluciones.
3) Se colocan todas las restricciones
4) Se implementa la búsqueda de soluciones.

Para problemas complejos, la dificultad para abordar esta fase dependerá de la etapa de diseño.

Conclusiones.

Fundamentalmente, la metodología planteada, ha demostrado en una forma simple como construir un programa utilizando la programación con restricciones. En cuanto a la aplicabilidad, las actividades propuestas son suficientes para problemas simples, mientras que para problemas reales la misma establece pautas generales de como abordarlos, las cuales son el fruto de la investigación aplicada que desarrolla el LISI [**Rueda,95**], [**Ddiaz,98**].
Con respecto a la arquitectura esta tiene como objetivo mostrar de manera sencilla como se alcanza una implementación de un problema sencillo, como así también cuales son artefactos básicos con los que cuenta una herramienta para programar por restricciones.

Referencias

[**Booch, 91**] Grady Booch, Object-Orient Design with Aplications, Benjamin/Cummings, Redwood City, Calif., 1991.
[**Hente,89**] P. Van Hentenryck, "Constraint Satisfaction in Logic Programming" MIT Press, 1989.
[**ILOG,Sch-R**] "ILOG SCHEDULE- Reference Manual Version 2.0", Ilog, France, 1995.
[**ILOG,Sch-U**] "ILOG SCHEDULE - User Manual Version 2.0", Ilog, France, 1995.
[**ILOG,Sol-R**] "ILOG SOLVER - Reference Manual Version 3.0", Ilog, France, 1995.
[**ILOG,Sol-U**] "ILOG SOLVER - User Manual Versión 3.0", Ilog, France, 1995.
[**Jaffa, 87**] J. Jaffar and J-L. Lassez. Constraint Logic Programming, ACM, 1987.
[**Rueda,95**] Rueda L. Klenzi R. Gutierrez L. Ibañez F. Forradellas R., "Tratamiento de Problemas de Combinatoria Discreta mediante el Paradigma CLP", 2das. Jornadas de Inteligencia Artificial, Universidad Nacional del Sur, Bahía Blanca, 1995.
[**Ddiaz,98**]Daniel Diaz Araya - Aplicación de Tecnologías Basadas en Restricciones a la Resolución de Problemas Industriales -Tesis de Grado , Universidad Nacional de San Juan, 1998.