Diseños de visualizaciones para programación lógica con restricciones

Manuel Carro, Manuel Hermenegildo
Facultad de Informática, Universidad Politécnica de Madrid

<{mcarro,herme}@fi.upm.es>

Resumen: *en este se estudia diseños y e implementaciones de paradigmas visuales para observar la ejecución de programas lógicos con restricciones, enfocados hacia la depuración, optimización y enseñanza. Nos centraremos en la representación de datos en ejecuciones* ***CLP****, donde perseguimos la representación de variables con restricciones y de las restricciones en sí mismas. Se han implementado dos herramientas,* ***VIFID*** *y* ***TRIFID****, que utilizan dichas representaciones y que se usan para mostrar la utilidad de las visualizaciones desarrolladas.*

Palabras clave: *programación lógica, programación lógica con restricciones, visualización, depuración, abstracción de representaciones.*

1. Introducción

La visualización de programas se ha utilizado en la Informática con muchos propósitos diferentes, que incluyen la enseñanza, la depuración y la optimización. Sin embargo, las visualizaciones clásicas son a menudo demasiado dependientes de los paradigmas de programación para las que se desarrollaron, y no se adaptan bien a la naturaleza de las computaciones realizadas en otros paradigmas: por ejemplo, la visualización de programas concurrentes se centra en aspectos que no están presentes en la programación secuencial. En particular, la naturaleza de la programación con restricciones (**CP**) difiere radicalmente de la de otros paradigmas, y la visualización debe solucionar otros problemas. Aún más, las necesidades de los usuarios de CP son también diferentes de aquellas de los usuarios de otros paradigmas.

En cualquier caso, una buena representación pictórica es fundamental para conseguir una visualización útil. Por tanto es importante diseñar representaciones que se ajusten a las características de los datos y el control de la CP. Adicionalmente, un problema que aparece ya en la visualización de ejecuciones de tamaño medio es la gran cantidad de información normalmente disponible. Para tratar estos (relevantes) casos se necesitan *abstracciones* de las representaciones. Idealmente, tales abstracciones deberían mostrar las características más interesantes (según los objetivos particulares de cada visualización, que pueden diferir de otras) sin sobrecarga de detalles innecesarios.

2. Programación lógica con restricciones: una brevísima introducción

La programación con restricciones (CP) es «uno de los desarrollos de lenguajes de programación más relevantes de la última década» [MS98]. La CP trata de programar utilizando las ecuaciones que caracterizan la solución de un problema. Esas ecuaciones, que son similares a las aritméticas, pueden variar sobre un amplio abanico de dominios: enteros, reales, términos (es decir, estructuras de datos), listas, conjuntos,... Un sistema CP resolvería automáticamente las ecuaciones, resultando en una solución al problema inicial. Esto está indudablemente relacionado con las matemáticas y la investigación operativa, pero se separa de ellas en dos características: la posibilidad de crear las ecuaciones dinámicamente (y posiblemente eliminar algunas de ellas en algún punto y añadir otras) y el uso de dominios que normalmente no se utilizan en matemáticas.

La programación lógica con restricciones (**CLP**) [Van89] reúne las bases de programación con restricciones con las ideas de la programación lógica (**LP**), lo que resulta en una combinación altamente sinergética. Las propiedades las variables lógicas (asignación única, unificación) y el control que normalmente se usa en LP (búsqueda automática con *backtracking*, retraso de objetivos) encaja particularmente bien con la programación con restricciones. El resultado es una familia de lenguajes que extiende naturalmente la LP en un entorno unificado, de modo que la LP puede verse como un caso particular de CLP, y resuelve algunas debilidades (especialmente en aritmética) que existen en los lenguajes LP.

Figura 1. Representación de una variable FD

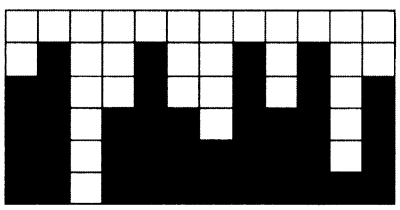


Figura 2. Historia de una variable

Uno de los dominios CLP más útiles es el denominado *dominios finitos* (**FD**): las variables FC toman valores sobre conjuntos finitos de enteros, y las operaciones permitidas sobre ellas son extensiones de las operaciones y relaciones aritméticas. Aunque no hay un procedimiento de eliminación similar al de las ecuaciones lineales, las ecuaciones sobre FD son decidibles gracias a la finitariedad del dominio, y se puede hallar (de existir) una solución mediante una mezcla de simplificación, propagación de valores y enumeración. Nos centraremos en este dominio debido a su importancia práctica.

3. Mostrando variables con restricciones

El concepto de valor de una variable en CLP es más complejo que en la programación imperativa y funcional: ese valor es, en realidad, un objeto complejo que representa un conjunto potencialmente infinito de valores más las restricciones que relacionan a esa variable con las demás del programa. Una representación textual es normalmente no muy informativa y difícil de interpretar y comprender, y una representación gráfica puede ofrecer una visión más fácil de entender. Además, si deseamos seguir la historia del programa es deseable que la representación sea animada en el tiempo, o dispuesta espacialmente como una serie de dibujos. Ésta última permita comparar diferentes comportamientos fácilmente, intercambiando tiempo por espacio.

3.1. Mostrando variables con dominios finitos

Las variables FD se instanciadas inicialmente a un dominio que se reduce según se añaden ecuaciones al conjunto de restricciones y éste se simplifica (mediante manipulación algebraica o mediante procedimientos de enumeración). En un estado cualquiera, cada variable FD tiene un dominio activo (el conjunto de valores que puede tomar) que suele estar accesible mediante primitivas del lenguaje. Por razones de espacio o velocidad, este dominio se representa habitualmente mediante una aproximación superior (un superconjunto) del verdadero conjunto de variables que la variable puede tomar.

Una posible representación gráfica asigna un punto (o, dependiendo de la visualización deseada, un cuadrado) a cada valor que la variable puede tomar, resaltando aquellos que están en el dominio actual. Un ejemplo de la representación de la variable con dominio actual de un dominio inicial se muestra en la **figura 1**.

Es extremadamente útil seguir la evolución de un conjunto de variables a lo largo de la ejecución. Probablemente la representación más útil simplemente apila las diferentes representaciones de los estados, como en la **figura 2**. El tiempo puede reflejarse con exactitud haciéndolo corresponder a la distancia entre las sucesivas filas, o ignorarse añadiendo una fila cada vez que hay un cambio en una variable o se realiza un paso en un proceso de enumeración. Esta representación permite comprar fácilmente distintos estados y tiene la ventaja adicional de permitir añadir más información relacionada con el tiempo. Otras posibilidades que no exploraremos son animar la representación, de modo que el tiempo se representa como tal, o usar diferentes tonos de color o matices de gris.

```
:- use_module(library(clpfd)).
:- use_module(library(tracing_library)).
program:-

    Variables
    = .... ,
    Names = .... ,
    open_log(Variables, Names, Handle), %% Added
    constrain_values(Variables, Handle),
    log_state(Handle) %%                          Added
    visual_labelling(Variables, Handle),
    close_log(Handle).                          %%Added
```

Figura 3. Un programa esquemático anotado

```
visual_labeling([ ], _Handle).
visual_labeling([ Q|Qs], Handle):-
                labeling([ Q] ),
                log_state(Handle),
                visual_labeling(Qs, Handle).
```

Figura 4. El predicado visual_labeling/2

El *apilamiento* es una de las visualizaciones disponibles en *VIFID*. *VIFID* es una librería Prolog que representa el estado de las variables en puntos del programa seleccionados por el usuario. La **figura 3** muestra un ejemplo esquemático de un programa anotado. La primitiva `open_log/4` inicializa la estructura de datos Handle que contiene las Variables para ser observadas y sus nombres (Names). `close_log/1` realiza las acciones necesarias para terminar la visualización (cerrar ficheros, mandar mensajes a las ventanas, etc.) La representación del estado se realiza mediante la primiti-

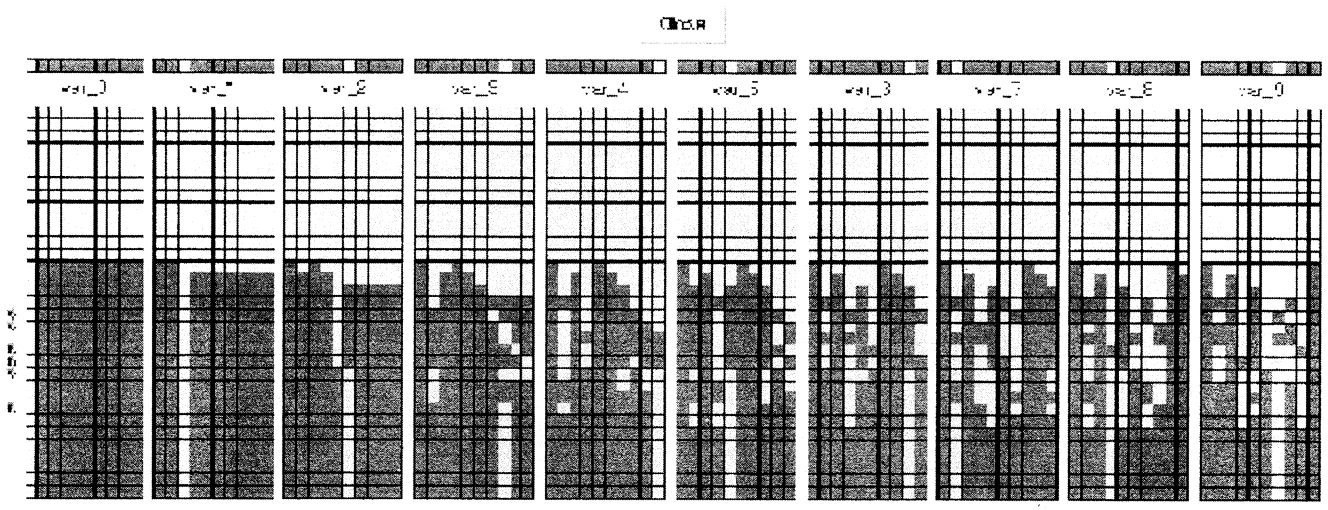


Figura 5. Evolución de variables FD para el problema de las 10 reinas

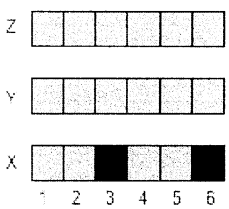


Figura 6. Varias variables adyacentes

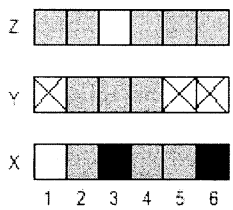


Figura 7. Cambiando un dominio

va `log_state/1`, que contacta con la parte visual de la herramienta para comunicarle el estado actual de las variables.

Una parte importante de la ejecución de programas CLP es la enumeración, que intenta asignar valores compatibles con las restricciones existentes. Esta enumeración se realiza normalmente mediante una primitiva que recibe una lista de variables y una indicación de la estrategia de enumeración. Podemos visualizar la evolución de las variables durante esta enumeración recodificando esa primitiva: la primitiva de la **figura 4** muestra una posible implementación, que recibe una lista de variables y el Handle y realiza una *enumeración que es visualizada. Este código está simplifi*cado, pero clarifica cómo esta y otras primitivas pueden tener un interfaz con las herramientas visuales sin demasiado esfuerzo.

La **figura 5** muestra un volcado de una ventana generada mediante *VIFID* que presenta la evolución de las variables en una resolución del problema de las reinas para un tablero de lado 10. Cada columna representa una de las variables del programa, y los valores posibles de cada una de ellas representan el número de las columnas que aún están en el dominio. Los cuadrados más oscuros representan valores descartados, y cada línea corresponde a un punto de espionaje en el programa fuente. Los puntos en que hubo *backtracking* están señalados con pequeñas flechas curvadas. Puede verse inmediatamente que fue necesario poco *backtracking* y que las variables están altamente restringidas, de modo que la enumeración (que tuvo lugar de izquierda a derecha) eliminó rápidamente valores iniciales.

4. Representando restricciones

Es obviamente interesante representar las relaciones entre varias variables según las restricciones que las afecten. Una representación textual no siempre es inmediata (o incluso posible, en determinados dominios de restricciones), puede ser computacionalmente cara, y da demasiados detalles para una comprensión intuitiva. Aún más, en general hay muchos estados de las variables que cumplen un conjunto dado de restricciones. Una solución general que utiliza la representación de los valores de una variable (y que es independiente de cómo se realice esta representación) es usar proyecciones para presentar los datos y permitir al usuario *cambiar los valores de las variables proyectadas, mientras se* observa como el resto de las variables se modifican. Esto puede dar al usuario una intuición de las relaciones entre las variables (y detectar, por ejemplo, la presencia de restricciones erróneas). Usaremos la restricción C1 en los ejemplos que siguen: $\mathbf{C1} \equiv X \in \{1..6\} \wedge X \neq 6 \wedge X \neq 3 \wedge Z \in \{1..6\} \wedge Z = 2X - Y \wedge Y \in \{1..6\}$

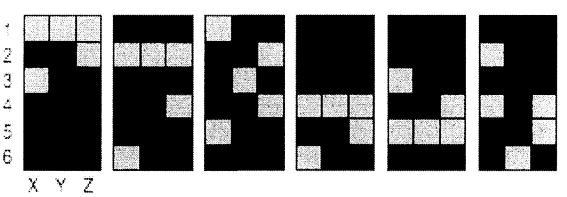


Figura 8. Enumeración de Y, representando los dominios para X y Z

La **figura 6** muestra los dominios de las variables FD X, Y

y Z sujetos a **C1**. Una modificación del dominio de una variable debería inducir cambios en los dominios de las variables relacionadas. Por ejemplo, podemos eliminar los valores 1, 5 y 6 del dominio de Y, que se reduce a representar la restricción C2 ... C1 | Y | 1 | Y < 5

La **figura 7** representa los nuevos dominios de las variables. Los valores directamente eliminados por **C2** se muestran con cruces en la caja; los valores eliminados por el efecto indirecto de esta restricción se muestran en un tono más claro. En este ejemplo, los dominios de X y Z se ven afectado por el cambio, y por tanto dependen de Y. Este tipo de visualización dirigida por el usuario está también disponible en *VIFID*. Puede hacerse una inspección más detallada dejando sólo un elemento en el dominio de una variable y observando cómo cambian los dominios de las otras. En la **figura 8**, Y recibe un valor definido desde 1 (en el rectángulo de más a la izquierda) a 6. Esto permite comprobar propiedades como la dependencia de la unicidad de valores del dominio entre variables --es decir, que una variable depende funcionalmente de otra.

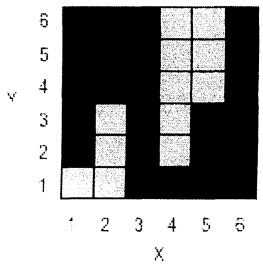


Figura 9. X contra Y

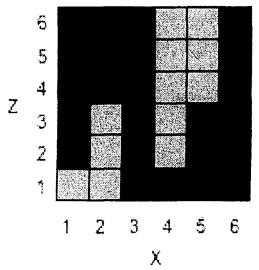


Figura 10. X contra Z

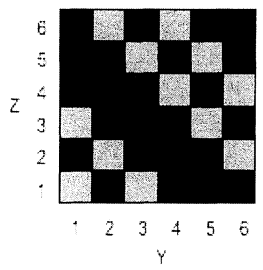


Figura 11. Y contra Z

Es posible obtener una versión estática dibujando pares de valores en una rejilla bidimensional. Esto se muestra esquemáticamente en las **figuras 9, 10 y 11**, donde las variables están sujetas a la restricción **C2**. De esas representaciones se puede deducir que los valores X = 3 y X = 8 no son admisibles, independientemente de los valores de Y y Z. También se ve que las proyecciones de X contra Z e Y contra Z son idénticas. De esto podría deducirse que la restricción Y = Z está implicada por el almacén de restricciones. Esta posibilidad está descartada por la **figura 11**, en la que vemos que hay pares consistentes donde X ' Z. Más aún, la pendiente de los cuadrados resaltados sugiere que hay una relación inversa entre Z y Y: incrementar uno de ellos posiblemente reduce el otro. Este es el caso, en realidad, como se ve en la restricción **C1**. Una ventana *VIFID* mostrando una representación bidimensional aparece en la **figura 12**; los botones en la parte inferior permiten seleccionar qué variables deben ser visualizadas.

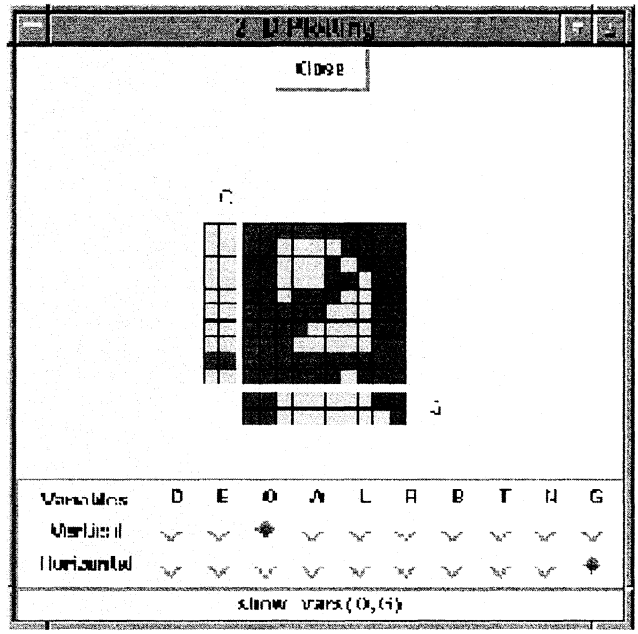


Figura 12. Relacionando variables en VIFID

5. Abstracción para la visualización de restricciones

El caso de ejecuciones de programas que terminan generando demasiados datos como para ser representados es frecuente. Incluso si la representación es muy fácil de comprender, la cantidad de datos puede dar al usuario un nivel de detalle innecesario o incluso contraproducente. La abstracción es un modo de tratar este problema.

5.1. Abstrayendo valores

Aunque la presencia de un gran número de variables puede resolverse parcialmente mediante una selección cuidadosa de las mismas, queda otro problema: la representación de variables con un gran número de posibles valores puede ser demasiado detallada. En el límite, la resolución de la pantalla puede ser insuficiente para asignar un *pixel* a cada valor del dominio. Esto se resuelve fácilmente mediante una ventana deslizable, aumento/disminución de la resolución, visiones de *ojo de pez*, etc. Sin embargo esos métodos son más físicos que abstracciones conceptuales de la información, que son más flexibles y ricas.

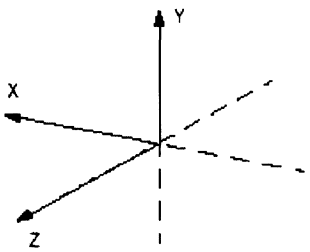
Una alternativa es usar un filtrado de los dominios orientado a la aplicación. Por ejemplo, si se confía en la corrección de algunas partes del programa, sus efectos pueden enmascararse suprimiendo de la representación de las variables los valores ya eliminados: por ejemplo, si se sabe que una variable puede tomar sólo valores pares, los valores impares simplemente no se muestran en la representación. Este filtrado puede especificarse utilizando el lenguaje fuente -- de hecho, la restricción que queremos abstraer es el filtro del dominio de las variables a representar.

Otra alternativa es realizar una compactación más semántica de determinadas partes del dominio. Como ejemplo, considérese presentar el dominio de una variable simplemente como un

número que denota cuantos valores quedan en el dominio, dando por tanto una indicación de su «grado de libertad». Esta idea es la base de nuestra siguiente visualización.

5.2. Compactación del dominio y nuevas dimensiones

Aparte de los problemas en aplicaciones con grandes dominios, la representación estática de la historia de la ejecución (**figura 5**) puede quedarse corta para mostrar intuitivamente cómo convergen las variables hacia sus valores finales, debido a un excesivo número de puntos en los dominios o porque una ejecución muestra un perfil «caótico». Una mejor opción es utilizar el número de valores activos en el dominio como coordenadas de una dimensión adicional. La **figura 14** muestra una programa CLP(FD) para el problema DONALD + GERALD = ROBERT. El programa se anotó con llamadas a los predicados que actúan como espías, y guarda los tamaños de los dominios de cada variable en el momento de la llamada.



X The CLP(FD) variables
Y An abstraction of the variable: the size of its domain
Z Time

Figura 13. Significado de las dimensiones la representación 3-D

```
:- use_module(library(clpfd)).
:- use_module(library(trifid)).

dgr(WhichOrder, ListOfVars):-
  ListOfVars = [G,O,B,N,E,A,R,L,T,D],
  open_log(ListOfVars, Handle),       %% Added
  domain(ListOfVars, 0, 9),
  log_state(Handle),                  %% Added
  D #> 0,
  log_state(Handle),                  %% Added
  G #> 0,
  log_state(Handle),                  %% Added
  all_different(ListOfVars),
  log_state(Handle),                  %% Added
  100000*D + 10000*O + 1000*N + 100*A + 10*L + D +
  100000*G + 10000*E + 1000*R + 100*A + 10*L + D #=
  100000*R + 10000*O + 1000*B + 100*E + 10*R + T,
  log_state(Handle),                  %% Added
  visual_labeling(ListOfVars, Handle),
  close_log(Handle).
```

Figura 14. El programa anotado DONALD + GERALD = ROBERT

La **figura 15** es una ejecución del programa de la **figura 14**. Las variables más cercanas al origen, que se enumeran antes, toman valores bastante pronto en la ejecución y permanecen sin cambios. Hay puntos de *backtracking* dispersos, que aparecen como bloques de variables saliendo del plano; hay también una variable (visible como una franja blanca en el medio de la representación) que aparece altamente restringida, de modo que es probablemente un buen candidato para ser enumerada al inicio. Otras variables tienen aparentemente una alta interdependencia: en caso de *backtracking* el cambio de una de llas afecta a las otras. Esto sugiere que las variables en este programa pueden ser clasificadas en dos categorías: una con variables fuertemente interrelacionadas y otra que contiene variables relativamente independientes de aquellas en el primer conjunto.

Esas figuras se generaron con una herramienta, **TRIFID**, integrada en el entorno ***VIFID***. Se produjeron mediante el paquete **ProVRML** [SCH99] que permite leer y escribir código VRML desde Prolog, con un enfoque similar al usado por PiLLoW [CH97]. Una ventaja de usar VRML es que hay visualizadores sofisticados de VRML disponibles para la mayor parte de las arquitecturas. El fichero VRML resultante puede ser cargado en un visualizador y rotado, expandido, etc. Otra razón para usar VRML es la posibilidad de utilizar hiperreferencias y mensajes para añadir información y animación a la ejecución sin sobrecargar la representación básica.

5.3. Abstrayendo restricciones

Según el número y la complejidad de las restricciones crece, visualizarlas como relaciones entre variables puede caer en los mismos problemas que encontramos al intentar representar las variables. Las soluciones que sugerimos para el caso de la representación de valores son aún válidas y pueden dar una intuición de cómo una variable se relaciona con las otras. Sin embargo, no es siempre fácil deducir de ellas cómo las variables se relacionan entre sí, debido a la falta de precisión en la representación de las variables.

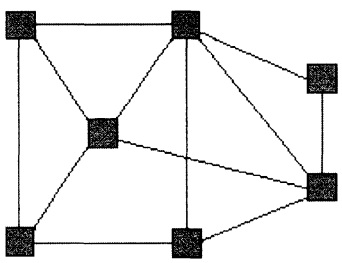


Figura 16. Restricciones representadas como un grafo

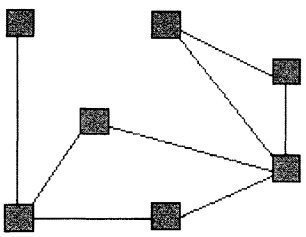


Figura 17. Los recuadros en negrita representan valores definidos

Un método diferente de abstraer las restricciones es mostrarlas como un grafo [MR91] donde las variables se representan como nodos que están enlazados si y sólo si las correspondientes variables están relacionadas mediante una restricción (**figura 16**, si bien esta figura es realmente sólo apropiada para el caso de relaciones binarias, siendo necesarios hipergrafos para restricciones de mayor aridad). Esta representación da al programador una comprensión aproximada de qué restricciones existen en el resolutor (pero no *cuáles* son). Más aún, dado que los diferentes resolutores se

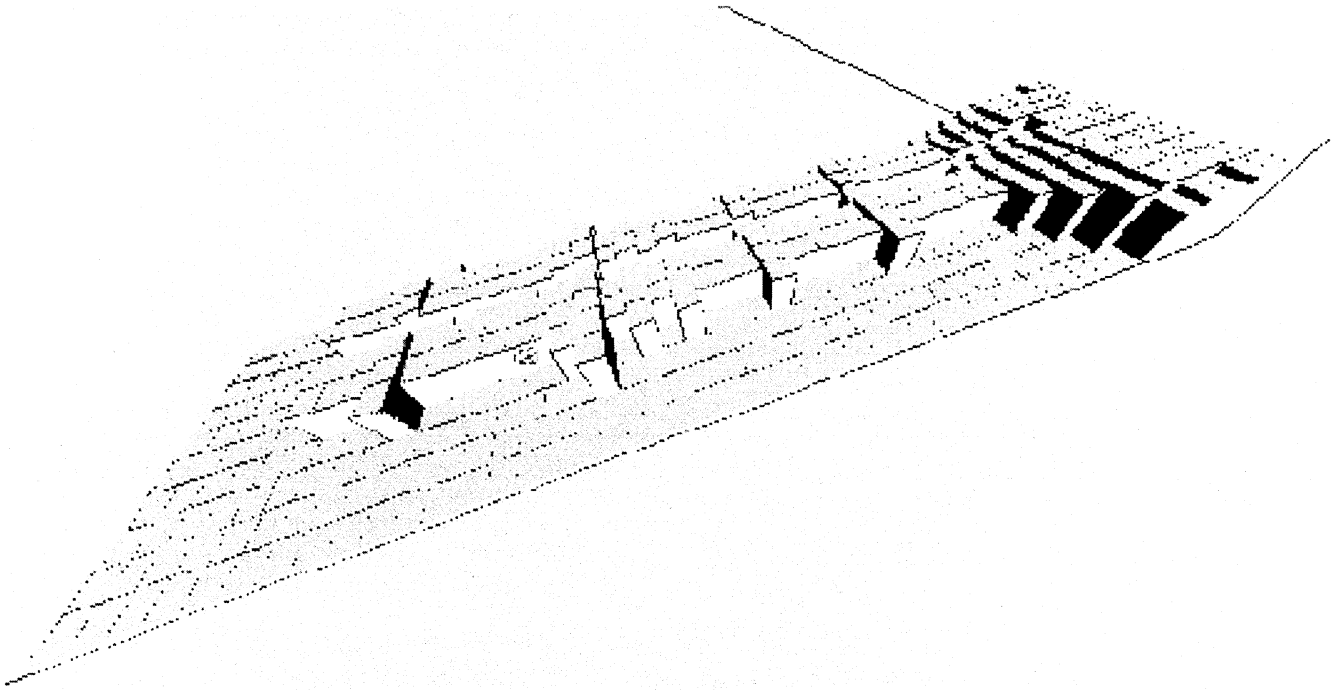


Figura 15. Ejecución del programa DONALD + GERALD = ROBERT

comportan de maneras distintas, esto puede proporcionar pistas sobre mejores modos de preparar las restricciones para un programa y resolutor determinado. La topología del grafo puede usarse para decidir si una reorganización del programa es ventajosa: por ejemplo, si hay subconjuntos de los nodos en el grafo con un alto grado de conectividad, pero esos subconjuntos están débilmente conectados entre si, puede ser beneficioso el preparar primero las secciones más fuertemente interconectadas y realizar una enumeración parcial antes, para después *enlazar* las diferentes regiones. Puede asimismo reflejarse la propagación de valores y la adquisición de valores únicos mediante una animación.

En la **figura 17** algunas variables han adquirido valores definidos, y como resultado algunas restricciones desaparecen: eso refleja la idea de un almacén de restricciones siendo progresivamente simplificado, y puede también visualizar cómo el *backtracking* afecta al almacén de restricciones. Un filtrado posterior puede seleccionar qué tipo de restricciones queremos visualizar (por ejemplo, solamente restricciones del tipo «mayor que», u otras marcadas en el texto del programa mediante anotaciones).

6. Detalles de implementación

VIFID y *TRIFID* están implementados en Prolog y Tcl/Tk, y usan un reducido número de primitivas para abrir conexiones a través de *sockets* y para crear y comunicarse con otros procesos (principalmente para el interfaz gráfico parte escrito en Tcl/Tk). *VIFID* es completamente interactivo, y dado qye la librería tiene acceso directo a las variables del programas, el usuario puede cambiarlas durante ejecución. Ésta puede continuar tras una modificación, pero el usuario no tiene que aceptar forzosamente este cambio: un botón de RESET fuerza al programa a retroceder al punto donde se introdujo el cambio. Sólo se escribieron directamente en Tcl/Tk unas pocas rutinas de uso común; la flexibilidad de Tcl/Tk fue suficiente, dado que la mayor parte de las ventanas tienen una disposición bastante sencilla. La velocidad de Tcl/Tk tampoco fue un problema, excepto cuando el número de objetos en la ventana se hacía muy grande. En conjunto la herramienta es suficientemente robusta como para ser usada rutinariamente.

TRIFID comparte muchas ideas con *VIFID*: es también una librería Prolog que analiza las variables a las que tiene acceso, pero en lugar de arrancar una visualización interactiva tridimensional, se decició aprovechar un interfaz existente de Prolog a VRML y generar VRML. Recoger los datos no fue problemático; sin embargo, encontramos problemas relacionados con el tamaño de los ficheros generados [3], y con la velocidad de los visualizadores de VRML disponibles libremente.

7. Trabajo relacionado

El sistema Eclipse [ECR93] fue uno de los primeros que contaron con sistemas de visualización de restricciones: **GRACE** [Mei96] representaba los valores de variables con restricciones de modo similar a como hicimos en la sección 3.1, conectado a un modelo de Byrd para la depuración de programas. El uso de tonos de color codificaba información adicional. Más recientemente, el proyecto **DisCiPl** [DHM00] fomentó el uso de herramientas de depuración basadas en la visualización y el uso de aserciones.

Algunas aplicaciones que usan restricciones necesitan generar relaciones complejas entre variables. En esos casos una visualización que imita el problema inicial ayuda a relacionar cuestiones de la resolución de restricciones con el problema inicial. La herramienta de *Global Constraint Visualization* [SABB00] hace precisamente esto, mediante la incorporación de visualizaciones a medida para algunos de las restricciones complejas diponibles en el **CHIP**. Aunque esto da una representación intuitiva, necesita que el

usuario convierta el problema en una las plantillas estándar de restricciones complejas.

La visualización de redes de restricciones, propuesta aquí como una abstracción, se ha implementado en el *Constraint Investigator* [TM99], que está enlazado con el *Oz Explorer* [Sch97]. Esta propuesta visualiza un grafo que es bastante cercano a la implementación. La posibilidad de expandir y colapsar la red y de filtrar las variables incrementa la utilidad de la herramienta en el caso de grandes ejecuciones. Da una buena representación del almacén de restricciones, pero posiblemente necesita una mayor estructuración para representar problemas complejos.

8. Conclusiones

Hemos visto algunas técnicas para visualizar la evolución de datos en CLP. Las representaciones gráficas se han escogido basándose en las necesidades de un programador que intenta analizar el comportamiento y características de una ejecución. Se han propuesto soluciones para la representación de valores y restricciones entre variables en tiempo de ejecución. Para poder manejar ejecuciones grandes, hemos visto algunas propuestas de abstracción, incluyendo la representación tridimensional de la evolución del tamaño del dominio de las variables. Las visualizaciones propuestas han sido probadas usando dos prototipos, *VIFID* y *TRIFID*. *VIFID* y en menor medida *TRIFID*, que es menos maduro, se han convertido en un sistema práctico. Algunas de las ideas propuestas han sido adoptadas en otras herramientas, como las desarrolladas por Cosytec para CHIP [SA00].

9. Bibliografía

[CH97] D. Cabeza y M. Hermenegildo. WWW Programming using Computational Logic Systems (and the PiLLoW/Ciao Library). En *Proceedings of the Workshop on Logic Programming and the WWW at WWW6*, San Francisco, CA, Abril 1997.

[DHM00] P. Deransart, M. Hermenegildo y J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming*. Número 1870 en LNCS. Springer-Verlag, Septiembre 2000.

[ECR93] ECRC. *Eclipse User's Guide*. European Computer Research Center, 1993.

[Mei96] M. Meier. Grace User Manual, 1996. Available at http://www.ecrc.de/eclipse/html/grace/grace.html.

[MR91] U. Montanari y F. Rossi. True-concurrency in Concurrent Constraint Programming. En V. Saraswat y K. Ueda, editores, *Proceedings of the 1991 International Symposium on Logic Programming*, páginas 694-716, San Diego, USA, 1991. The MIT Press.

[MS98] Kim Marriot y Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[SA00] H. Simonis y A. Aggoun. Search Tree Visualization. En P. Deransart, M. Hermenegildo y J. Maluszynski, editores, *Analysis and Visualization Tools for Constraint Programming*, número 1870 en LNCS. Springer-Verlag, Septiembre 2000.

[SABB00] H. Simonis, A. Aggoun, N. Beldiceanu y E. Bourreau. Complex Constraint Abstraction: Global Constraint Visualization. En P. Deransart, M. Hermenegildo y J. Maluszynski, editores, *Analysis and Visualization Tools for Constraint Programming*, número 1870 en LNCS. Springer-Verlag, Septiembre 2000.

[Sch97] Christian Schulte. Oz explorer: A visual constraint programming tool. En Lee Naish, editor, *ICLP'97*. MIT Press, Julio 1997.

[SCH99] G. Smedbäck, M. Carro y M. Hermenegildo. Interfacing Prolog and VRML and its Application to Constraint Visualization. En *The Practical Application of Constraint Technologies and Logic programming*, páginas 453-471. The Practical Application Company, Abril 1999.

[TM99] Tobias Müller. Practical Investigation of Constraints with Graph Views. Poster in International Conference on Logic Programming, Diciembre 1999.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.