Integración de ILOG CP en $\mathcal{TOY}$

Ignacio Castiñeiras[†] y Fernando Sáenz-Pérez[‡]

[†]*Dept. Sistemas Informáticos y Computación.* [‡]*Dept. Ingeniería del Software e Inteligencia Artificial.*
Universidad Complutense de Madrid
ncasti@fdi.ucm.es, fernan@sip.ucm.es

Resumen

Actualmente, el sistema lógico-funcional con restricciones $\mathcal{TOY}$ incorpora el resolutor de restricciones de dominios finitos ($\mathcal{FD}$) subyacente de SICStus Prolog. En este trabajo integramos la tecnología de resolución de restricciones $\mathcal{FD}$ de ILOG CP en $\mathcal{TOY}$ con el objetivo del aumento del rendimiento. Se revisa la arquitectura y funcionamiento de $\mathcal{TOY}$ relativa a $\mathcal{FD}$. Se explica la tecnología ILOG CP, que integra bibliotecas para la resolución de restricciones $\mathcal{FD}$. Se describe la implementación de ILOG CP en $\mathcal{TOY}$ haciendo énfasis en la comunicación entre ambos. Finalmente, se muestra una comparativa de rendimiento de $\mathcal{TOY}$ con ambos resolutores.

Keywords: Programación lógico-funcional, Programación con restricciones, Resolutores de restricciones, Rendimiento.

1. Introducción

$\mathcal{TOY}$[1] es un sistema implementado en SICStus Prolog 3.12.8 [11] cuya semántica operacional es un cálculo de estrechamiento perezoso que, actualmente, permite trabajar sobre el dominio de restricciones de igualdad y desigualdad de Herbrand ($\mathcal{H}$), el dominio de los reales ($\mathcal{R}$) con restricciones aritméticas (lineales y no lineales), los dominios finitos ($\mathcal{FD}$: Finite Domains) con restricciones sobre números enteros, conjuntos ($\mathcal{S}$) y dominios de cooperación ($\mathcal{M}$) con restricciones de comunicación para la resolución cooperativa entre $\mathcal{H}$, $\mathcal{R}$, $\mathcal{FD}$ y $\mathcal{S}$ [5,3]. Tanto los dominios $\mathcal{R}$ como $\mathcal{FD}$ hacen uso de los resolutores de restricciones de SICStus Prolog, mientras que el resto de dominios han de gestionar explícitamente sus propios mecanismos de resolución. El repertorio de restricciones $\mathcal{FD}$ que permite el sistema es comparable al existente en muchos sistemas CLP($\mathcal{FD}$). Para manejar estas restricciones, $\mathcal{TOY}$ usa un sistema de restricciones como uno de sus componentes [6].

En la parte izquierda de la Figura 1 se muestra la arquitectura de componentes genérica que conecta $\mathcal{TOY}$ con el sistema $\mathcal{FD}$ externo. Durante la resolución de

[1] Este trabajo ha sido parcialmente financiado por los proyectos TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407 y UCM-BSCH-GR58/08-910502

IX Jornadas sobre Programación y Lenguajes
I Taller de Programación Funcional
P. Lucio, G. Moreno y R. Peña, Eds.
San Sebastián, 8-11 de septiembre de 2009

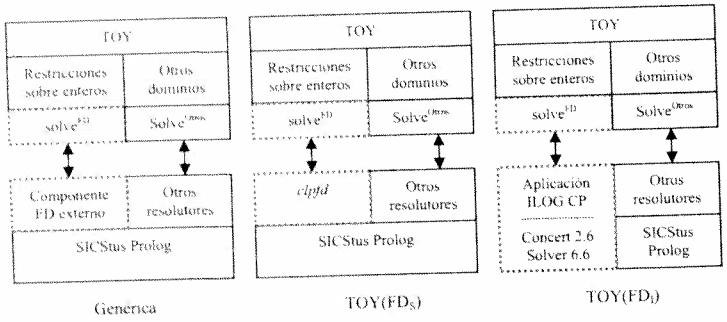


Figura 1 Arquitectura de componentes de $\mathcal{TOY}(\mathcal{FD})$. Genérica, $\mathcal{TOY}(\mathcal{FD}s)$, $\mathcal{TOY}(\mathcal{FD}i)$

objetivos. $\mathcal{TOY}$ identifica las restricciones sobre enteros, descomponiéndolas e introduciendo nuevas variables producidas si es preciso, hasta conformar restricciones primitivas. Cuando al fin encuentra una restricción $\mathcal{FD}$ primitiva, la transfiere a $solve^{\mathcal{FD}}$, que actúa como intermediario en la comunicación entre $\mathcal{TOY}$ y el sistema $\mathcal{FD}$ externo. $solve^{\mathcal{FD}}$ envía las restricciones a este sistema y recoge sus respuestas computadas.

1.1. *TOY con SICStus Prolog: $\mathcal{TOY}(\mathcal{FD}s)$*

Actualmente, $\mathcal{TOY}$ utiliza el sistema $\mathcal{FD}$ proporcionado por SICStus Prolog, que cuenta con un almacén de restricciones y un resolutor $\mathcal{FD}$, proporcionados por la biblioteca *clpfd*. La parte central de la Figura 1 muestra la arquitectura de componentes concreta que conecta $\mathcal{TOY}$ con el sistema $\mathcal{FD}$ de SICStus Prolog. A continuación se muestra un ejemplo básico de uso del sistema $\mathcal{TOY}$ con dominios finitos:

Example 1.1 Considérese el siguiente problema: `X` es un entero entre `5` y `12`, `Y` un entero entre `2` y `17`, y `X+Y = 17`, `X - Y = 5`. Se puede resolver en $\mathcal{TOY}(\mathcal{FD}s)$ en la siguiente sesión interactiva, donde la respuesta ofrecida por el sistema constituye una solución de intervalos. Ésta muestra los posibles valores que las variables pueden tomar así como las restricciones que dichos valores deben satisfacer.

```
TOY(FDs)> X #>= 5, X #<= 12, Y #>= 2, Y #<= 17,
          X #+ Y == 17, X #- Y == 5
   yes
   { 5 # + Y #= X,
     X # + Y #= 17,
     X in 10..12,
     Y in 5..7 }
   Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
   no
   Elapsed time: 0 ms.
```

Sin embargo, el sistema $\mathcal{FD}$ de SICStus Prolog presenta algunos inconvenientes:

- Trabajos recientes [2] han demostrado limitaciones en su capacidad de cómputo para hacer frente a problemas complejos.
- No es posible interactuar con el resolutor de restricciones *durante* los procesos predefinidos de búsqueda con objeto de podar el árbol de búsqueda más eficazmente con información específica del problema a resolver.
- La respuesta generada por el propio resolutor no incluye ninguna noción que facilite la depuración. De este modo, si el conjunto de restricciones tomado como entrada es insatisfactible, el resolutor no muestra el subconjunto de restricciones que no ha podido ser satisfecho.

1.2. *ILOG CP para mejorar $\mathcal{TOY}$*

ILOG CP 1.4 [7] ofrece una de las tecnologías industriales más competitivas del mercado. Su naturaleza es declarativa y dispone de un API C++ para el acceso a sus bibliotecas. Su resolutor trabaja como caja transparente, permitiendo la interacción durante el proceso de resolución. Incluye técnicas de depuración que ayudan al usuario a descubrir los fragmentos insatisfactibles del modelo de restricciones. Además, permite usar diferentes resolutores para el mismo dominio de aplicación. A pesar de su dificultad de uso como biblioteca C++, su integración en $\mathcal{TOY}$ oculta este inconveniente, ofreciéndose todo el poder expresivo de este sistema.

La estructura de las aplicaciones C++ ILOG CP 1.4, se basa en un aislamiento entre los objetos que modelan el problema planteado por el usuario y los objetos encargados de su posterior resolución. Siguiendo esta idea, los problemas se modelan en un lenguaje genérico, donde el usuario goza de una mayor flexibilidad al centrarse exclusivamente en definir las restricciones que caracterizan su problema, sin especificar cómo se debe resolver. Así, un mismo modelo puede ser resuelto por diferentes resolutores, que extraen la información contenida en el modelo, y crean, por cada uno de sus objetos, un nuevo objeto semánticamente equivalente, pero con una estructura interna dirigida a cada resolutor. Se puede acceder a cada elemento creado por el resolutor a través del elemento asociado contenido en el modelo. ILOG OPL Studio [8] es la herramienta paradigmática representante de esta filosofía. ILOG CP 1.4 proporciona la biblioteca ILOG Concert 2.6 con la tecnología necesaria para modelar problemas $\mathcal{FD}$. Asimismo, dispone para la resolución de estos modelos las siguientes bibliotecas, las últimas versiones a fecha actual (usadas también por CP Optimizer):

- ILOG Solver 6.6, para la resolución de problemas $\mathcal{FD}$ genéricos.
- ILOG Scheduler 6.6, con algoritmos específicos para la resolución de problemas de planificación.
- ILOG Dispatcher 4.6, con algoritmos específicos para la resolución de problemas de rutas.

La estructura de las aplicaciones ILOG CP se muestra en la Figura 2.

A continuación se ofrece una relación de los principales objetos de ILOG Concert 2.6 e ILOG Solver 6.6 necesarios en cualquier aplicación (véase [7] para más detalles):

- `IloModel` ***model***`(env)` Es el objeto principal de la fase de modelado. Contiene

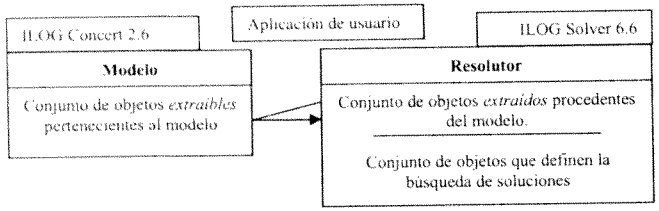


Figura 2: Estructura de una aplicación ILOG CP

el conjunto de objetos que formulan el problema $\mathcal{FD}$.

- `IloIntVarArray` ***vars***. Este vector referencia el conjunto de variables de decisión que componen el problema. Cada una de estas variables se modela mediante un objeto `IloIntVar` ***v*** indicando la cota inferior y superior de su dominio.
- `IloConstraint c`. Cada restricción involucra a un subconjunto de las `IloIntVar` contenidas en el vector `vars`.

- `IloSolver` ***solver***`(env)` Es el objeto principal de la fase de resolución. Extrae la información contenida en `model`, creando un nuevo objeto `IlcIntVar` asociado a cada `IloIntVar` y un nuevo objeto `IlcConstraint` asociado a cada `IloConstraint`. Estos nuevos objetos son semánticamente equivalentes a los contenidos en `model`, pero su estructura interna está orientada a las técnicas concretas de resolución que utiliza `solver` para computar las soluciones del problema.

2. $\mathcal{TOY}$ con ILOG CP: $\mathcal{TOY}(\mathcal{FD}i)$

A lo largo de esta sección se explica en detalle cómo integrar la tecnología $\mathcal{FD}$ ILOG CP en el sistema $\mathcal{TOY}$. En primer lugar, $\mathcal{TOY}$ es un sistema implementado en SICStus Prolog mientras que ILOG CP una tecnología implementada en C++. Estudiamos cómo realizar una conexión entre $\mathcal{TOY}$ e ILOG CP conectando SICStus Prolog y C++ desde la perspectiva de la integración de un recurso C++ externo (*foreign*) en una aplicación SICStus Prolog. Debido a la distinta naturaleza de ambos lenguajes, analizamos las dificultades existentes para establecer una correcta comunicación entre $\mathcal{TOY}$ e ILOG CP, así como las decisiones adoptadas para solventarlas. Por último, ilustramos el comportamiento del nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$ sobre un ejemplo.

2.1. *Conexión de SICStus Prolog con C++*

SICStus Prolog permite comunicar una aplicación escrita en este lenguaje con un componente C++ externo mediante vinculaciones entre hechos Prolog contenidos en la aplicación y funciones C++ definidas en el componente. Para que este componente sea integrado dentro de la aplicación Prolog debe consistir en una biblioteca dinámica con una estructura interna de archivos determinada. Asimismo, las funciones C++ de este componente externo deben ceñir sus parámetros al conjunto de conversiones *argumentos Prolog* <=> *argumentos C++* permitidos por SICStus.

Cada hecho Prolog debe indicar para cada uno de sus argumentos si es de entrada (enviado a la función C++) o de salida (calculado por la función C++). En particular, existe una conversión inmediata entre un término Prolog (sea cual sea) y el tipo C++ `SP_term_ref`. Mediante su gestión como `SP_term_ref`, las funciones C++ pueden hacer uso de:

- Métodos para la creación y asignación de un término Prolog.
- Métodos para la obtención del contenido de un término Prolog.
- Métodos de comparación y unificación de términos Prolog.

En este escenario se dan las condiciones necesarias para conectar $\mathcal{TOY}$ con ILOG CP realizando las siguientes modificaciones sobre la arquitectura de componentes, cuyo resultado puede verse en la parte derecha de la Figura 1.

- Desde el punto de vista de $\mathcal{TOY}$, se coloca un hecho Prolog en cada punto de $solve^{\mathcal{FD}}$ donde sea preciso comunicarse con ILOG CP (envío de una nueva restricción, aviso de nuevas variables, etc).
- Por otro lado, se construye una aplicación ILOG CP que integra las bibliotecas ILOG Concert 2.6 e ILOG Solver 6.6 conteniendo instancias de los principales objetos que permiten el modelado y resolución de problemas $\mathcal{FD}$. Esta aplicación incluye además el conjunto de funciones C++ vinculadas a los hechos Prolog existentes en $solve^{\mathcal{FD}}$.

Cada vez que $solve^{\mathcal{FD}}$ evalúa uno de estos hechos Prolog, realiza la conversión de sus argumentos y transfiere el control a la función C++, que los utiliza o computa dentro de su cuerpo. Al finalizar su ejecución, se transfiere de nuevo el control a $solve^{\mathcal{FD}}$, que pasa a evaluar la siguiente cláusula.

2.1.1. *Creación de un recurso externo C++ en una aplicación SICStus Prolog*

Para crear una biblioteca dinámica, e.g. `interface.dll`, y que pueda ser utilizada dentro de una aplicación SICStus Prolog se necesitan dos archivos:

- `interface.pl` Vincula cada hecho Prolog con una función C++. Agrupa todas las funciones C++ bajo un único recurso externo. Por ejemplo:
```
foreign(f1,p1(+integer)).
foreign(f2,p2(+term,-term)).
foreign_resource(interface,[f1,f2]).
```
- `interface.cpp` Codifica todas las funciones C++ vinculadas a un hecho Prolog. Añade las funciones auxiliares e importación de bibliotecas que sean necesarias. Por ejemplo:
```
void f1(long l){...}
void f2(SP_term_ref t1, SP_term_ref t2){...}
```

SICStus Prolog proporciona una herramienta, `splfr`, que actúa como macro que despliega una serie de comandos, utilizando las herramientas de compilación y vinculación que ofrece Microsoft Visual C++ [9]. `splfr` toma como entrada `interface.pl` e `interface.cpp` y permite crear la biblioteca dinámica, `interface.dll`. Para ello crean, a partir del contenido de `interface.pl`, dos nuevos ficheros, `interface_glue.c` e `interface_glue.h`, que codifican el código de interfaz

necesario para la aplicación SICStus Prolog. Para el ajuste de los parámetros utilizados en la creación de la biblioteca dinámica, SICStus Prolog recomienda ejecutar `splfr` una primera vez, estudiar las llamadas internas que realiza y reproducirlas alterando los parámetros deseados [10].

2.2. *Comunicación entre $\mathcal{TOY}$ e ILOG CP*

Se realiza una implementación $\mathcal{TOY}(\mathcal{FD}i)$ que reproduce el comportamiento de $\mathcal{TOY}(\mathcal{FD}s)$, aceptando como entrada el mismo repertorio de expresiones en los objetivos $\mathcal{TOY}$ y respetando el formato de respuestas ofrecidas al usuario. Para conseguir esto es necesario solventar dificultades surgidas de dos focos diferentes:

- Mientras que en $\mathcal{TOY}(\mathcal{FD}s)$ la comunicación entre $\mathcal{TOY}$ y su tecnología $\mathcal{FD}$ era más natural al estar ambos implementados en SICStus Prolog, en $\mathcal{TOY}(\mathcal{FD}i)$ la distinta naturaleza de los lenguajes SICStus Prolog y C++ hace necesario el uso de estructuras de datos adicionales para conseguir una correcta comunicación entre $\mathcal{TOY}$ e ILOG CP.
- La noción de solución a un problema $\mathcal{FD}$ de ILOG CP es diferente a la noción de solución de la tecnología $\mathcal{FD}$ de SICStus Prolog.

Comentamos los cuatro puntos que más dificultades han supuesto en la comunicación del nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$. Para ILOG CP se hace uso de la nomenclatura de objetos utilizada en la sección 1.2. Así, `model` se refiere al modelo de ILOG Concert 2.6, `solver` al resolutor $\mathcal{FD}$ genérico de ILOG Solver 6.6 y `vars` al vector que referencia a todas las variables de decisión contenidas en `model`.

2.2.1. *Gestión de las variables de decisión.*

El conjunto de restricciones $\mathcal{FD}$ existentes en un objetivo $\mathcal{TOY}$ involucran un conjunto de variables lógicas a las que denominaremos en lo sucesivo variables lógicas $\mathcal{FD}$. Para modelar este conjunto de restricciones $\mathcal{FD}$ con ILOG CP, es necesario:

- Crear tantas variables de decisión `IloIntVar` como variables lógicas $\mathcal{FD}$ se encuentren involucradas en el conjunto de restricciones $\mathcal{FD}$. Estas variables son añadidas a `model` y a `vars`, para poder ser referenciadas desde un único objeto.
- Encontrar una aplicación biyectiva que asocie de manera unívoca cada variable lógica $\mathcal{FD}$ existente en el objetivo $\mathcal{TOY}$ con una variable de decisión del vector `vars` de ILOG CP.
- Modelar cada restricción $\mathcal{FD}$ en ILOG CP sobre las variables de decisión del vector `vars` asociadas a las variables lógicas $\mathcal{FD}$ involucradas en dicha restricción $\mathcal{FD}$.

Es preciso hacer notar que para poder comunicar $\mathcal{TOY}$ con ILOG CP es necesario generar por cada variable del objetivo $\mathcal{TOY}$ hasta tres variables diferentes:

- La variable lógica $\mathcal{FD}$ contenida en $\mathcal{TOY}$.
- La variable de decisión modelada como objeto `IloIntVar` en `model`.
- La variable de decisión `IlcIntVar` generada como objeto específico por `solver`.

Para la asociación entre las variables lógicas $\mathcal{FD}$ y `vars` se realizó un primer intento que pretende gestionar un vector de `SP_term_ref` paralelo a `vars`. Los elementos de este vector de `SP_term_ref` contienen la conversión de las variables lógicas $\mathcal{FD}$. Así, cada vez que $solve^{\mathcal{FD}}$ envía una nueva restricción a ILOG CP, se buscan las variables lógicas $\mathcal{FD}$ en este vector. Si no se encuentra la variable, es seguro que se trata de una nueva no enviada previamente por $solve^{\mathcal{FD}}$. Por tanto, se añade esta variable al vector, pasando a ser su último elemento, de posición i. A continuación se crea una nueva variable de decisión `IloIntVar`, y se añade a `model` y a `vars[i]`. Cuando todas las variables lógicas $\mathcal{FD}$ están contenidas en el vector `SP_term_ref`, se detectan los índices que ocupan en el vector y se modela la restricción sobre las variables de `vars` asociadas a esos índices.

Sin embargo, este primer intento resulta fallido porque las reglas que rigen la gestión de dichos términos Prolog como `SP_term_ref` establecen que, cuando la función C++ termina su cómputo, el contenido de todos los `SP_term_ref` pasados a dicha función o creados dinámicamente dentro de ella deja de ser válido. Considérese como ejemplo una interfaz entre los siguientes tres predicados Prolog, `p1`, `p2` y `p3` asociados a las funciones C++ `f1`, `f2` y `f3`. Las funciones `f1` y `f2` reciben un término Prolog, mientras que `f3` recibe dos términos Prolog.

- Llamamos a `p3` con la variable lógica `X` por duplicado, es decir, `p3(X,X)`. Si en `f3(SP_term_ref t1, SP_term_ref t2)` efectuamos `SP_compare(t1,t2)`, el resultado indica que son el mismo término.
- Llamamos a `p1` con la variable lógica `X`, esto es, `p1(X)`. En `f1(SP_term_ref t1)` almacenamos `t1` dentro de una variable global vector `<SP_term_ref>`. La función `f1` termina, y devuelve el control a Prolog. Llamamos a `p2` de nuevo con la variable lógica `X`, esto es, `p2(X)`. En `f2(SP_term_ref t2)` efectuamos `SP_compare` entre `t2` y el término `t1` almacenado en el vector`<SP_term_ref>`. El resultado indica que son términos distintos. Sin duda se trata de la misma variable lógica en ambos casos, pero tras terminar la ejecución de `f1`, el `SP_term_ref` almacenado en el vector deja de ser válido. Por ello, la comparación posterior en `f2` da como resultado que se trata de términos distintos.

Por tanto, es necesario un segundo intento de asociación entre las variables lógicas $\mathcal{FD}$ y las variables de decisión de `vars`. En esta segunda ocasión, de resultado satisfactorio, la gestión se realiza dentro de la aplicación Prolog. Se utiliza la variable lógica **Cin** [4], común a todos los predicados Prolog del gestor $solve^{\mathcal{FD}}$, y que representa a un almacén mixto de restricciones. Se genera una lista con las variables lógicas $\mathcal{FD}$ y se añade dentro de **Cin**. Así, cada vez que se gestione el envío de una nueva restricción en $solve^{\mathcal{FD}}$, se contrastará si las variables lógicas $\mathcal{FD}$ pertenecen o no a la lista de variables contenida en **Cin**. Cada nueva variable se añade al final de dicha lista, con posición i, generando de inmediato una llamada a una función C++ que crea un nuevo objeto `IloIntVar`, que se añade a `model` y a `vars[i]`. Tras comprobar que todas las variables lógicas involucradas en la restricción $\mathcal{FD}$ pertenecen a la lista de variables contenida en **Cin**, $solve^{\mathcal{FD}}$ encuentra los índices que ocupan estas variables en la lista y llama a la función C++ que modela la restricción sobre las variables contenidas en estos índices.

2.2.2 *Sincronización de ILOG CP con $\mathcal{TOY}$*

$\mathcal{TOY}$ puede vincular sus variables lógicas $\mathcal{FD}$ mediante una restricción de igualdad en el resolutor de Herbrand. Por ejemplo, en el siguiente objetivo `TOY(FDi)> X #>= 0, X == 3` la variable `X` es vinculada al valor 3 debido a la unificación de términos Prolog que realiza el resolutor de Herbrand mediante su restricción de igualdad `X == 3`. Este hecho provoca que en la lista de variables lógicas $\mathcal{FD}$ contenida en **Cin**, `X` se vincule al valor 3, provocando una falta de sincronización entre esta variable y su variable de decisión asociada en el vector `vars`. Para sincronizar de nuevo el estado de ambas es preciso enviar a ILOG CP la restricción que iguala la variable asociada de `vars` al valor al que se ha unificado la variable lógica $\mathcal{FD}$.

Un primer intento pretende realizar esta sincronización mediante un demonio que capture el evento de vinculación. Para la captura de eventos, SICStus Prolog utiliza el módulo de variables atribuidas. Este módulo permite asignar atributos a una serie de variables lógicas. Además, ofrece el predicado *verify_attributes(+ Var, - Value, + Goals)*, que se dispara cada vez que una variable lógica atribuida se vincule a un valor. Se atribuye con el atributo `fd` a todas las variables lógicas $\mathcal{FD}$ existentes en el objetivo $\mathcal{TOY}$. Cuando mediante una restricción de igualdad el resolutor de Herbrand vincula alguna de estas variables, el demonio activa la llamada a la función C++ que sincroniza la variable asociada en el vector `vars`.

Sin embargo, este primer intento resulta fallido. Para poder conocer el índice de la variable del vector `vars` sobre la que actuar es necesario conocer la posición de la variable lógica $\mathcal{FD}$ dentro de la lista de variables de **Cin**. Sin embargo, el método `verify attributes` es rígido con respecto a sus parámetros de entrada, por lo que no se puede utilizar **Cin** como argumento. SICStus Prolog tampoco permite el uso de variables globales, por lo que es imposible utilizar el demonio *verify_attributes* para sincronizar la variable del vector `vars`.

Una segunda alternativa consiste en que sea el resolutor de Herbrand quien realice esa llamada a la función C++. Sin embargo, esta alternativa se debe desechar, ya que no cumple el principio básico de independencia de los distintos resolutores del sistema $\mathcal{TOY}$. La solución debe pasar por gestionar la sincronización de ILOG CP dentro del gestor de restricciones $\mathcal{FD}$ $solve^{\mathcal{FD}}$.

El tercer intento, de resultado satisfactorio, modifica la estructura de la lista de variables lógicas contenida en **Cin**, convirtiéndola en una lista de pares, donde el primer elemento de cada pareja contiene la variable lógica $\mathcal{FD}$ y el segundo elemento contiene un indicador (*flag*) que determina si la variable lógica $\mathcal{FD}$ vinculada ha sido sincronizada con la variable contenida en el vector `vars`. De este modo, mientras la variable lógica $\mathcal{FD}$ de cada pareja no se instancie, el valor del indicador permanece a `0`. Si se instancia, el valor del indicador cobra especial relevancia. Toda pareja con una variable vinculada y con valor 0 en su indicador representa que la variable de `vars` no se encuentra sincronizada con la variable de **Cin**. Si la variable está vinculada y con valor 1 en su indicador representa que la variable `vars` se encuentra sincronizada con la variable de **Cin**.

Cada vez que $solve^{\mathcal{FD}}$ envíe una restricción a ILOG CP debe sincronizar previamente todas las parejas de la lista que respondan al patrón [Variable lógica $\mathcal{FD}$ vinculada, 0]. Una vez que todas las variables de `vars` están sincronizadas, la nueva

restricción que $solve^{\mathcal{FD}}$ está gestionando puede ser enviada. Si no llegan más restricciones $\mathcal{FD}$, la sincronización se produce tras computar la última expresión del objetivo. Esta alternativa de sincronización presenta fuertes síntomas de ineficiencia. Observemos el siguiente objetivo:

`Toy(FDi)> X #>= 2, X == 1, X1 == 1, X2 == 1, ___ , X1000 == 1`

Por ser `X` la primera variable lógica del objetivo ocupará la posición `vars[0]` en el modelo ILOG CP. La sincronización de `X == 1` como `vars[0] == 1` convierte al problema modelado en insatisfactible. Por lo tanto, el objetivo $\mathcal{TOY}$ fallará, no continuando la evaluación de las restantes expresiones del objetivo.

Sin embargo, nuestra filosofía de sincronización hace que no se sincronice `vars[0] == 1` hasta el final del objetivo, tras evaluar `X1000 == 1`. Hasta ese momento, el objetivo $\mathcal{TOY}$ ha evaluado con éxito todas las expresiones, y sólo tras esta última sincronización encontrará inconsistente el problema $\mathcal{FD}$ planteado, devolviendo 'fallo'.

2.2.3 *Sincronización de $\mathcal{TOY}$ con ILOG CP*

ILOG CP debe sincronizar los valores obtenidos para las variables de decisión contenidas en `vars` con la lista de variables lógicas $\mathcal{FD}$ contenida en **Cin**. Para ello, $solve^{\mathcal{FD}}$ envía esta lista como argumento de entrada a todas las funciones C++ que implementan el modelado y resolución de una nueva restricción. Además, $solve^{\mathcal{FD}}$ coloca otro argumento, en este caso de salida, para que la función C++ compute en ella la nueva lista de variables lógicas $\mathcal{FD}$.

ILOG CP define una variable global con tipo vector `<int,int>`. Este vector es vacíado al inicio del cuerpo de cada una de las funciones C++. Cada uno de los pares almacena:

- El índice dentro del vector `vars` de la variable que ha resultado vinculada fruto de la propagación de la última restricción.
- El valor al que ha sido vinculada.

Una vez que la nueva restricción ha sido propagada, ILOG CP contendrá en este vector el conjunto de variables que deben ser sincronizadas con la lista de variables lógicas $\mathcal{FD}$. Además puede procesar esta lista (como una lista de `SP_term_ref`) dentro del cuerpo de la función. Por ello se utiliza un método auxiliar que recorre la lista de variables lógicas unificando las que hayan sido vinculadas en sus variables asociadas del vector `vars`.

Una posible forma de implementar este comportamiento dentro de ILOG CP sería comprobar una a una qué variables tenían un valor deducido antes de haberse añadido la nueva restricción y cuáles lo tienen después. Pero este método es ineficiente. Sin embargo, resulta mejor encontrar la manera de asociar un evento a cada variable del problema, permitiendo que cuando ésta se vincule se ejecute una cierta porción de código.

Las variables `IlcIntVar` (sobre las que actúa nuestro resolutor *solver*) tienen un método, `whenValue(IlcDemon d)`, que permite ejecutar un cierto demonio cuando se detecta el evento de que dicha variable ha sido acotada a un único valor. Un demonio `d` es un objeto asociado al método `m` de una restricción `c0`. Cuando se detecta el evento que activa el demonio, el control de programa pasa a ejecutar

inmediatamente el método `c0::m`.

Definimos un nuevo tipo de restricción unaria, `IlcCheckWhenBound`, que involucra una variable de decisión `v`. Esta restricción contiene un método `varDemon()` que añade al `vector<int,int>` la posición en `vars` de `v`, así como el valor al que `v` ha sido vinculada. Definimos el método de propagación de `IlcCheckWhenBound` para que propague, ejecutando `whenValue(IlcDemon d)` (disparando un demonio) cuando `v` haya sido acotada.

Definimos un nuevo demonio, `RealizeVarBound`, que ejecuta el método `varDemon()` de la restricción `IlcCheckWhenBound` que lo ha activado.

Cada vez que una nueva variable de decisión `vi` se añade al modelo, imponemos la restricción `IlcCheckWhenBound check_vi` sobre ella. Así, cuando `vi` se acota, `check vi` propaga, disparando el demonio `RealizeVarBound`. Éste ejecuta el método `check_vi::varDemon()`, añadiendo al `vector<int,int>` la posición en `vars` de `vi`, así como el valor al que `vi` ha sido vinculada.

2.2.4. *Noción de solución en ILOG CP*

Tras modelar y resolver cada nueva restricción enviada por $\mathcal{TOY}$, ILOG CP debe mostrar su estado interno, que indique los nuevos dominios que ha obtenido para las variables de decisión. Para ello, podemos utilizar los métodos `getMin()` y `getMax()`, que nos ofrezcan sus cotas mínimas y máximas. Asimismo, podemos utilizar `getMin()` y `getNextHigher()`, que muestre uno a uno los posibles valores del dominio. Finalmente, la solución escogida mostrará todos los intervalos de valores que puede tomar la variable, indicando para cada intervalo su cota mínima y cota máxima.

Sin embargo, dada la naturaleza del resolutor ILOG CP, éste no permite mostrar las restricciones enviadas al modelo en un formato simplificado. Esto es debido a que el resolutor de ILOG CP está pensado para encontrar una solución al problema planteado, donde por solución entiende asignar un valor a cada variable, de modo que la combinación de valores satisfaga el conjunto de restricciones.

Sin embargo, la utilización implícita que pretende hacer $\mathcal{TOY}$ del resolutor será aplicar tan sólo propagación sobre las restricciones. $\mathcal{TOY}$ sólo pretende obtener un etiquetado de las variables si explícitamente se lo pide al resolutor. Bajo el contexto de resolución por intervalos, es fundamental mostrar en la solución las restricciones simplificadas impuestas sobre el modelo. Como ILOG CP no permite la visualización de estas restricciones, es necesario guardar en la aplicación Prolog la lista de restricciones primitivas que van surgiendo en el objetivo $\mathcal{TOY}$, usando el almacén `Cin`.

2.3. *Ejemplo de aplicación*

En este apartado se detalla paso a paso la resolución del objetivo del ejemplo 1.1 en el nuevo contexto $\mathcal{TOY}(\mathcal{FD}i)$:

```
Toy(FDi)> X #>= 5, X #<= 12, Y #>= 2, Y #<= 17,
          X #+ Y == 17, X #- Y == 5
```

- Ejecución de `X #>= 5`

  Con la gestión de `X #>= 5`, $solve^{\mathcal{FD}}$ añade la nueva restricción a su lista de

restricciones contenida en **Cin**, generando **Cin1**. Además, dentro de esta nueva restricción detecta una nueva variable $\mathcal{FD}$ Prolog no incluida en la lista de variables Prolog contenida en **Cin1**. Por ello:

- Añade la variable `X` a la lista de variables de **Cin1**, generando **Cin2**. Esta variable ocupa la posición 0 dentro de la lista.
- Realiza la llamada a la función `C++` que crea una nueva variable entera ILOG dentro del vector de variables `vars`, hasta ahora vacío. La nueva variable será `vars[0]`. Con esto se consigue asociar la variable `X` Prolog y `vars[0]`.

En este momento todas las variables de la expresión $\mathcal{TOY}$ tienen su variable homóloga en el vector `vars`, por lo que $solve^{\mathcal{FD}}$ está en disposición de enviar la restricción a ILOG CP. Pasa a ejecutarse la función `C++ post_greater_equal`.

```
post_greater_equal: vars[0] >= 5
Feasible = 1
vars[0] in 5..2147483647;
```

Tras finalizar la ejecución de la función `C++`, el control vuelve de nuevo a $solve^{\mathcal{FD}}$. Éste finaliza la gestión de la restricción almacenando el nuevo contenido de **Cin2** como **Cout**, para continuar con la evaluación de la siguiente expresión.

- Ejecución de `X #<= 12`

```
post_lower_equal: vars[0] <= 12
Feasible = 1
vars[0] in 5..12
```

- Ejecución de `Y #>= 2`

```
post_greater_equal: vars[1] >= 2
Feasible = 1
vars[0] in 5..12, vars[1] in 2..2147483647
```

- Ejecución de `Y #<= 17`

```
post_lower_equal: vars[1] <= 17
Feasible = 1
vars[0] in 5..12, vars[1] in 2..17
```

- Ejecución de `X #+ Y == 17`

  La expresión `X #+ Y == 17` incluye una restricción compuesta que debe ser dividida hasta estar formada exclusivamente por restricciones primitivas. Así, es dividida en `X #+ Y == _Z, _Z == 17`

  - La primera es una restricción $\mathcal{FD}$ mientras que la segunda es una igualdad de Herbrand. Como restricción $\mathcal{FD}$, `X #+ Y== _Z` exige añadir previamente `X #+ Y == _Z` a la lista de restricciones de **Cin**, generando **Cin1**, añadir `_Z` a la lista de variables **Cin1**, generando **Cin2** y crear como variable homóloga a `_Z` la variable `vars[2]` en el vector de variables de ILOG CP. Tras estas acciones se activa la función `C++` que gestiona `X #+ Y== _Z`:

    ```
    post_addition: vars[0] + vars[1] == vars[2]
    Feasible = 1
    vars[0] in 5..12, vars[1] in 2..17, vars[2] in 7..29
    ```

  - Tras recuperar el control, $\mathcal{TOY}$ resolverá la segunda restricción `_Z == 17` en su resolutor de igualdades y desigualdades de Herbrand. Esto vinculará la variable

_Z en la expresión $\mathcal{TOY}$, y por ello también en la lista de variables Prolog de **Cin2**. Sin embargo, este cambio no será transmitido a ILOG CP inmediatamente. Se sincronizará antes de enviar la próxima restricción $\mathcal{FD}$ o al finalizar completamente la computación del objetivo $\mathcal{TOY}$.

- Ejecución de `X #- Y == 5`
  De nuevo la expresión `X #- Y == 5` se divide en `X #- Y == _T, _T == 5`:
  - La restricción `X #- Y == _T` es gestionada por $solve^{\mathcal{FD}}$. Como hemos indicado, de manera previa a proceder con la gestión de una nueva restricción, $solve^{\mathcal{FD}}$ sincroniza aquellas variables que han sido vinculadas en la lista de variables **Cin** pero no en el vector de variables `vars`. Por tanto, $solve^{\mathcal{FD}}$ comunica a ILOG CP que `_Z -> 17` mediante la llamada a la función C++ *post_equal*.
    ```
    post_equal: vars[2] == 17
    Feasible = 1
    vars[0] in 5..12; vars[1] in 2..17, vars[2] = 17
    ```
    Tras sincronizar todas las variables, se gestiona la restricción $\mathcal{FD}$ `X #- Y == _T`:
    ```
    post_subtraction: vars[0] - vars[1] == vars[3]
    Feasible = 1
    vars[0] in 6..12, vars[1] in 5..11, vars[2] = 17, vars[3] in 1..7
    ```
  - De nuevo la segunda restricción resuelve `_T == 5` en el resolutor de Herbrand. Esto permite vincular la variable _T en la expresión $\mathcal{TOY}$, y por ello también en la lista de variables Prolog de **Cin**, pero de nuevo, no inmediatamente en el vector de variables `vars`.
- El cálculo del objetivo no contiene más expresiones. Por último, es necesario sincronizar aquellas variables que han sido vinculadas en la lista de variables **Cin** pero no en el vector de variables `vars`:
  ```
  post_equal: vars[3] == 5
  Feasible = 1
  vars[0] in 10..12, vars[1] in 5..7, vars[2] = 17, vars[3] = 5
  ```
  Tras añadir esta última restricción, el cómputo del objetivo $\mathcal{TOY}$ termina completamente, mostrando como resultado el conjunto de restricciones contenido en **Cin** y las variables del objetivo $\mathcal{TOY}$, cuyos dominios tras la resolución del problema son requeridos al resolutor `solver` a través de las variables del modelo contenidas en `vars`.

```
      yes
      { X #+ Y #= 17,
      X #- Y #= 5,
      X in 10..12,
      Y in 5..7 }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```


3. Análisis de rendimiento

En este apartado se muestra el resultado de evaluar el rendimiento del nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$ con respecto al existente $\mathcal{TOY}(\mathcal{FD}s)$. Para ello se usan dos programas de prueba que modelan sistemas de `n` ecuaciones linealmente independientes de la forma $A * X = b$ cuyas `n` variables enteras `[X1, ..., Xn]`, de dominio $\{1..n\}$, tienen una única solución. La matriz `n` × `n` A está fijada y el vector b depende de la solución que se haya predeterminado. Toma para cada fila i el valor i en su coeficiente de la diagonal principal, y el valor 1 para el resto de sus coeficientes. Comparamos los tiempos de resolución (expresados en milisegundos) de los sistemas $\mathcal{TOY}(\mathcal{FD}s)$ y $\mathcal{TOY}(\mathcal{FD}i)$ (denotados $\mathcal{FD}s$ y $\mathcal{FD}i$ en las tablas, respectivamente) para instancias de ejemplos con 4, 12 y 15 variables. En cada caso distinguimos entre el uso de un procedimiento de búsqueda estático que selecciona las variables en el orden textual en que aparecen en el programa y el procedimiento de búsqueda dinámico 'First Fail'(denotado por ^{ff}). Ambos etiquetan los valores del dominio de cada variable en sentido ascendente. Los ejemplos escogidos son:

- La solución `[X1,...,Xn]` cumple: $\forall i \in \{1 \ldots n\}$ `Xi` $= i$. Los tiempos de resolución para este ejemplo son:

n	$\mathcal{FD}s$	$\mathcal{FD}s^{ff}$	$\mathcal{FD}i$	$\mathcal{FD}i^{ff}$
4	0	15	0	0
12	31	1.750	156	516
15	297	299.312	423	67.376

La ganancia de velocidad de $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a $\mathcal{TOY}(\mathcal{FD}s)$ para el primer ejemplo es:

n	$\mathcal{FD}i/\mathcal{FD}s$	$\mathcal{FD}i^{ff}/\mathcal{FD}s^{ff}$
4	1.0	-
12	5.0	0.29
15	1.42	0.22

Para este primer ejemplo, $\mathcal{TOY}(\mathcal{FD}i)$ precisa un mayor tiempo de resolución que $\mathcal{TOY}(\mathcal{FD}s)$ para la estrategia de búsqueda estática, mientras que para la estrategia de búsqueda dinámica su tiempo de resolución es menor. Esta tendencia aumenta cuanto mayor es el número de variables del problema. Observando los dominios resultantes tras la propagación inicial de las restricciones, se concluye que la estructura de la solución de este ejemplo favorece notablemente la estrategia que implementa el procedimiento de búsqueda estático y perjudica gravemente a la que implementa 'First Fail'. Esto nos indica que para problemas donde la exploración necesaria para encontrar la solución es escasa, el tiempo invertido en $\mathcal{TOY}(\mathcal{FD}i)$ en la gestión de las estructuras de datos que posibilitan la correcta comunicación entre $\mathcal{TOY}$ e ILOG CP convierte a este sistema en menos eficiente

que $\mathcal{TOY}(\mathcal{FD}s)$. Sin embargo, a medida que la exploración necesaria para encontrar la solución es mayor, esta pérdida de tiempo se compensa haciendo que el nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$ sea mucho más eficiente.

- La solución `[X1, ..., Xn]` cumple: $\forall i \in \{1 \ldots n\}$ `Xi` $:: n-(i-1)$.

n	$\mathcal{FD}s$	$\mathcal{FD}s^{ff}$	$\mathcal{FD}i$	$\mathcal{FD}i^{ff}$
4	16	16	16	31
12	531	250	437	126
15	15.563	21.968	13.937	3.406

La ganancia de velocidad de $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a $\mathcal{TOY}(\mathcal{FD}s)$ para el segundo ejemplo es:

n	$\mathcal{FD}i/\mathcal{FD}s$	$\mathcal{FD}i^{ff}/\mathcal{FD}s^{ff}$
4	1.0	1.93
12	0.83	0.50
15	0.90	0.16

La tendencia apuntada en el ejemplo anterior se confirma con este segundo ejemplo, donde $\mathcal{TOY}(\mathcal{FD}i)$ precisa para ambas estrategias menor tiempo de resolución que $\mathcal{TOY}(\mathcal{FD}s)$. Ahora, la estructura de la solución del ejemplo perjudica gravemente a la estrategia de búsqueda estática, mientras que ni perjudica ni beneficia excesivamente a la estrategia 'First Fail'. Para la primera estrategia, $\mathcal{TOY}(\mathcal{FD}i)$ obtiene un ligero mejor tiempo de resolución. Para la segunda estrategia, mucho más representativa al no tratar un caso extremo, los tiempos de resolución de $\mathcal{TOY}(\mathcal{FD}i)$ son mejores un órden de magnitud frente a los de $\mathcal{TOY}(\mathcal{FD}s)$.

4. Conclusiones y trabajo futuro

En este trabajo se ha estudiado cómo integrar la tecnología $\mathcal{FD}$ ILOG CP dentro del sistema $\mathcal{TOY}$. Se ha constatado que esta tecnología ofrece una serie de ventajas sobre la tecnología $\mathcal{FD}$ de SICStus Prolog. Se ha estudiado en detalle cómo realizar una conexión entre ambos sistemas, demostrando que la diferente naturaleza de los lenguajes sobre los que están implementados ambos sistemas dificulta una correcta comunicación entre ambos. Se ha comprobado, mediante la ejecución de programas de prueba, que el nuevo sistema $\mathcal{TOY}(\mathcal{FD}i)$ es más rápido que $\mathcal{TOY}(\mathcal{FD}s)$ conforme aumenta el tamaño del problema. No obstante, se ha observado una penalización en el rendimiento debido a la gestión de las estructuras de datos que sincronizan $\mathcal{TOY}$ con su nuevo componente $\mathcal{FD}$, por lo que este aspecto será objeto de trabajo futuro.

Referencias

[1] Arenas, P., S. Estévez, A. Fernández, A. Gil, F. López-Fraguas, M. Rodríguez-Artalejo and F. Sáenz-Pérez, *TOY: a multiparadigm declarative language - version 2.3.1*, (2007), r. Caballero and J. Sánchez (Eds.), Available at `http://toy.sourceforge.net`

[2] del Campo, R. G. and F. Sáenz-Pérez, *Programmed search in a timetabling problem over finite domains*, ENTCS **177** (2007), pp. 253–267

[3] Estévez, S., A. Fernández and F. Sáenz, *Cooperation of the Finite Domain and Set Solvers in TOY*, in: *Prole'09*, San Sebastián, Spain, 2009, accepted for publication

[4] Estévez-Martín, S., A. J. Fernández and F. Sáenz-Pérez, *About implementing a constraint functional logic programming system with solver cooperation*, in: *proc. of CICLOPS'07*, 2007, pp. 57–71

[5] Estévez-Martín, S., T. Hortalá-González, M. Rodríguez-Artalejo, R. del Vado Vírseda, F. Sáenz-Pérez and A. J. Fernández, *On the Cooperation of the Constraint Domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ in CFLP*, Theory and Practice of Logic Programming (2009), accepted for publication

[6] Fernández, A., T. Hortalá-González, F. Sáenz-Pérez and R. del Vado-Vírseda, *Constraint Functional Logic Programming over Finite Domains*, Theory and Practice of Logic Programming **7** (2007), pp. 537–582.

[7] ILOG, *ILOG Solver 6.6, Reference Manual* (2008)

[8] ILOG, *ILOG OPL Studio 6.1, Reference Manual* (2009).

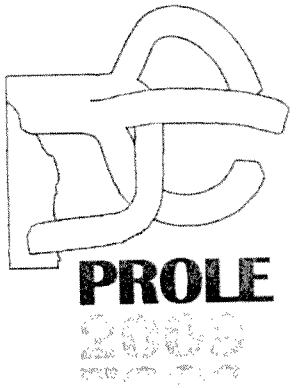
[9] Microsoft (2005), `http://msdn.microsoft.com/en-us/visualc/default.aspx`

[10] SICStus Prolog, *Using SICStus Prolog with newer Microsoft C compilers*. `http://www.sics.se/isl/sicstuswww/site/dontpanic.html`

[11] SICStus Prolog (2007), `http://www.sics.se/isl/sicstus`

Programación y Lenguajes

IX Jornadas sobre Programación y Lenguajes, PROLE'09
I Taller de Programación Funcional, TPF'09



San Sebastián, España

del 8 al 11 de Septiembre de 2009

Editores:

Paqui Lucio, Ginés Moreno y Ricardo Peña

Edita:
Paqui Lucio Carrasco
Ginés Moreno Valverde
Ricardo Peña Marí

Filmación e impresión:
Gráficas Michelena

Depósito Legal:
SS-990-2009

SBN:
978-84-692-4600-9

Comité de Programa de PROLE-2009

Presidente Comité: Ginés Moreno (U. de Castilla-La Mancha)

Elvira Albert (U. Complutense de Madrid)
Jesús Almendros (U. de Almería)
María Alpuente (U. Politécnica de Valencia)
Gilles Barthe (IMDEA)
Miquel Bertran (U. Ramón Llull)
Santiago Escobar (U. Politécnica de Valencia)
Antonio Fernández (U. de Málaga)
Victor Gulías (U. de A Coruña)
Paqui Lucio (U. del País Vasco)
Narciso Martí-Oliet (U. Complutense de Madrid)
Susana Muñoz (U. Politécnica de Madrid)
Marisa Navarro (U. del País Vasco)
Manuel Núñez (U. Complutense de Madrid)
Fernando Orejas (U. Politécnica de Catalunya)
Yolanda Ortega (U. Complutense de Madrid)
Francisco Ortín (U. de Oviedo)
Ernesto Pimentel (U. de Málaga)
Germán Puebla (U. Politécnica de Madrid)
Enric Rodríguez (U. Politécnica de Catalunya)
Jaime Sánchez (U. Complutense de Madrid)
Germán Vidal (U. Politécnica de Valencia)

Comité de Programa de TPF-2009

Presidente Comité: Ricardo Peña (U. Complutense de Madrid)

Victor Gulías (U. de A Coruña)
Francisco Gutiérrez (U. de Málaga)
Salvador Lucas (U. Politécnica de Valencia)
Pablo Nogueira (U. Politécnica de Madrid)
Julio Rubio (U. de la Rioja)
Alberto Verdejo (U. Complutense de Madrid)
Mateu Villaret (U. de Girona)

Comité Organizador de JISBD/PROLE-2009

Presidenta Comité: Goiuria Sagardui (U. de Mondragón)
Vicepresidenta Comité: Paqui Lucio (U. del País Vasco)
Gentzane Aldekoa (U. de Mondragón)
Ana Altuna (U. de Mondragón)
Javier Álvez (U. del País Vasco)
Lorea Belategui (U. de Mondragón)
Leire Etxeberria (U. de Mondragón)
Joxe Gaintzarain (U. del País Vasco)
Montserrat Hermo (U. del País Vasco)
Marisa Navarro (U. del País Vasco)
Xabier Sagarna (U. de Mondragón)

Comité Ejecutivo de PROLE-2009

Presidente Comité: Fernando Orejas (U. Politécnica de Cataluña)

Jesús Almendros (U. de Almería)
María Alpuente (U. Politécnica de Valencia)
Manuel Hermenegildo (U. Politécnica de Madrid)
Paqui Lucio (U. del País Vasco)
Juan José Moreno (U. Politécnica de Madrid)
Ginés Moreno (U. de Castilla-La Mancha)
Ricardo Peña (U. Complutense de Madrid)
Ernesto Pimentel (U. de Málaga)

Revisores adicionales de PROLE/TPF-2009

Javier Álvez, David Castro, Antonio Cansado, Victor Pablos Ceruelo, Javier Cámara, Henrique Ferreiro, Jose Gaintzarain, Miguel Gómez-Zamalloa, Raúl Gutierrez, Montserrat Hermo, Pablo López, Susana Nieva, Manuel Ojeda-Aciego, Miguel Palomino, Jaime Penabad, Ivan Pérez, Juan Rodríguez-Hortalá, Irakli Rogava, Daniel Romero, Fernando Rubio, Gwen Salaün, Damiano Zanardini.

Índice

Prólogo

Las Jornadas sobre PROgramación y LEnguajes (PROLE) se vienen consolidando como un marco propicio de reunión, debate y divulgación para los grupos españoles que investigan en temas relacionados con la programación y los lenguajes de programación. La investigación en este campo está en continuo desarrollo y comprende todo el estudio de conceptos, métodos, técnicas, fundamentos y aplicaciones relativos a la tarea de programar y a los lenguajes que se utilizan en ella. El evento, de carácter anual, pretende fomentar tanto el intercambio de experiencias y resultados, como la comunicación y cooperación entre los grupos de investigadores españoles que trabajan en el área de programación y lenguajes, manteniendo un año más la enriquecedora trayectoria de las ocho ediciones previas celebradas en Almagro (2001), El Escorial (2002), Alicante (2003), Málaga (2004), Granada (2005), Sitges (2006), Zaragoza (2007) y Gijón (2008).

En esta ocasión, la IX edición de las Jornadas (PROLE'09) va precedida por primera vez en su historia del I Taller sobre Programación Funcional (TPF'09). Ambos eventos se celebran entre el 8 y el 11 de septiembre de 2009, dentro de la XXVIII edición de los Cursos de Verano de San Sebastián. Como en ocasiones previas, la organización de esta conferencia se realiza en paralelo con las Jornadas de Ingeniería del Software y Bases de Datos (JISBD'09), compartiendo conferencias invitadas, actos sociales, publicidad, etc. Para información más detallada puede consultarse `http://www.mondragon.edu/prole2009/`. La organización conjunta de ambos eventos ha sido auspiciada por la Sociedad de Ingeniería del Software y Tecnologías de Desarrollo de Software (SISTEDES). Agradecemos desde aquí el soporte, la infraestructura y el apoyo prestado por todos los agentes arriba mencionados.

En el ámbito de PROLE'09 se han seleccionado este año un total de 31 trabajos, que cubren tanto aspectos teóricos como prácticos relativos a la especificación, diseño, implementación, análisis y verificación de programas y lenguajes de programación, además de herramientas tangibles y sistemas software que incrementan el carácter pragmático del área. Por su parte, el Taller de Programación Funcional TPF'09 que precede a PROLE'09, inicia su recorrido como una actividad independiente y complementaria a PROLE, con un comité de programa propio que ha seleccionado 5 trabajos recogidos también en estas actas, y que se centran en aspectos relacionados con lenguajes de programación con una fuerte componente funcional (en esta edición los trabajos se refieren a los lenguages Haskell, Lisp y Maude), incluyendo herramientas y experiencias docentes y de investigación en torno a este tipo de lenguajes. Como parte del Taller, se celebra también en esta ocasión una mesa redonda acerca de la inclusión de la programación funcional y lógica en los nuevos planes de Grado que empezarán a impartirse en 2009.

Este volumen recopila por tanto un total de 36 trabajos que fueron rigurosamente revisados cada uno de ellos por 3 miembros de ambos comités de programa y/o revisores adicionales, a los cuales es necesario agradecer su inestimable ayuda y reconocer su gran profesionalidad. También en consonancia

con este agradecimiento, es justo felicitar a los autores por la calidad de sus trabajos y su contribución a que esta edición sea la de mayor participación en la evolución histórica de PROLE, lo que garantiza la buena salud del evento.

Por otro lado, además de las tres conferencias invitadas que compartimos con la planificación de JISBD'09, el programa de PROLE'09 cuenta este año con una excelente conferencia específica (un resumen de la misma se incluye en este volumen) que, bajo el título de "Understanding program verification", será impartida por K. Rustan M. Leino, de Microsoft Research, USA, a quien agradecemos el haber aceptado tan amablemente nuestra invitación.

Finalmente, queremos agradecer la confianza que han depositado en nosotros, para conducir la presente edición de estas jornadas, a todos los miembros del comité ejecutivo de PROLE, y esperamos no haberles defraudado. En el desempeño de esta tarea, ha sido determinante la ayuda y experiencia prestada por Jesús Almendros, quien presidió la anterior edición de PROLE en Gijón, y a quien aprovechamos para dar mil gracias desde aquí.

Septiembre de 2009

Paqui Lucio
Ginés Moreno
Ricardo Peña

PATROCINADORES