Integrating ILOG CP technology into $\mathcal{TOY}$*

Nacho Castiñeiras[1] and Fernando Sáenz-Pérez[2]

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

Abstract. The constraint functional logic programming system $\mathcal{TOY}$ has been using the SICStus Prolog finite domain (FD) constraint solver. In this work, we show how to integrate the ILOG CP FD constraint solving technology into this system, with the aim of improving its application domain and performance. We describe our implementation emphasizing the synchronization between Herbrand computations in the $\mathcal{TOY}$ side and FD constraint solving in the ILOG CP side. Finally, performance results are reported and discussed.

1 Introduction

$\mathcal{TOY}$[1] is a system implemented in SICStus Prolog 3.12.8 [10]. Its operational semantics is based on a lazy narrowing calculus and includes several constraint domains allowing its cooperation. This system allows Herbrand equality and disequality constraints (managed by the constraint domain H), linear and non-linear arithmetic constraints over reals (R), finite domain constraints over integers (FD), and a communication domain (M) which makes possible the cooperation among H, R and FD. Whereas R as FD rely on the constraint solvers provided by SICStus Prolog, solving in H and M needs an explicitly management [3]. $\mathcal{TOY}$ offers a wide range of finite domain constraints comparable to many CLP(FD) systems, using a concrete constraint solving system as one of its components [5]. Here, we focus on this particular constraint domain for integrating a new constraint solving system based on ILOG CP technology.

The generic component architecture of the connection between $\mathcal{TOY}$ and its external FD constraint system is shown to the left of Fig. 1. $\mathcal{TOY}$ identifies each FD constraint during goal solving, and factorizes this (possibly) composed contraint into primitive ones, adding new produced variables if necessary [3]. Then, it posts these primitive constraints to $solve^{FD}$, which acts as an intermediary between $\mathcal{TOY}$ and the external FD system. $solve^{FD}$ sends the constraints to this system and collects its computed answers.

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407 and UCM-BSCH-GR58/08-910502

1.1 $\mathcal{TOY}$ with SICStus Prolog: $\mathcal{TOY}(FDs)$

$\mathcal{TOY}$ (referred to as $\mathcal{TOY}(FDs)$ from now on) has been using the FD constraint system provided in the library `clpfd` of SICStus Prolog, which is basically composed of a constraint store and solver. The component architecture of the connection between $\mathcal{TOY}$ and SICStus Prolog FD constraint system is shown in the middle of Fig. 1. Next, we show a basic example for illustrating the use of the system $\mathcal{TOY}(FDs)$ with finite domains constraints.

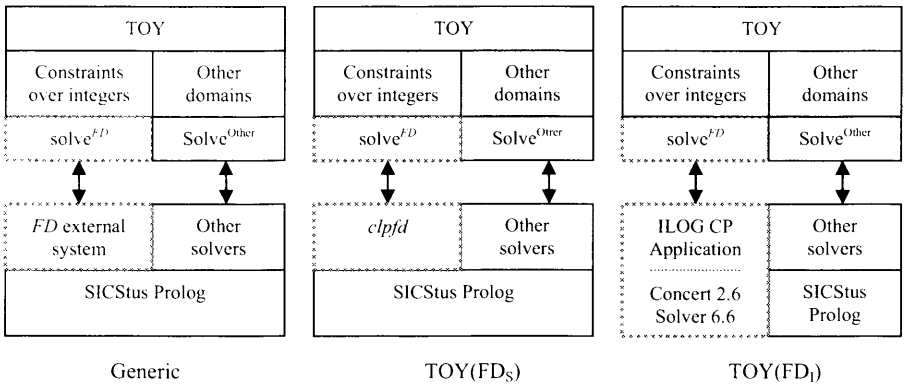


Fig. 1. Architectural Components

Example 1. Let's consider that `X` is an integer between `5` and `12`, `Y` is an integer between `2` and `17`, `X+Y=17` and `X-Y=5`. It is possible to solve this problem in $\mathcal{TOY}(FDs)$ as shown in the following interactive session:

```
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.
```

However, the use of the SICStus Prolog FD system has some disadvantages:

- Recent works [2] have proved that its performance can be enhanced, needed when dealing with complex problems.
- The constraint solver works as a black-box for predefined search processing. This precludes user-defined interactions for pruning the search tree.
- There are no debugging capabilities allowing, for instance, to derive the subset of infeasible constraints.

1.2 ILOG CP to improve $\mathcal{TOY}$

ILOG CP 1.4 [6] is an industrial technology market leader. Its nature is declarative and provides a C++ API to access its libraries. Its constraint solver works as a glass-box, allowing interactions during the solving process. It also includes debugging techniques helping the user to discover the unfeasible subset of the constraints set input. Its wide range of global constraints make possible to formulate different and complex properties. The use of different constraint solvers for a unique application domain is also allowed. Moreover, libraries for solving specific, efficient algorithms for complex scheduling problems are provided.

Any ILOG CP 1.4 application isolates objects responsible of modeling the user problem from objects responsible of solving any concrete model. Following this idea, the problem is modeled in a generic language, easing the task of expressing the constraints of the problem. Once the modeling phase is completed, the model can be solved by one or more different constraint solvers. The solver extracts all of the modeling objects contained into the model, creating a one-to-one object translation. This new objects belonging to the solver are semantically equivalent to the modeling objects, but their internal structure is targeted at the solver. It is possible to access each object created by the solver through the associated object contained into the model. The most paradigmatic tool representing this philosophy is ILOG OPL Studio [7]. ILOG CP 1.4 includes the library ILOG Concert 2.6 to provide the necessary interface for connecting models to solvers. Three libraries are provided for FD constraint solving:

- ILOG Solver 6.6, for generic FD problems solving.
- ILOG Scheduler 6.6, with specific algorithms for solving scheduling problems.
- ILOG Dispatcher 4.6, with specific algorithms for solving routing problems.

As a first approach, we will consider only ILOG Solver 6.6. For this case, any ILOG CP application needs the following set of ILOG Concert 2.6 and ILOG Solver 6.6 objects (see [6] for a detailed explanation):

- `IloEnv` ***env*** It manages the memory of any object of the application.
- `IloModel` ***model***`(env)` Is the main modeling object. Contains the set of objects responsible of formulating the FD problem, which are:
  - `IloIntVarArray` ***vars***`(env)` This vector is intended to make possible to reference all of the decision variables of the model from a unique object. Each variable must be created previously by `IloIntVar` ***v***`(env, int lowerBound, int upperBound)`.
  - `IloConstraint` ***c*** Each `IloConstraint` involves some `IloIntVar` of `vars`. It can be added directly to the model, without being created previously.
- `IloSolver` ***solver***`(env)` It is the main solving object. It contains an object `IloGoal` ***goal*** which specifies the concrete search procedure to be used. ***solver*** main methods are:
  - `solver.extract(model)` Extracts the information contained into `model`. For each `IloIntVar` and `IloConstraint` contained in `model` it creates an associated new `IlcIntVar` or `IlcConstraint` object.
  - `solver.solve(goal)` Solves the extracted model.

2 $\mathcal{TOY}$ with ILOG CP: $\mathcal{TOY}(FDi)$

In this section, we explain in detail how to integrate ILOG CP FD technology into the system $\mathcal{TOY}$ (referred to as $\mathcal{TOY}(FDi)$ from now on). $\mathcal{TOY}$ is implemented in SICStus Prolog while ILOG CP is a technology implemented and available in C++. So, first we study how to make a connection between $\mathcal{TOY}$ and ILOG CP by connecting SICStus Prolog and C++. Our approach is based on the integration of a C++ foreign resource into a SICStus Prolog application. Due to the different nature of both languages, we study the emerging difficulties to establish a communication between $\mathcal{TOY}$ and ILOG CP, as well as the decisions we have made to solve them. Also, an example of the behavior of the new system $\mathcal{TOY}(FDi)$ is shown.

2.1 Connecting SICStus Prolog with C++

It is possible to communicate a SICStus Prolog application with a C++ component. This communication is done by mapping a set of linking Prolog facts (contained in the Prolog application) with a set of C++ functions (defined in the C++ component). The C++ component needs to be a dynamic library with a specific internal file structure. SICStus Prolog also defines a set of possible conversions between Prolog arguments and C++ arguments. Each arguments of a linking Prolog fact must also indicate if it is either an input argument (sent to the C++ function) or an output argument (computed by the C++ function). There is a bidirectional conversion between a Prolog term and the C++ type `SP_term_ref`. By invoking `SP_term_ref` object methods, C++ functions can perform the following actions:

- Create and assign Prolog terms.
- Obtain the contents of a Prolog term.
- Compare and unify Prolog terms.

This context supports the necessary conditions to connect $\mathcal{TOY}$ and ILOG CP by making just a few changes in the component architecture of $\mathcal{TOY}$, whose new structure can be seen on the right hand side of Fig. 1.

- From the point of view of $\mathcal{TOY}$, it is necessary to put a new Prolog fact in any place of $solve^{FD}$ where a communication with ILOG CP is needed (posting a new constraint, declaring a new ILOG decision variable, etc.)
- On the other hand, we build a new ILOG CP application which integrates ILOG Concert 2.6 and ILOG Solver 6.6 libraries. This application contains instances of the basic modeling and solving objects explained in Section 1.2. It also includes the set of C++ functions linked to the existing Prolog facts in $solve^{FD}$.

Each time $solve^{FD}$ calls any interfaced predicate, first, it turns all Prolog arguments into C++ arguments. Next, it transfers the program control to the C++ function, which uses and/or computes them within its body. Once the C++ function has finished, the execution control comes back to $solve^{FD}$, which continues with the evaluation of the next call.

Creating a SICStus Prolog C++ Foreign Resource

SICStus Prolog needs two files for creating a dynamic library as, for instance `interface.dll`, which could be used within a SICStus Prolog application:

- `interface.pl` Declares the mapping of each Prolog predicate to each C++ function. It groups all of these functions in a unique resource. For example:
  ```
  foreign(f1,p1(+integer)).
  foreign(f2,p2(+term,-term)).
  foreign_resource(interface,[f1,f2]).
  ```
- `interface.cpp` Includes the C++ functions mapped to Prolog facts. It adds as many auxiliary functions and libraries as needed. For example:
  ```
  void f1(long l){...}
  void f2(SP_term_ref t1, SP_term_ref t2){...}
  ```

SICStus Prolog supplies a tool, `splfr` [9], to create a dynamic library (say `interface.dll`), taking as input `interface.pl` and `interface.cpp`. The macro `splfr` is used as a shortcut to the execution of some compiling and linking commands offered by Microsoft Visual C++ [8]. First of all, taking `interface.pl` as input, it creates two new files, `interface_glue.c` and `interface_glue.h`, which provides the necessary glue code for the SICStus application.

2.2 Communication between $\mathcal{TOY}$ and ILOG CP

In this section we explain in detail how to implement $\mathcal{TOY}(FDi)$ in such a way it accepts any $\mathcal{TOY}(FDs)$ input goal, including all FD constraints managed by the existing $solve^{FD}$ in $\mathcal{TOY}(FDs)$. Also, $\mathcal{TOY}(FDi)$ uses the same goal solution structure as $\mathcal{TOY}(FDs)$ does. To achieve that behavior is necessary to solve the following difficulties:

- As $\mathcal{TOY}$ is a system implemented in SICStus Prolog, in $\mathcal{TOY}(FDs)$ the communication between $\mathcal{TOY}$ and its FD technology is quite natural. However, as ILOG CP is implemented in C++, some glue code is needed to fix the impedance mismatch problem.
- ILOG CP and SICStus Prolog differ on their notion of solution of a FD problem.

There have been four difficult tasks to achieve in the new system $\mathcal{TOY}(FDi)$. We explain each of them in the next subsections. When we make reference to any ILOG CP application object, we use the notation of Section 1.2. To this end, we use `model` if we refer to the ILOG Concert 2.6 model object, we use `solver` if we refer to the ILOG Solver 6.6 generic FD solver, and we use `vars` if we refer to the decision variables contained in `model`.

Managing Decision Variables

The set of FD constraints of a $\mathcal{TOY}$ goal involves a set of logic variables that we denote as 'FD logic variables'. To model the FD constraint set with ILOG CP, some points must be taken into account:

- We need to create as many `IloIntVar` decision variables as FD logic variables take part into the FD constraint set. These variables must be added to `model` and `vars` (the former to model the FD problem properly and the latter to make possible to refer to each variable of the model from a unique object).
- We must find a bijective relation that associates each FD logic variable of the $\mathcal{TOY}$ goal with each decision variable existing in the ILOG CP vector `vars`.
- We model each FD constraint in ILOG CP over the set of decision variables of the vector `vars` associated to the set of FD logic variables involved in that FD constraint.

Whatever way of communication between $\mathcal{TOY}$ and ILOG CP, for each FD logic variable we have three variables:

- The FD logic variable contained in $\mathcal{TOY}$.
- The decision variable modeled as an `IloIntVar` object in `model`.
- The specific `IlcIntVar` object created by `solver` from its associated `IloIntVar` object contained in `model`.

A first attempt for mapping a FD logic variable to a decision variable of `vars` is tried. It intends to manage `vars` and a `SP_term_ref` vector, making them evolve simultaneously. The elements of the `SP_term_ref` vector are in fact the `SP_term_ref` conversion of the FD logic variables. Each time $solve^{FD}$ sends a new FD constraint to ILOG CP, the associated C++ function will first look for its FD logic variables into the `SP_term_ref` vector. If it can not find any variable, we can assure that the C++ function is dealing with a new FD logic variable not treated before. So, the C++ function adds this new FD logic variable to the `SP_term_ref` vector last position, say `i`. Immediately, a new `IloIntVar` decision variable is created and added to `model` and `vars[i]`. When each FD logic variable of the FD constraint sent by $solve^{FD}$ is contained at an index of the `SP_term_ref` vector, the FD constraint is modeled over the decision variables of `vars` associated to these indexes.

However, this first attempt fails. This is due to the rules which govern the scope of a `SP_term_ref`. When a C++ function containing `SP_term_ref`s (as arguments or dynamically created within it) finishes its execution, all these `SP_term_ref`s become invalid. Let's see the next example, where an interface between the Prolog predicates `p1`, `p2` and `p3` and the C++ functions `f1`, `f2` and `f3`, resp, is defined. Functions `f1` and `f2` receive a Prolog term as an argument, while `f3` receives two Prolog terms.

- Let's call `p3` with to occurrences of the logic variable `X`, as `p3(X,X)`. If we make `SP_compare(t1,t2)` within `f3(SP_term_ref t1, SP_term_ref t2)` the result says that both `SP_term_ref`s are in fact the same Prolog term.
- But, let's do the call `p1(X)`. We store `t1` of `f1(SP_term_ref t1)` into a global vector <SP_term_ref>. When `f1` finishes, the program control comes back to Prolog. Now, we call `p2` with the logic variable `X` again, `p2(X)`. If

we make `SP_compare(t1,t2)` within `f2(SP_term_ref t2)` between `t2` and the `SP_term_ref` stored in the vector during `f1`, the result says that both `SP_term_ref`s are different. There is no doubt that both are in fact the same Prolog term. The problem is that, when `f1` finish, the `SP_term_ref` stored in the vector becomes invalid.

The second and successful attempt relies on the management of the bijective relation, which is done into the Prolog application by the use of a list of FD logic variables (referred to as `L` from now on). We want `L` to be used in each $solve^{FD}$ predicate. On one hand SICStus Prolog does not allow global variables. On the other hand, there is a logic variable `Cin` [4], which represents a mixed constraints store and is common to each $solve^{FD}$ predicate. Our plan is to store any data structure demanded by the communication between $\mathcal{TOY}$ and ILOG CP, specifically `L`, into `Cin`. Each time a $solve^{FD}$ predicate manages a new FD constraint, we can check whether a FD logic variable belongs to `L` or not by accessing to it within `Cin`. Any new FD logic variable is automatically added to the end of `L`, say at position `i`. Here, a new call to the C++ function which creates a new `IloIntVar` is done. This function adds this decision variable to `model` and `vars[i]`. Once all FD logic variables of the FD constraint belongs to `L`, $solve^{FD}$ determines their indexes, and put them as arguments to the C++ function, which models the FD constraint by adding to `model` a new `IloConstraint` over the associated positions of `vars`.

Synchronizing ILOG CP with $\mathcal{TOY}$

TOY can also bind its FD logic variables through an equality constraint in the Herbrand solver. For example, in the goal `TOY(FDi)> X #>= 0, X == 3` the variable `X` is bound to the value `3`. This is done by the Prolog terms unification which results from the Herbrand equality constraint `X == 3`. This unification is visible at any occurrence of that FD logic variable, particularly the one in `L`. This causes an inconsistency between the contents of `L` and `vars`. To repair this lack of synchronization we must send an equality constraint to ILOG CP, making the mapped decision variable in `vars` equals to the bound value.

A first attempt tries to synchronize by an event-driven approach. To capture events, SICStus Prolog provides the module of attributed variables. This module assigns attributes to a set of logic variables. Each time an attributed logic variable is bound, the predicate `verify_attributes(+Var, +Value, +Goals)` is triggered. We use the attribute `fd` for each FD logic variable. Thus, each time the Herbrand solver binds a FD logic variable, `verify_attributes(+Var, +Value, +Goals)` will automatically call the C++ function which synchronizes the associated decision variable of `vars`.

However, this first attempt fails. For this synchronization we need to know which index does the associated decision variable have in `vars`. We can only get this index by looking for the FD logic variable in `L`. But, unfortunately, the arguments of `verify_attributes(+Var, +Value, +Goals)` are fixed. SICStus Prolog does not allow global variables, so there is no way to get access to `L`.

A second attempt consists of making the Herbrand solver responsible of calling the C++ synchronization function. But this idea must be rejected, because there is a basic principle of independency between the different solvers of the system $\mathcal{TOY}$. Any solution to this problem must respect the idea of solving the synchronization within $solve^{FD}$.

The third (and successful) attempt modifies the internal structure of `L`. Now it becomes a list of pairs. The first element of each pair contains the FD logic variable, and the second one contains a flag which determines if the bound FD logic variable has been synchronized with `vars`. Thus, while the FD logic variable is not bound, the value of the flag remains at `0`. When the FD logic variable becomes bound, the value of the flag indicates whether the variable of `vars` is synchronized or not.

Each time $solve^{FD}$ sends a new FD constraint to ILOG CP, it must previously:

- Look for any pair in `L` (say at position `i`) whose pattern is `[`*`value`*`,0]`
- Add to `model` the new `IloConstraint vars[i]==`*`value`*.
- Change the pair at position `i` of `L` by `[`*`value`*`,1]`

Once there is no pairs with the pattern `[`*`value`*`,0]` in the list, $solve^{FD}$ is able to send the new FD constraint. If there are no more FD constraints, the pairs `[`*`value`*`,0]` will be synchronized at the end of the $\mathcal{TOY}$ goal. This synchronization attempt is clearly inefficient, making it a task to be improved in new releases of $\mathcal{TOY}(FDi)$. Let's see the next goal:

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

The first FD logic variable of the goal is `X`, which occurs at the first position of `L` and `vars`. The synchronization of `X == 1` as `vars[0] == 1` makes the FD problem infeasible. So, the $\mathcal{TOY}$ goal will fail after `X == 1`, and there is no need of computing the rest of the goal expressions. However, the first equality `vars[0] == 1` is not computed until the next FD constraint is posted.
As `X == 1, X1 == 1, X2 == 1, ... , X1000 == 1` are computed by Herbrand solver there are no more FD constraints in the goal, so the synchronization will not occur until the end of the goal. The goal will useless compute a thousand of successful expressions. After that, it synchronizes `vars[0] == 1` and fails.

Synchronizing $\mathcal{TOY}$ with ILOG CP

ILOG CP can bind variables in `vars` via the set of C++ functions concerning the management of FD constraints. This produces a lack of synchronization between the vector `vars` and `L`. To achieve the synchronization, whenever any of this C++ functions binds to *value* `vars[i]`, the pair contained at position `i` of `L` must be automatically unified with `[`*`value`*`,1]`.

To this end, $solve^{FD}$ sends `L` to a C++ function as an input argument, and puts an output argument to obtain the new state of `L` computed within the C++ function. A new global variable of type `vector<int,int>` must be created

in ILOG CP. This vector of pairs is cleared at the beginning of each C++ function. Each pair of the vector contains:

- The index `i` in `vars` of the decision variable.
- The *value* that `solver` has obtained for this variable.

A C++ function manages any new constraint by adding it to `model`, and propagates its new FD constraint set. Next, the C++ function accesses to the contents of the vector`<int,int>`, to see whether there are any `IloIntVar` that has been bound. Using the content of vector`<int,int>` and `L`, the C++ function builds the new state of `L` by unifying as many FD logic variables as vector`<int,int>` demands.

The only remaining task to be explained is how to add each pair to the global vector`<int,int>`. To do so, we use demons to capture bind events. Thus, a new demon object `IlcDemon RealizeVarBound` is created. It concerns on how to insert each new pair into the vector`<int,int>`. This demon is triggered by the propagation of a constraint `IlcCheckWhenBound`. Each `IlcCheckWhenBound` constraint involves one `IloIntVar`. This constraint propagates when its `IloIntVar` becomes bound. ILOG CP associates a demon to a method of a constraint class. When the demon is triggered, the method of this constraint class is automatically executed. We associate `RealizeVarBound` to the method `varDemon` of the `IlcCheckWhenBound` constraint class. This method checks the index in `vars` of the bounded `IloIntVar` and its value, adding both of them as a new pair of integers to the global vector`<int,int>`. We summarize how our ILOG CP application adds the pairs to the vector`<int,int>` in the next three steps:

- For each new decision variable `IloIntVar` added to `vars` and `model`, we impose the constraint `IlcCheckWhenBound`.
- When this `IloIntVar` becomes bound, `IlcCheckWhenBound` propagates, triggering the demon `RealizeVarBound`.
- `RealizeVarBound` executes the `IlcCheckWhenBound` method `varDemon`, which adds the pair `<index of the variable, value of the variable>` to vector`<int,int>`.

Solutions in ILOG CP

Any $\mathcal{TOY}(FDs)$ solution is expressed in general with constraints (equality, disequality, FD constraints –including ranges–). Of course, $\mathcal{TOY}(FDs)$ accepts to label FD variables by calling the FD labeling enumeration procedure.

In $\mathcal{TOY}(FDi)$, to show the remaining values of the FD logic variables we access to each `IlcIntVar` of `solver` by its associated `IloIntVar` contained in `model`. There are some methods to check the remaining values of these variables. However, ILOG Solver does not grant access to simplified constraints (i.e., solved forms). The ILOG philosophy of a solution is to select a value for each decision variable while satisfying the constraint set. Of course, you can use no search procedure, obtaining the same structure as in an interval solution, but again without accessing the simplified constraints. As in our context we have to show them, we store within `Cin` a list with the FD constraints (referred to as `C` from now on) appearing in the $\mathcal{TOY}$ goal.

2.3 A $\mathcal{TOY}(FDi)$ Example

In this section we detail how goal solving works with the new system $\mathcal{TOY}(FDi)$ over the example 1:

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

We specify how the data structures of $solve^{FD}$ and ILOG CP evolve with each expression evaluation. On one hand we look at the state of `L` and `C` within `Cin`. On the other hand we look at the state of `vars`, `model`, `solver` by pointing out any `IloIntVar`, `IloConstraint`, `IlcIntVar`, `IlcConstraint` object accessed through them. For each goal expression any new element added to each data structure is remarked in boldface. Figure 2 tries to make it clearer:

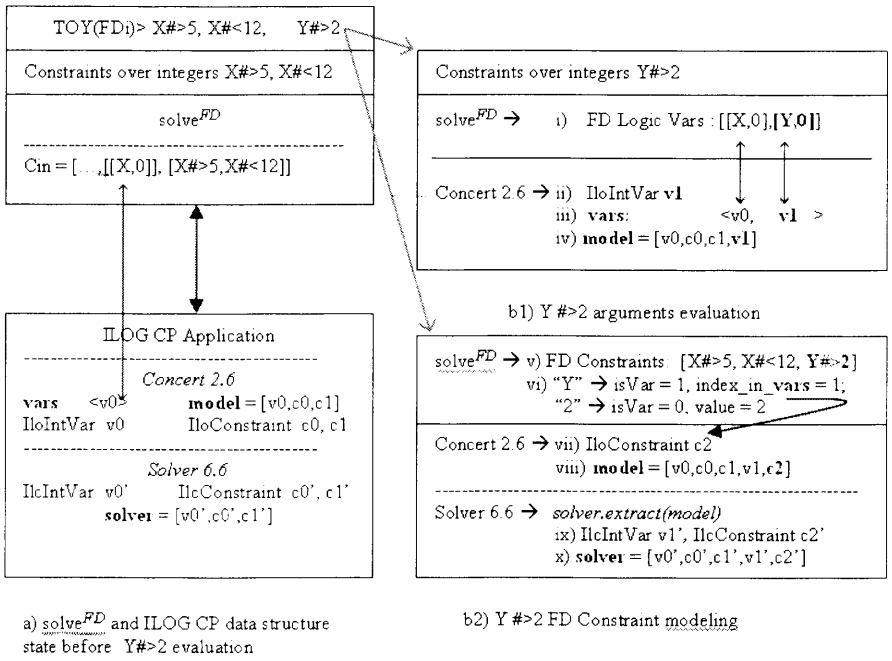


Fig. 2. $TOY(FDi)$ data structures evolution over FD Constraint expression evaluation

- Figure 2a) represents the internal state of $solve^{FD}$ and ILOG CP data structures at the end of `Toy(FDi)> X #>= 5, X #<= 12` evaluation.
- Figure 2b1) and 2b2) describes which actions must be done for the correct management of the new FD Constraint `Y #>= 2`.

Before evaluating any goal expression, in the *solve*^{FD} side L=[] and C=[]. In the ILOG CP side model=[], vars=<> and solver=[]. There is also no IloIntVar, IloConstraint, IlcIntVar, IlcConstraint objects.

- Execution of X #>= 5
  The new *FD* constraint is added to C=[**X#>=5**]. The new *FD* logic var is added to L=[[**X,0**]]. A new IloIntVar **v0** is created and added to vars=<**v0**> and model=[**v0**]. A new IloConstraint **c0** is created, involving vars[0] and the value 5. This IloConstraint **c0** is added to model=[v0,**c0**]. solver extracts the new state of model and creates a new IlcIntVar **v0'** and a new IlcConstraint **c0'**. solver=[**v0'**,**c0'**]. Its constraint propagation technique prunes the domain of **v0'**=5..sup. The state of the solver remains 'Feasible'. $\mathcal{TOY}$ continues evaluating next goal expression.
- Execution of X #<= 12
  C=[X#>=5,**X#<=12**]. L=[[X,0]]. vars=<v0>. A new IloConstraint **c1** is created involving vars[0] and 12. model=[v0,c0,**c1**]. solver extracts model creating IlcConstraint **c1'**. solver=[v0',c0',**c1'**]. Constraint propagation prunes **v0'**=5..12. solver state='Feasible'.
- Execution of Y #>= 2
  By managing Y#>=2 arguments, *solve*^{FD} adds L=[[X,0],[**Y,0**]]. By adding a new *FD* Logic Var to L, a new IloIntVar **v1** is created and added to vars=<v0,**v1**> and model[v0,c0,c1,**v1**]. There is a correspondence between **Y** and **v1** because both are at the same position of L and vars respectively. *solve*^{FD} adds Y#>=2 to C=[X#>=5,X#<=12,**Y#>=2**]. The relevant information to modeling the *FD* constraint into ILOG CP is the tuple <1,1,0,2> which says if the arguments are variables or not and its index/value respectively. Then a new IloConstraint **c2** is created involving vars[1] and the value 2. This IloConstraint c2 is added to model=[v0,c0,c1,v2,**c2**]. solver extracts the new state of model creating a new IlcIntVar **v1'** and IlcConstraint **c2'**. solver=[v0',c0',c1',**v1'**, **c2'**]. Constraint propagation prunes **v1'**=2..sup. solver state='Feasible'. After constraint propagation, the program control comes back to *solve*^{FD}. It finishes the management of the *FD* constraint by storing the new state of L=[[X,0],[Y,0]] and C=[X#>=5,X#<=12,Y#>=2] into Cin.
- Execution of Y #<= 17
  C=[X#>=5,X#<=12,Y#>=2,**Y#<=17**]. L=[[X,0],[Y,0]]. vars=<v0,v1>. A new IloConstraint **c3** is created involving vars[1] and 12.
  model=[v0,c0,c1,v2,c2,**c3**]. solver extracts model creating IlcConstraint **c3'**. solver=[v0',c0',c1',v1',c2',**c3'**]. Constraint propagation prunes **v1'**=2..17. solver state='Feasible'.
- Execution of X#+Y==17
  This expression includes a compound constraint. This constraint must be decomposed into primitive constraints. In this case: X#+Y==_Z, _Z==17
  - Execution of X#+Y==_Z
    C=[X#>=5,X#<=12,Y#>=2,Y#<=17,**X#+Y==_Z**].
    L=[[X,0],[Y,0],[**_Z,0**]]. A new IloIntVar **v0** is created and added

to vars=<v0,v1,**v2**>. A new IloConstraint **c4** is created involving vars[0] and vars[1]. model=[v0,c0,c1,v2,c2,c3,**c4**]. solver extracts model creating IlcIntVar **v2'** and IlcConstraint **c4'**.
solver=[v0',c0',c1',v1',c2',c3',**v2'**,**c4'**]. Constraint propagation prunes **v2'**=7..29. solver state='Feasible'.

- Execution of _Z==17
  $\mathcal{TOY}$ sends _Z == 17 to the Herbrand solver. This will bind the variable _Z to 17, L=[[X,0],[Y,0],[**17,0**]],
  C=[X#>=5,X#<=12,Y#>=2,Y#<=17,**X#+Y==17**]. However, this value will not be automatically synchronized with ILOG CP. The synchronization will happen before either a new FD constraint is sent or at the end of the $\mathcal{TOY}$ goal.

– Execution of X#-Y==5
This expression is decomposed again into X#-Y==_T, _T==5

- Execution of X#-Y==_T
  As we have pointed out, before the new FD constraint is sent to ILOG CP, any pattern [*value*,0] contained in L at position i will be synchronized with model by adding the new IloConstraint vars[i]==*value*.
  C=[X#>=5,X#<=12,Y#>=2,Y#<=17,X#+Y==17].
  L=[[X,0],[Y,0],[17,0]]. A new IloConstraint **c5** is created involving vars[2] and 17.
  model=[v0,c0,c1,v2,c2,c3,c4,**c5**]. solver extracts model creating IlcConstraint **c5'**. solver=[v0',c0',c1',v1',c2',c3',v2',c4',**c5'**].
  Constraint propagation bounds vars[2] to 17. L=[[X,0],[Y,0],[**17,1**]].
  solver state='Feasible'.

  As there is no more patterns [*value*,0] in L, $solve^{FD}$ is now able to manage the constraint X#-Y==_T. So the new FD constraint is added to
  C=[X#>=5,X#<=12,Y#>=2,Y#<=17,X#+Y==17,**X#-Y==_T**].
  L=[[X,0],[Y,0],[17,1],[**_T,0**]]. A new IloIntVar **v0** is created and added to vars=<v0,v1,v2,**v3**>. model=[v0,c0,c1,v2,c2,c3,c4,c5,**v3**]. A new IloConstraint **c6** is created involving vars[0] and vars[1].
  model=[v0,c0,c1,v2,c2,c3,c4,c5,v3,**c6**].solver extracts model creating IlcIntVar **v3'** and IlcConstraint **c6'**.
  solver=[v0',c0',c1',v1',c2',c3',v2',c4',c5',**v3'**,**c6'**].
  Constraint propagation prunes **v0'**=6..12, **v1'**=5..11', **v3'**=1..7.
  solver state='Feasible'.
- Execution of _T==5
  $\mathcal{TOY}$ sends _T == 5 to the Herbrand solver. This will bind the variable _T to 5, making L=[[X,0],[Y,0],[17,1],[**5,0**]],
  C=[X#>=5,X#<=12,Y#>=2,Y#<=17,X#+Y==17,**X#-Y==5**].
  Again, the synchronization will happen before either a new FD constraint is sent or at the end of the $\mathcal{TOY}$ goal.

– The $\mathcal{TOY}$ goal is almost finished. To completely finish the goal computation we synchronize the pairs L with the pattern [*value*,0].
C=[X#>=5,X#<=12,Y#>=2,Y#<=17,X#+Y==17].

L=[[X,0],[Y,0],[17,1],[5,0]]. A new `IloConstraint` **c7** is created involving `vars[3]` and 5. `model`=[v0,c0,c1,v2,c2,c3,c4,c5,v3,c6,**c7**]. `solver` extracts `model` creating `IlcConstraint` **c7'**.
`solver`=[v0',c0',c1',v1',c2',c3',v2',c4',c5',v3',c6',**c7'**]. Constraint propagation bounds `vars[3]` to 5, **v0'**=10..12, **v1'**=5..7'. L=[[X,0],[Y,0],[17,1],[**5,1**]].
`solver` state='Feasible'.
After this synchronization, the $\mathcal{TOY}$ goal is completely finished. It shows as the computed answer the set of non-ground FD constraints of `C` as well as the (unbound) variables of `L`. For each of these variables, $\mathcal{TOY}$ shows its domain. These values are obtained from the `IlcIntVar` contained in `solver` through the associated `IloIntVar` contained in `model`. Each decision variable of `model` is accessed through its position of `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 Measuring Performance

In this section we use two test parametric, scalable (on `n`) benchmark programs which model systems of linear equations $A * X = b$. Each system has `n` independent equations with `n` variables `[X1,...,Xn]` whose domains are {`1..n`}. Each system has a unique integer solution. The matrix A takes the value i on its diagonal coefficients $A_{i,i}$ and the value 1 for the rest of them.

Both benchmark programs have been run in a machine with an Intel Dual Core 2.4Ghz processor and 4GB RAM memory. The SO used is Windows XP SP3. The SICStus Prolog version used is 3.12.8. The ILOG CP application used is ILOG CP 1.4, with ILOG Concert 2.6 and ILOG Solver 6.6 libraries. Microsoft Visual C++ 6.0. tools are used for compiling and linking the application.

We show performance results (expressed in miliseconds) for the following systems: both $\mathcal{TOY}(FDs)$ and $\mathcal{TOY}(FDi)$ just described, and also for a C++ program directly modelling the problems using the ILOG CP libraries (denoted by FDs, FDi and ILOG in the tables, respectively). The latter will help us in analysing the overhead due to $\mathcal{TOY}$ implementation of lazy narrowing.

For each benchmark, we show three instances of `n`: 4, 12 and 15 variables. In each case, we present results for two labeling strategies: a static search procedure which selects the variables in the textual order they occur in the program, and the dynamic search procedure 'first fail' (denoted by ff), which selects first the variable with minimum domain size. For a given variable, both of them selects first the minimum value in its domain.

Also, we show the speedups of $\mathcal{TOY}(FDi)$ with respect to $\mathcal{TOY}(FDs)$ and ILOG CP respectively. Specifically, we denote as:

- (a) to the speedup of $\mathcal{TOY}(FDi)$ with respect to $\mathcal{TOY}(FDs)$ using the static search procedure to solve the problem.
- (b) to the speedup of $\mathcal{TOY}(FDi)$ with respect to $\mathcal{TOY}(FDs)$ using the 'first fail' search procedure.
- (c) to the speedup of $\mathcal{TOY}(FDi)$ with respect to ILOG CP C++ program using the static search procedure.
- (d) to the speedup of $\mathcal{TOY}(FDi)$ with respect to ILOG CP C++ program using the 'first fail' search procedure.

The benchmarks programs are:

- The solution `[X1,...,Xn]` holds: $\forall i \in \{1 \ldots n\}$ `Xi` $= i$. Performance measurement gives the following results:

n	FDs	FDs^{ff}	FDi	FDi^{ff}	$ILOG$	$ILOG^{ff}$	(a)	(b)	(c)	(d)
4	0	15	0	0	15	15	1.0	-	0	0
12	31	1.750	156	516	15	281	5.0	0.29	10.4	1.83
15	297	299,312	423	67,376	63	20,578	1.42	0.22	6.7	3.27

  For this first benchmark, $\mathcal{TOY}(FDi)$ takes more time than $\mathcal{TOY}(FDs)$ for solving with the static search procedure, but less time for the dynamic search procedure. The solving time difference between them grows as we increase the number of variables for the benchmarks. Looking at how the domains of the variables evolve after the initial constraint propagation, we can conclude that the structure of the solution for this first benchmark fits quite well into the static search procedure, while it is dramatically harmful to the dynamic search procedure. This help us to realize that, for problems where the needed exploration to obtain the solution is really small, then $\mathcal{TOY}(FDi)$ is slower than $\mathcal{TOY}(FDs)$. This is because of the time involved in the communication between the Prolog implementation of $\mathcal{TOY}(FDi)$ and ILOG CP. However, as the nodes needed to be explored increase slightly, this waste of time is balanced, making $\mathcal{TOY}(FDi)$ more efficient than $\mathcal{TOY}(FDs)$.
- The solution `[X1,_,Xn]` holds: $\forall i \in \{1..n\}$ `Xi` $= n - (i - 1)$. Performance measurement gives the following results:

n	FDs	FDs^{ff}	FDi	FDi^{ff}	$ILOG$	$ILOG^{ff}$	(a)	(b)	(c)	(d)
4	16	16	16	31	31	15	1.0	1.93	0.51	2.06
12	531	250	437	126	109	63	0.83	0.50	4	2
15	15,563	21,968	13,937	3,406	843	1,765	0.90	0.16	16.53	1.93

The above conclusions are clearly confirmed in this second benchmark, where $\mathcal{TOY}(FDi)$ is faster than $\mathcal{TOY}(FDs)$ for both search procedures. In this case, the structure of the solution is dramatically harmful for the static strategy, while it behaves better for the dynamic strategy. In the former, $\mathcal{TOY}(FDi)$ takes

slightly less solving time than $\mathcal{TOY}(FDs)$. In any case, these measurements point out that our first approach to integrate the ILOG CP technology into $\mathcal{TOY}(FDi)$ is encouraging, but also that the management of the additional data structures used for the interface should be optimized.

4 Conclusions and Future Work

In this work, we have studied how to integrate the FD ILOG CP technology into the system $\mathcal{TOY}$. We have shown that this technology offers some advantages over the existing system $\mathcal{TOY}$ based on the FD technology of SICStus Prolog. We have described in detail our implementation, showing that the application architecture of $\mathcal{TOY}$ and ILOG CP are hard to integrate in terms of a correct communication between them. We have shown by means of two scalable benchmarks that the new system $\mathcal{TOY}(FDi)$ is faster than $\mathcal{TOY}(FDs)$ as the benchmark increases its size. However, we have concluded that there is a performance penalization due to the management of the data structures that make possible the connection of $\mathcal{TOY}$ with its new FD component. Therefore, optimizing this management will be the target of our immediate future work. In addition, backtracking management will be covered in a next work, together with an extended set of benchmarks. Another subject of interest is to test other constraint libraries, as Gecode [11].

References

1. P. Arenas, S. Estévez, A. Fernández, A. Gil, F. López-Fraguas, M. Rodríguez-Artalejo, and F. Sáenz-Pérez. $\mathcal{TOY}$. a multiparadigm declarative language. version 2.3.1., 2007. R. Caballero and J. Sánchez (Eds.), Available at `http://toy.sourceforge.net`.
2. R. G. del Campo and F. Sáenz-Pérez. Programmed Search in a Timetabling Problem over Finite Domains. *Electr. Notes Theor. Comput. Sci.*, 177:253–267, 2007.
3. S. Estévez-Martín, A. Fernández, M. Hortalá-González, F. Sáenz-Pérez, M. Rodríguez-Artalejo, and R. del Vado-Vírseda. On the Cooperation of the Constraint Domains H, R and FD in $CFLP$. *Theory and Practice of Logic Programming*, 2009. Accepted for publication.
4. S. Estévez-Martín, A. J. Fernández, and F. Sáenz-Pérez. About implementing a constraint functional logic programming system with solver cooperation. In *proc. of CICLOPS'07*, pages 57–71, 2007.
5. A. J. Fernández, T. Hortalá-González, F. Sáenz-Pérez, and R. del Vado-Vírseda. Constraint Functional Logic Programming over Finite Domains. *Theory Pract. Log. Program.*, 7(5):537–582, 2007.
6. ILOG. ILOG Solver 6.6, Reference Manual, 2008.
7. ILOG. ILOG OPL Studio 6.1, Reference Manual, 2009.
8. Microsoft, 2005. `http://msdn.microsoft.com/en-us/visualc/default.aspx`.
9. SICStus Prolog. Using SICStus Prolog with newer Microsoft C compilers.
10. SICStus Prolog, 2007. `http://www.sics.se/isl/sicstus`.
11. The Gecode team. Generic constraint development environment. Available from http://www.gecode.org, 2006.

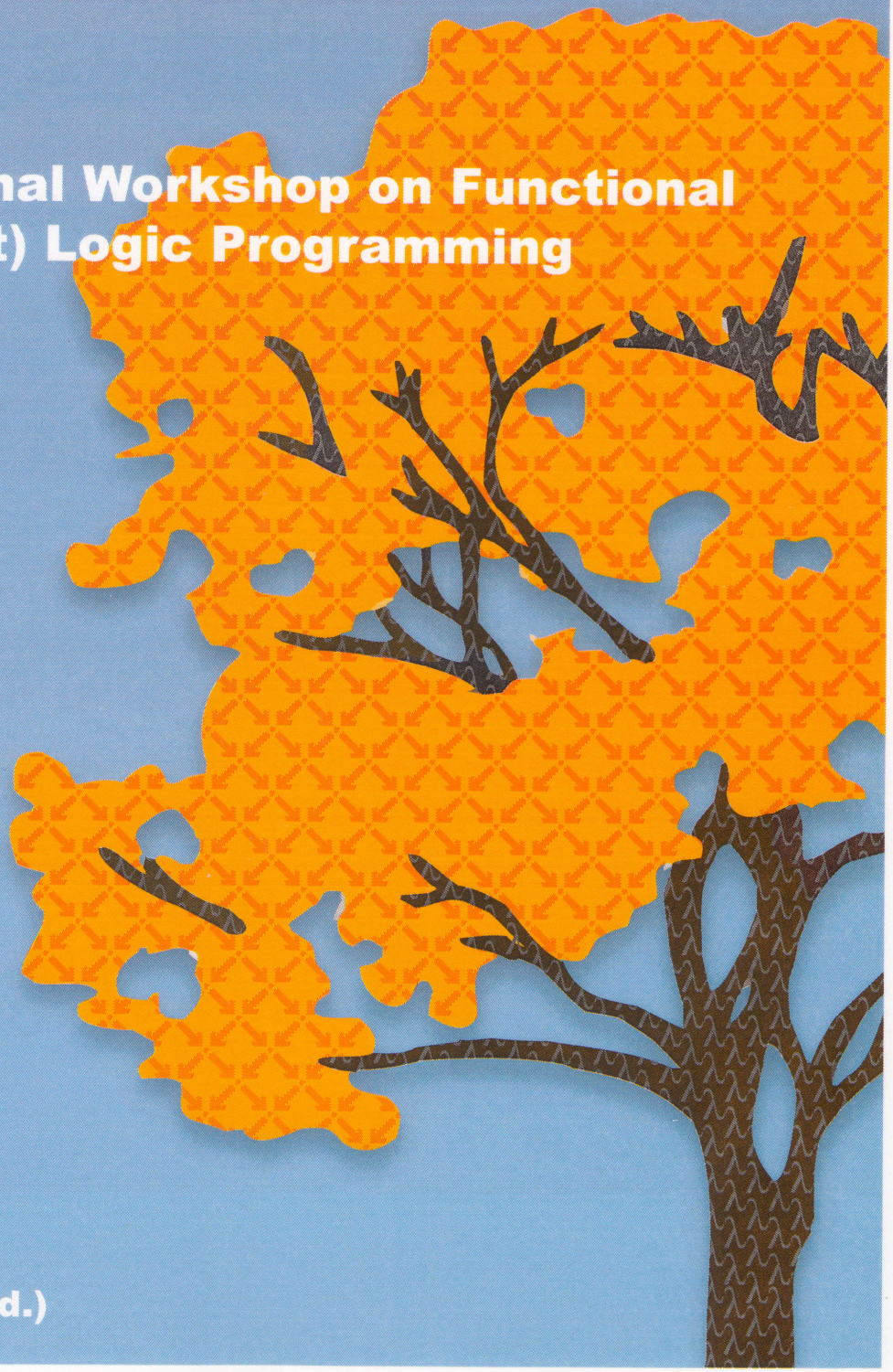
Federated Conference on Rewriting, Deduction & Programming

RDP 2009

Universidade de Brasília, Brasília D.F., Brazil
June 28th – July 03rd

Pre-proceedings

18th International Workshop on Functional and (Constraint) Logic Programming WFLP 2009



Santiago Escobar (Ed.)

Sponsorship
CNPq CAPES fapdf

Scientific sponsorship
SBC Sociedade Brasileira de Computação

Hosted by
UnB
Grupo de Teoria da Computação
Departamento de Matemática e Ciência da Computação

Santiago Escobar (Ed.)

Functional and (Constraint) Logic Programming

18th International Workshop, WFLP'09

part of the Federated Conference on Rewriting, Deduction, and Programming (RDP'09)

Brasília, Brazil, June 28, 2009.

Informal Proceedings

Preface

This report contains the informal workshop proceedings of the *18th International Workshop on Functional and (Constraint) Logic Programming* (WFLP'09), held at Brasília, Brazil, during June 28. 2009. WFLP'09 is part of the Federated Conference on Rewriting, Deduction, and Programming (RDP'09). Previous meetings are: WFLP 2008 (Siena, Italy), WFLP 2007 (Paris, France), WFLP 2006 (Madrid, Spain), WCFLP 2005 (Tallinn, Estonia), WFLP 2004 (Aachen, Germany), WFLP 2003 (Valencia, Spain), WFLP 2002 (Grado, Italy), WFLP 2001 (Kiel, Germany), WFLP 2000 (Benicassim, Spain), WFLP'99 (Grenoble, France), WFLP'98 (Bad Honnef, Germany), WFLP'97 (Schwarzenberg, Germany), WFLP'96 (Marburg, Germany), WFLP'95 (Schwarzenberg, Germany), WFLP'94 (Schwarzenberg, Germany), WFLP'93 (Rattenberg, Germany), and WFLP'92 (Karlsruhe, Germany).

The aim of the WFLP workshop is to bring together researchers interested in functional programming, (constraint) logic programming, as well as the integration of the two paradigms. It promotes the cross-fertilizing exchange of ideas and experiences among researchers and students from the different communities interested in the foundations, applications and combinations of high-level, declarative programming languages and related areas.

The Program Committee of WFLP'09 collected three reviews for each paper and held an electronic discussion during May 2009. The Program Committee selected 12 regular papers for presentation at the workshop. In addition to the selected papers, the scientific program includes two invited lectures by Claude Kirchner from the Centre de Recherche INRIA Bordeaux - Sud-Ouest, France and Roberto Ierusalimschy from the Departamento de Informática, PUC-Rio, Brazil. I would like to thank them for having accepted our invitation.

I would also like to thank all the members of the Program Committee and all the referees for their careful work in the review and selection process. Many thanks to all authors who submitted papers and to all conference participants. We gratefully acknowledge the *Departamento de Sistemas Informáticos y Computación* of the *Universidad Politécnica de Valencia*, who has supported this event. Finally, we express our gratitude to all members of the local organization of the Federated Conference on Rewriting, Deduction, and Programming (RDP'09), whose work has made the workshop possible.

Brasília, Brazil, Santiago Escobar
June 2009 WFLP'09 Chair

Organization

WFLP'09 is part of the Federated Conference on Rewriting, Deduction, and Programming (RDP'09).

Program Committee

María Alpuente	Universidad Politécnica de Valencia, Spain
Sergio Antoy	Portland State University, USA
Christiano Braga	Universidade Federal Fluminense, Brazil
Rafael Caballero	Universidad Complutense de Madrid, Spain
David Déharbe	Universidade Federal do Rio Grande do Norte, Brazil
Rachid Echahed	CNRS, Laboratoire LIG, France
Moreno Falaschi	Università di Siena, Italy
Michael Hanus	Christian-Albrechts-Universität zu Kiel, Germany
Frank Huch	Christian-Albrechts-Universität zu Kiel, Germany
Tetsuo Ida	University of Tsukuba, Japan
Wolfgang Lux	Westfalische Wilhelms-Universitat Munster, Germany
Mircea Marin	University of Tsukuba, Japan
Camilo Rueda	Universidad Javeriana-Cali, Colombia
Jaime Sánchez-Hernández	Universidad Complutense de Madrid, Spain
Anderson Santana de Oliveira	Universidade Federal do Rio Grande do Norte, Brazil

Additional Referees

Gloria Álvarez	Iliano Cervesato	Albert Rubio	Rafael del Vado Vírseda
Demis Ballis	Yukiyoshi Kameyama	Clara Segura	Toshiyuki Yamada
Bernd Braßel	Temur Kutsia	Peter Sestoft	Hans Zantema
Linda Brodo	Miguel Palomino	Thierry Boy de la Tour	

Sponsoring Institution

Departamento de Sistemas Informáticos y Computación (DSIC)
Universidad Politécnica de Valencia (UPV)

Table of Contents