Integrating ILOG CP Technology into $\mathcal{TOY}$[⋆]

Ignacio 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 ($\mathcal{FD}$) constraint solver. In this work, we show how to integrate the ILOG CP $\mathcal{FD}$ 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 $\mathcal{FD}$ 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 [11]. 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 $\mathcal{H}$), linear and non-linear arithmetic constraints over reals ($\mathcal{R}$), finite domain constraints over integers ($\mathcal{FD}$), and a communication domain $\mathcal{M}$ which makes possible the cooperation among $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$. Whereas $\mathcal{R}$ as $\mathcal{FD}$ rely on the constraint solvers provided by SICStus Prolog, solving in $\mathcal{H}$ and $\mathcal{M}$ needs an explicit management [3]. $\mathcal{TOY}$ offers a wide range of finite domain constraints comparable to many CLP($\mathcal{FD}$) 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 $\mathcal{FD}$ constraint system is shown to the left of Fig. 1. $\mathcal{TOY}$ identifies each $\mathcal{FD}$ 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 $\mathcal{FD}$ 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 CLP($\mathcal{FD}$): $\mathcal{TOY}(\mathcal{FD}s)$

$\mathcal{TOY}$ (referred to as $\mathcal{TOY}(\mathcal{FD}s)$ from now on) has been using the $\mathcal{FD}$ 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 $\mathcal{FD}$ constraint system is shown in the middle of Fig. 1. Next, we show a basic example for illustrating the use of the system $\mathcal{TOY}(\mathcal{FD}s)$ with finite domains constraints.

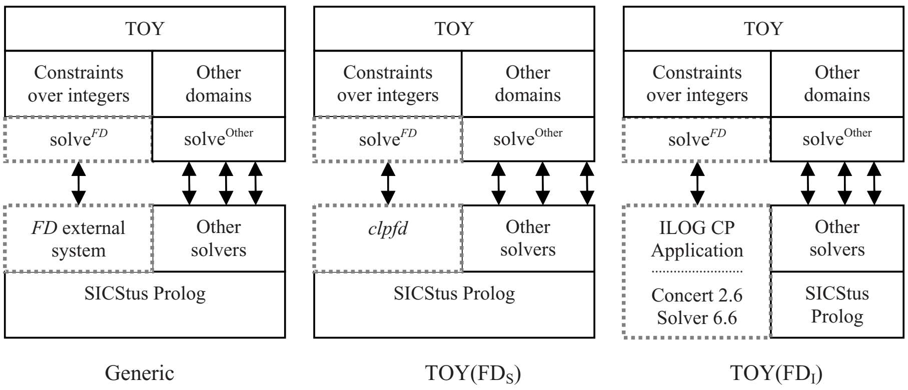


Fig. 1. Architectural Components

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 $\mathcal{TOY}(\mathcal{FD}s)$ 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 $\mathcal{FD}$ system reveals some disadvantages:

– Recent works [2] have proved that its performance can be enhanced, which is 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 [7] is an industrial technology market leader. It has a declarative nature. It provides a C++ API to access its libraries. Its $\mathcal{FD}$ 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 input constraint set. It allows the user to define new classes of constraints in order to formulate different and complex properties. The use of different constraint solvers for a unique application domain is also allowed. Moreover, libraries using specific, efficient algorithms for solving 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 [8]. 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 $\mathcal{FD}$ constraint solving:

* ILOG Solver 6.6, for generic $\mathcal{FD}$ problems solving.
* ILOG Scheduler 6.6, with specific algorithms for solving scheduling problems.
* ILOG Dispatcher 4.6, with specific algorithms for solving routing problems.

In this work we will consider only ILOG Solver 6.6. Fig. 2. shows the basic objects needed to model and solve the $\mathcal{FD}$ problem proposed in Example 1:

– `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 $\mathcal{FD}$ problem, which are:
  • $\mathcal{FD}$ constraints, each of them modeled as an `IloConstraint` object.
  • $\mathcal{FD}$ decision variables, each of them modeled as an `IloIntVar` object.
– `IloIntVarArray` ***vars***`(env)`. This vector is intended to make possible to reference, from a unique object, any `IloIntVar` contained in `model`.
– `IloSolver` ***solver***`(env)`. Is the main solving object. Contains the set of objects responsible of solving the $\mathcal{FD}$ problem, which are:
  • $\mathcal{FD}$ constraints, each of them modeled as an `IlcConstraint` object.
  • $\mathcal{FD}$ decision variables, each of them modeled as an `IlcIntVar` object.

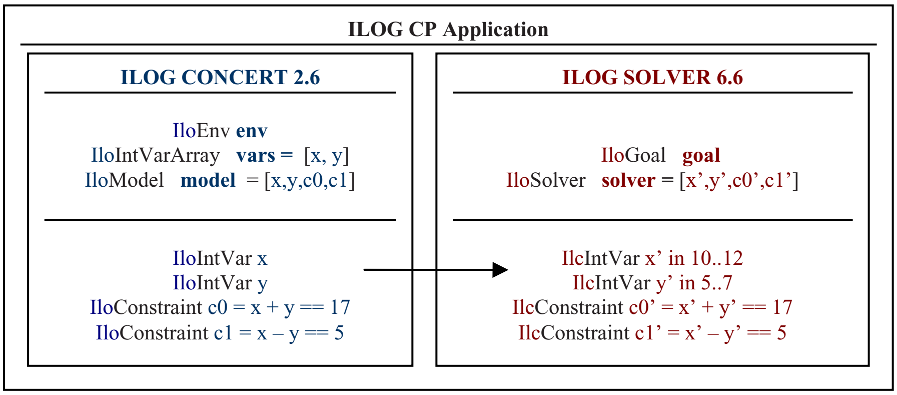

ILOG CP Application
ILOG CONCERT 2.6
IloEnv env
IloIntVarArray vars = [x, y]
IloModel model = [x,y,c0,c1]
IloIntVar x
IloIntVar y
IloConstraint c0 = x + y == 17
IloConstraint c1 = x – y == 5
ILOG SOLVER 6.6
IloGoal goal
IloSolver solver = [x',y',c0',c1']
IlcIntVar x' in 10..12
IlcIntVar y' in 5..7
IlcConstraint c0' = x' + y' == 17
IlcConstraint c1' = x' – y' == 5

Fig. 2. Generic ILOG CP Application

The main methods of `solver` we use in this work are:

- `solver.extract(model)`. For each `IloIntVar` and `IloConstraint` contained in `model` it creates an associated new `IlcIntVar` or `IlcConstraint` object, whose internal structure is targeted at `solver` solving techniques.
- `solver.propagate()`. Propagates the `IlcConstraint` set contained in `solver`. This propagation prunes some values of the `IlcIntVar` set contained in `solver` by using limit consistency techniques. In Fig. 2. we can see the remaining values of the `IlcIntVar` set contained in `solver` after the propagation of the `IlcConstraint` set.
- `solver.solve(goal)`. Uses the labeling enumeration procedure defined in `goal` to look for a first concrete solution of the $\mathcal{FD}$ problem contained in `solver`.

– `IloGoal` ***goal***`(env,vars,Strategy)`. This object represents a labeling enumeration procedure which labels the `IlcIntVars` contained in `solver` associated to the `IloIntVars` contained in `vars`. By this labeling procedure, `solver` is able to find the different extensional solutions to the $\mathcal{FD}$ problem. In this work we use two labeling strategies predefined in ILOG Solver 6.6:
  - A static search procedure `IloChooseFirstUnboundInt`, which selects the variables in the textual order they occur in `vars`.
  - A dynamic search procedure 'first fail' `IloChooseMinSizeInt`, which selects first the variable of `vars` with minimum domain size.

  For a given variable, both strategies select first the minimum value in its domain.

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

In this section, we explain in detail how to integrate ILOG CP $\mathcal{FD}$ technology into the system $\mathcal{TOY}$ (referred to as $\mathcal{TOY}(\mathcal{FD}i)$ 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 $\mathcal{TOY}(\mathcal{FD}i)$ is shown.

2.1 Connecting SICStus Prolog to C++

It is possible to communicate a SICStus Prolog application to a C++ component. The C++ component needs to be a dynamic library with a specific internal file structure. This communication is done by mapping a set of linking Prolog predicates (contained in the Prolog application) to a set of C++ functions (defined in the C++ component). SICStus Prolog also defines a set of possible conversions between Prolog arguments and C++ arguments. Each argument of a linking Prolog predicate 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 predicate 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 predicates 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 supplies a tool, `splfr` [10], for creating a dynamic library as, for instance `interface.dll` taking as input two files:

- `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){...}
  ```

The macro `splfr` is used as a shortcut to the execution of some compiling and linking commands offered by Microsoft Visual C++ [9]. 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 solve the communication difficulties between SICStus Prolog and ILOG CP in the system $\mathcal{TOY}(\mathcal{FD}i)$. As $\mathcal{TOY}$ is a system implemented in SICStus Prolog, the communication between $\mathcal{TOY}$ and its $\mathcal{FD}$ technology is quite natural in $\mathcal{TOY}(\mathcal{FD}s)$. However, as ILOG CP is implemented in C++, some glue code is needed to fix the impedance mismatch problem in $\mathcal{TOY}(\mathcal{FD}i)$.

There have been four difficult tasks to overcome in the new system $\mathcal{TOY}(\mathcal{FD}i)$. 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 $\mathcal{FD}$ solver, and we use `vars` if we refer to the decision variables contained in `model`.

Managing $\mathcal{FD}$ constraints

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

- We need to create as many `IloIntVar` decision variables as $\mathcal{FD}$ logic variables take part into the $\mathcal{FD}$ constraint set. These variables must be added to `model` and `vars`.
- We must find a bijective relation that associates each $\mathcal{FD}$ logic variable of the $\mathcal{TOY}$ goal to each decision variable existing in the ILOG CP vector `vars`.
- We model each $\mathcal{FD}$ constraint in ILOG CP over the set of decision variables of the vector `vars` associated to the set of $\mathcal{FD}$ logic variables involved in that $\mathcal{FD}$ constraint.

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

- The $\mathcal{FD}$ constraint and $\mathcal{FD}$ logic variable contained in $\mathcal{TOY}$.
- The `IloConstraint` and `IloIntVar` contained in `model`.
- The `IlcConstraint` and `IlcIntVar` created by `solver` from its associated `IloConstraint` and `IloIntVar` contained in `model`, respectively.

Fig. 3. shows the association between the different instances of an element in $\mathcal{TOY}$, ILOG Concert 2.6 and ILOG Solver 6.6.

TOY	ILOG CONCERT 2.6	ILOG SOLVER 6.6
X	IloIntVar x	IlcIntVar x’
Y	IloIntVar y	IlcIntVar y’
X #> Y	IloConstraint c0 = x > y	IlcConstraint c0’ = x’ > y’

Fig. 3. Association between $\mathcal{TOY}$ and ILOG CP

A first attempt for mapping a $\mathcal{FD}$ 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` conversions of the $\mathcal{FD}$ logic variables. Each time $solve^{FD}$ sends a new $\mathcal{FD}$ constraint to ILOG CP, the associated C++ function will first look for its $\mathcal{FD}$ logic variables in the `SP_term_ref` vector. If it can not find any variable, we can ensure that the C++ function is dealing with a new $\mathcal{FD}$ logic variable not handled before. So, the C++ function adds this new $\mathcal{FD}$ 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 $\mathcal{FD}$ logic variable of the $\mathcal{FD}$ constraint sent by $solve^{FD}$ is contained at an index of the `SP_term_ref` vector, the $\mathcal{FD}$ 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_refs` (as arguments or dynamically created within it) finishes its execution, all these `SP_term_refs` become invalid. Let’s see the next example, where we define an interface between the Prolog predicates `p1`, `p2` and `p3` and the C++ functions `f1`, `f2` and `f3`, respectively. Functions `f1` and `f2` receive a Prolog term as an argument, while `f3` receives two Prolog terms.

– Let’s call `p3` with two occurrences of the logic variable `X`, as `p3(X,X)`. If we call `SP_compare(t1,t2)` within `f3(SP_term_ref t1, SP_term_ref t2)` the result says that both `SP_term_refs` are in fact the same Prolog term.
– But, let’s do the call `p1(X)`. We store `t1` of `f1(SP_term_ref t1)` in a global vector with type `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 call `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_refs are different. There is no doubt that both are in fact the same Prolog term. The problem is that, when f1 finishes, 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 in the Prolog application by the use of a list of $\mathcal{FD}$ logic variables (referred to as V from now on). We want V to be used in each $solve^{FD}$ predicate. On the one hand, SICStus Prolog does not allow global variables. On the other hand, there is a logic variable Cin [4], which represents a mixed constraint 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 our $\mathcal{FD}$ logic variables list V, into Cin. Each time a $solve^{FD}$ predicate manages a new $\mathcal{FD}$ constraint, we can check whether a $\mathcal{FD}$ logic variable belongs to V or not by accessing it within Cin. Any new $\mathcal{FD}$ logic variable is automatically added to the end of V, 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 $\mathcal{FD}$ logic variables of the $\mathcal{FD}$ constraint belong to V, $solve^{FD}$ determines their indexes, and puts them as arguments to the C++ function, which models the $\mathcal{FD}$ constraint by adding to model a new IloConstraint over the associated positions of vars.

Synchronizing ILOG CP with $\mathcal{TOY}$

$\mathcal{TOY}$ can also bind its $\mathcal{FD}$ 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 term unification which results from the Herbrand equality constraint X == 3. This unification is visible at any occurrence of that $\mathcal{FD}$ logic variable, particularly the one in V. This causes an inconsistency between the contents of V 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 $\mathcal{FD}$ logic variable. Thus, each time the Herbrand solver binds a $\mathcal{FD}$ 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 $\mathcal{FD}$ logic variable in V. But the arguments of verify_attributes(+Var, +Value, +Goals) are fixed. As SICStus Prolog does not allow global variables, there is no way to get access to V.

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 `V`. Now it becomes a list of pairs. The first element of each pair contains the $\mathcal{FD}$ logic variable, and the second one contains a flag which determines if the bound $\mathcal{FD}$ logic variable has been synchronized with `vars`. Thus, while the $\mathcal{FD}$ logic variable is not bound, the value of the flag remains at `0`. When the $\mathcal{FD}$ 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 $\mathcal{FD}$ constraint to ILOG CP, it must previously:

- Look for any pair in `V` (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 `V` by `[`*`value`*`,1]`

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

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

The first $\mathcal{FD}$ logic variable of the goal in the narrowing order is `X`, which occurs at the first position of `V` and `vars`. The synchronization of `X == 1` as `vars[0] == 1` makes the $\mathcal{FD}$ problem infeasible. So, the $\mathcal{TOY}$ goal will fail after `X == 1`, and there is no need for computing the rest of the goal expressions. However, the first equality `vars[0] == 1` is not computed until the next $\mathcal{FD}$ constraint is posted.
As `X == 1, X1 == 1, X2 == 1, ... , X1000 == 1` are computed by the Herbrand solver there are no more $\mathcal{FD}$ 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 `IlcConstraint` set propagation produced by `solver.propagate()` or a labeling enumeration procedure `goal` used by `solver.solve(goal)`. This produces a lack of synchronization between the vector `vars` and `V`. To synchronize, whenever any `IlcIntVar var'` (associated to the `IloIntVar var` contained in `vars[i]`) is bound to *value*, the pair `[`*`Var`*`,0]` contained at position `i` of `V` must be automatically unified with `[`*`value`*`,1]`.

To this end, the predicates of *solve*^{FD} send the list `V` as an input argument to the C++ functions managing the $\mathcal{FD}$ constraints in ILOG CP. An output argument is also added to obtain the new state of `V` computed by the C++ function after `solver` propagation or labeling. A new global variable `vector<int,int>` must be created in ILOG CP. Each pair of the vector contains:

– The index `i` in `vars` of the `IloIntVar` associated to the `IlcIntVar` treated.
– The *value* that `solver` has obtained for this `IlcIntVar`.

Any C++ function clears `vector<int,int>` at the beginning of the managment of a new $\mathcal{FD}$ constraint. After the solving techniques used, the C++ function accesses to the content of `vector<int,int>`, to see whether there are any variable that has been bound. By using `vector<int,int>` and `V`, the C++ function builds the new state of `V`, unifying as many $\mathcal{FD}$ logic variables as `vector<int,int>` demands.

The only remaining task to be explained is how the solving techniques add automatically each pair to `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 the `IlcIntVar` associated to this `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 associated `IloIntVar` of the bound `IlcIntVar` 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[i]` and `model`, we impose the constraint `IlcCheckWhenBound`.
– When the `IlcIntVar` associated to `vars[i]` is bound to *value*, `IlcCheckWhenBound` propagates, triggering the demon `RealizeVarBound`.
– `RealizeVarBound` executes the `IlcCheckWhenBound` method `varDemon`, which adds the pair `<i,value>` to `vector<int,int>`.

Solutions in $\mathcal{TOY}(\mathcal{FD}i)$

Any $\mathcal{TOY}(\mathcal{FD}s)$ solution is expressed in general with constraints (equality, disequality and $\mathcal{FD}$ constraints –including ranges–). Of course, $\mathcal{TOY}(\mathcal{FD}s)$ accepts to label $\mathcal{FD}$ variables by using a $\mathcal{FD}$ labeling enumeration procedure, in order to obtain the extensional solution to a goal.

The system $\mathcal{TOY}(\mathcal{FD}i)$ presented in this work reproduces the solution structure of $\mathcal{TOY}(\mathcal{FD}s)$. But, as a first approach, we do not support the use of backtracking, so we can only use labeling enumeration procedures to look for the first concrete solution to a goal. When the goal is completely finished, we show to the user the set of non-ground $\mathcal{FD}$ constraints as well as the remaining values of the $\mathcal{FD}$ variables.

ILOG Solver 6.6 does not grant access to simplified constraints (i.e., solved forms). So, to show the solution to the user, we do not parse the `IlcConstraint` set contained in `solver`. Instead of that, we store in `Cin` a list with the $\mathcal{FD}$ constraints (referred to as `C` from now on) appearing in the $\mathcal{TOY}$ goal. When a

$solve^{FD}$ predicate manages a new $\mathcal{FD}$ constraint of the goal, this constraint is added to `C`. In the solution we show any non-ground element of `C`.

To show the remaining values of the $\mathcal{FD}$ logic variables, we access to each `IlcIntVar` contained in `solver` throughout its associated `IloIntVar` contained in `model`. ILOG Solver 6.6 provides some methods to check the remaining values of these variables.

2.3 A $\mathcal{TOY}(\mathcal{FD}i)$ Example

In this section we detail how goal solving works with the new system $\mathcal{TOY}(\mathcal{FD}i)$ following 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 goal expression evaluation. On the one hand, we look at the state of `V` and `C` within `Cin`. On the other hand, we look at the state of `vars`, `model` and `solver` by pointing out any `IloIntVar`, `IloConstraint`, `IlcIntVar`, `IlcConstraint` object accessed through them. Any new element added by the evaluation of a goal expression is highlighted in boldface. At the beginning of the computation, all data structures are empty, as we can see in Fig. 4.

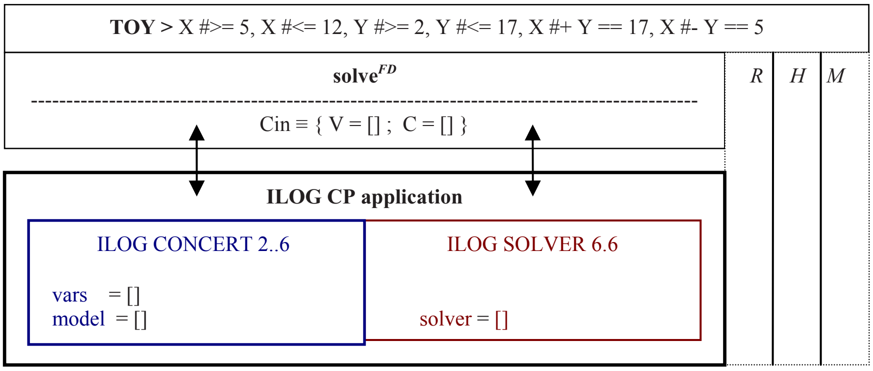

TOY > X #>= 5, X #<= 12, Y #>= 2, Y #<= 17, X #+ Y == 17, X #- Y == 5
solve^FD
Cin ≡ { V = [] ; C = [] }
R
H
M
ILOG CP application
ILOG CONCERT 2..6
vars = []
model = []
ILOG SOLVER 6.6
solver = []

Fig. 4. Beginning of the computation

Now, we detail in Fig. 5. the evaluation of the goal expression `X #>= 5`. First, we model the $\mathcal{FD}$ constraint in ILOG Concert 2.6:

1. As `X` is not contained in `V`, we add the new pair `[X,0]` in the position 0 of `V`. Then, we create a new `IloIntVar x` and we add it to `vars[0]` and `model`. Now, `X` and `x` are associated due to they are at the same index (0) of `V` and `vars`, respectively.
2. We add the propagator `IloCheckWhenBound c1(x,0)` to be able to synchronize the $\mathcal{FD}$ logic variable `X` when the `IlcIntVar` associated to `x` becames bound to a value.

3. We add the new $\mathcal{FD}$ constraint `X #>= 5` to `C`. Then, we add to `model` the `IloConstraint c0`.

Then, we solve the $\mathcal{FD}$ problem contained in `model`:

4. We use `solver.extract(model)` to make a one-to-one object translation of the `model` content. This creates the new objects `x',c0'` and `c1'`.
5. We use `solver.propagation()`. It prunes some values of the domain of `x'`.
6. As the state of `solver` remains feasible, $\mathcal{TOY}$ continues with the evaluation of the next goal expression.

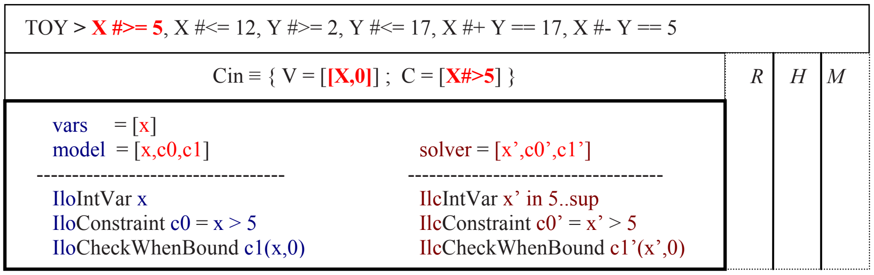

TOY > X #>= 5, X #<= 12, Y #>= 2, Y #<= 17, X #+ Y == 17, X #- Y == 5
Cin ≡ { V = [[X,0]] ; C = [X#>5] }
R
H
M
vars = [x]
model = [x,c0,c1]
solver = [x',c0',c1']
IloIntVar x
IloConstraint c0 = x > 5
IloCheckWhenBound c1(x,0)
IlcIntVar x' in 5..sup
IlcConstraint c0' = x' > 5
IlcCheckWhenBound c1'(x',0)

Fig. 5. Evaluation of the first $\mathcal{FD}$ constraint

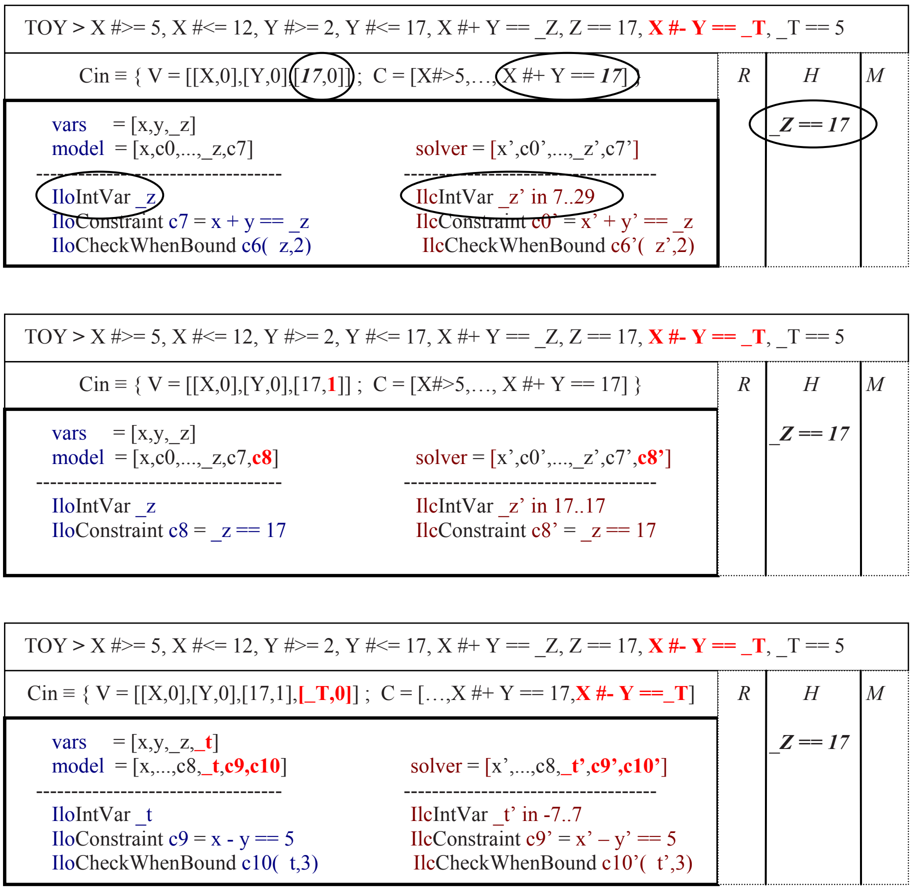
We do not detail here the evaluation of `X #<= 12, Y #>= 2` and `Y# <= 17`, which are quite similar to `X #>=5`. The evaluation continues with the expression `X #+ Y == 17`. As this is a compound constraint, $\mathcal{TOY}$ decomposes it into the primitive constraints `X #+ Y == _Z` and `_Z == 17`.

– The evaluation of `X #+ Y == _Z` adds this constraint to `C`. It also adds `[_Z,0]` to `V`, `_z` to `model` and `_z'` to `solver`.
– In the evaluation of `_Z == 17` the constraint is sent to the Herbrand solver $\mathcal{H}$, which binds the variable `_Z` to the value 17. This causes that the instances of `_Z` in `V` and `C` are also unified to 17, producing a lack of consistency between `_Z` and its associated variables in ILOG `_z` and `_z'`. The synchronization of `_z` and `_z'` will happen with the management of the next $\mathcal{FD}$ constraint or at the end of the $\mathcal{TOY}$ goal.

The evaluation continues with the expression `X #- Y == 5`. As this is a compound constraint, $\mathcal{TOY}$ decomposes it into the primitive constraints: `X #- Y == _T` and `_T == 5`. We detail here the evaluation of the goal expression `X #- Y == _T`, which can be seen in Fig. 6.

In the top of Fig. 6. we see the state before the evaluation starts. We can see that the constraint `_Z == 17` appears in the Herbrand solver of $\mathcal{TOY}$. Also, the instances of `_Z` in `V` and `C` are highlighted, because they have been unified to 17. The variables associated to `_Z` in ILOG are also highlighted, because they have not been bound yet to the value `17`.

In the middle of Fig. 6. we see the state after the synchronization of `_Z` with `_z` and `_z'`. Now the `IlcIntVar _z'` is also bound to the value `17`. The pair



```
TOY > X #>= 5, X #<= 12, Y #>= 2, Y #<= 17, X #+ Y == _Z, Z == 17, X #- Y == _T, _T == 5
Cin ≡ { V = [[X,0],[Y,0],[17,0]] ;  C = [X#>5,…, X #+ Y == 17] }      R    H         M
                                                                           _Z == 17
vars   = [x,y,_z]
model  = [x,c0,...,_z,c7]                  solver = [x',c0',...,_z',c7']
-----------------------------------        -----------------------------------
IloIntVar _z                               IlcIntVar _z' in 7..29
IloConstraint c7 = x + y == _z             IlcConstraint c0' = x' + y' == _z
IloCheckWhenBound c6( z,2)                 IlcCheckWhenBound c6'( z',2)
```

```
TOY > X #>= 5, X #<= 12, Y #>= 2, Y #<= 17, X #+ Y == _Z, Z == 17, X #- Y == _T, _T == 5
Cin ≡ { V = [[X,0],[Y,0],[17,1]] ;  C = [X#>5,…, X #+ Y == 17] }      R    H         M
                                                                           _Z == 17
vars   = [x,y,_z]
model  = [x,c0,...,_z,c7,c8]               solver = [x',c0',...,_z',c7',c8']
-----------------------------------        -----------------------------------
IloIntVar _z                               IlcIntVar _z' in 17..17
IloConstraint c8 = _z == 17                IlcConstraint c8' = _z == 17
```

```
TOY > X #>= 5, X #<= 12, Y #>= 2, Y #<= 17, X #+ Y == _Z, Z == 17, X #- Y == _T, _T == 5
Cin ≡ { V = [[X,0],[Y,0],[17,1],[_T,0]] ;  C = […,X #+ Y == 17,X #- Y ==_T]   R    H         M
                                                                           _Z == 17
vars   = [x,y,_z,_t]
model  = [x,...,c8,_t,c9,c10]              solver = [x',...,c8,_t',c9',c10']
-----------------------------------        -----------------------------------
IloIntVar _t                               IlcIntVar _t' in -7..7
IloConstraint c9 = x - y == 5              IlcConstraint c9' = x' – y' == 5
IloCheckWhenBound c10( t,3)                IlcCheckWhenBound c10'( t',3)
```

Fig. 6. Constraint management with synchronization

`[17,0]` of `V` has been changed by `[17,1]`, because the variables associated in ILOG are now synchronized.

After synchronizing ILOG CP with the equalities produced by the Herbrand solver, we manage the constraint `X #- Y == _T`. We can see this in the bottom of Fig. 6.

$\mathcal{TOY}$ continues with the evaluation of the next goal expression. The constraint `_T == 5` is sent to the Herbrand solver $\mathcal{H}$. This will bind the variable `_T` to the value `5`. The instances of `_T` in `V` and `C` will be unified to `5`. This produces a lack of consistency between `_T` (now bound to 5) and its associated variables in ILOG `_t` and `_t'`. As there are no more $\mathcal{FD}$ constraints, the synchronization will happen at the end of the $\mathcal{TOY}$ goal.

With this last synchronization we create a new `IloConstraint c = _t == 5` in `model`. Then `solver` will translate and propagate this new constraint, binding the `IlcIntVar _t'`. We modify the pair `[5,0]` of the list `V` to `[5,1]`.

Now the goal is completely finished. As the state of `solver` remains feasible, $\mathcal{TOY}$ shows the solution of the computation to the user. First we show the $\mathcal{FD}$ constraints by displaying any non-ground term contained in `C`. Then we show the values for the $\mathcal{FD}$ logic vars by accessing its associated `IlcIntVars` contained in ILOG. We do not show the extra variables produced during narrowing.

```
Toy(FDi)> X #>= 5, X #<= 12, Y #>= 2, Y #<= 17,
          X #+ Y == 17, X #- Y == 5
      yes
      { X #>= 5
        X #=< 12
        Y #>= 2
        Y #=< 17
        X #+ Y == 17
        X #- Y == 5
        X in 10..12
        Y in 5..7 }
      Elapsed time: 0 ms.
```

In Fig. 7. we see the state of $\mathcal{TOY}(\mathcal{FD}i)$ after the computation.

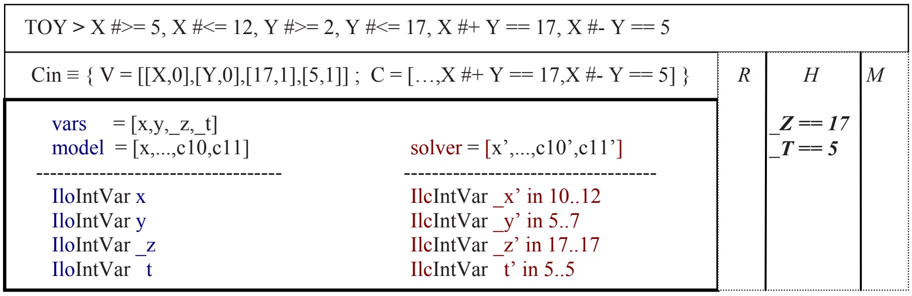


Fig. 7. $\mathcal{TOY}(\mathcal{FD}i)$ state after computation

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 $\mathcal{TOY}(\mathcal{FD}s)$ and $\mathcal{TOY}(\mathcal{FD}i)$ 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 the two labeling strategies `IloChooseFirstUnboundInt` and `IloChooseMinSizeInt` (denoted by ff) presented in the section 1.2.

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

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

The benchmark programs are:

– First: 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, $\mathcal{TOY}(\mathcal{FD}i)$ takes more time than $\mathcal{TOY}(\mathcal{FD}s)$ 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 $\mathcal{TOY}(\mathcal{FD}i)$ is slower than $\mathcal{TOY}(\mathcal{FD}s)$. This is because of the time involved in the communication between the Prolog implementation of $\mathcal{TOY}(\mathcal{FD}i)$ and ILOG CP. However, as the nodes needed to be explored increase slightly, this waste of time is overcome, making $\mathcal{TOY}(\mathcal{FD}i)$ more efficient than $\mathcal{TOY}(\mathcal{FD}s)$.

– Second: The solution `[X1,_,Xn]` holds: $\forall i \in \{1..n\}$ `Xi` $= n-(i-1)$. The above conclusions are clearly confirmed in this second benchmark, as $\mathcal{TOY}(\mathcal{FD}i)$ is faster than $\mathcal{TOY}(\mathcal{FD}s)$ 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, $\mathcal{TOY}(\mathcal{FD}i)$ takes slightly less solving time than $\mathcal{TOY}(\mathcal{FD}s)$. In any case, these measurements point out that our first approach to integrate the ILOG CP technology into $\mathcal{TOY}(\mathcal{FD}i)$ is encouraging, but also that the management of the additional data structures used for the interface should be optimized. 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

4 Conclusions and Future Work

In this work, we have studied how to integrate the $\mathcal{FD}$ ILOG CP technology into the system $\mathcal{TOY}$. We have shown that this technology offers some advantages over the existing system $\mathcal{TOY}(\mathcal{FD}s)$ based on the $\mathcal{FD}$ 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 $\mathcal{TOY}(\mathcal{FD}i)$ is faster than $\mathcal{TOY}(\mathcal{FD}s)$ 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 $\mathcal{FD}$ component. Therefore, optimizing this management will be the target of our immediate future work. As many practical $\mathcal{FD}$ problems covered by $\mathcal{TOY}(\mathcal{FD}s)$ require the use of non-deterministic functions, backtracking management will be covered in a next work. This will also allow us to use labeling enumeration procedures to find the different extensional solutions of a goal. So, we will be able to deal with an extended set of benchmarks (as the one seen in [5]) in $\mathcal{TOY}(\mathcal{FD}i)$ future releases. Another subject of interest is to test the constraint libraries ILOG Scheduler 6.6 and ILOG Dispatcher 4.6 bundled in ILOG CP 1.4, as well as other constraint libraries, as Gecode [6].

References

1. Arenas, P., Estévez, S., Fernández, A., Gil, A., López-Fraguas, F., Rodríguez-Artalejo, M., Sáenz-Pérez, F.: $\mathcal{TOY}$. A multiparadigm declarative language. version 2.3.1 (2007); Caballero, R., Sánchez, J. (eds.), `http://toy.sourceforge.net`
2. del Campo, R.G., Sáenz-Pérez, F.: Programmed search in a timetabling problem over finite domains. Electronic Notes in Theoretical Computer Science 177, 253–267 (2007)
3. Estévez-Martín, S., Fernández, A., Hortalá-González, M., Sáenz-Pérez, F., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: On the Cooperation of the Constraint Domains H, R and FD in $CFLP$. Theory and Practice in Logic Programming 9(4), 415–527 (2009)

4. Estévez-Martín, S., Fernández, A.J., Sáenz-Pérez, F.: About implementing a constraint functional logic programming system with solver cooperation. In: Proc. of CICLOPS 2007, pp. 57–71 (2007)
5. Fernández, A.J., Hortalá-González, T., Sáenz-Pérez, F., del Vado-Vírseda, R.: Constraint Functional Logic Programming over Finite Domains. Theory and Practice in Logic Programming 7(5), 537–582 (2007)
6. Gecode. Gecode, `http://www.gecode.org/`
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`

Santiago Escobar (Ed.)

Functional and Constraint Logic Programming

18th International Workshop, WFLP 2009
Brasilia, Brazil, June 28, 2009
Revised Selected Papers

Springer

Volume Editor

Santiago Escobar
Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación
Camino de vera, s/n, 46022 Valencia, Spain
E-mail: sescobar@dsic.upv.es

Library of Congress Control Number: 2010923100

CR Subject Classification (1998): F.4, F.3.2, D.3, I.2.2-5, I.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-11998-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-11998-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains a selection of the papers presented at the 18th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2009), held on June 28, 2009 in Brasília, Brazil. Previous WFLP meetings were held in Siena (2008), Paris (2007), Madrid (2006), Tallinn (2005), Aachen (2004), Valencia (2003), Grado (2002), Kiel (2001), Benicassim (2000), Grenoble (1999), Bad Honnef (1998), Schwarzenberg (1997, 1995, and 1994), Marburg (1996), Rattenberg (1993), and Karlsruhe (1992).

The aim of the WFLP series 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.

WFLP 2009 solicited papers in all areas of functional and (constraint) logic programming, including but not limited to:

- Foundations: formal semantics, rewriting and narrowing, non-monotonic reasoning, dynamics, and type theory.
- Language Design: modules and type systems, multi-paradigm languages, concurrency and distribution, and objects.
- Implementation: abstract machines, parallelism, compile-time and run-time optimizations, and interfacing with external languages.
- Transformation and Analysis: abstract interpretation, specialization, partial evaluation, program transformation, and meta-programming.
- Software Engineering: design patterns, specification, verification and validation, debugging, and test generation.
- Integration of Paradigms: integration of declarative programming with other paradigms such as imperative, object-oriented, concurrent, and real-time programming.
- Applications: declarative programming in education and industry, domain-specific languages, visual/graphical user interfaces, embedded systems, WWW applications, knowledge representation and machine learning, deductive databases, advanced programming environments and tools.

The WFLP 2009 workshop was part of the Federated Conference on Rewriting, Deduction, and Programming (RDP 2009), which grouped together different events such as the 20th International Conference on Rewriting Techniques and Applications (RTA 2009), the 9th International Conference on Typed Lambda Calculi and Applications (TLCA 2009), the 4th Workshop on Logical and Semantic Frameworks, with Applications (LFSA 2009), the 10th International Workshop on Rule-Based Programming (RULE 2009), and the 9th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2009).

There were 14 original contributions to the workshop, the Program Committee selected nine papers for publication, and revised versions of these selected papers are included in this volume. Each contribution was reviewed by at least three Program Committee members. This volume also includes two invited contributions 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 for both the scientific program and this volume. I am also grateful to Andrei Voronkov for his extremely useful EasyChair system for automation of conference chairing.

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. I gratefully acknowledge the *Departamento de Sistemas Informáticos y Computación* of the *Universidad Politécnica de Valencia*, who supported this event. Finally, I express our gratitude to all members of the Local Organization of the Federated Conference on Rewriting, Deduction, and Programming (RDP 2009), whose work made the workshop possible.

December 2009 Santiago Escobar

Organization

Program Chair

Santiago Escobar
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de vera, s/n
E-46022 Valencia, Spain
`sescobar@dsic.upv.es`

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-Universität Münster, 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 Reviewers

Hassan Aït-Kaci
Gloria Alvarez
Demis Ballis
Bernd Braßel
Linda Brodo
Iliano Cervesato
Yukiyoshi Kameyama
Temur Kutsia
Michael Maher
Miguel Palomino
Cody Roux
Albert Rubio
Clara Segura
Peter Sestoft
Nikhil Swamy
Thierry Boy de la Tour
Rafael del Vado Vrseda
Toshiyuki Yamada
Hans Zantema

Table of Contents