Improving the Search Capabilities of a CFLP(FD) System*

Ignacio Castiñeiras[1], **Fernando Sáenz-Pérez**[2]

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

Abstract: In this work we focus on the CFLP system $\mathcal{TOY}(\mathcal{FD})$, implemented in SICStus Prolog and supporting $\mathcal{FD}$ constraints by interfacing the external CP($\mathcal{FD}$) solvers of Gecode and ILOG Solver. We extend $\mathcal{TOY}(\mathcal{FD})$ with new search primitives, in a setting easily adaptable to other Prolog CLP or CFLP systems. We describe the primitives from a solver-independent point of view, pointing out some novel concepts not directly available in any CP($\mathcal{FD}$) library we are aware of, as well as how to specify some search criteria at $\mathcal{TOY}(\mathcal{FD})$ level and how easily these strategies can be combined to set different search scenarios. We describe the implementation of the primitives, presenting an abstract view of the requirements and how they are targeted to the Gecode and ILOG libraries. We evaluate the resulting $\mathcal{TOY}(\mathcal{FD})$ architecture and we use some benchmarks to prove that the use of the search strategies improve its solving performance.

Keywords: CFLP, FD Search Strategies, Solver Integration

1 Introduction

The use of *ad hoc* search strategies has been identified as a key point for solving Constraint Satisfaction Problems (CSP's) [Tsa93], allowing the user to interact with the solver in the search of solutions (exploiting its knowledge about the structure of the CSP and its solutions). Different paradigms provide different expressivity for specifying search strategies: Constraint Logic Programming CLP($\mathcal{FD}$) [JM94] and Constraint Functional Logic Programming CFLP($\mathcal{FD}$) [Han07] provide a declarative view of this specification, in contrast to the procedural one offered by Constraint Programming CP($\mathcal{FD}$) [MS98] systems (which make the programming of a strategy to depend on low-level details associated to the constraint solver, and even on the concrete machine the search is being performed). Also, due to their model reasoning capabilities, CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) treat search primitives as simple expressions, making possible to: (1) Place a search primitive at any point of the program, (2) Combine several primitives to develop complex search heuristics, (3) Intermix search primitives with constraint posting, and (4) Use indeterminism to apply different search scenarios for solving a CSP.

The main contribution of this paper is to present a set of search primitives for CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems implemented in Prolog, and interfacing external CP($\mathcal{FD}$) solvers with a

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, UCM-BSCH-GR58/08-910502, and S2009TIC-1465

C++ API. The motivation of this approach is to take advantage of: (i) The high expressivity of CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) for specifying search strategies, and (ii) The high efficiency of CP($\mathcal{FD}$) solvers. The paper focuses on the CFLP($\mathcal{FD}$) system $\mathcal{TOY}(\mathcal{FD})$ [FHSV07], more precisely in the system versions $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ [CS12], interfacing the external CP($\mathcal{FD}$) solvers (with C++ API) of Gecode 3.7.3 [STL12] and IBM ILOG Solver 6.8 [IBM10], resp. $\mathcal{TOY}(\mathcal{FD})$ is completely developed in SICStus Prolog [Mat12]. Their programs follow a syntax mostly borrowed from Haskell [PJ02], with the remarkable exception that program and type variables begin with upper-case letters whereas data constructors, types and functions begin with lower-case. Instead of using an abstract machine for running byte-code or intermediate code from compiled programs, the $\mathcal{TOY}$ compiler uses SICStus Prolog as an object language [LLR93]. Regarding search, $\mathcal{TOY}(\mathcal{FD})$ offers two possibilities up to now: (1) Defining a new search from scratch at $\mathcal{TOY}(\mathcal{FD})$ level (with a $\mathcal{TOY}(\mathcal{FD})$ function that use reflection functions to represent the search procedure), and, (2) Using the search primitive `labeling`, which simply relies on predefined search strategies already existing in Gecode and ILOG, resp. The use of external CP($\mathcal{FD}$) solvers (with C++ API) opens a third possibility, which we exploit in this paper: Enhancing the search language of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ with new parametric search primitives, implementing them in Gecode and ILOG by extending their underlying search libraries.

The paper is organized as follows: Section 2 presents an abstract description of the new parameterizable $\mathcal{TOY}(\mathcal{FD})$ search primitives, pointing out some novel concepts not directly available in any CP($\mathcal{FD}$) library we are aware of, as well as how to specify some search criteria at $\mathcal{TOY}(\mathcal{FD})$ level and how easily these strategies can be combined to set different search scenarios. Section 3 describes the implementation of the primitives in $\mathcal{TOY}(\mathcal{FD})$, presenting an abstract view of the requirements, how they are targeted to the Gecode and ILOG libraries and a evaluation of the resulting architecture of the system. Section 4 presents some preliminary although encouraging case studies, showing that the use of the search strategies improve the solving performance of both $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$. Finally, Section 5 presents some conclusions and future work.

2 Search Primitives Description

This section presents eight new $\mathcal{TOY}(\mathcal{FD})$ primitives for specifying search strategies, allowing the user to interact with the solver in the search for solutions. These primitives bridge the gap between the other two classical approaches available in $\mathcal{TOY}(\mathcal{FD})$: Defining a whole search procedure at $\mathcal{TOY}$ level (by using reflection functions), and relying on the set of predefined search strategies available in the solver library. Each primitive has its own semantics, and it is parameterizable by several basic components. The search primitives are considered by the language as simple expressions, so intermixing search strategies with the regular posting of constraints is allowed. The section describes the primitives and their components (including its type declaration) from an abstract, solver independent point of view. It emphasizes: (1) Some novel search concepts arisen, which are not available in the predefined search strategies of any CP solver, (2) How easy and expressive it is to specify some search criteria at $\mathcal{TOY}(\mathcal{FD})$ level, and (3) The appealing possibilities $\mathcal{TOY}(\mathcal{FD})$ offers to apply different search strategies for solving a CP problem.

2.1 Labeling Primitives

Primitive lab

`lab :: varOrd -> valOrd -> int -> [int] -> bool`

This primitive collects (one by one) all possible combination of values satisfying the set of constraints posted to the solver. It is parameterized by four basic components. The first and second ones represent the variable and value order criteria to be used in the search strategy, resp. To express them we have defined in $\mathcal{TOY}$ the enumerated datatypes `varOrd` and `valOrd`, covering all the predefined criteria available in the Gecode documentation [STL13]. They also include a last case (`userVar` and `userVal`, resp.) in which the user implements its own variable/value selection criteria at $\mathcal{TOY}(\mathcal{FD})$ level. The third element `N` represents how many variables of the variable set are to be labeled. This represents a novel concept not available in the predefined search strategies of any CP solver. The fourth argument represents the variable set `S`. Thus, the search heuristic uses `varOrd` to label just `N` variables of `S`.

Next $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom) show how expressive, easy and flexible is to specify a search criteria in $\mathcal{TOY}(\mathcal{FD})$. In the example, the search strategy of the goal uses the `userVar` and `userVal` selection criteria (specified by the user in the functions `myVarOrder` and `myValOrder`, resp.) The `lab` search strategy computes partial solutions to the $\mathcal{TOY}(\mathcal{FD})$ goal `domain [X,Y, Z] 0 4, Y /= 1, Y /= 3, Z /= 2`. Then, "rest of $\mathcal{TOY}(\mathcal{FD})$ goal" is processed to compute complete solutions. Our search strategy acts over the set of variables `[X,Y,Z]`, but it is only expected to label one of them.

```
myVarOrder:: [int] -> int
myVarOrder V = fst (foldl cmp (0,0) (zip (take (length V) (from 0))
                                         (map (length . get_dom) V)))
%
myValOrder:: [[int]] -> int        | from:: int -> [int]
myValOrder D = head (last D)       | from N = [N | from (N+1)]
%
cmp:: (int,int) -> (int,int) -> (int,int)
cmp (I1,V1) (I2,V2) = if (V1 >= V2) then (I1,V1) else (I2,V2)
-----------------------------------------------------------------------
TOY(FD)> domain [X,Y,Z] 0 4, Y /= 1, Y /= 3, Z /= 2,
         lab userVar userVal 2 [X,Y,Z], ... (REST OF TOY(FD) GOAL)
```

The function `myVarOrder` selects first the variable with more intervals on its domain. It receives the list of variables involved in the search strategy, returning the index of the selected one. To do so it uses: (i) The auxiliary functions `from` and `cmp`. (ii) The predefined functions `fst`, `foldl`, `zip`, `take`, `length`, `map`, `head`, `last` and `(.)` (all of them with an equivalent semantics as in Haskell). (iii) The reflection function `get_dom`, which accesses the internal state of the solver to obtain the domain of a variable (this domain is presented as a list of lists, where each sublist represents an interval of values).

The function `myValOrder` receives as its unique argument the domain of the variable, returning the lower bound of its upper interval. So, in conclusion, the first two (partial) solutions obtained by the `lab` strategy are: `(X in 0..4, Y -> 4, Z -> 3)` and `(X in 0..4, Y -> 4, Z -> 4)`.

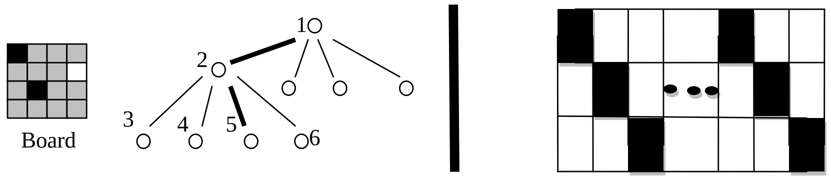

Figure 1: Applying `labB` and `fragB` to the *n*-queens problem

Primitive labB

```
labB :: varOrd -> valOrd -> int -> [int] -> bool
```

This primitive uses the same four basic elements as `lab`. However, its semantics is different, as it follows the `varOrd` and `valOrd` criteria to explore just one branch of the search tree, with no backtracking allowed. Let us explain it by using the *4-queens* example.

Using `lab unassignedLeftVar smallestVal 0 [X1,X2,X3,X4]` (where `0` in the third argument stands for labeling all the variables) we obtain two solutions: `X1 -> 1, X2 -> 3`, `X3 -> 2`, `X4 -> 4` and `X1 -> 2`, `X2 -> 4`, `X3 -> 1`, `X4 -> 3`. However, if we use `labB unassignedLeftVar smallestVal 0 [X1,X2,X3,X4]` the strategy fails, getting no solutions. Left hand side of Fig. 1 (4×4 square `Board` and tree) shows the computation process. First, the selected criteria assigns `X1 -> 1` at root node (`1`), leading to node `2`. Propagation reduces search space to `(X2 in 3..4, X3 in 2` $\vee$ `4, X4 in 2..3)`, pruning nodes `3` and `4`. Then, computation assigns `X2 -> 3` (leading to node `5`), and propagation leads to an empty domain for `X3`. So, the explored tree path leads to no solutions, and so it does the computation. As we have seen, propagation during search modifies the intended branch to be explored (in our example, it explores the branch `1-2-5` instead of the `1-2-3`).

Primitive labW

```
labW :: varOrd -> bound -> int -> [int] -> bool
```

This primitive performs an exhaustive breadth exploration of the search tree, storing the satisfiable leaf nodes achieved to further sort them by a specified criteria. Let us consider a first example to understand the semantics of `labW`. The following $\mathcal{TOY}(\mathcal{FD})$ goal has four variables, where two implication constraints relate `X` and `Y` with `V1` and `V2`, resp.

```
TOY(FD)> domain [X,Y] 0 1, post_implication X (#=) 1 V1 (#>) 1,
         domain [V1,V2] 0 3, post_implication Y (#=) 0 V2 (#>) 0,
         labW unassignedLeftVar smallestSearchTree 2 [X,Y,V1,V2], ...
```

If we had used a `lab unassignedLeftVar smallestVal 2 [X,Y,V1,V2]` strategy (instead of the `labW` one) to label the first two unbound vars of `[X,Y,V1,V2]`, then the search would have explored the search tree obtaining (one by one) the next four feasible solutions: `X -> 0, Y -> 0`, `X -> 0, Y -> 1`, `X -> 1, Y -> 0` and `X -> 1, Y -> 1`. Fig 3 represents the exploration of the search tree for obtaining those four solutions, where each black node represents a solution, and the triangle it has below represents the remaining size of the search space (product of cardinalities of `V1` and `V2`). As we see, whereas the first solution computed by `lab` leads to compute the "rest of the $\mathcal{TOY}(\mathcal{FD})$ goal" from a 12 candidates search space, the third solution leads to a 6 candidates one.

The primitive `labW` explores exhaustively the search tree in breadth, storing in a data structure `DS` each feasible node leading to a solution. Once the tree has been completely explored, the

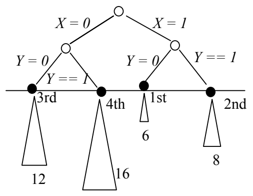

Figure 2: Search Tree

solutions are obtained (one by one) by using a criteria to select and remove the *best* node from `DS`. In the example, the selected criteria `smallestSearchTree` selects first the node with smaller product of cardinalities of `V1` and `V2` (returning first the solution of the 6 candidates). The order in which the `labW` strategy of the goal delivers the solutions is presented in Fig. 2.

Coming back to the definition of `labW`, the first parameter represents the variable selection criteria (no value selection is necessary, as the search would be exhaustive for all the values of the selected variables). The second parameter represents the *best* node selection criteria. To express it we have defined in $\mathcal{TOY}$ the enumerated datatype `ord`, ranging from the smallest/largest remaining search space of the product cardinalities of the labeling/solver-scope variables. Again, a last case (`userBound`) allows to specify the bound criteria at $\mathcal{TOY}(\mathcal{FD})$ level. The third parameter sets the breadth level of exhaustive exploration of the tree (represented as a horizontal black line in Fig. 2). Finally, as usual, the last parameter is the set of variables to be labeled.

Next $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom) presents a second example, with a bound criteria specified in the user function `myBound`. The *best* node procedure selection traverses all the obtained nodes in `DS`, selecting first the one with minimal bound value. In this context, the user criteria specified in `myBound` assigns to each node (minus) the number of its singleton value search variables. Once again, the function `myBound` also relies on auxiliary, prelude and reflection functions. The first two obtained solutions are `(X -> 1, Y -> 1, A -> 0, B -> 0, C -> 0)` and `(X -> 2, Y -> 1, A in 0..1, B -> 0, C -> 0)`, resp.

```
isBound:: [[int]] -> bool
isBound [[A,A]] = true
isBound [[A,B]] = false <== B /= A
isBound [[A,B] | RL] = false <== length RL > 0
%
myBound:: [int] -> int
myBound V = - (length (filter isBound (map get_dom V)))
----------------------------------------------------------------------
TOY(FD)> domain [X,Y] 1 2, domain [A,B,C] 0 5, A #< X, B #< Y, C #< Y,
         labW unassignedLeftVar userBound 2 [X,Y,A,B,C]
```

In summary, `labW` represents a novel concept not available in the predefined search strategies of any CP solver. However, it must be used carefully, as exploring the tree very deeply can lead to a explosion of feasible nodes, producing memory problems for `DS` and becoming very inefficient (due to the time spent on exploring the tree and selecting the *best* node).

Primitive labO

`labO :: optType -> varOrd -> valOrd -> int -> [int] -> bool`

This primitive performs a standard optimization labeling. The first parameter `optType` contains the optimization type (minimization/maximization) and the variable to be optimized. The other four parameters are the same as in the `lab` primitive.

2.2 Fragmentize Primitives

```
frag  :: domFrag -> varOrd -> intervalOrd -> int -> [int] -> bool
fragB :: domFrag -> varOrd -> intervalOrd -> int -> [int] -> bool
fragW :: domFrag -> varOrd -> bound -> int -> [int] -> bool
fragO :: domFrag->optType->varOrd->intervalOrd->int->[int]->bool
```

These four new primitives are mate to the `lab⋆` ones, but each variable is not labeled (bound) to a value, but *fragmented* (pruned) to a subset of the values of their domain. Let us consider an introductory example to motivate the usefulness of these new primitives. We suppose that: (i) A goal contains `V` variables and `C` constraints, with `V′` ≡ {`V1`, `V2`, `V3`} a subset of `V`, (ii) The constraint `domain V′ 1 9` belongs to `C`, (iii) No constraint of `C` relates the variables of `V′` by themselves, but some constraints relate `V′` with the rest of variables of `V`.

The left and right hand sides of Fig. 3 present the search tree exploration achieved by `frag⋆` and `lab⋆` search primitives, resp. In the case of `frag⋆`, the three variables of `V′` have been fragmented into the intervals (`1`,...,`3`), (`4`,...,`6`) and (`7`,...,`9`), leading to exponentially less leaf nodes (27) that the `lab⋆` exploration (729). On the one hand, if it is known that there is only one solution to the problem, the probabilities of finding the right combination of `V′` values is thus bigger in `frag⋆` than in `lab⋆`. On the other hand, the remaining search space of the leaf nodes of `lab⋆` are expected to be exponentially smaller than the ones of `frag⋆`, due to the more propagation in `V′` (also expecting to lead to more pruning in the rest of `V` variables). Thus, the `frag⋆` search strategies can be seen as a more conservative technique, where there are less expectations of highly reducing the search space, as variables are not bound, but there is more probability of choosing a subset containing values that lead to solutions (in what can be seen as a sort of generalization of the *first-fail* principle [MS98]).

Coming back to the definition of each `frag⋆` primitive, two differences arise w.r.t. its mate `lab⋆` primitive: (1) It contains as an extra basic component (first argument) the datatype `domFrag`, which specifies the way the selected variable is fragmented. The user can choose between `partition n` and `intervals`. The former fragments the domain values of the variable into `n` subsets of the same cardinality. The latter looks for already existing intervals on the domain of the variables, splitting the domain on them. For example, let us suppose that a goal computes `domain [X] 0 16, X /= 9, X /= 12`. Whereas applying

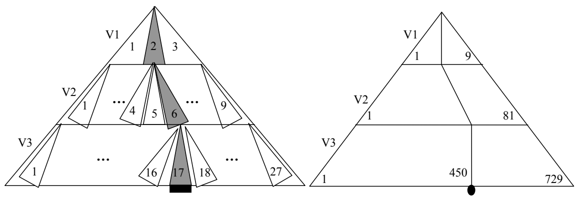

Figure 3: `frag` vs `lab` Search Tree

`partition 3` to `X` fragments the domain in the subsets `S1` $\equiv \{0\ldots4\}$, `S2` $\equiv \{5\ldots8\} \cup \{10\}$ and `S3` $\equiv \{11\} \cup \{13\ldots16\}$, applying `intervals` fragments the domain in the subsets `S1'` $\equiv \{0\ldots8\}$, `S2'` $\equiv \{10\ldots11\}$ and `S3'` $\equiv \{13\ldots16\}$. (2) An enumerated datatype `intervalOrd` (replacing the `lab*` argument `valOrd`), to specify the order in which the different intervals should be tried: First left, right, middle or random interval.

In summary, we claim the `frag*` primitives to be an interesting tool, to be taken into account in the context of search strategies as an alternative or a complement to the use of exhaustive labelings. Also, its use in $\mathcal{TOY}(\mathcal{FD})$ represents a novel concept not available in the predefined library of any CP solver.

2.3 Applying Different Search Scenarios

$\mathcal{TOY}(\mathcal{FD})$ supports non-deterministic functions, with possibly several reductions for given, even ground, arguments. The rules are applied following their textual order, and both failure and user request for a new solution trigger backtracking to the next unexplored rule. In this setting, different search strategies can be sequentially applied for solving a CP problem. For example, after posting `V` and `C` to the solver, the $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom) presented below uses the non-deterministic function `f` to specify three different scenarios for the solving of the CP problem described in Section 3.5. Each scenario ends with an exhaustive labeling of the set of variables `V`. However, the search space s this exhaustive labeling has to explore can be highly reduced by the previous evaluation of `f`.

```
f:: [int] -> bool
f [V1,V2,V3] = true <==
  fragB (partition 4) unassignedLeftVar random 0 [V1],
  labB unassignedLeftVar smallestVal 0 [V2,V3]
f [V1,V2,V3] = true <==
  fragW (partition 4) unassignedLeftVar smallestTree 0 [V1],
  labB unassignedLeftVar smallestTotalVars 0 [V2,V3]
f [V1,V2,V3] = true
-------------------------------------------
TOY(FD)> Post of (V,C), f V', lab userVar userVal 0 V
```

Scenario 1: The first rule of `f` performs the search heuristic h_1 over `V'` $\equiv$ {`V1`,`V2`,`V3`}. (1) h_1 fragments the domain of `V1` into 4 subsets, selecting one randomly. If propagation succeeds, (2) then h_1 bounds `V2` and `V3` to their smallest value. If propagation succeeds (with a remaining search space s_1), (3) then h_1 succeeds, and the exhaustive labeling explores s_1. If propagation fails in (1) or (2), or the exhaustive labeling does not find any solution in s_1, then h_1 completely fails (as so the first rule of `f`), as both the `labB` and `fragB` primitives just explore one branch.

Scenario 2: The second rule of `f` is tried, performing the heuristic h_2 over `V'`. Here a `fragW` primitive is first applied. So, if further either `labB` of h_2 or the exhaustive `lab` (acting over s_2) fails, backtracking is done over `fragW`, providing the next *best* interval of `V1` (according to the smallest search tree criteria, as in Fig. 2). If, after trying all the intervals a solution is not found, then h_2 completely fails (as so the second rule of `f`).

Scenario 3: If both h_1 and h_2 fail, the third rule of `f` trivially succeeds, and the exhaustive labeling is performed over the original search space obtained after posting `V` and `C` to the solver.

3 Search Primitives Implementation

The integration of the eight new search primitives into $\mathcal{TOY}(\mathcal{FD})$ is based on the scheme presented in [CS12], which supported the coexistence in the goal computations of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ of multiple `labeling` primitives (using the predefined search strategies provided in Gecode and ILOG) interleaved with the posting of constraints. To achieve the integration, the scheme: (1) Uses the Prolog-C++ connection provided by SICStus, to gain access from the Prolog predicate which manages the `labeling` primitive to the C++ function which implements the search (by accessing to the API of Gecode and ILOG). (2) Relies on a vector of auxiliary search managers $\mathtt{ss}_1 \ \ldots \ \mathtt{ss}_l$ to perform the search of the labelings $\mathtt{l}_1 \ \ldots \ \mathtt{l}_l$ arisen along the goal computation. This includes synchronizing the constraint store of the main solver with $\mathtt{ss}_i$ before performing $\mathtt{l}_i$ for first time, and vice versa each time $\mathtt{ss}_i$ finds a solution.

In this work we reuse this scheme, but, instead of relying on the predefined search strategies of Gecode and ILOG, we use their underlying search mechanisms to model new search strategies implementing the intended behavior of the primitives. As the implementation of a new search strategy is different in Gecode and ILOG, we present first an abstract specification of the requirements the new search strategies must fulfill, and then we present separately the adaptation of this specification to the technological framework provided by each library. The current versions of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ are available at: http://gpd.sip.ucm.es/ncasti/TOY(FD).zip

3.1 Abstract Specification of the Search Strategy

We specify a single entry point (C++ function) for the different primitives. Its proposed algorithm is parameterizable by the primitive type and its basic components. It fulfills the following requirements: (1) The algorithm explores the tree by iteratively selecting a variable `var` and a value `v`, creating two options: (a) Post `var == v`. (b) Post `var /= v` to continue the exploration taking advantage of the previously explored branch, recursively selecting another value to perform again (a) and (b). (2) For `frag*` strategies it selects an interval `i` instead of a value, posting in (a) both `var #>= i.min` and `var #<= i.max`. However, the (b) branch can not take advantage by posting `var #< i.min` and `var #> i.max`, as the constraint store would become inconsistent. Thus, (b) just removes `i` from the set of intervals, and continue the search by selecting a new interval. (3) For `labB` and `fragB` strategies only the (a) option is tried. (4) For `labO` and `fragO` strategies branch and bound techniques are used to optimize the search. (5) Specific functions are devoted to: (i) Variable and (ii) value/interval selection strategies, as well as to (iii) the bound associated to a particular solution found by `labW` and `fragW`. Those functions include the possibility of accessing Prolog, to follow the criteria the user has specified at $\mathcal{TOY}(\mathcal{FD})$ level (using $\mathcal{TOY}(\mathcal{FD})$ functions compiled to mate Prolog predicates). (6) The primitives `labW` and `fragW` perform the breadth exploration of the upper levels of the search tree, storing all the satisfiable leaf nodes to further give them (one by one) on demand. Thus, `ss` contains: (i) An entity performing the search, (ii) A vector `DS` (cf. Section 2.3) containing the solutions. The notion of solution is abstracted as the necessary information to perform the synchronization from `ss` to the main constraint solver. (iii) A status indicating whether the exploration has finished or not. (7) The algorithm finishes (successfully) as it founds a solution, except for `labW` and `fragW` strategies, where it stores the solution node and triggers an explicit

Search Concept	Gecode	ILOG Solver
Search trigger	`Search Engine`	`IloGoal` stack
Tree node	`Space`	`IloGoal` attributes
Node exploration	`Brancher Commit`	`IloGoal` execution
Child generation	`Brancher Choice`	`IloGoal` constructor
Solution check	`Brancher Status`	Stack with no pending `IloAnd`
Solution abstraction	`Space`	Tree path (var,value) vector register

Table 1: Different Search Concept Abstractions in Gecode and ILOG Solver

failure, continuing the breadth exploration of the tree. (8) A counter is used to control that only the specified amount of variables of the variable set is labeled/pruned.

Next two sections adapt this specification to Gecode and ILOG Solver, resp. Table 1 summarizes the different notions provided by both libraries.

3.2 Gecode

We have selected Gecode 3.7.3 as the external solver for $\mathcal{TOY}(\mathcal{FD}g)$ as it is a free software constraint solver with state-of-the-art performance. Search strategies in Gecode are specified via `Branchers`, which are applied to the constraint solver (`Space`) to define the shape of the search tree to be explored. The `Space` is then passed to a `Search Engine`, whose execution method looks for a solution by performing a depth-first search exploration of the tree. This exploration is based on cloning `Spaces` (two `Spaces` are said to be equivalent if they contain equivalent stores) and hybrid recomputation techniques to optimize the backtracking. As `Spaces` constitute the nodes of the search tree, a solution found by the `Search Engine` is a new `Space`. The library allows to create a new class of `Brancher` by defining three class methods: (1) `status`, which specifies if the current node is a solution, or their children must be generated, to continue with their exploration. (2) `choice`, which generates an object `o` containing the number of children the node has, as well as all the necessary information to perform their exploration. (3) `commit`, which receives `o` and the concrete children identifier to perform its exploration (generating a new `Space` to be placed at that node).

Adaptation to the Specification. The search strategies are implemented via two layers: (I) A new class of `Brancher MyGenerate`, which carries out the tree exploration by the combination of the `status`, `choice` and `commit` methods. As each node of the tree is a `Space`, the methods are applied to it. (II) A `Search Engine`, controlling the search by receiving the initial `Space` and making the necessary clones to traverse the tree. Regarding (1), the `choice` method deals with the selection of: (i) The variable `var` and (ii) The value `v`, creating an object `o` with them as parameters, as well as the notion of having two children. The variable selection must rely on an external register `r`, being controlled by the `Search Engine` and thus independent on the concrete node (`Space`) the `choice` method is working with. The register is necessary to ensure that, whether a father generates its right hand child by posting `var /= v`, this child will reuse `r` to select again `var` (as a difference to the left hand child, which removes the `r` content to select a new variable). Regarding (2), for `frag*` strategies, instead of passing `val` to `o`, the `choice` method generates a vector with all the different intervals to be tried, and the size of this vector is passed as its number of children. Regarding (3), for `labB` and `fragB` only one child is considered. Regading (4), for `labO` and `fragO` we use a specialized branch and bound `Search Engine` provided by Gecode. Regarding (6), the search entity is the `Search Engine` and the notion of solution is a `Space`. Regarding (7), for `labW` and `fragW` the `Search Engine` uses a loop, requesting solutions one by one until no more are found (the breadth exploration of the search tree has finished). Regarding (8), only the left hand child of `lab*` strategies increments the counter value, and the `status` method checks the counter to stop the search at the precise moment.

3.3 ILOG Solver

We have selected IBM ILOG Solver 6.8 as the external solver for $\mathcal{TOY}(\mathcal{FD}i)$ as it is an industrial market leader for solving generic $\mathcal{FD}$ problems. It belongs to the ILOG CP 1.6 package, which contains the ILOG Concert 12.2 modeling library and two other solver libraries for specific scheduling and routing problems. Thanks to the IBM academic initiative these products are free for academic purposes. Search strategies in ILOG Solver are performed via the execution of `IloGoals`. An `IloGoal` is a daemon characterized by its constructor and its execution method. The constructor creates the goal, initializing its attributes. The execution method triggers the algorithm to be processed by the constraint solver (`IloSolver`), and can include more calls to goal constructors, making the algorithm processed by `IloSolver` to be the consequence of executing several `IloGoals`. We say that an `IloGoal` fails if `IloSolver` becomes inconsistent by running its execution method; otherwise the goal succeeds. The library allows to create a new class of `IloGoal` by defining its constructor and execution method. Four basic goal classes are provided for developing new goals with complex functionality. Goals `IlcGoalTrue` and `IlcGoalFalse` make the current goal succeed and fail, resp. Goals `IlcAnd` and `IlcOr`, both taking two subgoals as arguments, make the current goal succeed depending on the behavior of its subgoals. While `IlcAnd` succeeds only if its two subgoals succeed, `IlcOr` creates a restorable choice point which executes its first subgoal, restores the solver state at the choice point on demand, and executes its second subgoal.

Adaptation to the Specification. The search strategies are implemented via the new `IloGoal` classes `MyGenerate` and `MyInstantiate`. Whereas the former deals with the selection of a variable, the latter deals with its binding/prunning to a value/interval. Regarding (1), the control of the tree exploration is carried out by `MyGenerate`, which selects a variable and uses the recursive call `IlcAnd(MyInstantiate, MyGenerate)` to bind it and further continue processing a new variable. In `MyInstantiate`, the alternatives (a) and (b) are implemented with an `IlcOr(var == val, IlcAnd(var /= var, MyInstantiate))`. Regarding (2), it dynamically generates a vector with the available intervals on each different `MyGenerate` call. Regarding (3), only the goal `var == val` is tried. Regarding (4), we explicitly implement the branch and bound. Thus, before selecting each new variable, we check if the current optimization variable can improve the bound previously obtained; otherwise an `IloGoalFail` is used to trigger backtracking (as well as if, after labeling the required variables, the obtained solution does not bind the optimization variable). Regarding (6), the entity performing the search is an `IloSolver`. Also, the notion of solution is given by: (i) A vector of integers, representing the indexes of the labeled/pruned variables. (ii) A vector of pairs, representing the assigned value or bounds of these variables. This explicit solution entity is built towards the recursive calls of `MyGenerate`, which adds on each call the index of the variable being labeled. Once found the solution, it stores (i) and (ii) in `DS`. Regarding (7), after storing a solution in `labW` or `fragW` an `IloGoalFalse` is used, triggering backtracking to continue the breadth exploration. Regarding (8) each call to `MyGenerate` increments the counter value.

3.4 TOY(FD) Architecture

The resulting $\mathcal{TOY}(\mathcal{FD})$ architecture supporting the search primitives is presented in Fig. 4. It contains five different layers:

(1) **The $\mathcal{TOY}(\mathcal{FD})$ interpreter**. It allows to impose the user commands. In Fig. 4 the goal proposed in Section 3.1 is to be solved. Besides its $\mathcal{FD}$ constraints `domain` and `/=`, there is a `lab` strategy. The user is specifying its own variable and value selection criteria by

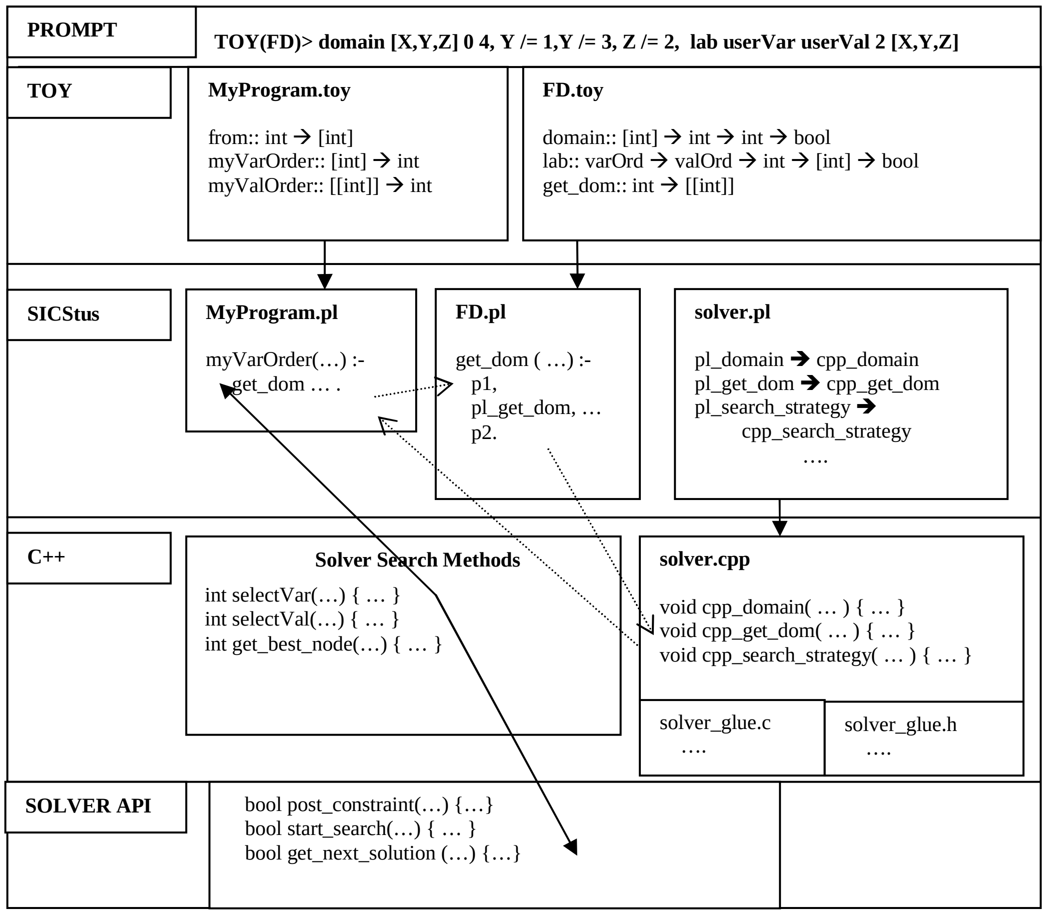

Figure 4: $\mathcal{TOY}(\mathcal{FD}i)$ Architecture

using the functions `myValOrder` and `myVarOrder`, resp., which rely on auxiliary, prelude and reflection functions (as, for example, `from`, `get_dom` and `map`, respectively).

(2) **The $\mathcal{TOY}(\mathcal{FD})$ files defining the language**. They include: (i) A file `Prelude.toy`, to specify the prelude functions (as `map`), (ii) A devoted file `FD.toy`, specifying the set of $\mathcal{FD}$ constraints supported, and (iii) A file `MyProgram.toy` with the user definitions (including `from` and `myVarOrder`).

(3) **The SICStus implementation of $\mathcal{TOY}(\mathcal{FD})$**. It includes Prolog mate files for (i), (ii) and (iii), implementing all the $\mathcal{TOY}(\mathcal{FD})$ datatypes, functions and operators supported by the system and defined by the user. The file `solver.pl` supports the communication from SICStus to C++ by specifying the set of SICStus predicates `S` being implemented in C++ functions.

(4) **The C++ interface to the solver**. It includes: (a) The file `solver.cpp`, containing the set of C++ functions implementing `S`, and (b) The auxiliary files containing the extra C++ functions, data structures and new solver specific classes extending the library (needed to implement `S`). This includes the new C++ class `MyGenerate` in Gecode and ILOG Solver (the latter also including `MyInstantiate`). They are devoted to implement the `lab` strategy, and contain methods for the variable and value selection. Fig. 4 shows how these methods can access either to the solver API (if a predefined criteria is being selected), or (focusing on the variable selection criteria) come back to the SICStus file `MyProgram.pl`, executing the SICStus predicate `myVarOrder` (implementing the user $\mathcal{TOY}(\mathcal{FD})$ function `myVarOrder`). In our example, the latter case holds, and we can see how the execution traverses the SICStus and C++ layers, as a cycle is performed between the SICStus predicate `myVarOrder`, its auxiliary SICStus predicate `get_dom` (which belong to `S`) and its C++ implementation in `solver.cpp`.

(5) **The C++ API of the solver**. Accessed by the C++ methods interfacing the solver. In the case of Gecode, this layer also includes the proper solver implementation, as it is open software.

4 Performance Analysis

In this section we present a preliminary although encouraging performance analysis of the $\mathcal{TOY}(\mathcal{FD})$ search primitives, devoting a specific case study for each novel concept already presented. On each case we select a constraint satisfaction/optimization problem (either a pure classical CP($\mathcal{FD}$) benchmark from the CSPLib [CSP12] or a real-life problem) and we describe how the use of a concrete search strategy increases the solving efficiency of the problem.

labB: *n-magic_sequence.* When $n \geq 9$ the sequence follows the pattern: $L \equiv [(n-4), 2, 1, 0, 0, \ldots, 1, 0, 0, 0]$. Although the use of first-fail (`labeling` [ff] L) turns the solving of any n-sequence into $\simeq 0ms$, the initial search space this search departs from can be huge. For example, $n = 9$ contains an initial search space of $77,760$ candidates. In this context, for each $n \geq 9$, applying before `labB unassignedRightVar smallestVal 3 L, labB unassignedRightVar largestVal 1 L` *captures* the last four variable $1,0,0,0$ pattern, whose propagation lead to $L \equiv [(n-4), A, B, C, 0, \ldots, 1, 0, 0, 0]$ (with A in 1..3, B in 0..1 and C in 0..1), dramatically reducing the search space the `labeling` has to deal with. However, we are interested in examples in which search space reduction (because of the application of our search strategies) also implies a better solving efficiency for the problem.

Employee Timetabling Problem. The real-life problem [CS13] optimizes the timetabling of a department. Relying on the *seed* approach presented in [R. 07] for solving a former nonoptimization version of the problem, we use now the `labB` strategy to find *an optimal seed*, i.e., a variable-subset-binding (a different one for each of the independent subproblems being solved) which matches the one of the optimal timetabling solution. For example, for the 21 timetabling instance of the [CS13] $\mathcal{TOY}(\mathcal{FD})$ model, applying `labB unassignedLeftVar smallestVal 2 (extract TCal)` before performing the `labeling` of each subproblem: (1) Finds an optimal seed, (2) Reduces the solving time of the problem to a 94% in Gecode.

labW: *langford's number problem.* A deep breath exploration of the search tree with `labW` supposes a tradeoff between: (1) Obtaining an ordered hierarchy of interesting intermediate tree-level nodes and (2) The computational effort to obtain such this hierarchy. In this context, applying `labW smallestMinValueVar smallestSearchTree 3 L` to `langFord (2,4)` directly finds a solution, i.e., not even a further `labeling` is necessary. However, the time `labW` spends is bigger than the one of running `labeling` to the whole search tree, so the use of `labW` does not pay-off. Fortunately it does when applying `labW largestMinRegretVar smallestSearchTree 2 L` to `langFord (3,19)`, where the use of `labW` does not find directly a solution, but the sum of time for obtaining the hierarchy and applying the `labeling` to the remaining space takes a 65% of time less in Gecode than applying straight the `labeling`.

fragB: *n-queens.* The formulation of the n-queens problem based on global ($all\_different$) constraints becomes much more efficient than the one using single disequality constraints, with some n-queens instances for which the former finds a solution in a few seconds whereas the latter can not find anyone after hours. The right hand side of Fig. 1 (cf. Section 2.1) presents an intuitive way for reducing the initial search space of the problem: (1) Split the n variables into k variable sets $(vs_1, vs_2, \ldots, vs_k)$ (where consecutive variables are placed in different variable sets), (2) Split the initial domain $1 \ldots n$ into k different intervals $(1..(n/k), \ldots, (n/k)*(i-1)+1..(n/k)*i, \ldots, (n/k)*(k-1)+1..n)$, (3) Assign the variables of vs_i to the i^{th} interval. The application of `split_into_3 L ([],[],[]) == (K1,K2,K3), fragB (partition 3) unassignedLeftVar firstRight 0 K1, fragB (partition 3) unassignedLeftVar firstMiddle 0 K2, fragB (partition 3) unassignedLeftVar firstLeft 0 K3` implements the approach with $k = 3$ sets, solving the 75-queens instance in just one second in Gecode (whereas it is not solved after twelve hours without using the strategy).

fragW: *n-Golomb rulers.* The classical formulation of the problem leads to a huge initial search space. The initial domain of the last three rulers in 11-Golomb is H in 36..1020, I in 45..1021 and J in 55..1023 (with know optimal solution 64, 70 and 72, resp.) whereas the one of the first three rulers is 0, A in 1..977 and B in 3..987 (with know optimal solution 0, 1 and 4, resp.) In this context, an intuitive way of reducing the initial search space is by reducing so much the upper bound of these variables. Applying `fragW (partition 3) unassignedRightVar smallestSearchTree 3 L, fragW (partition 15) unassignedLeftVar largestSearchTree 2 L` fragments first the last three variables and then the first three. Note that, whereas the former selects as *best* intermediate node the one minimizing the remaining search space, the latter select the one maximizing it (which intuitively makes sense, as the smaller interval is the one pruning the less the upper bound of the first three variables). The use of these strategies reduces the solving time of the problem to a 88% in Gecode.

Results: The obtained results are summarized in Table 2. The first column represents the problem being solved. Next two blocks of three columns represent respectively the results of Gecode and ILOG: Elapsed time (measured in milliseconds) without/with using the strategy and

Problem	G	G*	G*/G	I	I*	I*/I	G/I	G*/I*
ETP	24,710	1,465	0.06	54,351	4,570	0.08	0.45	0.32
Langford	624	218	0.35	1,014	827	0.82	0.62	0.26
Golomb	42,869	5,320	0.12	75,365	9,687	0.13	0.57	0.55
Queens	-	1,622	$\simeq 0.00$	-	4,306	$\simeq 0.00$	$\simeq 0.00$	$\simeq 0.00$

Table 2: Case Studies of $\mathcal{TOY}(\mathcal{FD})$ Search Strategies Application

slow-down of the latter w.r.t. the former, resp. Finally, columns 8 and 9 represent the slow-down of Gecode w.r.t. ILOG Solver without/with using the search strategy, resp.

The results show that the use of the search strategies improve the solving performance of the case studies both in Gecode and ILOG Solver (making them from 1.25 to 20 times faster, or even solving instances that were not possible before). However, the impact of the search strategies is not the same in both systems, i.e., Gecode is faster than ILOG Solver both with/without the search strategies, but the ratio is bigger when using the search strategies. As both systems are running exactly the same $\mathcal{TOY}(\mathcal{FD})$ model, we claim that the approach Gecode offers to extend the library with new search strategies is more efficient than the ILOG Solver one.

Benchmarks are run in a machine with an Intel Dual Core 2.4Ghz processor and 4GB RAM memory. The OS used is Windows 7 SP1. The SICStus Prolog version used is 3.12.8. Microsoft Visual Studio 2008 tools are used for compiling and linking the $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}g)$ C++ code. The different models being used as case studies are available at: http://gpd.sip.ucm.es/ncasti/models.zip.

5 Conclusions and Future Work

We have described the integration of new parametric search primitives in the systems $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$. Our approach benefits both from the high expressivity of $\mathcal{TOY}(\mathcal{FD})$ and of the high efficiency of Gecode and ILOG Solver, and can be easily adapted to other CLP or CFLP systems implemented in Prolog and interfacing external CP($\mathcal{FD}$) solvers with a C++ API.

We have described the primitives, pointing out novel concepts they include, as perform an exhaustive breadth exploration of the search tree further sorting the satisfiable solutions by an specified criteria, fragment the variables pruning each one to a subset of its domain values instead of binding it to a single value, and applying the labeling or fragment strategy only to a subset of the variables involved. We have seen how expressive, easy and flexible it is to specify some search criteria at $\mathcal{TOY}(\mathcal{FD})$ level, as well as how easy is to combine some search strategies to set different search scenarios. We have described an abstract view of the eight requirements needed to integrate the search strategies in $\mathcal{TOY}(\mathcal{FD})$. We have presented the implementation in Gecode and ILOG Solver, by matching each abstract concept to the concrete one provided in the library. We have seen the resulting architecture of the system, pointing out its five layers and the interaction between them. We have presented five case studies (using classical CP($\mathcal{FD}$) benchmarks and a real-life problem) to point out that the use of the search strategies improve the $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ solving performance, and that the approach Gecode offers to extend the library with new search strategies is more efficient than the ILOG Solver one.

As future work, we will use scripting for applying the search strategies to classical CP($\mathscr{FD}$) benchmarks under multiple and very precisely controlled scenarios. We will use data mining techniques over the obtained results, to find out some patterns about the relation between the structure of a problem and the concrete search strategies to be applied.

Bibliography

[CS12] I. Castiñeiras, F. Sáenz-Pérez. Improving the Performance of FD Constraint Solving in a CFLP System. In *FLOPS'12, 88–103*. LNCS 7294. Springer, 2012.

[CS13] I. Castiñeiras, F. Sáenz-Pérez. Applying CP(FD), CLP(FD) and CFLP(FD) to a Real-Life Employee Timetabling Problem. *Procedia Computer Science* 18(0):531 – 540, 2013.

[CSP12] CSPLib. A Problem Library for Constraints. 2012. http://www.csplib.org/.

[FHSV07] A. J. Fernández, T. Hortalá-González, F. Sáenz-Pérez, R. del Vado-Vírseda. Constraint Functional Logic Programming over Finite Domains. In *TPLP 7, 5, 537–582*. Cambridge University Press, 2007.

[Han07] M. Hanus. Multi-Paradigm Declarative Languages. In *ICLP'07, 45–75*. LNCS 4670. Springer, 2007.

[IBM10] IBM ILOG CP 1.6. 2010. http://www-947.ibm.com/support/entry/portal/Overview/Software/WebSphere/IBM_ILOG_CP.

[JM94] J. Jaffar, M. Maher. Constraint Logic Programming: A Survey. In *The Journal of Logic Programming, 19–20. 503–581*. Elsevier, 1994.

[LLR93] R. Loogen, F. López-Fraguas, M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *PLILP'93, 184–200*. LNCS 714, 1993.

[Mat12] Mats Carlsson et. al. SICStus Prolog User's Manual. 2012. http://www.sics.se/.

[MS98] K. Marriott, P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[PJ02] S. Peyton-Jones. Haskell 98 Language and Libraries: the Revised Report. 2002. http://www.haskell.org/onlinereport/.

[R. 07] R. González-del-Campo and F. Sáenz-Pérez. Programmed Search in a Timetabling Problem over Finite Domains. In *ENTCS, 177, 253–267*. Elsevier, 2007.

[STL12] C. Schulte, G. Tack, M. Z. Lagerkvist. Gecode 3.7.3: Generic Constraint Development Environment. 2012. http://www.gecode.org/.

[STL13] C. Schulte, G. Tack, M. Z. Lagerkvist. Modeling and Programming with Gecode. 2013. http://www.gecode.org/doc-latest/MPG.pdf.

[Tsa93] E. Tsang. *Foundations of constraint satisfaction*. Academic Press, 1993.