A Metalevel Compiler of CLP(FD) and its Combination with Intelligent Backtracking

Philippe Codognet [1] [2]
François Fages [3] [2]
Thierry Sola [2]

Abstract

We propose an implementation of constraint solving over finite domains, as pioneered by CHIP, on top of any Prolog system that provides a delay mechanism and backtrackable assignment. The aim is to propose a simple, portable, easily-maintenable, but yet efficient package. The performances of our system are of the same order of magnitude than a "wired" implementation of finite domains such as CHIP, even on quite large programs. We have also designed some basic low-level (WAM) extension to Prolog's delay mechanism to make it more suited for handling constraint solving, such as priority queues for scheduling woken goals, that allow to preferentially treat cheap primitive constraints and investigate various constraint solving heuristics.
Moreover, we propose an intelligent backtracking scheme for Finite Domain CLP Languages. Intelligent backtracking consists in determining, upon unsolvability of the constraint system, a pertinent choice-point which can "cure" the failure, and departs from the naive approach usually found in logic programming which always goes to the most recent choice-point, whether it is pertinent or not. As non-determinism is used in CLP to express disjunctive constraints (eg. disjunctive scheduling) or heuristic search techniques, our method can be used to improve the efficiency in such problems. We propose an implementation of this scheme by using the finite domains constraint solver on top of a Prolog compiler with intelligent backtracking.
Interestingly, our framework also leads to a new (and efficient) execution model for Hierarchical Constraint Languages, and can be the basis for general incremental constraint solving.

[1]INRIA, domaine de Rocquencourt, 78153 Le Chesnay, FRANCE
[2]LCR Thomson-CSF, domaine de Corbeville, 91404 Orsay, FRANCE
[3]LIENS CNRS, 45 rue d'Ulm, 75005 Paris, FRANCE

1 Introduction

Constraint Logic Programming (CLP) languages provide an attractive paradigm which combines the advantages of Logic Programming (declarative semantics, non-determinism, partial answer) with the efficiency of special-purpose constraint-solving algorithms over specific domains such as reals or rationals, finite domains or booleans. Several languages, such as Prolog-III [9], CHIP [12], CLP($\mathcal{R}$) [16], show the usefulness of this approach for real applications in various domains as combinatorial problems, scheduling, cutting-stock, circuit simulation, diagnosis, finance, etc. Interesting domains of computation proposed by the CHIP language are finite domains together with linear equations, inequations and disequations. Such finite domains allow a very efficient constraint solving, even for large combinatorial problems, and are useful for a variety of applications, and are even competitive on some applications most suited for rational or boolean domains.

Finite domain constraint solving typically considers a discrete domain D, such as the set of integers or an enumerated type, and requires that each variable used in the constraint system has an associated domain which is a finite subset of D. The usual constraints over finite domains are (linear) equations ($=$), inequations ($<, \leq$), disequations ($\neq$), list membership (e.g. `element(I,[x1,...,xn],V)` meaning $x_I = V$), cardinalities (e.g. `atmost(n,[X1,...,Xk],v)` meaning $card\{i|X_i\} \leq v$), but more complicated constraints can be considered, eg. the user-defined active constraints of CHIP. The usual constraint solving mechanism is the *propagation* of the constraints in order to mutually reduce the domains of the variables by *forward-checking* or *look-ahead* techniques as proposed in [15] and applied to CHIP in [21]. One can consider for instance the propagation of the min and max values of variables for equations and inequations or the removal of values for disequations, see [21] for a comprehensive treatment. Indeed, as proposed by [22], the treatment of monotonic constraints on totally ordered domains, eg. $<$ on integers, can be made even more efficient. Also observe that these propagation techniques are very close to the treatment of interval arithmetics over reals proposed in BNR-Prolog [3].
The purpose of our approach was to design a finite domain constraint solver on top of Prolog, aiming at proposing a simple, portable, easily-maintenable, but yet efficient package. Such a package could be used to experiment various and algorithms and heuristics for the constraint solving process while keeping with the declarativity of the Prolog languages (instead of going into a "wired" implementation in C). It should however remain efficient enough to give credible results, with a constant time slowdown w.r.t. high-speed systems such as CHIP, even for large programs.
Another interesting feature of this approach is that we could thus benefit for free of various extensions for Prolog systems, such as for instance an intelligent backtracking scheme. We thus have a CLP language with intelligent backtrack-

ing, which could be used to efficiently treat disjunctive constraints.

The key-point for efficiently implementing finite domains on top of Prolog is to use a *delay mechanism*, as pioneered by Prolog-II and now offered by some (most ?) other Prolog systems, to handle the constraints and let this coroutining mechanism take care of the constraints propagation. This idea indeed emerges from seminal work at SICS [19], and is also based on the "backtrackable assignment" facility of Sicstus Prolog [20]. [11] also describes a finite domain package for Prolog with a delay mechanism, which is however quite different from our implementation and uses a pre-processing phase. In our approach, each constraint which is still active (i.e. not solved and still present in the current constraint system) corresponds to a delayed predicate which will be woken up as soon as the domain of one variable of the constraint is modified. This predicate will then enforce its own domain modifications and propagations and will be delayed again if the constraint is not completely solved. We use the usual forward-checking and (partial) look-ahead techniques on finite domain to perform the constraint propagation, see [21].

However we have experienced that the delay mechanism of Prolog systems was not completely adequate to implement constraint systems, which is not very surprising as it was not intented to do so anyhow. Indeed we would have like more flexibility in the scheduling of woken goals, and we have thus designed an enhanced delay mechanism. The most important feature we needed was to allow some control over the scheduling and wakening of constraints, and to use *priorities*. Hence, for a set of constraint that should be woken up at a given time, the constraints with high priority will be treated first and those with low priority second. We could thus treat preferentially cheap primitive constraints, such as $\neq$ or binary $<$, and perform their pruning before managing more costly constraints such as linear equations. Another feature that we have introduced at a low-level (into the WAM) is the ability for a goal to be waiting on a disjunction of variables, corresponding for a constraint in being waiting for a domain modification of any of its variable.

We experiment our implementation with programs taken from [21], and compare it with the timings given for the implementation of CHIP. The finite domain implementation on top of Prolog is less than 7 times slower than the "wired" implementation, even for large program such as the disjunctive scheduling of the bridge construction.

Another interesting feature of our approach was to experiment the finite domain package with our Prolog compiler with intelligent backtracking [8] in order to investigate the use of intelligent backtracking (IB) in CLP languages. Indeed, the nondeterminism of the underlying Prolog language is often used in CLP languages to express for instance disjunctive constraints (eg. disjunctive

scheduling) or heuristic search techniques such as domain-splitting, see various examples in [21]. The CLP languages however rely, as Prolog, on naive or *blind* backtracking, and always backtrack to the most recent choice-point when a failure (inconsistency constraint system) is found. However this choice-point is not necessarily related to the current failure, and such backtracking can amount to useless computation work and lead to trashing behavior. Intelligent backtracking consist, upon failure of the computation, in analyzing the causes of the failure and determining a pertinent backtrack point that can "cure" the failure. It thus avoids useless computation work when compared to naive backtracking. Intelligent backtracking has been investigated in Logic Programming for more than one decade [10] [4], [17], [5], and some efficient methods have been developed [7] [8]. Indeed [8] describes an implementation inside a WAM-based Prolog compiler where the overhead of the IB machinery is limited to 20 %, leading to interesting speedups for non-deterministic programs. IB is related to dependency-directed backtracking of the TMS framework [13].
The backbone of our method is to record for every domain modification which constraint is responsible of it (whodunnit ...). This allows to easily determine upon unsolvability of the constraint system the (minimal) unsolvable subsystem, i.e. the subsets of constraints responsible of the failure. This gives a set of pertinent backtrack points that can cure the failure. When the constraint system derives from a proof-tree, as in CLP languages, it is however not sufficient to simply backtrack to the most recent *intelligent* backtrack point, and a special backtracking process is needed to retain the completeness of the method. This mechanism consists in managing sets of intelligent backtrack points attached to the nodes of the proof-tree, and is identical to its counterpart in intelligent backtracking methods for Logic Programming. We show how, by using the finite domain package on top of our Prolog compiler with intelligent backtracking, we can implement the proposed IB scheme.
An other application of our framework is to consider constraint hierarchies and Hierarchical Constraint Logic Programming (HCLP) languages [2]. HCLP is an extension that associates a *strength* level to each constraint and allows to relax some constraints (those *preferred* but not required) when their introduction leads to inconsistency. Our scheme allows to handle *all* constraints as they occurred, having thus a more constrained system and a better pruning of the search space, and, thanks to the intelligent backtracking information, to selectively relax some of them upon failure. This seems to be an adequate framework for a general treatment of incrementality in constraint solving.

This paper is organized as follows. Section 2 describes the basis of the implementation of finite domains on top of a Prolog system with a delay mechanism and backtrackable assignment, and Section 3 proposes some extensions for the implementation of the delay mechanism in the WAM in order to increase efficiency. The performances of our system are presented in Section 4. Section 5 describes an intelligent backtracking scheme for a finite domain CLP and

presents a simple implementation by using the previous finite domain programs on a Prolog compiler with intelligent backtracking.

2 Constraint Solving over Finite Domains on Top of Prolog

2.1 Constraints over finite domains

A constraint in Logic Programming is an n-ary predicate whose ground instances either succeed or finitely fail. Domain variables are declared with a domain of possible values. We shall consider Constraint Logic Programming over finite domains, CLP(FD), and for simplicity the only finite domains we shall consider are *finite sets of natural numbers*. In addition to arithmetic constraints, such as linear equalities, one can define also symbolic constraints, such as set membership [21], or higher order constraints.

The declarative semantics of an answer to a goal G and a program P is a conjunction of constraints C that entail G:

$$P \models_{FD} \forall(C \Rightarrow G)$$

In order to cope with incomplete constraint solvers we do not suppose that the answer constraints C are FD-satisfiable, this has to be checked independently. In this way constraints are used actively to reduce the search space before the instanciation of domain variables. To perform this behavior new inference rules have to be added to the standard resolution rule of Logic Programming:

$$RES\frac{:-A_1,...,A_i,...,A_m}{:-(A_1,...,A_{i-1},B_1,...,B_n,A_{i+1},...,A_m)\sigma}$$

where $B :- B_1,...,B_n$ is a rule whose head unifies with goal A_i and σ is the most general unifier.

This inference rule is well suited to implement "generate and test" procedures but does not use domain informations to simplify constraints and achieve a priori pruning of the search space. Following [21] two inference rules are thus added for constraint propagation: forward-checking and look-ahead.

2.1.1 Forward Checking

A constraint is forward checkable if it contains exactly one domain variable all the other arguments being ground. The forward checking inference rule consists in solving a constraint forward checkable on x^d (d denotes the domain of variable x) by substituting to x^d a new variable with the appropriate domain.

$$FC\frac{:-A_1,...,A_i,...,A_m}{:-(A_1,...,A_{i-1},A_{i+1},...,A_m)\sigma}$$

where A_i is forward checkable on x^d,
$e = \{a \in d | P \models A_i[x^d \leftarrow a]\}$ is non-empty,
$\sigma = \{x^d \leftarrow c\}$ if $e = \{c\}$, $\sigma = \{x^d \leftarrow Y^e\}$ otherwise.

This definition can be directly implemented for forward checking arbitrary constraint predicates over finite domains, but of course much more efficient implementations exist for specific constraints.

2.1.2 Look Ahead

A constraint is look-ahead checkable if it contains at least one domain variable. Depending on the number of domain variables remaining in the constraint after looking-ahead the constraint is solved and can be eliminated, or is used only for reducing the search space. This is formalized by two inference rules:

$$LA\frac{:-A_1,...,A_i,...,A_m}{:-(A_1,...,A_i,...,A_m)\sigma} \qquad LA'\frac{:-A_1,...,A_i,...,A_m}{:-(A_1,...,A_{i-1},A_{i+1}...,A_m)\sigma}$$

where A_i is look-ahead checkable,
$e_j = \{a_j \in d_j | \exists a_k \in d_k, k \neq j, P \models A_i[x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]\}$
$\sigma = \{x_1 \leftarrow v_1, ..., x_n \leftarrow v_n\}$ where $v_j = c$ if $e_j = \{c\}$, $v_j = y^{e_j}$ otherwise.

LA applies if $A_i\sigma$ contains at least two domain variables, otherwise LA' applies and as $A_i\sigma$ contains at most one domain variable the goal A_i is solved.

In practice it is often more cost effective to compute an approximation of the exact domain of each variable in a constraint used in look-ahead. The *partial look ahead* inference rule (PLA) differs from LA by the domains used in the substitution:

$\sigma = \{x_1 \leftarrow v_1, ..., x_n \leftarrow v_n\}$ where $v_j = c$ if $e_j = \{c\}$, otherwise $v_j = y^{e'_j}$ with $e_j \subseteq e'_j \subseteq d_j$.

Partial look ahead is employed for example for solving linear equalities by reasoning only on the minimum and maximum values of each domain variable.

2.2 Efficient Constraint Propagation on Top of Prolog

2.2.1 Representation of Domain Variables

Domain variables can be represented by Prolog variables with their domain attached to them as an attribute. The attributes of a variable can be represented by frozen goals on that variable, and can be accessed by the standard `freeze`, `frozen` predicates. When a variable is declared with a domain, a goal recording information about its domain is frozen. Then some extra information can be added by constraints in which the variable occurs. When a domain variable gets instanciated the goal recording its domain is woken up to check membership of

the value to the domain. The general representation of domain variables ranging over finite sets of natural numbers is the following[4]:

```
freeze(X,domain-check(X,Inst,nat(min,max,Vmin,Vmax,BV)))
```

Variable `Inst` is used to wake the constraints in forward checking, `Inst` is instanciated as soon as a value is assigned to X. Components `min, max` indicate the extremum values of the domain of X, these values can be changed by using the backtrackable assignment instruction `setarg`. Variables `Vmin` and `Vmax` are used to implement look ahead propagation, they get instanciated as soon as the minimum, resp. the maximum value, of the domain of X changes.

`BV` provides optional extra informations used by some constraints. When it is instanciated `BV` is an explicit representation of the domain of X by a boolean vector, used for example for forward checking disequalities. `BV` contains also a variable `Vmid` used to wake-up some constraints in look-ahead (e.g. $a * X = b * Y + c$ or `element`) as soon as a value is deleted from the middle of the domain of a variable.

```
BV=bv(Bias,BooleanVector,Vmid)
```

2.2.2 Representation of Constraints

Constraints are represented by Prolog goals on which standard delay mechanisms can apply. For example a linear equation, $\sum a_i X_i = \sum b_j Y j$, is represented by a goal `eqln(Lx,Ly)` where the couple `(Lx,Ly)` is some *canonical form* of the equation.

A constraint used in forward checking is represented by a goal frozen on the variable `Inst` of each domain variable occurring in the constraint. In this way the constraint is woken-up at each instanciation of one of its domain variables to check whether the constraint is forward checkable, and then apply the inference rule FC.

A constraint used in look ahead, or partial look ahead, is represented by a goal frozen on the disjunction of variables associated to the different kinds of domain modification that may occur to each domain variable. In the case of natural numbers the variables `Vmin, Vmax` indicate a modification of the extremum values of a domain variable, and when it is present `Vmid` indicates the deletion of a value in the middle of the domain. Note that the freezing of constraints on a disjunction of variables can be implemented on top of Prolog by using the freeze predicate in two steps:

[4] In Prolog systems in which `freeze` and `frozen` are costly predicates, declarativity may be sacrified to efficiency by representing the domain by a term bound to the variable instead of a frozen goal.

```
freeze_or(L,G) :-
         freeze(V,G),
         freez_or(L,V).

freez_or([],_).
freez_or([X|L],V) :-
         freeze(X,V=go),
         freez_or(L,V).
```

A wired implementation of this metapredicate in our Prolog compiler is responsible for a 50% speed-up in the N-queens problem but is neglectable in the other benchmarks of section 4.

The general scheme for *constraint propagation* is:

1. wake the constraint
2. simplify
3. resolve and propagate
4. iterate on 2) and 3) until no change
5. refreeze the constraint

This general scheme can be specialized into two global versus selective constraint propagation schemes presented in the next sections.

2.2.3 Global Constraint Propagation Scheme

In this scheme a constraint on several variables is represented by a unique goal frozen on a disjunction of variables. When the constraint is woken-up it is checked for consistency independently of the causes for its waking. The global propagation process is iterated as long as some domains of the variables in the constraint are reduced. These domain reductions are caused either directly by the propagation of the constraint itself, or by the waking of other constraints during the propagation. The possibility to interrupt the propagation of a constraint to resolve another one is an important feature of the metaprogramming approach. We shall come back on this point in section 3.
For example the program for solving linear equations is basically the following one:

```
wake_eqln(LX,LY) :-
        elim_cst(LX,LXs),
        elim_cst(LY,LYs),
        res_eqln(LXs,LYs).
```

```
res_eqln(LX,LY) :-
        minmaxln(LX,Minx,Maxx),
        minmaxln(LY,Miny,Maxy),
        min(Maxx,Maxy,Max),
        max(Minx,Miny,Min),
        Max >= Min,
        propagate_minmax(LX,Minx,Maxx,Min,Max),
        propagate_minmax(LY,Miny,Maxy,Min,Max),
        ((minmaxln(LX,Minx,Maxx),minmaxln(LY,Miny,Maxy))
        ->
                freeze_eqln(LX,LY)
        ;
                res_eqln(LX,LY)).

freeze_eqln(LX,LY):-
        varminmax([LX,LY],LV),
        freeze_or(LV,wake_eqln(LX,LY)).
```

The purpose of predicate `elim-cst` is to simplify linear terms by eliminating instanciated variables. Here constraint solving consists in propagating the extremum values of the terms in each member of the equation. This propagation is iterated as long as the extremum values of linear terms are modified (notice that these domain reductions may cause the waking of other constraints). Then the constraint is frozen again on the list of variables attached to the domain variable in the constraint.

The constraint solver recognizes also some simplified forms of the constraint, such as linear equations with less than three variables. For these small constraints a more efficient propagation scheme is preferable.

2.2.4 Selective Constraint Propagation Scheme

Some simple constraints, such as $a * X = b * Y + c$, are better represented by several specialized goals frozen as daemons, instead of by one unique goal. Instead of executing the global propagation scheme on these constraints, it is better to record the cause for a waking, perform the propagation limited to this cause, and refreeze the constraint only on this cause, keeping unchanged the other references to the constraint.

For example the predicate to implement looking ahead on $X \geq Y + c$ uses in the first argument a flag which indicates the cause of the waking (i.e. either the minimum of Y increased, either the maximum of X decreased). The program is basically the following one:

```
wake_supXYC(_,X,Y,C):-
        unique(X,X1),
        unique(Y,Y1),
        !,
        X1 >= Y1+C.

wake_supXYC(_,X,Y,C):-
        unique(X,X1),
        !,
        M is X1-C,
        decrease_max(Y,M).

wake_supXYC(_,X,Y,C):-
        unique(Y,Y1),
        !,
        M is Y1+C,
        increase_min(X,M).

wake_supXYC(minY,X,Y,C):-
        frozen(Y,domain-check(_,_,nat(Miny,_,_,_,_))),
        Min is Miny+C,
        increase_min(X,Min),
        (Y=d(_,nat(_,Miny,_,LAy,_))
        ->
                vararg(1,LAy,Vmin),
                freeze(Vmin,wake_supXYC(minY,X,Y,C))
        ;
                wake_supXYC(minY,X,Y,C)).

wake_supXYC(maxX,X,Y,C):- ... symmetrical ...
```

Note that the predicate `vararg` uses backtrackable assignment to replace an instanciated variable, such as `Vmin`, by a fresh variable used to refreeze the constraint again.

In general the selective propagation scheme is preferable to the global propagation scheme for constraints on less than three variables.

2.2.5 Compilation with the WAM

Constraint solvers are deterministic programs. Many features of the WAM are precisely done for optimizing such Prolog programs, but as pointed out in [1] particular attention has to be paid on the trailing of intermediate values during constraint propagation. In our approach these intermediate values are modified by the backtrackable assignment instruction `setarg`. Thus the general

optimisations of the trail mechanism in Prolog are inherited, furthermore we shall see in section 5 that the labels used to implement intelligent backtracking can be used as time stamps [1] to avoid the trailing of intermediate values between two choice points.

In the previous programs the refreezing of a constraint leads to the reconstruction of a frozen goal on the heap, while the old instance becomes inaccessible. It is thus desirable to reuse always the same instance, avoiding space consumption and recourse to the garbage collector. This can be achieved in our approach on top of Prolog by keeping a reference to the frozen goal in an extra argument (this was responsible for dividing by 2 the space required on the heap in an application with 60 variables and 2500 linear constraints over 6 variables each).

3 Extension to the WAM for an Adequate Delay Mechanism

3.1 Interruptions during constraint solving

One important aspect of the constraint propagation scheme on top of Prolog is the interruption of one constraint solver by another, the number of interruption levels being bounded by the number of running constraints. During the propagation of a constraint involving a large number of variables, if one domain is modified, several small constraints can be woken-up. The propagation is then interrupted, inconsistencies are detected earlier and in case of consistency the global propagation of the large constraint continues with updated domain variables. Note that the iteration check has to take care of the fact that the domains may be changed by other constraints during the propagation phase. However if it is not controlled this interruption mechanism may have the opposite effect as well. If several small constraint in forward checking are woken-up by the instanciation of a domain variable, then during the solving of the first constraint, a costly constraint can be woken-up, but then the solving of the other small constraints will not occur before the end of the propagation of the costly constraint which can be woken-up in this way several times.
Clearly one has to distinguish at least two levels of priorities according to the two propagation schemes. Constraints of priority 1, constraints of low cost, are those used in forward checking, or those containing few variables relying on the selective propagation scheme in look-ahead. Constraints of priority 2 are those containing several variables for which the global propagation scheme is preferable. It is important that the constraints of priority 2 should be interrupted only by constraints of priority 1, but not by other constraints of same priority. The propagation of a constraint of priority 2 has to be terminated before the execution of any other lower priority constraint. Indeed by postponing the execution of lower priority woken constraints the additional causes of waking

(that may occur during the execution of higher priority constraints or of the preceding lower priority woken constraints) are simply ignored and the costly constraint is executed only once.

Priority	Constraints	Interrupts
Freezed pred. prio. 1	selective propag.	priority 1
Freezed pred. prio. 2	global propag.	priority 1
Standard predicate		priorities 1 and 2

Of course after propagation and simplification a lower priority constraint may be refreezed as a higher priority constraint.

3.2 Extension to the WAM for a `wait` with two-level priorities

The method presented in [6] to incorporate a delay mechanism into the WAM can be extended to implement two-level priorities.

Constrained variables have attached to them two lists of frozen goals, corresponding to the two priorities. The manipulation of these variables is similar to [6]. Two wakeup registers W1 and W2 are introduced. These registers play the same game as register W in [6], i.e. they contain the woken goals to execute at the next inference. W1 and W2 contain respectively the goals of priority 1 and those of priority 2.

The particularity of our mechanism appears when a list of frozen goals is woken up. If there is a list of woken goals of priority 1 to execute (i.e. $W1 <> nil$) then the execution is always interrupted at the next inference to execute this list of goals. It is therefore necessary to save the current environment in order to restore it at the end of the execution. This is achieved by the instruction "continue" introduced by [6] for the restoration of the goal after its interruption.

For the waking-up of goals of priority 2, the current resolution step is tested to know whether it has been caused by the execution of a woken goal or not. For this check a new register RQ is introduced. RQ is a pointer to the last element of the list of woken goals that are executed after an interruption. This register is equal to *nil* initially and after the demonstration of the last woken goal. If $RQ = nil$ then the waking-up of goals of priority 2 causes an interruption, the list pointed by W1 and W2 are merged to form the list of goals to execute, and RQ is initialized to the last element:

$$last(W1) = *W2$$

$$RQ = last(W2)$$

If $RQ <> nil$ then the list of woken goals pointed by W2 is added *at the end* of the list of woken goals that are currently executed:

$$*RQ = *W2$$

$$RQ = last(W2)$$

In this way the list of woken goals is treated as a diff-list instead of an ordinary list. The woken goals of priority 1 cause always an interruption while those of priority 2 cause an interruption only if no woken goals are currently executed ($RQ = nil$). Otherwise they are simply added at the end of the list of currently executed woken goals. In particular with priority 1 several levels of interruption are possible while with priority 2 only one level of interruption is possible.

4 Performance Results

In this section we compare the performances of our implementation on top of Prolog on a Sun-3/50, to the performances of the Chip system on a Vax/785 given in [21][5]. Both machines are given for 1.5 Mips and can be considered as equivalent. The first benchmark is the N-queens problem. This program uses only forward checking on disequalities, together with the first-fail principle. The second example is the puzzle "send+more=money" given in [21]. This program uses both forward checking on disequalities between variables, and look ahead on one linear equation. The third example is the search of a magic series of length 7 with (CAC) and without (CWC) additional redundant constraints. The fourth example is a disjunctive scheduling problem for the construction of a bridge, consisting of 44 variables and 400 constraints. That program uses a branch and bound procedure with look ahead on precedence constraints and choice points on disjunctive constraints. The timings for this benchmark indicate the time taken to find the first solution (cost 110), the optimal solution (cost 104), and the proof of optimality.

Two timings are given on our implementation. Meta-1 indicates the running time on top of Sicstus-Prolog 0.6. Meta-2 is on top of our Prolog compiler with the highest level of priority for constraints involving less than three variables, The last column indicates the ratio between Meta-2 and Chip.

[5]Since the publication of the timings given in [21] the Chip system has been improved but new timing results are not available and this book is still the main reference.

Bench	Chip	Meta-1	Meta-2	Meta-2/Chip
8 queens	0.77	2.14	1.02	1.4
32 queens	4.05	10.1	4.75	1.2
96 queens	36.2	88.3	39.6	1.1
send	0.06	0.48	0.42	7
crypta	0.15	0.8	0.72	4.8
magic 7 CAC	12.9	14.0	14.0	1.1
magic 7 CWC	171	131	37.9	0.22
bridge				
110	1.5	2.92	2.82	1.9
104	6	6.71	6.5	1.1
proof	90	161	159	1.8

The N-queens problem is not a representative benchmark, however on this example we obtain nearly the same performances as Chip. The speed-up by a factor 2 between Meta-1 and Meta-2 on the N-queens problem is due to the wired implementation of the freezing of a goal on a disjunction of variables.

On the cryptarithmetic puzzle the speed-up between Meta-1 and Meta-2 is limited to 10%. This is quite deceiving as in this example a large linear equation of priority 2 is mixed with several small inequalities of priority 1. However a speed-up by a factor 2.5 has been measured on a different version where the problem is written with *several* linear equations of priority 2. On the magic series program, the good speedup for CWC is due to the use of priority 2 for the delayed predicate `occur` defined in the program, all constraints being in priority 1. In CAC however, a better result (8.93 seconds) can be achieved by using priority 1 for `occur`.

5 Intelligent Backtracking in Finite Domains

We will now describe the basic techniques to perform intelligent backtracking in CLP(FD). Naive backtracking upon failure (unsolvability of the constraint system) consists in simply going back to the most recent choice-point, removing the corresponding constraints and choosing an alternative clause. This may however not be enough to cure the failure, as the removed constraints may be independent of the previous failure, and this will lead to redo the same failure and backtrack further. In order to have a better behavior and avoid useless computation/backtracking steps, one has to determine upon unsolvability of the constraint system the subsystem consisting of the constraints which are the "causes" of the failure. Such a subsystem is called a conflict, and the removal of a single constraint of the conflict from the original system can "cure" the failure, i.e. restore satisfiability. One has however to take care of the management of intelligent backtrack points as some hidden dependencies exists between constraints due to previous conflicts and as the constraint system indeed derives

from a proof-tree, which also induces a dependency relation.
We will first present the extensions that are needed to achieve such a behavior, and then see how they are indeed enjoyed by running the previous Prolog implementation of finite domain constraint solving on top of our Prolog compiler with intelligent backtracking.

5.1 Finding the Causes of Unsolvability

Due to the propagation techniques, the unsolvability (or inconsistency) of the constraint system is discovered as soon as the domain of (at least) one variable becomes empty during the propagation phase. In order to be able to analyze the failure upon unsolvability of the constraint system, some extra-information has to be attached to the variables. This information should be computed during forward execution, when the domain of the variable is updated, and will contain the "history" of the domains modification.
We thus associate to each element e in the domain of a variable V an set of (references to) constraints called the *label* of e, that will refer to the constraints that leads to the removal of e from the domain of V. Observe that, due to the presence of unification constraints (simple equalities), a set of constraints (corresponding to a chain of aliased variables) instead of simply a single constraint can be responsible of the removal of a single value from a domain.
When, during the propagation phase, the domain of some variable becomes empty, the failure analysis is performed as follows.
Consider a variable X whose domain becomes empty due to a constraint c_i. In order to determine the constraints responsible of the inconsistency, one has first to compute the *clashing subdomain* of X, i.e. the set of values that have been removed from the domain but that are compatible with c_i alone. The union of c_i and the labels of all the values in the clashing subdomain forms the *conflict* corresponding to the unsolvability and represents the set of all constraints responsible of it.
The clashing subdomain, and hence the conflict can be easily determined. If c_i is a disequation constraint, it remove only a single value from the domain of X, and this value forms the clashing subdomain. For equations and inequations, one just need to consider the values ranging from the value imposed by the constraint to the extremity of the original domain of X to determine the clashing subdomain. Remark however that, as these constraints are treated by propagation on the min and max values of domains only, and that it is possible to associate only two labels to each domain, one to the min value and another to the max value. The clashing subdomain is then determined by considering either the label associated to the min or that of the max, depending on which the inconsistency occurs.

Observe that, of course, c_i is always an element of the conflict, because the consistency check is performed after each domain modification. Removal of any

of the constraint of the conflict (if it is possible, i.e. if they correspond to a non-deterministic node) will give a solvable constraint system, and the conflict hence represents the set of intelligent backtracking points associated to the failure.

5.2 Backtracking Process

Let us now precise the backtracking process, i.e. how to manage intelligent backtracking points that represent nodes of a proof-tree and handle deterministic and non-deterministic nodes.

5.3 Backtrack Points and Constraints

In the previous section, we have assumed that labels and conflicts contain references to constraints. However, as our language does not contain an explicit disjunction operation between constraints but relies on the non-determinism of the underlying constraint language, backtracking takes place not directly between constraints but between non-deterministic nodes of the proof-tree corresponding to the current computation. Indeed, at the implementation level, the system relies on a Prolog-like backtracking mechanism. The simplest way to match both visions is to consider that a constraint implicitly refers to the choice-point which is the closest non-deterministic ancestor above it in the proof-tree. Therefore backtracking to this choice-point is the minimal way of removing the constraint. Such a scheme is simple to implement and corresponds indeed to the handling of deterministic predicates in intelligent backtracking for logic programming. We will thus assume for the rest of this section that labels contain references to non-deterministic nodes as described above.

5.4 Backtracking Process

In logic programming, all intelligent backtracking methods are essentially based on two ideas : the first is that of computing a conflict in case of failure by some failure-analysis, and the second is that, while backtracking to one element of the conflict, one must store the remaining elements of the conflict as alternative solutions to cure the failure.
It is indeed not enough to just backtrack to the most recent intelligent backtrack point and forget about the other elements of the conflict, as this will lead to incompleteness of the method w.r.t. naive backtracking, see [5] [7] for a deep analysis and illustration of this phenomenon. Intuitively, if the most recent intelligent backtrack point does not lead to cure the failure, one then needs another alternative and, as distinct conflicts can occur and be mixed during the computation, considering always the current conflict is not enough. Recording alternative backtrack points for every conflict – and selectively consider which are pertinent for a given failure – is necessary.
In intelligent backtracking methods, a set containing intelligent backtrack points

is thus attached to this end to each node of the proof-tree. These sets are called *Alt* (for *Alt*ernative backtrack points) in [7], and have similar counterparts in other methods. One *Alt* set has to be attached to every (non-deterministic) node, and it has to contain all intelligent backtrack points related to it. That is, roughly, all conflicts that occurred during the resolution of the call literal and also all intelligent backtrack points associated to its father node in the proof-tree as the father is always a pertinent backtrack point. The *Alt* set is thus computed as follows. When a new node is created, its *Alt* set is initialized by that of its father node. Now consider a failure with associated conflict K that leads to a backtracking step to a node $i = \max(K)$, i.e. to the most recent intelligent backtrack point. $Alt(i)$ has to be updated to include the remaining of the conflict $K - \{i\}$.
The backtracking process consists, when backtracking to a node i that has no more alternative clauses (i.e. becomes deterministic), in inspecting $Alt(i)$ and in further backtracking to $\max(Alt(i))$, i.e. to the most recent alternative solution, see for instance [7] [8] for all the details.

5.5 Implementation

The idea, to experiment the integration of intelligent backtracking in CLP languages, was simple : just use the finite-domain algorithms written in Prolog on top of a Prolog compiler with intelligent backtracking [8]. We have however to take care that the two above mentioned schemes are compatibles ...

5.5.1 Domains

A finite domain of size n is implemented as a Prolog term that contains n slots representing the elements of the domain and slots for the min and max values. The IB machinery will roughly associate to each variable a label containing references to the literals that modified its value. In fact such a label will be associated to a *class* of (aliased) variables sharing the same value and implemented as an extra field for the value. We thus have the desired labels for the finite domains.

5.5.2 Domain Modification

The removal of a value in a domain (assignment of a free variable to false) will results, due to the IB machinery, in the recording of the current literal (the constraint) in the label of the variable. One point to take care about is the treatment of variables updated by the backtracking assignment, such as min and max. The associated label must be the union of the labels associated to all previous values and to the current one, so that the label keep track of all modifications to the value.

Observe that the IB machinery takes care of recording for a determinate literal (eg. a constraint) the nearest non-deterministic node above it in the proof tree.

5.5.3 Labels and Alt Sets

Labels are implemented as bitvectors with one bit corresponding to each choice-point in the current computation (i.e. in Prolog's local stack). Hence the size of the bitvector is proportional to the number of real choice-points and not to the number of constraints.
Alt sets are also implemented as bitvectors, and thus basic operations on *Alt* sets, such as taking the union or extracting the maximal element, are very efficient (bitwise OR and shift). Indeed our experiments show that in a simple implementation of *Alt* sets as lists of references to choice-points these operations are quite costly, and take up to 30 % of the total execution time in a compiled system (although it was 5 % in an interpreted system). Our bitvector scheme cuts down the overhead to less than 5 % allows standard memory reclaim techniques for deterministic parts of the proof-tree, see [8] for details. The conflict is accordingly a bitvector, which is used to scan down the stack of choice-points until reaching a pertinent backtrack point.

5.6 Evaluation

Our implementation of IB is very efficient and limits the overhead of the IB machinery to 20 %, giving good speedup (from a factor 2 to 9) for non-deterministic Prolog programs, see [8].
To investigate the impact of IB in CLP languages, we have used as a benchmark program a disjunctive planning example due to [21]. The application consists in constructing a bridge, and the use of shared resources leads to disjunctive constraints, which are implemented as backtrack points [21]. However in this problem every task is related to another one by some constraint in such a way that intelligent backtracking does not provide any speedup.

The speedup due to intelligent backtracking with constraint propagation can be measured on problems having a graph of constraints with more than one strongly connected components. Otherwise all variables are connected by constraints and nearly all backtrack points are intelligent. The following table provides experimental results on the 6-queens problems with up to 4 chessboards played simultaneously, and on a version of the bridge construction problem where unconnected components are created. Of course the independence is idealized in these examples, but this phenomenon appears when resource allocation is mixed with task scheduling.

Bench	Nb reductions without IB	Time without IB	Nb. reductions with IB	Time with IB	Ratio (time)
6-queens	1989	0.03	1989	0.03	1.0
2×6-queens	27294	0.58	5204	0.12	4.8
3×6-queens	439 468	9.43	9 634	0.24	39
4×6-queens	7 106 360	151.18	15 286	0.43	352
Bridge*	40 039	1.28	7 068	0.22	5.8
2×Bridge*	1 554 364	53.09	13 624	0.39	136

5.7 Application to Hierarchical Constraint Logic Programming

An extension to CLP languages, proposed in [2], consists in associating a *strength* to each constraint, that is a positive integer indicating the degree of requirement of this constraint to hold. A strength 0 denotes constraints that are required (i.e. as usual), while other strengths denote constraints that are more or less *preferred*, i.e. that should hold if possible, but that can be removed if their introduction leads to inconsistency. A *constraint hierarchy* consists in a multiset of such labelled constraints. The method to handle constraint hierarchies in a Hierarchical CLP language [2] consists in adding only vrequired constraints when they occur during the computation, whereas preferred constraints are delayed until a global solution is found for all the required constraints. They are then added by decreasing preference until it leads to inconsistency. However this approach suffers some deficiencies as for instance the preferred constraints satisfied by the final solution have not been used actively for pruning the search space during computation. It can be better to consider preferred constraint as they occur, especially for those easy to satisfy that have good chances to be part of the final solution. This allows indeed a further pruning of the search space and early discovery of inconsistencies (first-fail principle). This scheme is feasible in our system as one can isolate upon failure the responsible constraints (conflict) and relax first the preferred ones. We are currently designing such a language extension in our system.

6 Conclusion

Metaprograms for constraint solving can be surprisingly efficient provided some critical primitives are implemented in the logic programming shell. We have shown that the addition of backtrackable assignment in Prolog together with the introduction of priorities in the standard delay mechanisms of Prolog are sufficient to limit the overhead due to the programming of constraints over finite domains on top of Prolog, to a factor less than 7 in comparison to an integrated system as Chip, even on quite large problems.

Having defined a very limited extension of the WAM for constraint propagation on top of Prolog it is possible to benefit from other improvements. This point has been illustrated by the combination of constraint solving with intelligent backtracking. We have described the basic techniques to perform intelligent backtracking in CLP(FD), and have shown how these techniques are indeed realized by the finite domain package on top of our Prolog compiler with intelligent backtracking.

This framework is also appropriate to investigate other constraint propagation techniques. We are particularly interested in constraint relaxation for hierarchical CLP and reactive CLP. We are currently investigating these techniques in the light of the basic mechanisms developed for intelligent backtracking.

References

[1] A. Aggoun, N. Beldiceanu, "Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems", Se'minaire Programmation en Logique CNET, Tregastel. 1990.

[2] A. Borning, M. Maher, A. Martindale and M. Wilson : "Constraint Hierarchies and Logic Programming", proceeding of the 6th Int. Conf. on Logic Programming, Lisboa, MIT Pres 1989.

[3] BNR-Prolog User's Manual, Bell Northern Research, Canada, 1988.

[4] M. Bruynooghe : "Intelligent backtracking for an interpreter of Horn Clause logic programs", report CW 16, K.U. Leuven, September 1978.

[5] M. Bruynooghe and L. M. Pereira : "Deduction revision by intelligent backtracking", in Implementations of Prolog, J. A. Campbell (Ed.), 194-215, Ellis Horwood 1984.

[6] M. Carlsson, "Freeze, Indexing and Other Implementation Issues in the WAM", proceedings of the 4th Int. Conf. on Logic Programming, Melbourne, MIT Press 1987.

[7] C. Codognet, P. Codognet and G. Filè : "Yet Another Intelligent Backtracking Method", proceedings of the 5th Int. Conf. on Logic Programming, Seattle, MIT Press 1988.

[8] P. Codognet and T. Sola : "Extending the WAM for Intelligent Backtracking", proceedings of the 8th Int. Conf. on Logic Programming, Paris, MIT Press 1991.

[9] A. Colmerauer : "Opening the Prolog-III universe", Byte, August 1987.

[10] P. T. Cox : "Deduction plans, a graphical proof procedure for first-order predicate calculus", Ph.D. dissertation, Dept. of Computer Science, University of Waterloo, Canada, 1977.

[11] D. De Schreye and M. Bruynooghe : "implementation of finite domains for Prolog with a delay mechanism", proceeding ESOP 90.

[12] M. Dincbas, H. Simonis and P. Van Hentenryck : "Solving large combinatorial problems in Logic Programming", ECRC technical report TR-LP-21, 1987, and Journal of Logic Programming, 1988.

[13] J. Doyle : "A Truth Maintenance System", Artificial Intelligence 24 (1980).

[14] F. Fages. J. Fowler: "Programmation logique avec contraintes sur les domaine¡s finis: manuels de Meta(F) version 2.3", Rapport LCR Thomson-CSF, LACS-91-6. Sept. 1991.

[15] R. M. Haralick and G. L. Elliot : "Increasing tree search efficiency for constraint satisfaction problems", Artificial Intelligence 14 (1980), 263-313.

[16] J. Jaffar and J-L. Lassez : "Constraint Logic Programming", research report, University of Melbourne, 1986, also in the proceedings of POPL 87.

[17] L. M. Pereira and A. Porto : "intelligent Backtracking and Sidetracking in Horn clause programs - the theory", research report CINUL 2/79, Universitade Nova de Lisboa, 1979.

[18] V. J. Saraswat : "Concurrent Constraint Programming Languages", research report CMU-CS-89-108, Carnegie Mellon University, 1989,
to be published in MIT Press series in Logic Programming, 1991.

[19] S. Haridi, personal communication, Dec 89.

[20] Sicstus Prolog User's Manual, SICS, Sweden, 1989.

[21] P. Van Hentenryck : "Constraint Satisfaction in Logic Programming", MIT Press 1989.

[22] P. Van Hentenryck and Y. Deville : "Efficient Arc Consistency Algorithm for a class of CSP Problems", proc. IJCAI 91, Sidney, 1991.