A Minimal Extension of the WAM for `clp(FD)`

Daniel Diaz and **Philippe Codognet**
INRIA-Rocquencourt
Domaine de Voluceau
78153 Le Chesnay
FRANCE
{Daniel.Diaz, Philippe.Codognet}@inria.fr

Abstract

We present an abstract instruction set for a constraint solver over finite domains, which can be smoothly integrated in the WAM architecture. It is based on the use of a single primitive constraint `X in r` which embeds the core propagation mechanism. Complex user constraints such as linear equations or inequations are compiled into `X in r` expressions which encode the propagation scheme chosen to solve the constraint. The uniform treatment of a single primitive constraint leads to a better understanding of the overall constraint solving process and makes possible three main global optimizations which encompass many previous particular optimizations of "black box" finite domains solvers. Implementation results show that this approach combines both simplicity and efficiency. Our `clp(FD)` system is more than twice as fast as CHIP on average, with peak speedup reaching seven.

1 Introduction

Constraint Logic Programming (CLP) has shown to be a very active field of research over recent years, and languages such as CHIP [10] [18] [2], CLP($\mathcal{R}$) [12] [14] or PrologIII [7] have proved that this approach opens Logic Programming (LP) to a wide range of real-life applications.

The basic idea of CLP [12] is to replace unification by constraint solving over a particular domain of interest, considering the constraint solver as a "black box" that is responsible for checking the consistency of a set of constraints and, possibly, for reducing it into some normal form. Although this dichotomy is very important from the theoretical point of view, and makes it possible to import many results from LP semantics into CLP, it is not very satisfactory from the practical point of view. It may be noted that there is a curious lack of literature about the practical side of CLP...

One of the major breakthroughs of the last decade in LP has arguably been, the definition of the Warren Abstract Machine (WAM) [21] which became a de facto standard for the compilation of Prolog and has helped many researchers to gain a better understanding of Prolog's execution and to develop efficient LP systems. Moreover the WAM proved to be flexible enough to remain the backbone of various extensions such as Higher-Order, parallel or concurrent LP. To return to CLP, we could but deplore the fact that the "black box" approach does not give much information about the architecture of a real CLP system, and does not lead to the design of an abstract machine for constraints. One of the main issues is that there should be as many abstract machines as constraint domains and solvers...

We chose to focus on Finite Domains (FD), as introduced in LP by the CHIP language, where constraint solving is done by propagation and consistency tech-

niques originating from Constraint Satisfaction Problems [20] [15] [16]. Very close to those methods are the interval arithmetic constraints of BNR-Prolog [4]. Happily, a recent paper [19] broke the black box monopoly to unveil a "glass box" for FD constraints. The basic idea is to have a *single constraint* `X in r`, where `r` is a *range* (e.g. `t1..t2`). More complex constraints such as linear equations and inequations are then defined in terms of this primitive constraint. The `X in r` constraint can be seen as embedding the *core* propagation mechanism for constraint solving over FD, and should be a good basis for an abstract machine for CLP(FD)[1].
We have thus developed an extension of the WAM for FD based on the `X in r` constraint, and we propose an instruction set to implement this constraint which is very much more in the spirit of the WAM. It is also worth noticing that the basic WAM architecture and data structures are left untouched, e.g. the representation of choice points, environments and non-FD terms is not changed. Complex FD constraints are translated at compile-time into a set of `X in r` constraints, which really encodes the propagation scheme chosen to solve the constraint. This makes it possible to express at a high level the constraint solving scheme and to change it very simply if desired. Indeed the `X in r` expressions give us a *language* to express propagation methods, which is obviously not the case with the black box approach of CHIP or BNR where one has to get down to C for any change.
Also the uniform treatment of a single primitive for all complex "user" constraints leads to a better understanding of the overall constraint solving process and allows for (a few) global optimizations, as opposed to the many local and peculiar optimizations hidden inside the black box. Hence, we have designed three simple but powerful optimizations for the `X in r` constraint which encompass many previous particular optimizations for FD constraints. Implementation results show that this approach was sound and can be competitive in efficiency. On a traditional set of benchmark programs, our `clp(FD)` engine is more than twice as fast as the CHIP system, with peak speedup reaching seven.

The rest of this paper is organized as follows. Section 2 presents the (unique) constraint `X in r` and its use to define high-level constraints. Section 3 describes the integration of `X in r` in the WAM and presents the compilation scheme. It also details what is performed when a constraint is told. Section 4 presents the results of the implementation in the `clp(FD)` system. Results of a basic implementation are first presented and analyzed; and we then propose three optimizations whose impact is then detailed. A short conclusion and perspectives end the paper.

2 From primitive constraints to user constraints

2.1 The constraint `X in r`

The main idea is to use a single primitive constraint `X in r`, where `X` is a finite domain variable and `r` is a *range*, which can be not only a *constant range*, e.g. `1..10` but also an *indexical range* using:

- `min(Y)` which represents the minimal value of `Y` in the current store,
- `max(Y)` which represents the maximal value of `Y`,
- `dom(Y)` which represents the whole domain of `Y`.

[1] although the authors introduced it in the context of concurrent constraint languages [17].

The complete syntax of this (simple) constraint system is given below.

```
c ::=  X in r    (constraint)

r ::=  t..t      (interval range)
       t..       (interval t..∞)
       {t}       (singleton range)
       dom(X)    (indexical domain)
       r : r     (union)
       r & r     (intersection)
       - r       (complementation)
       r + t     (pointwise addition)
       r - t     (pointwise subtraction)
       r mod t   (pointwise modulo)

t ::=  C         (parameter)
       n         (integer)
       min(X)    (indexical min)
       max(X)    (indexical max)
       val(X)    (delayed value)
       t + t     (addition)
       t - t     (subtraction)
       t * t     (multiplication)
       t // t    (division rounded by default)
       t / t     (division rounded by excess)
       t mod t   (modulo)
```

The intuitive meaning of such a constraint is: "`X` must always belong to `r`". There are some monotonic restrictions about constraints so that the range `r` can only decrease cf. [19]. During computation, a constraint can *succeed*, *fail* or *suspend*. For example, in the store `X in 3..20, Y in 5..7:10..100` the constraint:

- `X in 10..50` succeeds and the new store is:
  `X in 10..20, Y in 5..7:10..100`.
- `X in 30..50` fails.
- `X in min(Y)..40` suspends and the new store is:
  `X in 5..20, X in min(Y)..40, Y in 5..7:10..100`.
- `X in dom(Y)+1` suspends and the new store is:
  `X in 6..8:11..20, X in dom(Y)+1, Y in 5..7:10..100`.

A constraint `c` can be removed from the current store only if it succeeds. If `c` suspends, it must remain in the store. Hence in the third example, `X in min(Y)..40` must remain in the store as long as min(`Y`) can be greater than min(`X`). Indeed, at each modification of min(`Y`), that constraint will be actived to check consistency with the domain of `X`, reducing it if necessary.

2.2 User constraints

From constraints `X in r`, it is possible to define high-level constraints (called user constraints) as Prolog predicates. Each constraint specifies how the *constrained variable* must be updated when the domains of other variables change. In the following examples `X`, `Y`, `Z` are *FD variables* and `C`, `L`, `U` *parameters* (runtime constant values).

```
'x=y+c'(X,Y,C):-   X in min(Y)+C..max(Y)+C,
                   Y in min(X)-C..max(X)-C.

'x+y=z'(X,Y,Z):-   X in min(Z)-max(Y)..max(Z)-min(Y),
                   Y in min(Z)-max(X)..max(Z)-min(X),
                   Z in min(X)+min(Y)..max(X)+max(Y).

'x≠y'(X,Y):-       X in -(dom(Y)),
                   Y in -(dom(X)).
```

In the user constraint `'x≠y'(X,Y)` the constraint `X in -(dom(Y))` must be awoken only when `Y` is bound (in a *forward checking* manner, cf. [11]) because it is not monotonic (as the domain of `Y` decreases, its complementary increases). The most elegant solution to deal with that phenomenon is to move to the concurrent constraint framework and to use an `ask` mechanism, but staying in the CLP approach a simple solution is to use some well known delay mechanism (`freeze`, `wait`, ...)[13]. In our approach this is achieved using a new indexical term `val(X)` which delays the activation of a constraint in which it occurs until `X` is *ground*, i.e. its domain is reduced to a singleton. So `dif` is defined as:

```
'x≠y'(X,Y):- X in -{val(Y)},
             Y in -{val(X)}.
```

The propagation scheme used in the user constraint `'x=y+c'` is a *partial lookahead*, namely only changes on min and max of `X` and `Y` are propagated. A *full lookahead* scheme could be used by:

```
'x=y+c'(X,Y,C):- X in dom(Y)+C,
                 Y in dom(X)-C.
```

3 Integration of `X in r` into the WAM

3.1 Modifying the WAM for FD variables

Here, we detail the necessary modifications of the WAM to manage a new data type: FD variables. They will be located in the heap, and an appropriate tag is introduced to distinguish from Prolog variables. Dealing with FD variables slightly affects data manipulation, unification, indexing and trailing instructions.

3.1.1 Data manipulation

FD variables, as standard WAM unbound variables, cannot be duplicated (as is done for terms by structure-copy). For example, loading an unbound variable into a register consists in creating a binding to the variable whereas loading a constant

consists in really copying it. In standard WAM, thanks to self-reference representation for unbound variables, the same copy instruction can be used for these two kinds of loading. Obviously, a FD variable cannot be represented by a self-reference, so we must take care of this problem. When a source word W_s must be loaded in a destination word W_d, if W_s is a FD variable then W_d is bound to W_s else W_s is physically copied in W_d.

3.1.2 Unification

A FD variable `X` can be unified with:

- an unbound variable `Y`: `Y` is just bound to `X`,
- an integer `n`: equivalent to `X in n..n`[2],
- another FD variable `Y`: equivalent to `X in dom(Y)` and `Y in dom(X)`.

3.1.3 Indexing

The simplest way to manage a FD variable is to consider it as an ordinary unbound variable and thus try all clauses. Obviously, doing more complex indexing based on the actual values of the domain would be useful, e.g. for optimizing the declarative definition of piecewise functions.

3.1.4 Trailing

In the WAM, unbound variables only need one word (whose value is fully defined by their address thanks to self-references), and can only be bound once, thus trailed at most once. These key properties make it possible to use a simple-entry trail. With FD variables these two properties no longer hold and a multiple-entry trail is needed.

Multiple-entry trail. A tagged trail is used to record the multiple values for FD variables (e.g. min, max, ...). Hence we have three types of objects in the trail: one-word entry for standard Prolog variables, two-words entry for trailing one previous value, $(n+2)$-words entry for trailing n previous values.

Avoiding useless trailing. As the domain of an FD variable is gradually reduced (in many intermediate steps), the standard (WAM) criterion for trailing would lead to much useless trailing. Indeed, only one trailing is necessary per choice point for an FD variable. We thus use the time stamp method of [1] which consists in adding a new register `STAMP` to number the choice points[3] and an extra cell per FD variable which refers to the choice point of its last trailing. Then, an FD variable `X` needs to be trailed if `Stamp(X)` $\neq$ `STAMP`.

3.2 Data structures for constraints

3.2.1 Argument frame

An argument frame (A_Frame) represents the environment in which the constraint is called, it records addresses of FD variables and values of parameters. All the

[2] we will describe later how constraints are managed.
[3] i.e. `STAMP` is incremented at choice point creation and decremented at choice point deletion.

constraints defined in the same clause share the same A_Frame. In the following FD variables will be referred as `fv(i)` (Frame Variable) and parameters as `fp(j)` (Frame Parameter).
For example, `'x=y+c'` will be translated into the following pseudocode:

```
'x=y+c'(X,Y,C):-
          create a 3 elements A_Frame
          put X in A_Frame (fv(0))
          put Y in A_Frame (fv(1))
          put C in A_Frame (fp(2))
          fv(0) in min(fv(1))+fp(2)..max(fv(1))+fp(2),
          fv(1) in min(fv(0))-fp(2)..max(fv(0))-fp(2).
```

3.2.2 Constraint frame

A constraint frame (C_Frame) is created for every constraint and consists of:

- the pointer to the associated A_Frame,
- the address of the FD variable which is constrained[4],
- the address of the associated code.

3.2.3 FD variable frame

The frame associated to the FD variable `X` is divided in two main parts:

- the domain (a not empty range),
- the constraints depending on `X` (several distinct chains).

These two parts are not modified at the same times. Chains are created when the constraints are *installed* whereas the domain can be updated during execution. Each part has its own stamp and can thus be trailed independently.

More precisely, a range consists of:

- `Type_Range`: range features (one word, of which only 2 bits are used: is the range empty ? is the bit-vector used ?),
- `Min`: the min of the range (one word),
- `Max`: the max of the range (one word),
- `Vec`: the bit-vector associated to the range (`n` words), not used if the range is an interval. It will be switched on as soon as a hole appears in the range.

Several distinct chains are used, in order to avoid useless propagation[5]:

- `Chain_Min`: list of constraints depending on min(`X`) and not on max(`X`).
- `Chain_Max`: list of constraints depending on max(`X`) and not on min(`X`).
- `Chain_Min_Max`: list of constraints depending on min(`X`) and on max(`X`).
- `Chain_Dom`: list of constraints depending on dom(`X`).
- `Chain_Val`: list of constraints depending on val(`X`).

[4] for the absentminded reader: that is `X` in the constraint `X in r`.

[5] for instance, it is useless to reexecute a constraint depending on min(`X`) when only max(`X`) is changed.

3.2.4 Registers

In order to manage the previous data structures, we need to introduce new registers:

`AF`	: pointer to the current A_Frame.
`CF`	: pointer to the current C_Frame.
`CC`	: Continuation after Constraint.
`STAMP`	: choice point number.
`DATE`	: call constraint date (explained below).
`T(t)`	: Term registers.
`V(v)`	: Vector registers.

The `CC` register points the next instruction to execute after the call constraint.
There is a bank of one word term registers (`T(t)`) and a bank of `n` words vector registers (`V(v)`).
Remark that the standard WAM `CP` and `A(i)` registers could be used instead of `CC`, `T(t)` and `V(v)`. But in that case they should be saved in the standard way (`allocate` and `deallocate` for `CP`, `try` and `retry` for `A(i)` registers), while this can be avoided with these special registers.
A range is stored in three `T` registers and one `V` register: `T(tr)`=`Type_Range`, `T(min)`=`Min`, `T(max)`=`Max`, `V(v)`=`Vec` (see section 3.2.3).

3.3 Compilation scheme

The compilation of a clause which contains at least one `X in r` constraint gives rise to three groups of instructions:

- creating and loading the A_Frame. The space is reserved at the top of the heap and addresses of FD variables and values of parameters are loaded into this new A_Frame.
- installing a constraint. The installation code for a constraint creates and loads a C_Frame. It also initializes the appropriate chain lists for all FD variables used by this constraint. For example, in the constraint $c \equiv$ `X in min(Y)..`∞, the installation code will add a pointer to c in the list of constraints depending on the min of `Y`.
- calling a constraint. The code for a constraint consists of:
  - loading parameters, indexical terms and ranges into appropriate registers. Useless loading is avoided: for instance, if a constraint uses `dom(X)` and `min(X)`, only the domain of `X` needs to be loaded because it contains the min of `X`.
  - computing the range `r`. The compilation of `r` is very easy, driven by the constraint syntax. The syntactical tree of `r` is traversed bottom-up: each leaf and each node give rise to a specific instruction. For leaves corresponding to indexical terms (or parameters), copy instructions are produced to set the appropriate registers from those loaded in the previous part. The final code can benefit from register optimization[6].

[6] in our compiler we reused the register allocation written for the WAM.

```
'x=y+c':   fd_set_AF(3,X(3))                    3 elements X, Y, C
           fd_value_in_A_frame(X(0))            X is fv(0)
           fd_value_in_A_frame(X(1))            Y is fv(1)
           fd_parameter_in_A_frame(X(2))        C is fp(2)
           fd_install_constraint(inst_1,X(3))   install cstr_1
           fd_call_constraint                   call cstr_1
           fd_install_constraint(inst_2,X(3))
           fd_call_constraint
           proceed

inst_1:    fd_create_C_frame(0,cstr_1)
           fd_install_ind_min_max(fv(1))        uses min(Y) and max(Y)
           fd_proceed

cstr_1:    fd_ind_min_max(T(0),T(1),fv(1))      min(Y) and max(Y)
           fd_parameter(T(2),fp(2))             C
           fd_add(T(0),T(2))                    min(Y)+C
           fd_add(T(1),T(2))                    max(Y)+C
           fd_tell_interval_range(T(0),T(1))    X in min(Y)+C..max(Y)+C
           fd_proceed

inst_2:    (...)
```

Table 1: Fragment of code generated for `'x=y+c'`

– telling the constraint `X in r` (i.e. updating `X` w.r.t. `r`). See explanation in section 3.4.

Appendix A summarizes the complete description of the instruction set.
Table 1 shows an example of code generated for our typical example `'x=y+c'`.

3.4 Telling the constraint `X in r`

Here, we detail the work done to tell a constraint. Some issues will be labeled (in *italic*) for later references. For a constraint `X in r` we have the following possible behaviors:

If `X` is an integer, there are two possibilities:

- `X` $\in$ `r`: success (*INT Check Ok*).
- `X` $\notin$ `r`: failure (*INT Check Not Ok*).

else (X is an FD variable whose range is $\mathtt{r}_X$) let `r'` be `r` $\cap$ $\mathtt{r}_X$:

- `r'`$= \emptyset$: failure (*Empty Range*).
- `r'`$=\mathtt{r}_X$ (i.e. $\mathrm{r}_X \subset$ `r`): success (*False Tell (sic)*).
- otherwise: (*True Tell*) if `r'` only contains one element, `X` is assigned to this value else the domain of `X` is replaced by `r'`.
  In both cases propagation occurs. Namely, as the domain of `X` has been modified, some constraints should be reexecuted. Here, we take advantage of having separate constraint chains (cf. section 3.2.3). The current `CC` must be pushed on the stack (local or global) for restoring it after propagation.

It is important to note that *False Tell* and *INT Check OK* issues do not modify the current store. So, if we can detect that the tell of a constraint will give rise to one of these issues, it is possible to avoid it, as we will see later.
The propagation phase consists in awaking and executing a set of constraints which could themselves enrich this set by new constraints. As the overall order of execution is obviously irrelevant for correctness, we could thus either manage an explicit propagation stack (or bag, queue, heap,...) or handle an implicit continuation-based execution. This is very similar to the execution of goals in logic programs where one can choose between Prolog depth-first search and more complex handling of (active) goals in the resolvent, as in concurrent logic languages. We have chosen the latter and maintain an explicit propagation stack for flexibility reasons. The small overhead induced by this scheme is largely counterbalanced by the potential for order heuristics and optimizations (see below). Moreover, our experiments show that we reach more quickly the solution in that way than with depth-first search.

4 Implementation results

4.1 Basic implementation

The compiler used to implement `clp(FD)` was developed at INRIA in 1991 [8]. Its novelty is to translate Prolog to C via the WAM. Predicates give rise to C functions, WAM instructions to C macros. Its performances are similar to Sicstus Prolog (version 0.6). The extension to `clp(FD)` gives rise to boolean C functions for constraints.
Several traditional benchmark programs have been used:

- `crypta`: a cryptarithmetic on 10 variables (20 digit numbers) [18].
- `eq10`: a system of 10 linear equations with 7 variables.
- `eq20`: a system of 20 linear equations with 7 variables.
- `queens`: the well-known N-queens problem [18]
  (N=16 with standard labeling and N=64 with first-fail principle).
- `five`: the five houses puzzle [18].
- `cars`: the car sequencing problem of [9] with 10 cars.

In all programs, labeling is done in that standard way unless otherwise stated and only the first solution is computed.

Performances of the architecture above presented are pretty good, giving a 1.6 speedup factor w.r.t. CHIP on average. Full measurements for this implementation and for the forthcoming optimizations can be found in appendix B.

4.2 Optimization 1

If we analyze the decomposition of tells, we note that many of them (about 75 %) have *False Tell* or *INT Check Ok* as issues (see appendix B). A lot of *False Tell*s and *INT Check Ok*s come from the fact that many constraints are "equivalent", so that it is not necessary to reexecute them. Intuitively, all constraints inside a

single user constraint have the same declarative meaning and would lead to such a phenomenon, detailed by the following example.
Consider the constraint $X = Y + 5$, (`'x=y+c'(X,Y,5)`) with a current store:

```
X in 5..15, Y in 1..10
```

giving:

```
X in 5..15, Y in 1..10,

X in min(Y)+5..max(Y)+5 (C_X),
Y in min(X)-5..max(X)-5 (C_Y)
```

Let us now detail what is happening if the constraint `X in 12..100` is told. `X` is set to `12..15` and thus its min is propagated to `Y` via C_Y (`Y in 7..10`). Now, as the min of `Y` has been modified, C_X (`X in 12..15`) will be reexecuted giving rise to *False Tell* (i.e. it does not modify the domain of `X`). Is is obviously useless to evaluate `X` from `Y` because `Y` has just been computed from `X`. C_X and C_Y are "equivalent".
More formally, two constraints c and c' are *equivalent* if in any store S they have the same solutions. Consider a constraint $c \equiv X \ in \ r$ and an equivalent constraint $c' \equiv Y \ in \ r'$. Let $V \cup \{Y\}$ be the set of variables on which r depends, then obviously r' depends on $V \cup \{X\}$. If c has been executed due to a modification of Y then it is useless to call c' as it cannot reduce the domain of Y, because c and c' are equivalent. Otherwise (c has been executed due to some $Z \in V$) c' has also been pushed on the propagation stack (due to Z). In both cases it is useless to call c' once c has been executed.

Optimization 1: *telling* `c`*, it is useless to reexecute* `c'` *if* `c'` *is* equivalent *to* `c`.

In `clp(FD)`, we have designed all user constraints such as linear equations, inequations and disequations so that all constraints in the body of a user constraint definition are equivalent. We recall that all constraints defined in the same clause share the same A_Frame, and therefore to implement this optimization we only have to compare the current `AF` with the one used by the constraint to be called.

Tells avoided by optimization 1 w.r.t. initial version are:
average: 16 %, worst case: 2 % (`queens`), best case: 27 % (`cars`).

4.3 Optimization 2

Many *INT Check Ok*s come from the fact that it is not necessary to reexecute `X in r` when `X` is ground (i.e. bound to an integer). This can be done if the constraints are written in such way that when `X` becomes ground, domains of related variables are reduced so that no further check of this constraint is needed[7] (i.e. the constraint succeeds).
Consider for example the constraint $X \neq Y$ (`'x≠y'(X,Y)`) with a store:

```
X in 1..10, Y in 1..10
```

giving:

[7] of course this is the case for all user constraints defined in the system.

```
X in 1..10, Y in 1..10,

X in -{val(Y)} (C_X),
Y in -{val(X)} (C_Y)
```

When `X` is set to 5, C_Y is awoken and 5 is removed from the domain of `Y`.
Thus, the new store is:

`X=5, Y in 1..4:6..10,` C_X, C_Y

Suppose now that a constraint `Y=8` is told (*True Tell*). The propagation reexecute C_X giving rise to *INT Check Ok*. Since `X` has been set to `5`, any constraint on `Y` does not need to reexecute such C_X.
However, reexecution can be avoided only if the variable became ground before calling the top-level constraint and not in the current propagation.

Optimization 2: *it is useless to reexecute* `X in r` *if* `X` *became ground before the top-level call constraint.*

To do this we use a new register (`DATE`) which is incremented at each constraint call. When a variable becomes ground it is dated with the current date. For this purpose a special cell in the FD variable frame is reserved.

Tells avoided by optimization 2 w.r.t. initial version are:
average: 21 %, worst case: 5 % (`cars`), best case: 48 % (`queens`).

4.4 Optimization 3

A lot of useless tells result from the fact that we have in the propagation stack multiple occurrences of a single constraint awoken from several variables. Since the order in which constraints are executed is irrelevant, this leads to many redundant executions: only one instance of a constraint has to be present in the propagation stack at any time.

Optimization 3: *if a constraint is already present in the propagation stack, it is useless to add it again.*

This can be achieved efficiently without scanning the whole propagation stack by using some simple dating technique reusing the `DATE` register introduced for optimization 2.
Tells avoided by optimization 3 w.r.t. initial version are:
average: 18 %, worst case: 0 % (`queens`), best case: 30 % (`eq10`).

4.5 Final results

The last table in appendix B gives full statistics for all three optimizations together.
Tells avoided w.r.t. initial version are:
average: 43 %, worst case: 38 % (`cars`), best case: 50 % (`queens`).

We can now compare `clp(FD)` with other CLP systems over finite domains, such as the CHIP system developed at ECRC and then COSYTEC. Exactly the same programs were run on both systems. The machine used for `clp(FD)` is a Sony

Program	CHIP Time (ms)	clp(FD) Time (ms)	CHIP / clp(FD)
crypta	190	120	1.58
eq10	270	140	1.92
eq20	480	220	2.18
queens 16	4520	1770	2.55
queens 64 (ff)	1580	230	6.86
five	50	20	2.50
cars	190	70	2.71

Table 2: Benchmarks (comparison with CHIP)

workstation (processor MIPS R3000), and times for CHIP (commercial version v3.1) have been measured on a Sparc Station 1+. Both machines are given as equivalent.

The average speedup of clp(FD) w.r.t. CHIP is around a factor 2.2, with peak speedup reaching 7 for the N-queens problem with first-fail principle (see table 2). We can also compare clp(FD) with the CHIP compiler system [2] on the queens program. The CHIP compiler performs in 60 ms for N=16, 780 ms for N=64 and 23300 ms for N=256. clp(FD) performs respectively in 30, 230 and 11200 ms. Therefore clp(FD) is still more than twice as fast. However it has not been possible to make a full comparision because we had no access to the CHIP compiler.

5 Conclusion and perspectives

We have presented an abstract instruction set for a constraint solver over finite domains, which can be smoothly integrated in the WAM architecture. It is based on the idea of [19] of using a single primitive constraint X in r which embeds the core propagation mechanism, while complex constraints are compiled into X in r expressions.

Implementation results show that this approach is sound, as it combines both simplicity and efficiency. Our clp(FD) is more than twice as fast as the commercial version of CHIP on average, with peak speedup up to a factor seven, thanks to three main global optimizations. We have also designed a boolean constraint solver using X in r decompositions. This system performs pretty well, being seven times faster than the CHIP propagation-based solver and infinitely better than the CHIP boolean unification on usual boolean benchmarks [5].

Future work will consist in integrating more complex constraints such as cardinality and constructive disjunction, and a simple intelligent backtracking scheme on FD constraints [6].

Perspectives also include moving to the concurrent constraint framework [17] by defining a simple and efficient ask mechanism, and extending the constraint solver for incremental solving in reactive systems, i.e. for an intelligent handling of addition or deletion of constraints "from the outside" with minimal recomputation.

Acknowledgements.

We would like to thank Pascal Van Hentenryck for the initial idea of the "glass box" and his helpful answers to our questions.

References

[1] A. Aggoun and N. Beldiceanu. Time Stamps Techniques for the Trailed Data in CLP Systems. In *Actes du Séminaire 1990 - Programmation en Logique*, Tregastel, France, CNET 1990.

[2] A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In *8th International Conference of Logic Programming*, Paris, France, MIT Press 1991. Also in *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press 1993.

[3] H. Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* Logic Programming Series, MIT Press, Cambridge, MA, 1991.

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

[5] P. Codognet and D. Diaz. Boolean constraint solving using a finite domains propagation scheme. Research Report, forthcoming.

[6] P. Codognet, F. Fages and T. Sola. A metalevel compiler for CLP(FD) and its combination with intelligent backtracking. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). The MIT Press, 1993.

[7] A. Colmerauer. An introduction to Prolog-III. *communications of the ACM*, 33 (7), July 1990.

[8] D. Diaz. Compilation de Prolog: étude et réalisation d'un outil extensible. Mémoire de D.E.A., Université d'Orléans, 1991.

[9] M. Dincbas, H. Simonis and P. Van Hentenryck. Solving the Car-Sequencing Problem in Constraint Logic Programming. In *ECAI-88*, Munich, W. Germany, August 1988.

[10] M. Dincbas, H. Simonis and P. Van Hentenryck. Solving large combinatorial problems in Logic Programming. *Journal of Logic Programming*, 8 (1,2), 1990.

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

[12] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, FRG, January 1987.

[13] J. Jaffar and S. Michaylov. A Methodology for Managing Hard Constraints in CLP Systems. In *proceedings of Sigplan PLDI*, Toronto, Canada, ACM Press 1991.

[14] J. Jaffar, S. Michaylov, P. J. Stuckey and R. Yap. An Abstract Machine for CLP($\mathcal{R}$). In *proceedings of Sigplan PLDI*, San Francisco, USA, ACM Press 1992.

[15] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence 8 (1977)*, pp 99-118.

[16] B. A. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence 5 (1989)*, pp 188-224.

[17] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Research Report CMU-CS-89-108, Carnegie-Mellon University, 1989. Also (revised) MIT Press 1993.

[18] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[19] P. Van Hentenryck, V. Saraswat and Y. Deville. Constraint processing in cc(FD). Draft, 1991.

[20] P. Van Hentenryck, Y. Deville and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57 (1992)*, pp 291-321.

[21] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Oct. 1983.

A Instruction set

A.1 Interfacing with Prolog clause

`fd_set_AF(nb_arg,V(i))` reserves space, on the top of the heap, for an A_Frame whose size is `nb_arg`. `AF` and `V(i)` point the start of the A_Frame.

`fd_variable_in_A_frame(V(j))` binds `V(j)` to a new FD variable created on top of the heap and puts its address in the cell pointed by `AF`. `AF` is incremented.

`fd_value_in_A_frame(V(j))`, let `w` be the dereferenced word of `V(j)`, if `w` is:

- an unbound variable: similar to `fd_variable_in_A_frame(w)`.
- an integer: it is pushed on the heap and its address is stored in the cell pointed by `AF`. `AF` is incremented.
- an FD variable: its address is stored in the cell pointed by `AF`. `AF` is incremented.

`fd_parameter_in_A_frame(V(j))` the dereference of `V(j)` must be an integer and its value is copied in the cell pointed by `AF`. `AF` is incremented.

`fd_install_constraint(install_proc,V(i))` restores `AF` with `V(i)` and the control is given to the install procedure.

`fd_call_constraint` calls the constraint pointed by `CF`.

A.2 Installing constraints

`fd_create_C_frame(tell_fv,constraint_proc)` creates, on the top of the heap, a C_Frame. The code is located at the address `constraint_proc` and the constrained variable is the `tell_fv`*th*. `CF` points this C_Frame.

fd_install_$\left\{\begin{array}{c}\texttt{ind_min}\\ \texttt{ind_max}\\ \texttt{ind_min_max}\\ \texttt{ind_dom}\\ \texttt{dly_val}\end{array}\right\}$(fv)
adds a new element (with the content of CF) to the appropriate chain list of the fv*th* variable.

fd_proceed gives the control to the address pointed by CC.

A.3 Loading parameters, indexical terms and ranges

Let R(r) be the list of registers: T(tr), T(min), T(max), V(v) (see section 3.2.4).

fd_parameter(T(t),fp) loads the value of the fp*th* parameter in T(t).

fd_ind_$\left\{\begin{array}{c}\texttt{min}\\ \texttt{max}\end{array}\right\}$(T(t),fv) loads the $\left\{\begin{array}{c}\text{min}\\ \text{max}\end{array}\right\}$ of the fv*th* variable in T(t).

fd_ind_min_max(T(min),T(max),fv) loads the min and the max of the fv*th* variable in T(min) and T(max).

fd_ind_dom(R(r),fv) loads the domain (a range) of the fv*th* variable in R(r).

fd_dly_val(T(t),fv,lab_else) if the fv*th* variable is an integer, it is copied in T(t), else the control is given to the label lab_else.

A.4 Computing ranges

fd_interval_range(R(r)) creates a range from a min (in T(min)) and a max (in T(max)) (i.e. initializes T(tr), V(v) being unused, see section 3.2.3).

fd_$\left\{\begin{array}{c}\texttt{union}\\ \texttt{inter}\end{array}\right\}$(R(r),R(r1)) executes R(r)←R(r)$\left\{\begin{array}{c}\cup\\ \cap\end{array}\right\}$R(r1).

fd_compl(R(r)) executes R(r)←0..∞ \ R(r).

fd_$\left\{\begin{array}{c}\texttt{add}\\ \texttt{sub}\\ \texttt{mod}\end{array}\right\}$_in_range(R(r),T(t)) executes R(r)←R(r)$\left\{\begin{array}{c}+_{pointwise}\\ -_{pointwise}\\ mod_{pointwise}\end{array}\right\}$T(t).

fd_copy_V(V(v),V(v1)) executes V(v)←V(v1).

fd_integer(T(t),n) executes T(t)←n.

fd_$\left\{\begin{array}{c}\texttt{add}\\ \texttt{sub}\\ \texttt{mul}\\ \texttt{floor_div}\\ \texttt{ceil_div}\\ \texttt{mod}\end{array}\right\}$(T(t),T(t1)) executes T(t)←T(t)$\left\{\begin{array}{c}+\\ -\\ *\\ /_{default}\\ /_{excess}\\ mod\end{array}\right\}$T(t1).

fd_copy_T(T(t),T(t1) executes T(t)←T(t1).

A.5 Telling constraints

See section 3.4 for more details about telling `X in r`. We recall that the current constraint is pointed by `CF` and `X` can be reached from the C_Frame. So only `r` must be provided to `tell`. In order to optimize the execution we distinguish the case `X in` $t_1..t_2$ and the case `X in r` (with any `r`).

`fd_tell_range(r)` tells `X in r` where `r` is a range.

`fd_tell_interval_range(T(min),T(max))` tells `X in r` where `r` is an interval.

B Performances evaluation

Basic Implementation			Tell decomposition				
Program	Time (ms)	Tell (#)	*True Tell*	*False Tell*	*INT Check Ok*	*INT Check Not Ok*	*Empty Range*
`crypta`	170	8919	2073	4087	2707	11	41
`eq10`	190	15746	3018	8679	4000	6	43
`eq20`	300	24546	5154	12497	6846	12	37
`queens16`	2600	64619	21132	6954	34700	834	999
`five`	20	566	227	52	273	14	0
`cars`	90	2483	402	1271	810	0	0

Optimization 1 only			Tell decomposition				
Program	Time (ms)	Tell (#)	*True Tell*	*False Tell*	*INT Check Ok*	*INT Check Not Ok*	*Empty Range*
`crypta`	150	6897	2121	2883	1841	10	42
`eq10`	160	12568	3101	6750	2668	2	47
`eq20`	270	21625	5369	11039	5168	14	35
`queens16`	2600	63118	21132	6954	33199	834	999
`five`	20	489	227	41	207	14	0
`cars`	70	1796	402	958	436	0	0

Optimization 2 only			Tell decomposition				
Program	Time (ms)	Tell (#)	*True Tell*	*False Tell*	*INT Check Ok*	*INT Check Not Ok*	*Empty Range*
`crypta`	160	7599	2073	4087	1387	11	41
`eq10`	170	13700	3018	8679	1954	5	44
`eq20`	270	21038	5154	12497	3338	11	38
`queens16`	1800	33481	21132	6954	3562	834	999
`five`	20	401	227	52	108	14	0
`cars`	80	2364	402	1271	691	0	0

Optimization 3 only			Tell decomposition				
Program	Time (ms)	Tell (#)	*True Tell*	*False Tell*	*INT Check Ok*	*INT Check Not Ok*	*Empty Range*
`crypta`	130	6517	2140	2558	1767	13	39
`eq10`	150	11114	3135	5538	2392	8	41
`eq20`	270	18134	5303	8562	4220	15	34
`queens16`	2650	64619	21132	6954	34700	834	999
`five`	20	547	227	52	254	14	0
`cars`	70	1949	403	1093	453	0	0

Optimizations 1+2+3			Tell decomposition				
Program	Time (ms)	Tell (#)	*True Tell*	*False Tell*	*INT Check Ok*	*INT Check Not Ok*	*Empty Range*
`crypta`	120	5047	2155	2239	601	11	41
`eq10`	140	8739	3174	4747	769	6	43
`eq20`	220	14483	5479	7550	1405	9	40
`queens16`	1770	31980	21132	6954	2061	834	999
`five`	20	345	227	41	63	14	0
`cars`	70	1546	403	910	233	0	0