Tracing and analyzing execution of clp(fd) programs: a trace model and an experimental validation environment

Projet RNTL OADymPPaC, réalisation 1.2.1

November 23, 2001

Ludovic Langevine, Pierre Deransart, Mireille Ducassé and Erwan Jahier

INRIA/INSA, INRIA, IRISA/INSA, IRISA/IFSIC

{Ludovic.Langevine, Pierre.Deransart}@inria.fr, {Mireille.Ducasse, Erwan.Jahier}@irisa.fr

INRIA Rocquencourt
BP 105
FR-78153 Le Chesnay Cédex

IRISA
Campus universitaire de Beaulieu
Avenue du Général Leclerc
FR-35042 Rennes Cédex

Contents

1 Introduction **3**

2 Operational semantics of constraint programming **5**

3 Trace definition **10**

4 Deriving a tracer from the operational semantics **13**

5 Performance **17**

6 clp(fd) Trace Analyzer: a First Prototype **23**

7 Experimentation **24**

8 Discussion and Conclusion **26**

A Source code of the test suite **31**

B Source code of the program analyzer **33**

C Source Code of the Second and Third Meta-interpreters **35**

Abstract

Developing and maintaining CLP programs requires visualization and explanation tools. However, existing tools are built in an ad hoc way. Therefore porting tools from one platform to another is very difficult. We have shown in previous work that, from a fine-grained execution trace, a number of interesting views about logic program executions could be generated by trace analysis.

In this report, we propose a generic trace model for constraint resolution by narrowing and a methodology to study and improve it. The trace model is the first one proposed for clp(fd) and does not pretend to be the ultimate one. The methodology is based on the following steps: definition of a formal model of trace, extraction of relevant informations by a trace analyzer, utilization of the extracted informations in several debugging tools.

We present the trace model and an implementation which includes a tracer, based on a meta-interpreter written in ISO-Prolog, and an *opium*-like analyzer. The efficiency of the tracer is tested and some elementary debugging tools are experimented.

This work sets the basis for generic analysis of behavior of clp(fd) programs.

1 Introduction

Developing and maintaining CLP programs benefits from visualization and explanation tools such as the ones designed by the DiSCiPl European project [11]. For example, the CHIP search-tree tool [28] helps users understand the effect of a search procedure on the search space; and the S-Box model [18] allows users to inspect the store with graphical and hierarchical representations.

However, existing tools are built in an ad hoc way. Therefore porting tools from one platform to another is very difficult. One has to duplicate the whole design and implementation effort every time.

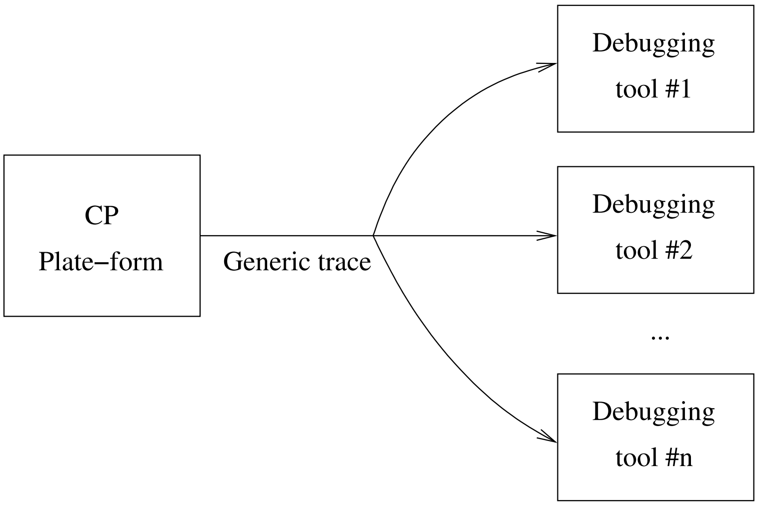

Figure 1: Broadcasting of the generic trace to several debugging tools

We have shown in previous work that, from a fine-grained execution trace, a number of interesting views about logic program executions could be generated by trace analysis [12, 21]. Here we want to generate a generic trace that can be used by several debugging tools. Each tool gives a useful view of the clp(fd) execution (e.g. the labelling tree or the domain state evolution during a propagation stage). As shown by Figure 1, the trace is broadcasted to every tools in a standard exchange format ([9] gives a description of a XML-based format). Each tool selects the relevant data in order to build its execution view. Therefore, the trace is supposed to contain all potentially useful information. The tools can be developed simultaneously and reused on different platforms [1]. This approach assumes that there is a generic trace model. In order to reach this objective, we start with a general model of finite domain constraint solving which allows to define generic trace events.

So far, no fine-grained tracers exist for constraint solvers. Implementing a formally defined tracer is a delicate task, if even at all possible. This is especially true in the context of constraint solving where the solvers are highly optimized. Therefore, before going for a "real" implementation, it is essential to elaborate trace models and experiment them.

In order to rigorously define the constraint solving trace model, and as we did for Prolog [22], we use an operational semantics based on [2] and [16]. We define execution events of interest with respect to this semantics. On one hand, this formal approach prevents the fined-grained trace model to be platform dependent. On the other hand, the consistence between the model and concrete clp(fd) platforms has to be validated.

In order to develop a first experimentation, the formal model is implemented as an instrumented meta-interpreter [2] which exactly reflects it.

The proposed instrumented meta-interpreter is useful to experimentally validate the model. It is based on well known Prolog meta-interpretation techniques for the Prolog part and on

[1]Our OADymPPaC [20] partners will use our trace model to build tools in the CHIP [8], CLAIRE [6] and GNU Prolog [17] platforms.

[2]The meta-interpreter approach to trace clp(fd) program executions has already been used in the APT tool by Carro and Hermenegildo [4]. However, APT does not access the propagation details. We propose a more informative trace.

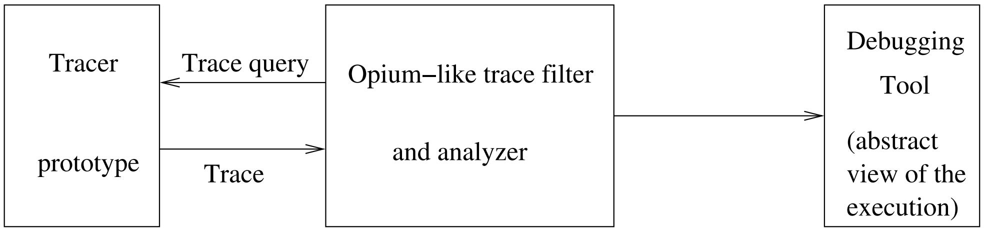

Figure 2: The experimentation chain: the tracer, the filter and the analyzer.

the described operational semantic for the constraint part.

The ultimate goal of the meta-interpreter (as in [22]) is to provide an executable specification of traces. The traces generated by the meta-interpreter could then be used to validate a real (and efficient) tracer. At this stage, efficiency of the meta-interpreter is not a key issue; it is used as a prototype rather than as an effective implementation. However we present some preliminary results of performance in order to assess the practicability of our approach.

Such a tracer may generate a large volume of data. The generic trace must be analyzed and filtered in order to condense this data down to provide the sufficient information to the debugging tools. To push further the validation we have mimicked a trace analyzer *à la* Opium [13] as shown by Figure 2. This Opium-like analyzer allows to obtain different kinds of abstract views of the execution. This is a way to show that the proposed trace model contains the necessary information to reproduce several existing clp(fd) debugging tools.

In this report, we concentrate on the constraint solving part, a trace of the logic programming part "*à la* Byrd" [3] could be integrated in the meta-interpreter (see for example [14]). In [9], additional ports are suggested to cope with other aspects of the computation.

In the following, Section 2 formally defines an execution model of clp(fd) narrowing. In particular, it defines a 8 steps operational semantics of constraint solving. These steps are the basis for the trace format defined in Section 3. Section 4 explains how to build an instrumented meta-interpreter in order to build an experimental environment. Section 5 gives some preliminary performance results. Section 6 presents an Opium-like trace analyzer based on the instrumented meta-interpreter. Section 7 briefly describes experiments with the proposed trace and our trace analyzer. Finally, Section 8 discusses the content of the trace with respect to existing debugging tools.

2 Operational semantics of constraint programming

In this section, we propose an execution model of (finite domain) constraint programming which is language independent. The operational semantics of constraint programming results from the combination of two paradigms: control and propagation. The control part depends on the programming language in which the solver is embedded, and the propagation corresponds to narrowing. Although the notions introduced here are essentially language independent, we will illustrate them in the context of clp(fd).

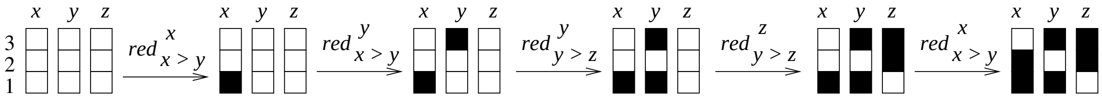

Figure 3: Application of reductions to the system $\{x > y; y > z\}$.

2.1 Basic notations

In the rest of the paper, $\mathcal{P}(A)$ denotes the power set of A; $r|_w$ denotes the *restriction* of the relation $r \subseteq A \times B$ to $w \subseteq A$: $r|_w = \{(x, y) \mid x \in w \wedge (x, y) \in r\}$.

The following notations are attached to variables and constraints: $\mathcal{V}$ is the set of all the finite domain variables of the problem; $\mathcal{D}$ is a finite set containing all possible values for variables in $\mathcal{V}$; D is a function $D : \mathcal{V} \to \mathcal{P}(\mathcal{D})$, which associates to each variable x its current domain, denoted by D_x; min_x and max_x are respectively the lower and upper bounds of D_x; $\mathcal{C}$ is the set of the problem constraints; **var** is a function $\mathcal{C} \to \mathcal{P}(\mathcal{V})$, which associates to each constraint $C \in \mathcal{C}$ the set of variables of the constraint.

2.2 Reduction operators

Many explanation tools focus on domain reduction (see for example [16, 23]). When constraints are propagated, the evolution of variable domains is a sequence of withdrawals of inconsistent values. At each step, for a given constraint, a set of inconsistent values is withdrawn from one (and only one) domain. These values can be determined by algorithms such as "node consistency", "arc consistency", "hyper-arc consistency" or "bounds consistency" described for example by Marriott and Stuckey [25].

Following Ferrand et al [16], we define reduction operators. The application of all the reduction operators of a constraint gives the *narrowing operator* introduced by Benhamou [2].

Definition 2.1 (Reduction operator). *A* reduction operator red_C^x *is a function attached to a constraint C and a variable x. Given the domains of all the variables used in C, it returns the domain D_x without the values of x which are inconsistent with the domains of the other variables. The set of withdrawn values is denoted W_x.*
$red_C^x(D|_{\mathbf{var}(C)}) = D_x - W_x$. □

There are as many reduction operators attached to C as variables in $\mathbf{var}(C)$. In general, for efficiency reasons, a reduction operator does not withdraw all inconsistent values.

A simple example of reduction operator for $C \equiv x > n$, where n is a given integer, is $red_C^x(D|_{\{x\}}) = D_x - \{v \mid v \in D_x \wedge v \leq n\}$.

The evolution of the domains can be viewed as a sequence of applications of reduction operators attached to the constraints of the store. Each operator can be applied several times until the computation reaches a fix-point [16]. This fix-point is the set of final domain states.

An example of computation with reduction operators is shown in Figure 3. There are three variables x, y and z and two constraints, $x > y$ and $y > z$. At the beginning, $D_x = D_y = D_z = \{1, 2, 3\}$, represented by three columns of white squares. Considering the first constraint, it appears that x cannot take the value "1", because otherwise there would be no value for y such that $x > y$; $\mathrm{red}^x_{x>y}$ withdraws this inconsistent value from D_x. This withdrawal is marked with a black square. In the same way, $\mathrm{red}^y_{x>y}$ withdraws the value 3 from the domain of y. Then, considering the constraint $y > z$, the operators $\mathrm{red}^y_{y>z}$ and $\mathrm{red}^z_{y>z}$ withdraw respectively the sets $\{1\}$ and $\{2,3\}$ from D_y and D_z. Finally, a second application of $\mathrm{red}^x_{x>y}$ reduces D_x to the singleton $\{3\}$. The fix-point is reached. The final solution is: $\{x = 3, y = 2, z = 1\}$.

2.3 Awakening and solved conditions

An essential notion of constraint propagation is the *propagation queue*. This queue contains all the constraints whose reduction operators have to be applied. At each step of the propagation, a constraint is selected in the propagation queue, according to a given strategy depending on implementation (for example a constraint with more variables first). The reduction operators of the selected constraint are applied. These applications can make new domain reductions. When a variable domain is updated, the system puts in the queue all the constraints where this variable appears. When the queue becomes empty, a fix-point is reached and the propagation ends. Thus, there are three fundamental operations: *selection* from the queue, *reduction* of variable domains and *awakening* of constraints.

It is actually not necessary to wake a constraint on each update of its variable domains. For example, it is irrelevant to wake the constraint $x > y$ at each modification of D_x or D_y. The reduction operators of this constraint are unable to withdraw a value in any domain if neither the upper bound of D_x nor the lower bound of D_y is updated. Hence, it is sufficient to wake the constraint only when one of those particular modifications occurs.

Definition 2.2 (Awakening condition). *The* awakening condition *of a constraint C is a predicate depending upon the modifications of $D|_{\mathbf{var}(C)}$. This condition holds when a new value withdrawal can be made by the constraint reduction operators. The condition is optimal when it holds* only when *a new value withdrawal can be made.*

The awakening condition of C is denoted by awake_cond(C). □

The actual awakening conditions are often a compromise between the cost of their computation and how many awakenings they prevent.

Another condition type permits irrelevant awakenings to be prevented. Let us consider a constraint C; if the domains of its variables are such that no future value withdrawals can invalidate the constraint satisfaction, the constraint is said to be *solved*. In that case, the reduction operators of this constraint cannot make any new value withdrawal anymore, unless the system backtracks to a former point. For example, if the two domains D_x and D_y are such that $D_x \cap D_y = \emptyset$, it is useless to apply the reduction operators of $x \neq y$. Thus, it is useless to wake a solved constraint.

Definition 2.3 (Solved condition). *The* solved condition *of a constraint C is a predicate depending upon the state of $D|_{\mathbf{var}(C)}$. This condition holds only when the domains are such that C is solved.*

The solved condition of C is denoted by solved_cond(C). □

We can note that a sufficient condition is that all variables in $\mathbf{var}(C)$ are ground to some values satisfying the constraint.

In the rest of the paper, a primitive constraint is defined by those three characteristics: its reduction operators, awakening condition and solved condition.

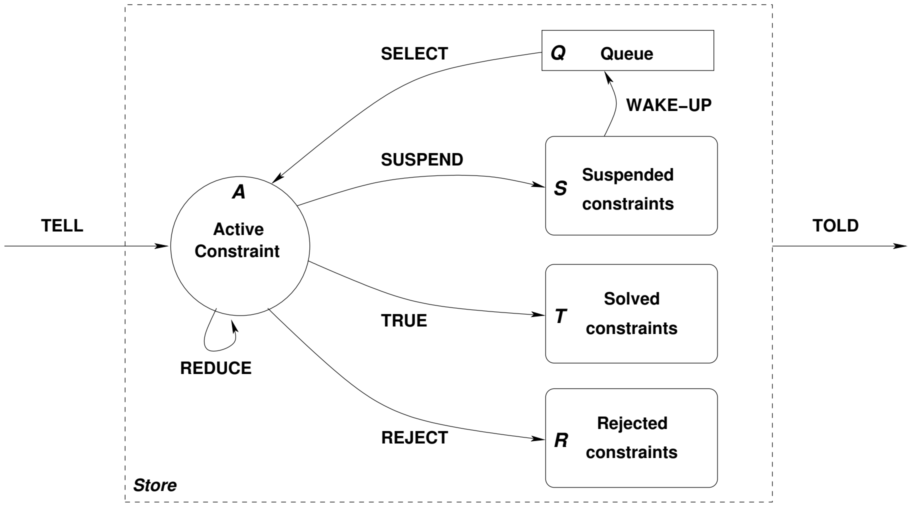

Figure 4: Events related to the structured store

2.4 Structure of the constraint store

The *constraint store* $\mathcal{S}$ is the set of all constraints taken into account by the computation at a given moment. When the computation begins, the store is empty. Then, constraints are individually added or withdrawn according to the control part. In the store, constraints may have different status and the store can be partitioned into five subsets denoted A, S, Q, T, R.

- A is the set of *active* constraints. It is either a singleton[3] (only one constraint is active) or empty (no constraint is active). The reduction operators associated to the active constraint will perform the reductions of variable domains.
- S is the set constraints which are said *suspended*, namely they are waiting to be "woken" and put into Q in the case of domain modifications of some of their variable (none has empty domain).
- Q is the set of constraints in the *propagation queue* [4], it contains the constraints waiting to be activated. In order to reach the fix-point all reduction operators associated to these constraints must be considered.
- T is the set of solved constraints (*true*), namely the constraints which hold whatever future withdrawals will be made.
- R is the set of rejected constraints, i.e. the constraints for which the domain of at least one variable is empty. In practice it is empty or a singleton, since as soon as there is one constraint in R the store is considered as "unsatisfiable" and the computations will continue according to the control.

2.5 Propagation

The evolution of the store can now be described as state transition functions or "events" acting in the store, in the style of Guerevitch's evolving algebras [19]. This is illustrated

[3]This restriction could be alleviated to handle multiple active constraints, for example to handle unification viewed as equality constraints on Herbrand's domains.

[4]The term *queue* comes from standard usage. It is in fact a set.

$$\textbf{select} \quad \frac{\exists C \in Q \ \wedge \ A = \emptyset \ \wedge \ R = \emptyset}{Q \leftarrow Q - \{C\}, \ A \leftarrow \{C\}}$$

$$\textbf{reject} \quad \frac{\exists C \in A, \exists x \in \mathbf{var}(C) \cdot (D_x = \emptyset)}{A \leftarrow A - \{C\}, \ R \leftarrow \{C\}}$$

$$\textbf{wake-up} \quad \frac{\exists C \in S \cdot \text{awake\_cond}(C) \ \wedge \ R = \emptyset}{S \leftarrow S - \{C\}, \ Q \leftarrow Q \cup \{C\}}$$

$$\textbf{reduce} \quad \frac{\exists C \in A, \exists x \in \mathbf{var}(C) \cdot (W_x \neq \emptyset) \ \wedge \ R = \emptyset}{D_x \leftarrow D_x - W_x} \quad W_x = D_x - red_C^x(D|_{\mathbf{var}(C)})$$

$$\textbf{true} \quad \frac{\exists C \in A \cdot \text{solved\_cond}(C) \ \wedge \ R = \emptyset}{A \leftarrow A - \{C\}, \ T \leftarrow T \cup \{C\}}$$

$$\textbf{suspend} \quad \frac{\exists C \in A \ \wedge \ R = \emptyset}{A \leftarrow A - \{C\}, \ S \leftarrow S \cup \{C\}}$$

Figure 5: Propagation events. Rule format : **Name** $\frac{\textit{Conditions}}{\textit{Actions}}$ *Definitions*

$$\textbf{tell(C)} \quad \frac{A = \emptyset}{push \ \langle \{C\}, S, Q, T, R, D \rangle}$$

$$\textbf{told} \quad \frac{\quad}{pop}$$

Figure 6: Control events

in Figure 4. When propagation begins, there is an active constraint. The active constraint applies its reduction operators as far as possible. Each application of a reduction operator is a reduce event. It narrows the domain of one and only one variable. If a domain becomes empty, the constraint is rejected (reject event). A rejected constraint is a sign of failure. If no failure occurs and no other reduction can be made, the constraint is either solved (true) or suspended (suspend). A solved constraint will not be woken anymore. A suspended constraint will be woken as soon as its awakening condition holds. On awakening, the constraint is put in the propagation queue (wake-up event). When there is no active constraint, one is selected in the queue (select) and becomes active. If the queue is empty, the propagation ends.

The propagation is completely defined by the rules given in Figure 5. The rules are applied in the order in which they are given (from top to bottom). Each rule specifies an event type. An event modifies the system state: $\langle A, S, Q, T, R, D \rangle$. An event occurs when its pre-conditions hold and no higher-priority event is possible. The rule priority prevents redundant conditions. For example, a suspend event is made only when no true event is possible. In the same way, if a reduce event wakes some constraints, all wake-up events are performed before any other reduce. The rule system still contains some indeterminism: the choice of C in the rules select and wake-up, and the choice of x in the rule reduce depends on the solver strategy.

The rules are applied until all the constraints are in T, R and S. No further application is possible. If R is non empty, the store is "rejected". Otherwise, it may be considered that a (set of) solution(s) has been obtained. All constraints in T are already solved. Therefore, any tuple of values of $\mathcal{V}$ which satisfies the constraints in S is a solution. The way the computation will continue depends of the control.

2.6 Control

The evolution of the store described so far assumes that the set of constraints in the store is invariant, i.e., only their status is modified. The host constraint programming language provides the way to build the store and to manage it, with possible interleaving of constraint management and propagation steps.

In order to remain as independent as possible from the host language, we restrict the control part to two events: the first one adds a constraint C into the store ($\mathsf{tell}(C)$ event) and the second one restores the store and the domains in some previous state (told event).

These events act on the store as illustrated in Figure 4. The tell event puts a new constraint in the store as the active one.

In clp(fd) the control part is particularly simple and can be described by a (control) stack of states of the system [5]. The state contains the current store and the domains. The propagation phase is always performed in the state on top of the control stack. Each tell event includes a constraint into the store, push the new state on the stack and is followed by a complete propagation phase. When the fix-point is reached either a new $\mathsf{tell}(C')$ is performed, or a told. In the later case the stack is popped, thus the former state is restored. As long as the control stack is not empty, new tell events can be performed on choice points and correspond to alternative computations.

This semantics is formalized by the two rules of Figure 6. The *push* and *pop* operations work on the control stack. With a different host language, the same told event could be used but with a possibly different meaning.

3 Trace definition

An execution is represented by a trace which is a sequence of events. An event corresponds to an elementary step of the execution. It is a tuple of attributes. Following the notation of Prolog traces, the types of events are called *ports*.

Most attributes are common to all events. For some ports, specific attributes are added. For example, reduce events have two additional attributes: the withdrawn values W_x, for the variable x whose domain is reduced, and the types of updates made, such a modification of the domain upper bound. The attributes are as follows.

Attributes for all events

- `chrono`: the event number (starting with 1);
- `depth`: the depth of the execution, starting with 0, it is incremented at each tell and decremented at each told;
- `port`: the event type as presented in Sections 2.5 and 2.6: one of reduce, wake-up, suspend, true, reject, select, tell and told;
- `constraint`: the concerned constraint, represented by a quadruple:
  - a unique identifier generated at its tell;
  - an abstract representation, identical to the source formulation of the program;

[5] This corresponds to the description of the visit of a standard search-tree of Prolog [10].

    - an concrete representation (e.g. `diffN(X, Y, N)` for `X ## Y + N` or `X - N ## Y`) ;
    - the invocation context, namely the Prolog goal from which the `tell` is performed.
- `domains`: the value of the variable domains before the event occurs;
- `store`: the content of the constraint store represented by the 5 components described in section 2.4. Each set of constraints is represented by a list of pairs (constraint identifier, external representation):
    - `store_A` the set of *active* constraints;
    - `store_S` the set of *suspended* constraints;
    - `store_Q` the *propagation queue*;
    - `store_T` the set of *solved* constraints;
    - `store_R` the set of *rejected* constraints.

Specific attributes for reduce events

- `withdrawn`: The withdrawn domain
- `update`: The list of updates, an update is of the form `(variable -> type)`, where `type` can be one of `ground, any, min, max`, see Section 4 for further explanation;

Specific attribute for wake-up events

- `cause`: The verified part of the awakening condition

The attributes are numerous and contain large chunks of information. Indeed, they aim at providing useful information to automatic trace analysis programs. The more contents the better. In a default display for users, only some attributes would be chosen. Furthermore, and as in Opium [13] the trace analysis will be mainly done on the fly, only the attributes relevant to a given analysis will be retrieved, and **no** trace will be stored. Therefore there is no a priori restriction on the number and size of attributes.

Figure 7 shows the source code of program `sorted(L)`. The program sorts three numbers between 1 and 3 in a very naive way. Following the convention of many systems, constraints operators are prefixed by a "`#`". The figure also shows a trace of the execution. All events are listed but only with a few event attributes: the event number and port, the constraint concerned by the event and its variable domains. At `reduce` events, the variable whose domain is being reduced as well as the withdrawn values are added.

The first two constraints are entered (`tell`) and suspended without any reduction (events #1 to #4). The `tell` of the third one, `Y #> Z` gives two value withdrawals, '1' from D_y (#6) and '3' from D_z (#8). The first reduction modifies the lower bound of D_y and so wakes the suspended constraint `X #>= Y` (#7). After those two reductions the constraint is suspended and the waiting one is selected (#10). At event #12, the domains are $D_x = \{2,3\}, D_y = \{2,3\}$ and $D_z = \{1,2\}$. Then the labelling phase begins. With our simple "first fail" strategy, the first added constraint is `X #= 2`. `X` is ground and equal to 2 and this constraint is solved (#17). Two other constraints are solved during the propagation, but it leads to $D_y = \{3\}, D_z = \{1,2\}$ and an empty domain for `X` (#25). Another labelling constraint is tried (#26), `X #= 3` and leads to the unique solution {`X:3, Y:2, Z:1`}.

```
sorted([X, Y, Z]):-
   [X, Y, Z] ::  1..3,        % At the beginning, D_x = D_y = D_z = [1..3]
   X ## Y, X #>= Y, Y #> Z,   % 3 constraints : x ≠ y, x ≥ y and y > z
   labelling([X, Y, Z]).      % labelling phase, with a "first fail" strategy
```

```
 1 [1] Tell    X##Y  X:[1,2,3] Y:[1,2,3]
 2 [1] Suspend X##Y  X:[1,2,3] Y:[1,2,3]
 3 [2] Tell    X#>=Y X:[1,2,3] Y:[1,2,3]
 4 [2] Suspend X#>=Y X:[1,2,3] Y:[1,2,3]
 5 [3] Tell    Y#>Z  Y:[1,2,3] Z:[1,2,3]
 6 [3] Reduce  Y#>Z  Y:[1,2,3] Z:[1,2,3] Y[1]
 7 [3] Wake-up X#>=Y X:[1,2,3] Y:[2,3]
 8 [3] Reduce  Y#>Z  Y:[2,3]   Z:[1,2,3] Z[3]
 9 [3] Suspend Y#>Z  Y:[2,3]   Z:[1,2]
10 [3] Select  X#>=Y X:[1,2,3] Y:[2,3]
11 [3] Reduce  X#>=Y X:[1,2,3] Y:[2,3]   X[1]
12 [3] Suspend X#>=Y X:[2,3]   Y:[2,3]
13 [4] Tell    X#=2  X:[2,3]
14 [4] Reduce  X#=2  X:[2,3]             X[3]
15 [4] Wake-up X#>=Y X:[2]     Y:[2,3]
16 [4] Wake-up X##Y  X:[2]     Y:[2,3]
17 [4] True    X#=2  X:[2]
18 [4] Select  X#>=Y X:[2]     Y:[2,3]
19 [4] Reduce  X#>=Y X:[2]     Y:[2,3]   Y[3]
20 [4] Wake-up Y#>Z  Y:[2]     Z:[1,2]
21 [4] True    X#>=Y X:[2]     Y:[2]
22 [4] Select  X##Y  X:[2]     Y:[2]
23 [4] Reduce  X##Y  X:[2]     Y:[2]     X[2]
24 [4] Reject  X##Y  X:[]      Y:[2]
25 [4] Told    X#=2  X:[]
26 [4] Tell    X#=3  X:[2,3]
27 [4] Reduce  X#=3  X:[2,3]             X[2]
28 [4] Wake-up X##Y  X:[3]     Y:[2,3]
29 [4] True    X#=3  X:[3]
30 [4] Select  X##Y  X:[3]     Y:[2,3]
31 [4] Reduce  X##Y  X:[3]     Y:[2,3]   Y[3]
32 [4] Wake-up Y#>Z  Y:[2]     Z:[1,2]
33 [4] True    X##Y  X:[3]     Y:[2]
34 [4] Select  Y#>Z  Y:[2]     Z:[1,2]
35 [4] Reduce  Y#>Z  Y:[2]     Z:[1,2]   Z[2]
36 [4] True    Y#>Z  Y:[2]     Z:[1]
37 [4] Told    X#=3  X:[3]
38 [3] Told    Y#>Z  Y:[2,3]   Z:[1,2]
39 [2] Told    X#>=Y X:[1,2,3] Y:[1,2,3]
40 [1] Told    X##Y  X:[1,2,3] Y:[1,2,3]
```

```
chrono     = 14
depth      =  4
port       = REDUCE
constraint = (4, X#=2, assign(var(1, X), 2), labelling([X, Y, Z]))
domains    = [X::[2..3], Y::[2..3], Z::[1..2]]
withdrawn  = X::[3]
update     = [X->any, X->ground, X->max]
store_A    = [(4, X#=2)]                      store_S = [(2, X#>=Y), (3, Y#>Z), (1, X##Y)]
store_Q    = []              store_T = []     store_R = []

chrono     = 16
depth      =  4
port       = WAKE-UP
constraint = (1, X##Y, diff(var(1, X), var(2, Y)), sorted([X, Y, Z]))
domains    = [X::[2], Y::[2..3], Z::[1..2]]
cause      = [X->ground]
store_A    = [(4, X#=2)]                      store_S    = [(3, Y#>Z), (1, X##Y)]
store_Q    = [(2, X#>=Y)]    store_T = []     store_R = []
```

Figure 7: A trace of the execution of program `sorted([X, Y, Z])`, all events are present with attributes (event number, [depth], constraint C, $D|_{\mathbf{var}(C)}$). Events #14 and #16 are displayed with all their attributes.

Constraint C	Reduction operators $(red_C^x(D\|_{\mathbf{var}(c)}) = D_x - W_x)$	Solved condition solved_cond(C)	Awakening cond. awake_cond(C)
$x = y$	$W_x = D_x - (D_x \cap D_y)$ $W_y = D_y - (D_x \cap D_y)$	$D_x = D_y = \{v\}$	$x_{\text{any}} \vee y_{\text{any}}$
$x \neq y$	$W_x = \begin{cases} \{v\} \cap D_x & \text{if } D_y = \{v\} \\ \emptyset & \text{otherwise} \end{cases}$ $W_y = \begin{cases} \{v\} \cap D_y & \text{si } D_x = \{v\} \\ \emptyset & \text{otherwise} \end{cases}$	$D_x \cap D_y = \emptyset$	$x_{\text{ground}} \vee y_{\text{ground}}$
$x = y + n$	$W_x = D_x - (D_x \cap \{v + n, v \in D_y\})$ $W_y = D_y - (D_x \cap \{v + n, v \in D_y\})$	$D_x = \{v_x\} \wedge$ $D_y = \{v_y\} \wedge$ $v_x = v_y + n$	$x_{\text{any}} \vee y_{\text{any}}$
$x \neq y + n$	$W_x = \begin{cases} \{v + n\} \cap D_x & \text{if } D_y = \{v\} \\ \emptyset & \text{otherwise} \end{cases}$ $W_y = \begin{cases} \{v - n\} \cap D_y & \text{if } D_x = \{v\} \\ \emptyset & \text{otherwise} \end{cases}$	$\forall v \in D_y, v{+}n \notin D_x$	$x_{\text{ground}} \vee y_{\text{ground}}$
$x > y$	$W_x = \{v \in D_x, v \leq min_y\}$ $W_y = \{v \in D_y, v \geq max_x\}$	$min_x > max_y$	$x_{\text{max}} \vee y_{\text{min}}$
$x \geq y$	$W_x = \{v \in D_x, v < min_y\}$ $W_y = \{v \in D_y, v > max_x\}$	$min_x \geq max_y$	$x_{\text{max}} \vee y_{\text{min}}$
$x = n$	$W_x = D_x - \{n\}$	$D_x = \{n\}$	
$x \neq n$	$W_x = \begin{cases} \{n\} & \text{if } n \in D_x \\ \emptyset & \text{otherwise} \end{cases}$	$n \notin D_x$	

Table 1: Characteristics of the primitive constraints implemented in the interpreter

4 Deriving a tracer from the operational semantics

In order to experimentally validate the trace defined in Section 3, we derive, from the operational semantics of Section 2, a clp(fd) interpreter in Prolog that we instrument with trace hooks. The resulting interpreter, which produces traces, is not meant to be an efficient clp(fd) system, but to be faithful to the semantics of Section 2. The faithfulness comes from the fact that the translation of the semantic rules into executable Prolog code is syntactical.

In Section 2, we left the primitive constraints undefined. We therefore first propose a definition for 8 primitive constraints by specifying, for each, its reduction operators, its solved condition, and its awakening condition. Note that the definitions we propose in Table 1 define reduction operators that perform a full-arc consistency.

Then, we show how to translate the primitive constraints and the semantic rules into Prolog. We also show how to interface this Prolog code with the Prolog underlying system.

4.1 Primitive constraint definitions

In order to define a primitive constraint, we need to define its reduction operators, its solved condition, and its awakening condition (see Section 2). We define in Table 1 the 8 primitive constraints $x = y$, $x \neq y$, $x = y + n$, $x \neq y + n$, $x > y$, $x \geq y$, $x = n$, and $x \neq n$, where x and y represent two finite domain variables, and n represents an integer constant. The reduction operator red_C^x is defined by the set of values W_x it withdraws from the domain of variable x.

The equality constraint between two variables x and y ($x = y$) withdraws from the domains of D_x and D_y the values which are not contained in both domains. The constraint is solved only when the two variables are ground and have the same value v. At each modification of D_x or D_y, the reduction operators may withdraw new values. The constraints must therefore be woken at each of their modification.

The difference constraint between two variables x and y ($x \neq y$) can only reduce the domains when one of them is ground. However, it is solved as soon as the two domains are disjoint.

4.2 Data structures

A constraint variable is represented by a term containing a unique integer, and a string (its name in the source). A constraint instance is represented by a 4-tuple containing a unique constraint number, a string (the constraint as displayed in the source), an internal form, a list of constraint variables, and an invocation context. The invocation context of a constraint is the Prolog goal from which it was invoked. The solver state is represented by a sextuplet: `(A, S, Q, T, R, D)`, where `A`, `S`, `Q`, `T` and `R` define the store as described in Section 2; they are lists of constraints. `D` is a list of domains.

In order to represent awakening conditions and domain narrowing, we define five types of domain modifications (following what is done in ECL^{i}PS^{e} [15]). Each type refers to a particular constraint variable x.

- x_{min} refers to a modification of the D_x lower bound, e.g., $\{1,2,4\} \rightarrow \{2,4\}$;
- x_{max} refers to a modification of the D_x upper bound, e.g., $\{1,2,4\} \rightarrow \{1,2\}$;
- x_{ground} refers to a grounding of x (D_x becomes a singleton), e.g., $\{1,2,4\} \rightarrow \{1\}$;
- x_{any} refers to any modification of D_x, e.g., $\{1,2,4\} \rightarrow \{1,4\}$;
- x_{empty} refers to an emptying of D_x, e.g., $\{1,2,4\} \rightarrow \emptyset$

Those five modification types respectively appear in the code as `X->min`, `X->max`, `X->ground`, `X->any`, and `X->empty`. Awakening conditions are disjunctions of such modification types ; such disjunctions are encoded by lists.

4.3 Translation of the primitive constraints

The reduction operators, the solved condition, and the awakening condition defining a primitive constraint are encoded by the following predicates:

- `cd_reduction(+C, +D, +X, -Wx)`[6]: takes as input a constraint C, a domain state D, and a constraint variable x; it succeeds iff the application of $red_C^x(D|_{\mathbf{var}(C)})$ withdraws a non-empty set (bound to `Wx`). There is one clause per reduction operator of `C`;
- `cd_solved(+C, +D)`: takes as input a constraint C and a domain state D; it succeeds iff the constraint C is solved in the domain state D;
- `cd_awake(+C, -Cond)`: takes as input a constraint C and outputs (in `Cond`) the list of awakening conditions of C.

Figure 8 shows the implementation of the primitive constraint $x \neq y$ which is simply a Prolog encoding of the first entry of Table 1. The functor `diff/2` is the internal encoding of $\neq$. Predicate `is_ground(+X, +D, -Vx)` takes a constraint variable x and a domain D, and succeeds iff D_x is a singleton (bound to `Vx`); `get_domain(+X, +D, -Dx)` takes a constraint variable x and a domain state D, and outputs the domain of x (D_x); `d_intersection(+D1, +D2, -D)` computes the intersection of two domains.

[6] As specified in the standard Prolog [10], `+` denotes input arguments and `-` denotes outputs arguments.

```
cd_reduction(diff(X,Y),D,Y,[Vx]) :-
  is_ground(X,D,Vx),
  get_domain(Y,D,Dy),
  member(Vx, Dy).
cd_reduction(diff(X,Y),D,X,[Vy]) :-
  is_ground(Y,D,Vy),
  get_domain(X,D,Dx),
  member(Vy,Dx).
cd_awake(diff(X,Y),Cond) :-
  Cond = [X->ground,Y->ground]).

cd_solved(diff(X,Y),D) :-
  get_domain(X,D,Dx),
  get_domain(Y,D,Dy),
  d_intersection(Dx,Dy,[]).
```

Figure 8: Definition of the primitive constraint $x \neq y$: its two reduction operators, its solved condition, and its awakening condition, as specified in Table 1.

4.4 Translation of the semantic rules

Figure 9 contains the translation of the semantic rules of Figures 5 and 6. Each rule is encoded by a predicate with the same name as the rule. The translation is merely syntactical, except for the tell and told rules, for which it is unnecessary to save and restore the solver states (*push* and *pop*) since this work is done by the Prolog backtracking mechanism.

Before paraphrasing the code for one rule, we give the meaning of all the (simple) predicates that are not defined elsewhere in the paper: `choose_in_queue(+Q0, -C)` takes as input a queue Q and outputs one of the queue constraints ; it succeeds iff Q is not empty. The choice of the constraint depends of the solver strategy; `subtract(+L1, +L2, -L)` computes the difference between two lists; `get_varC(+C, -V)` takes as input a constraint C and outputs a list of the constraint variables that appear in C; `trace(+Port, +C, +St0, +O1, +O2, +O3)` takes as input the different event attributes as described in Section 3; it calls the trace analysis system which can, for example, print a trace line; `put_end_of_queue(+C, +Q0, -Q)` puts a constraint at the end of a queue; `update_domain(+X, +Wx, +D0, -D, -Mod)` takes as input a constraint variable x, a value set W_x (to withdraw), and a domain state D^0; it outputs the state domain D such that $D_x = D^0_x - W_x$, and the list of modification types x_{mod_1}, ..., x_{mod_n} (where $mod_i \in \{min, max, ground, any, empty\}$) that characterizes the W_x value removals; `internal(+C, -Ci)` takes as input a constraint and outputs its internal representation.

All the predicates translating rules take as input a solver state St_0 and output a new solver state St. St_0 and St respectively denote the state of the solver before and after the application of a rule. The only exceptions are predicates `wake_up(+St0, -St, +ModIn)` and `reduce(+St0, -St, -ModOut)` that respectively inputs and outputs an additional argument: a list of modification types (x_{mod_1}, ..., x_{mod_n}). This list is computed in `reduce/3` and used in `wake_up/3` to check the awakening condition.

Predicate `select(+St0, -St)` translates the **select** rule. That rule needs to fulfill 3 conditions to be allowed to be applied: $\exists C \in Q$, that is checked line 17 by `choose_in_queue/2` (which fails iff the queue is empty); $A = \emptyset$ and $R = \emptyset$ that are checked line 15. The 2 actions to perform when the conditions of the rule hold are $Q \leftarrow Q - \{C\}$, which is done line 18 by `subtract/3`, and $A = \{C\}$ which is done line 16. The other rules are translated in the same way.

```
tell(C, St0, St) :-
  St0 = ( [], S, Q, T, R, D),
  St  = ([C], S, Q, T, R, D),
  trace(tell, C, St0, -, -, -).

told(C, St0) :-
  trace(told, C, St0, -, -, -).

select(St0, St) :-
  St0 = ( [], S, Q0, T, [], D),
  St  = ([C], S,  Q, T, [], D),
  choose_in_queue(Q0, C),
  subtract(Q0, [C], Q),
  trace(select, C, St0, -, -, -).

reject(St0, St) :-
  St0 = ([C], S, Q, T,  [], D),
  St  = ( [], S, Q, T, [C], D),
  get_varC(C, VarC),
  member(X, VarC),
  get_domain(X, D, []),
  trace(reject, C, St0, -, -, -).

wake_up(St0, St, ModIn) :-
  St0 = (A, S0, Q0, T, [], D),
  St  = (A, S,   Q, T, [], D),
  member(C, S0),
  awake_cond(C, ModIn, True),
  subtract(S0, [C], S),
  put_end_of_queue(C, Q0, Q),
  trace(wake_up, C, St0, True, -, -).

reduce(St0, St, ModOut) :-
  St0 = ([C], S, Q, T, [], D0),
  St  = ([C], S, Q, T, [], D),
  get_varC(C, VarC),
  member(X, VarC),
  reduction(C, D0, X, Wx),
  update_domain(X, Wx, D0, D, ModOut),
  trace(reduce, C, St0, X, Wx, ModOut).

true(St0, St) :-
  St0 = ([C], S, Q, T0, [], D),
  St  = ( [], S, Q, T,  [], D),
  solved_cond(C, D),
  T = [C|T0],
  trace(true, C, St0, -, -, -).

suspend(St0, St) :-
  St0 = ([C], S0, Q, T, [], D),
  St  = ( [], S,  Q, T, [], D),
  S = [C|S0],
  trace(suspend, C, St0, -, -, -).

reduction(C, D, X, Wx) :-
  internal(C, Ci),
  cd_reduction(Ci, D, X, Wx).

awake_cond(C, ModIn, True) :-
  internal(C, Ci),
  cd_awake(Ci, Cond),
  intersection(Cond, ModIn, True),
  not True = [].

solved_cond(C, D) :-
  internal(C, Ci),
  cd_solved(Ci, D).
```

Figure 9: Prolog translation of the semantic rules of Figures 5 and 6

```
call_constraint(C, St0, St) :-
  tell(C, St0, St1),
  propagation(none, St1, St),
  ( true ; told(C, St), fail ),
  St = (_, _, _, _, [], _).

propagation(Mod0, St0, St) :-
  prop_step(St0, St1, Mod0, Mod)
  -> ( St1 = (_, _, _, _, [], _)
       -> propagation(Mod, St1, St)
       ;  St = St1 )
  ;  St = St0.

prop_step(St0, St, Mod0, Mod) :-
    select(St0, St)  -> Mod = none
;   reject(St0, St)  -> Mod = none
;   wake_up(St0, St, Mod0)
                     -> Mod = Mod0
;   reduce(St0, St, Mod)
                     -> true
;   true(St0, St)    -> Mod = none
;   suspend(St0, St) -> Mod = none.
```

Figure 10: Integrating the constraint solver with the underlying Prolog system

4.5 Integration with the underlying Prolog system

The integration of our instrumented constraint solver with the underlying Prolog system is done by the predicate `call_constraint/3` which is given in Figure 10. After propagation and success, it returns the new state of the constraint part (`St`). If the propagation leads to a failure, the goal fails.

Predicate `prop_step/4` performs a propagation step, i.e., it applies one of the 6 propagation rules of Figure 5; it fails if no rule can be applied. The choice of the rule to apply is done according to the strategy discussed in Section 2.5. Predicate `propagation/3` calls `prop_step/4` in loop until either a propagation step fails (the fix-point is reached) or the solver rejects a constraint (the constraint goal is unsatisfiable).

5 Performance

In the previous section, we have described a meta-interpreter directly derived from the operational semantics. It is purely declarative. As this approach can be very inefficient we want to assess its practicability. Therefore we started some measurements, first comparing the execution of programs in a compiled mode (without tracing) and with the compiled meta-tracer; Then comparing with meta-interpreters trying to take advantage of proper features of ECL^{i}PS^{e}.

With our preliminary experiments, several properties can be observed:

- the number of events for some classical problems;
- the efficiency of the declarative meta-interpreter;
- the possibility of using meta-interpretation to prototype clp(fd) tracers.

5.1 The Two Other Meta-Tracers

We have implemented two other meta-interpreters using delay mechanism. This mechanism is provided by ECL^{i}PS^{e} plate-form. It may give efficient meta-implementation of constraint solving [7]. The first interpreter generates the same information as the declarative one. The second one does not manage some costly attributes.

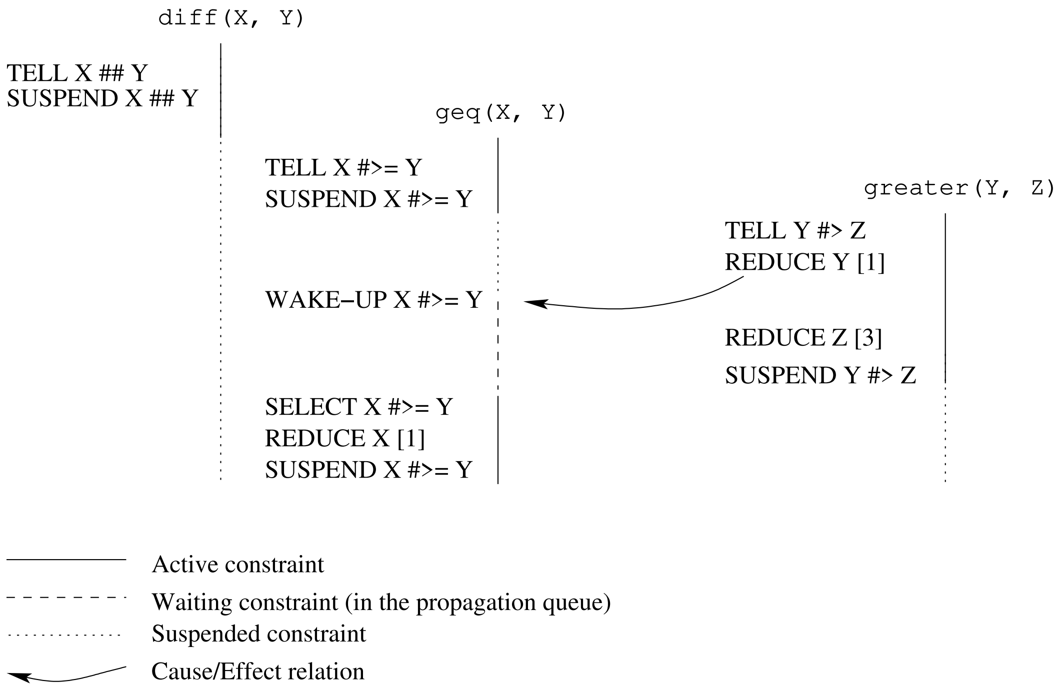

Figure 11: Events #1 to #12 of Figure 7, viewed as events in a thread based system.

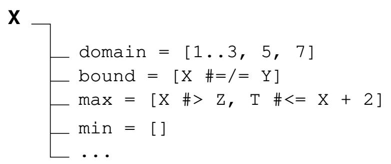

Figure 12: Example of a finite domain variable `X` and some of its attributes.

In those two meta-interpreters, constraints are still described by three characteristics: reduction operators, awakening condition and solved condition. But the events are managed by ECL$^{\text{i}}$PS$^{\text{e}}$ built-ins instead of strict rule translation.

5.1.1 Meta-interpretation Based on Goal Suspension

The mechanism of constraint management is inspired by constraint implementation in ECL$^{\text{i}}$PS$^{\text{e}}$ as described by Wallace et al. [29]. In ECL$^{\text{i}}$PS$^{\text{e}}$, a constraint is represented by a goal whose resolution is suspended until a given condition is verified. This condition generally relates to variable domains of the constraint. For example, the $x \neq y$ constraint uses a goal suspended to two events: modification of domain D_x or modification of domain D_y. The woken goal has to make the reductions due to the constraint. If those reductions are not sufficient to solve the constraint, the corresponding goal is re-suspended.

The ECL$^{\text{i}}$PS$^{\text{e}}$ built-in `suspend(+Goal, +Priority, +Cond)` explicitly delays the goal `Goal` and wakes it with priority `Priority` as soon as the condition `Cond` holds. The **priority** notion is fundamental to interrupt the current goal so as to resolve an other one.

The suspension mechanism can be viewed as a thread system, as shown by Figure 11. Each constraint is represented by a "thread" (a goal). The threads share the domains of the variables of the problem. Each thread makes the reductions due to the corresponding constraint. When it cannot make any more reduction, it is suspended. On domain reduction, all the threads whose awakening condition holds are woken. When a constraint is solved, the corresponding thread is stopped. For example, the Figure 11 shows the first twelve events of the trace described by Figure 7: the first event, a tell, creates the goal "`diff(X,Y)`" which is immediately suspended. The second tell creates the goal "`geq(X,Y)`" which is also immediately suspended. The third tell creates the third goal, "`greater(Y,Z)`" which makes two reductions. The first one causes the awakening of `geq(X,Y)`. The third goal is suspended. Then the second one is selected and makes its reduction before being suspended again.

The implantation of this mechanism is made easier by the *attributed variables* of ECL^{i}PS^{e}. Each logical variable can have one or more attributes as illustrated in Figure 12. Thus, a finite domain variable is a structure with several fields. The first attribute of `X` is the domain D_x, represented by an interval list. The other attributes of `X` are suspended goal lists. Among these lists, we find the list of goals which are suspended to the grounding of `X` (this list is named `bound`), the list of goals suspended to the modification of the domain lower bound modification (`min`) or upper bound (`max`).

The code of the meta-interpreter is given in Appendix C.

The goal suspension mechanism has already been used to build a clp(fd) meta-compiler as described by Codognet et al. [7]. In our case, the meta-interpreter conception is guided by facility of execution tracing instead of efficiency.

5.1.2 Ignored Attributes

The system state is partly embedded in the ECL^{i}PS^{e} internal state. Therefore it is costly to get some particular attributes of an event:

- the cause of an awakening is not directly available to the tracer because the interpreter loss hand between a reduce and the following wake-up. The interpreter has to store the domain modification on a reduce by an "`assert`" and to compare it with the awakening condition of the woken constraint on a wake-up.
- the store contents is represented by delayed, solved or failed goals. ECL^{i}PS^{e} does not allow the interpreter to inspect those goals. Thus, the interpreter has to record all the store alteration.

Besides the interpreter, extra-mechanisms are needed to make some attributes available, whereas in the declarative interpreter all event attributes are explicitly managed at the meta-level. For efficiency purpose, the following attributes are ignored in the third meta-interpreter:

- `store`:the store content (the sets A, S, Q, T and R);
- `cause`: the awakening cause (specific attribute of wake-up port).

5.1.3 How Equivalent are the Three Meta-Interpreters?

The three meta-interpreters are equivalent wrt the operational semantics. We have the same invoked goals and the same choice points. The traces are the same (same ports and same attributes). The third interpreter is less precise: there is less attributes attached to each event.

The computation of the propagation phase is not necessarily the same. In the first interpreter, we control the awakening strategy. In the other two ones, this strategy is up to the ECL^{i}PS^e scheduler. For a given CLP goal, we do not get the same traces because of several possible orders of constraint awakenings. From a problem to another, the two strategies are not equivalent. For a given problem, one of the strategies can be better than the other one and generate less propagation events.

5.2 Assessments

When executing a program by meta-interpretation, most advantages of compiling are lost. The execution is obviously less efficient. Three problems and several instances will be considered. For each program we compare the time spent in the execution of the program without trace (T_{comp}) and its execution and tracing with the declarative meta-interpreter (T_1). We also compare with the two less declarative meta-interpreters based on goal suspension. Our hypothesis is that the suspension-based interpreters are more efficient than the declarative one. If the time spent in the program tracing with the second meta-interpreter is denoted by T_2, then we expect to have $T_2 \ll T_1$. We also measure T_3, the time spent in the program tracing with the suspension-based meta-interpreter without the costly attributes.

In the following we compare the three measures: T_1, T_2 and T_3. We will also compute the corresponding ratios:

- the cost of the declarative meta-interpreter: $R_1 = \frac{T_1}{T_{comp}}$;
- the cost of the second meta-interpreter: $R_2 = \frac{T_2}{T_{comp}}$;
- the cost of the third meta-interpreter: $R_3 = \frac{T_3}{T_{comp}}$.

5.2.1 Methodology

Hardware and software. We perform our measurements on a SUN Ultra-10 (440MHz, 384MB of RAM). It runs under the SUN Solaris 7 operating system. The machine is very lowly loaded; the "idle" time of the CPU before an execution launching is more than 99%. The Eclipse compiler is the release 5.1.3 of the 12 May 2001.

Time measuring command. We use the elapsed time given by the `profile/1` Eclipse command. `profile/1` executes a given goal and displays execution statistics. Because of the intrinsic imprecision of the above command, we need to make sure that programs run in a period of time that is far above the clock accuracy. Each program in the benchmark suite is re-executed until it runs at least for 20 seconds. The execution time is the total execution time divided by the number of executions. More precisely, the profiled goal is:

`repeat(BenchGoal, N)` where `BenchGoal` is the goal whose execution time has to be measured and `N` is an integer such that $N \times T_{BenchGoal} \geq 20\,s$
To measure an execution time, 10 consecutive profiling are performed. The first execution is not taken into account because of time spent in library loading or module compiling. The "execution time" finally given in the following is the average of the nine other measures.

The CPU loading is checked thanks to the `top` Unix command before the test launching.

The results have been obtained with an uncertainty of 1%.

Traced programs. We chose three different traced programs. The first one solves the classical n-queens problem. It uses only dis-equality constraints ($x \neq y$ and $x \neq y+n$). Its size is easy to parameterize. Three problem sizes have been tested: $n = 10$, $n = 11$ and $n = 12$ in order to study the variation of the rates $R_{i \in \{1,2,3\}}$ with respect to the problem size. The second traced program is a 4×4 magic squares generator. Its constraints are mostly in the form $x + y + z + t = n$. The reduction operators of this kind of constraint is more subtle than constraints such as dis-equality or inequality. The third traced program is a naive generator of the integers between 1 and n which involves a lot of constraints and variables. Its size is also simple to change. Six problem sizes have been tested: $n \in \{50, 100, 200, 300, 400, 500\}$.

The source code of these programs is given in appendix A. Test programs are compiled with all optimizations provided by the Eclipse compiler (`nodbgcomp` Eclipse command), macro expansions excepted. In the case of `nqueens`, the execution means the search for *all solutions*. For the magic-square problem, we run until the first solution. For the `sorted` problem, there is only one solution for each instance.

In the meta-interpreters, the trace is deactivated by defining the `trace/6` predicate used in Figure 9 by the following clause:

```
trace(_,_,_,_,_,_):-
  incval(var_event_number).
```

Therefore, no trace is stored or printed but the basic trace mechanism is used and events are counted.

5.2.2 Results

Table 2 shows the characteristics of the ten benchmarks. For each experiment, we have: the corresponding goal, the number of constraints and the number of variables involved by this goal, the number of events with the declarative meta-interpreter and with the other two ones. The number of events of the declarative interpreter is greater than the one of the other two interpreters in the case of the n-queens problem. At the opposite, the best number of events for the magic-square problem is obtained by the declarative interpreter. It is a way to appreciate the efficiency of the two different awakening strategies.

Table 3 gives the execution times of the programs and the rates presented in Section 5.2. Some instances of the program `sorted` have not been measured with the second meta-interpreter. In fact, the first instances and the measurements made with the third interpreter are sufficient to conclude about the efficiency of the declarative interpreter. In most cases, and unexpectedly for R_2 ($T_2 > T_1$) , it appears that $R_2 > R_1 > R_3$. The ratio R_1 linearly increases with the size of data (e.g. 421 for `sorted(50,_)` and 4483 for `sorted(500,_)`). This increasing may be due to list operations. In fact, the constraint and variable sets are represented by lists in the declarative interpreter and list accesses are in linear time. Lists of integers are also a time-expensive representation in the case of large domains. These hypothesis are confirmed by the profiling details for the goal `sorted(200,_)`: 45% of execution time is spent in the predicate "`get_domain/3`".

In the case of the second and third meta-interpreters, the cost is more stable, but there is a clear loss of performance when adding extra-mechanisms: $R_3 << R_2$. The extra-mechanisms make the second interpreter less efficient than the declarative one.

Goal	Nb. constraints	Nb. variables	$Nb_{events}(decl)$	$Nb_{events}(susp)$
`nqueens(10,_)`	135	10	980 313	849 460
`nqueens(11,_)`	165	11	4 701 121	4 049 253
`nqueens(12,_)`	198	12	24 409 709	20 892 277
`ms4(_)`	134	16	47 872	112 281
`sorted(50,_)`	49	50	4 949	4 949
`sorted(100,_)`	99	100	19 899	19 899
`sorted(200,_)`	199	200	79 799	79 799
`sorted(300,_)`	299	300	179 699	179 699
`sorted(400,_)`	399	400	319 599	319 599
`sorted(500,_)`	499	500	499 499	499 499

Table 2: Size of the test programs: number of constraints and number of variables involved, number of events generated by the two meta-interpreters.

Goal	Nb sol.	T_{comp}	T_1	R_1	T_2	R_2	T_3	R_3
`nqueens(10,_)`	724	1.429	261.0	183	306.3	214	19.7	13.8
`nqueens(11,_)`	2680	6.746	1362.1	202	1619.9	240	92.5	13.7
`nqueens(12,_)`	14200	34.866	7464.9	214	9257.5	265	476.8	13.7
`ms4(_)`	1	0.19	22.7	119	69.74	367	27.3	144
`sorted(50,_)`	1	0.013	5.5	421	268.6	20661	1.9	146.2
`sorted(100,_)`	1	0.047	41.8	889	951.9	20253	8.1	172.3
`sorted(200,_)`	1	0.178	331.5	1863	-	-	32.8	184.3
`sorted(300,_)`	1	0.424	1123.5	2650	-	-	79.2	186.8
`sorted(400,_)`	1	0.811	2674.9	3298	-	-	130.0	160.3
`sorted(500,_)`	1	1.195	5357.3	4483	-	-	206.1	172.5

Table 3: Cost of the instrumented meta-interpretation on three benchmark programs. Execution times are given in seconds.

5.3 Synthesis and Discussion

We first observe a high number of events (at less 47 000 events to generate a 4×4 magic square, or about 21 millions to find all the solutions of the 12-queens problem).

In the case of small problems (about 20 variables and 150 primitive constraints), the declarative meta-interpreter gives an overhead-ratio of about 200. This ratio is highly dependent on the size of data, especially the number of variables. Some optimizations could be done on that point.

The suspension-based meta-interpretation is a more efficient meta-implementation of constraint solving but it is not appropriate for tracing purpose: retrieving some attributes is too expensive. The suspension-based meta-interpreter has no advantage in the tracer prototyping context: it is highly platform dependent; its strategies to awake and select constraints are difficult to tune; moreover, the validity of the suspension-based meta-interpreters wrt the operational semantics depends on Eclipse special features.

At the opposite, the declarative meta-interpreter explicitly manages each item appearing in the operational semantics. The strategies are encoded in some Prolog predicates. The declarative meta-interpreter is derived from the semantics in a straightforward way and is written in ISO-Prolog. Therefore its validity is easier to guarantee. It is a good way to prototype, experiment and tune formal trace format.

6 clp(fd) Trace Analyzer: a First Prototype

A first on the fly trace analyzer has been implemented. It is based on the declarative meta-tracer presented above. The conceptual model underlying the trace query mechanism is the same as Opium conceived by Ducassé [13].

The trace analyzer is an independent Prolog process. A console allows the user to formulate queries in order to investigate the execution. The execution is simultaneously running. The queries are formulated in the Prolog language extended by three primitives:

`next/0` executes the program until the next event is reached;

`fget/1` executes the program until the current event satisfies a given *filter*. For instance: "the next event of port `reduce` which modifies the upper bound of the domain of `X`";

`get_attr/2` retrieves some attributes of the current event.

These primitives allow to search a specific event forward in the execution trace (`next/0` and `fget/1`) and to retrieve data about this event (`get_attr/2`). Only forward search is available at this stage. For example, the following query goes to the first event whose port is `reduce` with a "chrono" greater than 150. The reduced variable and the withdrawn domain are then retrieved and stored in the variables `X` and `Rx`

```
:- fget([port = reduce, chrono>150]),
   get_attr([var, withdrawn], [X, Rx]).
```

Analysis programs can be written with these primitives and easily reused. For example, here is the code of a query that counts the number of failures encountered before the first solution and prints it:

```
:- setval(nb_reject, 0),
   repeat,
   fget(in(port, [reject, solution])),
   ( get_attr(port, reject)
     -> incval(nb_reject),
        fail
     ;  true
   ),
   getval(nb_reject, NbFailures),
   writeln(NbFailures).
```

The implementation architecture of this tracer/analyzer comes from Opium's one too. As shown by Figure 13, the analysis is done by two co-processes: the tracer prototype executing the examined program and the trace analyzer where the primitives described above are available. The two processes communicate by a Unix socket in a client-server scheme. At the beginning, the tracer waits for a query of the analyzer: the execution is frozen. When the

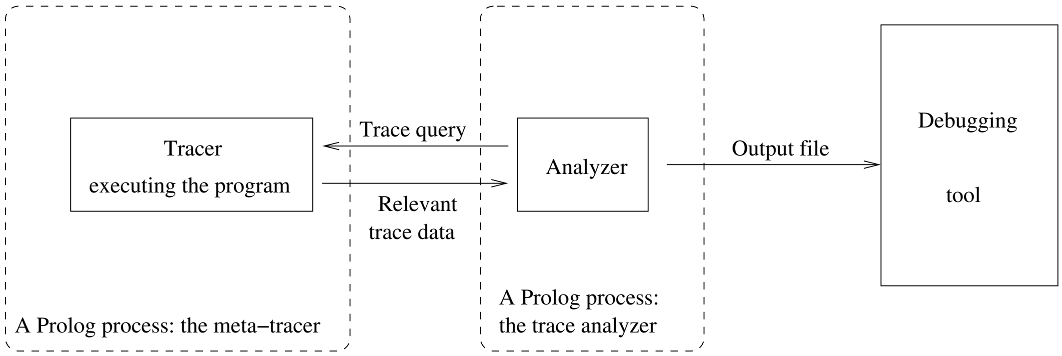

Figure 13: Architecture of the trace generation and analysis chain

analyzer has to retrieve some attributes of the current event, it sends a query to the tracer. The tracer searches for the corresponding data in the solver state and sends it to the analyzer. The execution is still frozen. When the analyzer has to examine an other event, it sends to the tracer an event-filter. The tracer resumes then the execution to the first event satisfying the filter. When a such event is reached, the execution is frozen again and the tracer waits for other queries. This mechanism allows the analyzer to examine the whole trace forward and to retrieve the necessary data in order to analyze the trace and to produce a specific view of the execution.

At this stage, the result is a file in an *ad hoc* format (for instance VRML for the computation space analysis or PostScript for the labelling tree) and a specific viewer is required to visualize it.

7 Experimentation

We have made some preliminary experiments of trace analysis with our tracer.

A first experiment consists in displaying the labelling (or search-) tree. The analyzed program solves the well known n-queens problem. Figure 14 shows a search-tree obtained for the 4-queens problem by trace analysis of a trace generated by our meta-interpreter, and displayed with *dot* [24]. Each variable corresponds to a row of the chess-board. The top node of the displayed tree contains the initial domains, the other nodes contain the reduced domains if any. The tree shows four failures in square boxes, two solutions and three choice points. The uppest arc sums up the tell operations for all permanent constraints of the programs. Each other arc represents a labelling constraint addition.

The trace analysis uses only the tell, told, reduce events. Few attributes were needed: the port, the concerned constraint and the domains of constraint variables. For reduce events, the updated variable and the withdrawn set were also needed.

The sequence of tell and told events in the trace corresponds to a depth-first left-to-right visit of the search-tree where tell and told events respectively correspond to downward and upward moves in the search-tree. Reconstructing a tree from its visit (for a fixed visit strategy) is easy. In Figure 14, we took advantage of reduce events to label the nodes with the propagation results.

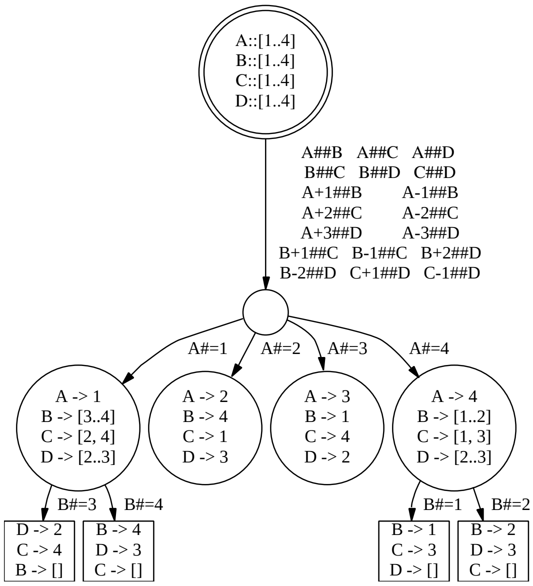

Figure 14: A search-tree of a 4-queens constraint program execution, obtained by trace analysis, and displayed by *dot*.

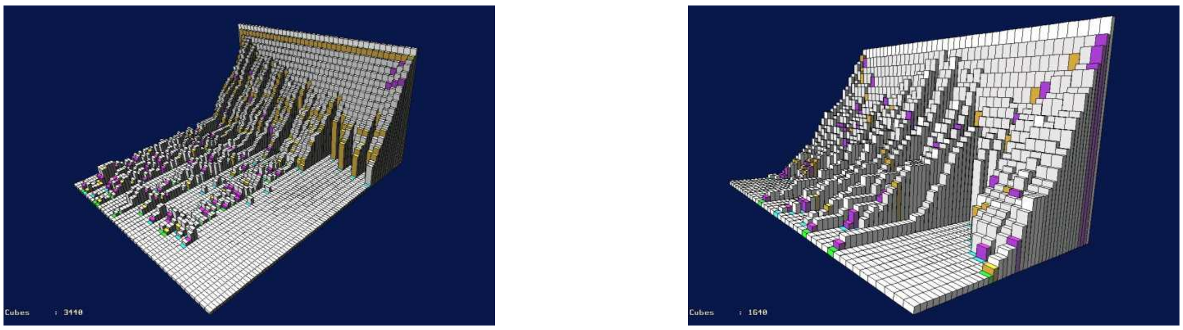

Figure 15: Comparing two search procedures for the 40-queens problem with VRML views computed by trace analysis.

A second experiment is the generation of a 3D variable update view [30]. The evolution of the domains of the variables during the computation is displayed in three dimensions. It gives a tool *à la* TRIFID [5] (here however colors are introduced to display specific events as in the variable update view of [28]). The trace analyzer makes a VRML file by computing domain size on each tell and reject event, and when a solution is found. The details of reduce events allow us to assign color to each kind of domain update (for example minimum or maximum value removed or domain emptied) as made by Simonis and Aggoun in the Cosytec Search-Tree Visualizer [28]. The trace analysis is implemented in about 125 lines of Prolog and generates an intermediate file. A program implemented in 240 lines of C converts this file into VRML format.

Figure 15 shows the resolution of the 40-queens problem with two different labelling strategies. We have three axes: variables (horizontal axis of the vertical "wall"), domain size (vertical axis) and time. The first strategy is a first-fail selection of the labelled variable and the first value tried is the minimum of its domain. The second strategy is also a first-fail strategy but variable list is sorted with the middle variable first and the middle of domain is preferred to its minimum. This strategy derives from one described by Simonis and Aggoun [28]. This approach allows to compare the efficiency of these two strategies by manipulating the 3D-model. With the first strategy, we get a quick decreasing of the domain size on one side of the chess-board and a long oscillation of the domain size on the other side. With the second strategy, the decreasing of domain size is more regular and more symmetrical, the solution is found faster. In fact, the second strategy, which consists in putting the queens from the center of the chess-board, benefits more from the symmetrical nature of the problem. The possibility of moving manually the figure facilitates observation of such property.

8 Discussion and Conclusion

This paper is a first attempt to define new ports for tracing finite domain solvers. We do not claim that the trace model presented here is the ultimate one. On the contrary we propose a methodology to experiment and to improve it. This methodology is based on the following steps: definition of an executable formal model of trace, extraction of relevant informations by a trace analyzer, utilization of the extracted informations in several debugging tools. It is illustrated by Figure 16 which combines Figures 1 and 2 of the introduction. It shows several debugging tools which extract from the same generic trace the informations they need.

The formal approach of trace modeling used here allows to clearly define the ports. Then the implementation of the model by a meta-interpreter almost written in ISO-Prolog, a logic programming language with a clear semantics, allows to preserve its correctness wrt the formal model. We have shown that this methodology is efficient enough on small examples and therefore is of practical interest. However to handle large realistic examples will require hard-coded implementation of the trace model.

Different aspects must be examined to estimate the results. First the formal model itself. Two ports are directly related to logic programming (tell and told) and correspond to the welknown ports *call* and *exit* of [3], the others correspond to a small number of different steps of computation of the reduction operator fix-point. On the other side we defined a great number of (possibly large sized) attributes to ensure that each event carries enough potentially useful informations. This immediately shows the limits of our model: it is probably general enough to take into account several finite domain solvers, but tracing the complete behavior

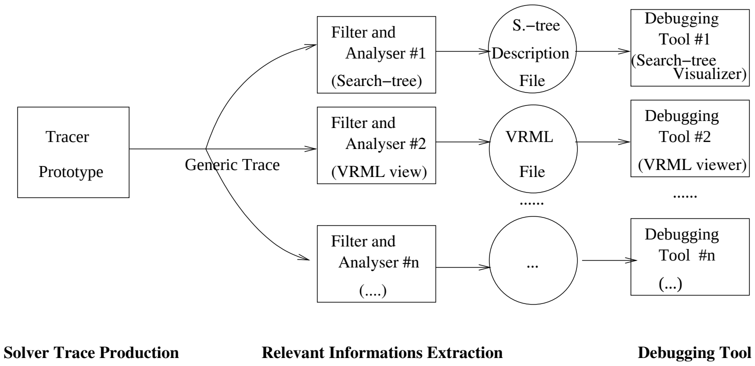

Figure 16: Trace study methodology

of different solvers will require new or different ports to take into account different kinds of control, specific steps of computation (e.g. constraint posting, labelling phase, ...), or different algorithms.

The proposed methodology shows the way to progress: defining the trace with a formal model makes easier to compare different trace models. It is thus easier to see which are the missing ports or attributes. Some solvers use a propagation queue with events instead of constraints (in this case new ports or same ports but with different attributes are necessary), or do not use backtracking (then `told` is never used). The question is also to find the right balance between the number of events and the attributes, in such a way that a hard-coded efficient implementation of (a part of) the trace model is still possible. For example the attribute `withdrawn` of the port `reduce` concerns one variable only. Therefore if several variables are involved by a single reduction step, there will be several `reduce` events. Another possibility would be to have a unique event with a port whose `withdrawn` attribute includes several variables at once. Gathering too many informations in one event may slow down the tracer considerably. The trace production must be as fast as possible in order to keep the best performances of the solver.

The ports and attributes presented here seem to be a good basis to start the study of a generic trace for CLP(FD). In [9] a sample ofadditional ports are suggested to cope with different finite domain solvers.

The second aspect concerns the extraction of tool specific relevant informations. The generic trace, by definition, contains too many informations if used by a unique debugging tool only. It is also not intended to be stored in a huge file, but it should be filtered on the fly and re-formatted for use by some given tool. The methodology we proposed here, *à la opium*, is well-known and has been shown to be efficient and general enough to be used in practice also in hard-coded implementations. It allows to specify the trace analysis in a high level language (here Prolog) in a way which is independent from the trace production.

For each experimented tool we have presented here there is a specific analyzer which needs only few lines of code. With this approach, building different views of the same execution requires only to modify the trace analyzer. Notice that the implementation presented here assumes that there is only one trace analyzer running in parallel with the solver. The question of how different analyzers, working simultaneously, could be combined is not considered here.

Finally the third aspect concerns the experimentation. The main challenge in constraint debugging is performance debugging. Our objective is to facilitate the development of constraint resolution analysis tools in a manner which is as independent as possible from the solvers platforms. The three steps method (generic tracer/trace analyzer/debugging tool) is a way to approach such a goal. We experimented it by building several analyzers and (limited) tools very easily, without having to change the trace format. Of course much more experiences are still needed.

Another way to estimate the results of this paper is to consider some of the existing debugging tools and to observe that most of the informations relevant for each of these tools are already present in the ports and their attributes. Here are briefly reviewed some platform independent tools[7], namely debugging tools using graphical interfaces where labelling, constraints and propagation can be visualized.

The Search-Tree Visualization tool for CHIP, described by Simonis and Aggoun [28] displays search-trees, variables and domain evolution. The whole of this information is present in the proposed ports. Our experimentation shows that search-tree and constraints can easily be displayed. The relevant information is also present for the Oz Explorer, a visual programming tool described by Schulte [27], also centered on the search-tree visualization with user-defined displays for nodes. It is also the case in the search-tree abstractor described by Aillaud and Deransart [1] where search-tree are displayed with constraints at the nodes.

In Grace, a constraint program debugger designed by Meier [26], users can get information on domain updates and constraint awakenings. This is available in our trace via our 8 ports. Grace also provides the ability to evaluate expressions using the current domain of the variables. As the domain of the variables is an attribute of all events, it can be obtained by any analyzer of our trace. The CHIP tool also provide update views in which, for example, useless awakenings are visible. In our environment useless awakenings could be detected by a select not followed by a reduce. The visualization tool by Carro and Hermenegildo [5] traces the constraint propagation in 3 dimensions according to time, variable, and cardinal of the domain. The required information is present in our trace. The S-boxes of Goualard and Benhamou [18] structure the propagation in and the display of the constraint store. Structuring the propagation requires to modify the control in the solver. Displaying the store according to the clausal structure would however cause no problem.

An important aspect, not presented in this paper, concerns the possibility of interactive tools. This means that some debugging tools may request to re-execute some already traced execution. Special mechanisms must be involved in order to re-execute efficiently and with the same semantics some part of the execution. How such capability may influences the form of the generic trace is still an open question and will be further investigated.

References

[1] C. Aillaud and P. Deransart. Towards a language for clp choice-tree visualisation. In Deransart et al. [11], chapter 8.

[2] F. Benhamou. Interval constraint logic programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, pages 1–21. Springer-Verlag, Lecture Notes in Computer Science 910, 1994.

[7]They are considered as (clp(fd)) platform independent tools in the sense that they are focused on general properties of finite domain solving: choice-tree, labelling, variables domain evolution.

[3] L. Byrd. Understanding the control flow of Prolog programs. In S.-A. Tärnlund, editor, *Logic Programming Workshop*, Debrecen, Hungary, 1980.

[4] M. Carro and M. Hermenegildo. Tools for search-tree visualisation: The apt tool. In Deransart et al. [11], chapter 9.

[5] M. Carro and M. Hermenegildo. The vifid/trifid tool. In Deransart et al. [11], chapter 10.

[6] Y. Caseau, F.-X. Josset, and F. Laburthe. Claire: combining sets, search and rules to better express algorithms. In D. De Schreye, editor, *Proc. of the 15th Int. Conference on Logic Programming*, pages 245–259. MIT Press, 1999.

[7] P. Codognet, F. Fages, and T. Solas. A meta-level compiler of CLP(FD) and its combination with intelligent backtracking. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, chapter 23. MIT Press, 1993.

[8] Cosytec. CHIP++ Version 5.2. documentation volume 6. http://www.cosytec.com, 1998.

[9] Romuald Debruyune, Jean-Daniel Fekete, Narendra Jussien, Mohammad Ghoniem, Pierre Deransart, Ludovic Langevine, and al. A proposal of concrete format for facing constraint programming (based on a dtd xml), October 2001. Deliverable D2.2.2.1 (in French).

[10] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog, The Standard; Reference Manual.* Springer Verlag, April 1996.

[11] P. Deransart, M. Hermenegildo, and J. Małuszyński, editors. *Analysis and Visualisation Tools for Constraint Programming.* Number 1870 in LNCS. Springer Verlag, 2000.

[12] M. Ducassé. Abstract views of Prolog executions with Opium. In P. Brna, B. du Boulay, and H. Pain, editors, *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, Cognitive Science and Technology, chapter 10, pages 223–243. Ablex, 1999.

[13] M. Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*, 39:177–223, 1999. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds).

[14] M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19/20:351–384, May/July 1994.

[15] ECLiPSe. Constraint logic programming system. http://www-icparc.doc.ic.ac.uk/eclipse/.

[16] G. Ferrand, W. Lesaint, and A. Tessier. Value withdrawal explanation in CSP. In M. Ducassé, editor, *AADEBUG'00 (Fourth International Workshop on Automated Debugging)*, pages 188–201, 2000. The COmputer Research Repository (CORR) cs.SE/0012005.

[17] GNU-Prolog. A clp(fd) system based on Standard Prolog (ISO) developed by D. Diaz. http://gprolog.sourceforge.net/ Distributed under the GNU license.

[18] F. Goualard and F. Benhamou. Debugging Constraint Programs by Store Inspection. In Deransart et al. [11], chapter 11.

[19] Y. Gurevitch. Evolving algebras, a tutorial introduction. *Bulletin of the European Association for Theoretical Computer Science*, 43:264–284, 1991.

[20] INRIA-Rocquencourt, EMN-Nantes, INSA-Rennes, University of Orléans, Cosytec, and ILOG. Tools for dynamic analysis and development of constraint programs (OADymPPaC), November 2000. An RNTL French Project. `http://contraintes.inria.fr/OADymPPaC`.

[21] E. Jahier. Collecting graphical abstract views of Mercury program executions. In M. Ducassé, editor, *Proceedings of the International Workshop on Automated Debugging (AADEBUG2000)*, Munich, August 2000. The COmputer Research Repository (CORR) cs.SE/0010038.

[22] E. Jahier, M. Ducassé, and O. Ridoux. Specifying Prolog trace models with a continuation semantics. In K.-K. Lau, editor, *Proc. of LOgic-based Program Synthesis and TRansformation*, London, July 2000. Springer-Verlag, Lecture Notes in Computer Science 2042.

[23] N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[24] E. Koutsofios and S. North. Drawing graphs with *dot*. TR 910904-59113-08TM, AT&T Bell Laboratories, 1991.

[25] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, Cambridge, Massachussets, 1998.

[26] M. Meier. Debugging constraint programs. In U. Montanari and F. Rossi, editors, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, number 976 in LNCS, pages 204–221. Springer Verlag, 1995.

[27] C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In *Proceedings of the Fourteenth International Conference on Logic Programming (*ICLP*'97)*, pages 286–300, Leuven, Belgium, June 1997. The MIT Press.

[28] H. Simonis and A. Aggoun. Search-Tree Visualisation. In Deransart et al. [11], chapter 7.

[29] M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming. Technical report, IC-Parc, August 1997.

[30] R. Zoumman. *Analyse du comportement du solveur de contraintes basée sur la visualisation de traces* (in french). Rapport de stage, INRIA, 2001.

A Source code of the test suite

There are three programs in the test suite. For each program, the types of basic constraints used, the number of constraints and the number of variables involved are given. The source code has been compiled with the release 5.1.3 of the $\mathrm{ECL^iPS^e}$ system.

The first section shows some predicates used in several programs. Each of the following sections presents only one program: the n-queens program in Section A.2, the 4×4-magic squares generator in Section A.3 and the strictly positive numbers generator in Section A.4.

A.1 Common predicates

The following predicates are used in several test programs. `ud_labelling/1` describes a user-defined labelling procedure for a list of variables. It is a naive procedure: the labelled variable is always the first in the list. `distinct(X, L)` gives the binary constraints which ensure that each variable in the list `L` is distinct from `X`. `all_diff/1` gives the binary constraints which ensure that all the variables in a list have distinct values. The macro expansion is switched off in order to prevent the compiler from making unknown program transformations. Performance differences are not significant. Therefore, the code presented below is exactly the code given to the interpreter.

```
:- set_flag(macro_expansion, off).

:- dynamic ud_labelling/1, distinct/2, all_diff/1.
ud_labelling([]).
ud_labelling([X|L]):-
        indomain(X),
        ud_labelling(L).

distinct(_, []).
distinct(X, [Y|L]):-
        X ## Y,
        distinct(X, L).

all_diff([]).
all_diff([X|L]):-
        distinct(X, L),
        all_diff(L).
```

A.2 n-queens program

Basic constraints used:

- $a \neq b$, where a and b are variables;
- $a \neq b + n$, where a and b are variables and n is a fixed integer.

Number of variables: n;

Number of constraints: $\frac{3}{2}(n^2 - n)$.

```
:- dynamic no_attack/3,
           safe/1,
           queens/2.

no_attack(_, [], _).
no_attack(X, [Y|R], I):-
        X ## Y,
        X ## Y + I,
        Y ## X + I,

        I1 is I + 1,
        no_attack(X, R, I1).

safe([_]).
safe([X|[Y|L]]):-
          no_attack(X, [Y|L], 1),
          safe([Y|L]).

nqueens(N, L):-
        length(L, N),
        L :: 1..N,
        safe(L),
        ud_labelling(L).
```

A.3 Generator of all the 4×4-magic squares

The following program generates all 4×4 magic squares: numbers from 1 to 16 are disposed in a square such that the sum of each row, column or diagonal is equal to the "magic sum", $\frac{4\times(4\times4+1)}{2} = 34$. There are 880 solutions (up to four symetric axis).

Basic constraints used:

- binary disequality $a \neq b$, where a and b are variables;
- equality $a+b+c+d = n$, where a, b, c and d are variables and n is a fixed integer;
- binary disequality $a > b$.

Number of variables: 16;

Number of constraints: 120 binary disequalities, 10 equalities, 4 inequalities (to prevent symetrical solutions from being found).

```
:- dynamic ms4/1.
ms4(L):-
        L = [A11, A12, A13, A14,
             A21, A22, A23, A24,
             A31, A32, A33, A34,
             A41, A42, A43, A44],
        L :: 1..16,

        % Computation of the ``magic sum''
        Som is 4 * (4 * 4 + 1) // 2,

        A11 + A12 + A13 + A14 #= Som,
        A21 + A22 + A23 + A24 #= Som,
        A31 + A32 + A33 + A34 #= Som,
        A41 + A42 + A43 + A44 #= Som,

        A11 + A21 + A31 + A41 #= Som,
        A12 + A22 + A32 + A42 #= Som,
        A13 + A23 + A33 + A43 #= Som,
        A14 + A24 + A34 + A44 #= Som,

        A11 + A22 + A33 + A44 #= Som,
```

```
        A14 + A23 + A32 + A41 #= Som,

        A11 #< A41,
        A11 #< A14,
        A11 #< A44,
        A14 #> A41,

        ud_labelling(L).
```

A.4 Generator of the n first strictly postitive integers

This program builds a list of n numbers between 1 and n such that each consecutive integers a and b (a is on the left of b) satisfy $a < b$. The result is the sorted list of integers between 1 and n (lower first). This program only works by propagation: no labelling procedure is used because the initial constraints are sufficient to deduce the unique solution.

Basic constraints used: strict inequality between two variables, $b > a$;

Number of variables: n;

Number of constraints: $n - 1$.

```
constrain([_]).
constrain([A|[B|L]]):-
        B #> A,
        constrain([B|L]).

sorted(N, L):-
        length(L, N),
        L :: 1..N,
        constrain(L).
```

B Source code of the program analyzer

The following Prolog program analyzes the trace of an execution and produces the corresponding search-tree. The program uses the primitives inspired by Opium (see Section 6): `next/1` (go to the next event) and `get_attr/2` (give some attributes of the current event). The first event is reached thanks to the `go/0` command.

`build_node(Node)` analyzes the trace after a `tell` until a `tell` or `told` is reached: it deals with the propagation phase after a `tell`. `Node` sums up the reductions occurring during this phase (list of pairs `V->D` where `V` is a variable whose domain has been reduced at least once and `D` is the final state of the domain of `V`). `isReduction(V->D)` succeeds iff the current event is a `reduce`. `V` is the concerned variable and `D` is the domain of `V` after reduction.

`sub_tree(SL)` constructs the list of all sons of the current node (list of pairs `Constraint - Son` where `Constraint` represents the arc between the current node and `Son`). This construction is made thanks to the `constraint_node/1` predicate.

`constraint_node(C - node(Sum_up, SonList))` is invoked on the `tell` of a constraint `C`. It sums up the resulting propagation phase in `Sum_up`. The list of eventual subtrees is `SonList`. Therefore, this predicate gives a search-tree.

`build_tree(A)` initializes the trace visit and constructs the search-tree. The root of the search-tree is labelled "`top`".

```
build_node(Node):-
        get_attr(port, Port),
        ( member(Port, [told, tell])
            -> Node = []
        ;
          % Result = V -> Dv
          isReduction(Result)
            -> next,
            build_node(Rest),
            update_node(Rest, Result, Node)
        ;
          Port = reject
            -> Node = [reject],
          next,
          build_node(_)
        ;
          next,
          build_node(Node) ).

sub_tree(SonList):-
        get_attr(port, Port),
        ( Port = told
            -> SonList = []
          ;
          Port = tell
            -> constraint_node(ASon),
               next,
               sub_tree(OtherSons),
               SonList = [ASon | OtherSons]
        ).

constraint_node(Constraint - node(Node, SonList)):-
        % On the first call, the current event
        % is tell(C). The tree is returned on told(C)
        get_attr([port, c], [tell, Constraint]),
        next,
        % Deal with the propagation phase
        build_node(Node),
        % Deal with the sub-tree of the current node
        sub_tree(SonList).

build_tree(A):-
        go,
        setof(OneSon, constraint_node(OneSon), SonList),
        A = node("Top", SonList).
```

C Source Code of the Second and Third Meta-interpreters

The following program is the non Prolog part of the suspension-based meta-interpreters presented in Section 5.1. The extra-mechanisms used to retrieve implicit attributes have been removed because they make the code understanding more difficult.

`tell(C)` is invoked by the Prolog part of the meta-interpreter on encountering a constraint `C`. A constraint number is generated and the depth is increased. Then, the constraint is activated.

`activate(C)` makes all the possible reductions and tests the solved condition of `C` before the eventual suspension. If the solved condition holds, the goal succeeds.

`reduce(C)` applies the reduction operators of `C`. The value withdrawals are given by `cd_reduction` as a list of pairs `X - Wx` where `X` is the variable to reduce and `Wx` is the withdrawn domain. Each simple withdrawal is made by `withdraw/3`.

`suspension(C)` suspends the constraint `C`. It suspends two goals with the same awakening condition: `wake_up(C)` with a high priority (1) and `select(C)` with a normal priority (3). When the awakening condition holds, the first goal is activated. Therefore, the `wake-up` event is immediately traced. The second goal is put in the queue and activated when another one succeeds or is suspended.

C.1 The Second Meta-interpreter

```
tell(C):-
        incval(var_constraint_number),
        getval(var_constraint_number, CNum),
        c_setnumber(C, CNum),
        (
          ( inc_depth,
            trace(tell, _, _, _),
            % Extra-mechanism to record the store alteration
            store_telling(C),
            call_priority(activate(C), 3) )
        ;
          ( trace(told, C, _, _),
            % Extra-mechanism to record the store alteration
            store_tolding(C),
            dec_depth,
            fail)
        ).

activate(C):-
        reduce(C),
        internal_representation(C, Int),
        ( cd_true(Int)
          ->  trace(true, _, _, _),
              % Extra-mechanism to record the store alteration
              store_solution(C)
          ; suspension(C) ).

reduce(C):-
        internal_representation(C, Int),
        cd_reduction(Int, RedList),
```

```
        (
            foreach((X, Wx), RedList), param(C) do
              ( DRetX = empty -> true
                ; withdraw(X, Wx, C) )
        ).

:- dynamic awakening_context/1.
withdraw(X, Wx, C):-
        get_domain(X, Dx),
        trace(reduce, X, Wx, _),

        % Extra-mechanism to record the awakening information
        assert(awakening_context(info(C, X, Wx))),

        d_difference(Dx, Wx, NewDomX, _),
        % All the corresponding 'wake-up' are done: retract the info
        retract_all(awakening_context(_)),
        (NewDomX = empty -> (trace(reject, _, _, _),
                             % Extra-mechanism to record the store alteration
                             store_rejection(C),
                             fail
                           ; true),
        call_priority((d_update(X, NewDomX), set_active_constraint(C)), 1),
        wake.

suspension(C):-
        internal_representation(C, Int),
        cd_wake_up_condition(Int, CondReveil),
        trace(suspend, _, _, _),
        suspend(wake_up(C, Info), 1, CondReveil),
        suspend(select(C, Info), 3, CondReveil),
        % Extra-mechanism to record the store alteration
        store_suspension(C).

wake_up(C, Info):-
        awakening_context(Info),
        trace(wake_up, C, Info, _),
        % Extra-mechanism to record the store alteration
        store_awakening(C).

select(C, _):-
        trace(select, C, _, _),
        % Extra-mechanism to record the store alteration
        store_selection(C),
        activate(C).
```

C.2 The Third Meta-interpreter

The following meta-interpreter does not manage all the event attributes as explained in Section 5.1.

```
tell(C):-
        incval(var_constraint_number),
```

```
        getval(var_constraint_number, CNum),
        c_setnumber(C, CNum),
        (
          ( inc_depth,
            trace(tell, _, _, _),
            call_priority(activate(C), 3) )
        ;
          ( trace(told, C, _, _),
            dec_depth,
            fail)
        ).

activate(C):-
        reduce(C),
        internal_representation(C, Int),
        ( cd_true(Int)
          ->  trace(true, _, _, _)
          ; suspension(C) ).

reduce(C):-
        internal_representation(C, Int),
        cd_reduction(Int, RedList),
        (
            foreach((X, Wx), RedList), param(C) do
              ( DRetX = empty -> true
                ; withdraw(X, Wx, C) )
        ).

withdraw(X, Wx, C):-
        get_domain(X, Dx),
        trace(reduce, X, Wx, _),
        d_difference(Dx, Wx, NewDomX, _),
        (NewDomX = empty -> (trace(reject, _, _, _), fail
                         ; true),
        call_priority((d_update(X, NewDomX), set_active_constraint(C)), 1),
        wake.

suspension(C):-
        internal_representation(C, Int),
        cd_wake_up_condition(Int, CondReveil),
        trace(suspend, _, _, _),
        suspend(wake_up(C, Info), 1, CondReveil),
        suspend(select(C, Info), 3, CondReveil).

wake_up(C, _):-
          trace(wake_up, C, _, _).

select(C, _):-
        trace(select, C, _, _),
        activate(C).
```