Comput Syst Sci & Eng (1994) 4: 255-261
© 1994 CRL Publishing Ltd

# CONSTRICTOR: a constraint-based language

**Giuseppina C Gini and C Rogialli**

Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milan, Italy

*It is well-known that programming by constraints is a powerful way of expressing problems; unfortunately it is not always an efficient way of solving them. Problems that can be defined as Constraint Satisfaction Problems are in the areas of planning, scheduling, image interpretation, temporal reasoning, and others. Constraint propagation also plays an important role in TMS and in qualitative physics. While the first examples of constraint satisfaction techniques in AI appeared as long ago as the 1970s, only recently has it been recognized that constraints can be the basis of programming languages. Our work is aimed at integrating CSP in the LISP environment as a general-purpose tool. A model of constraint satisfaction and a language are defined. The language implementation is discussed, with particular emphasis on the debugging facilities and their use to monitor the domains of the variables.*

*Keywords: constraint satisfaction problems, CONSTRICTOR, language*

## 1. INTRODUCTION

We have designed and developed CONSTRICTOR, a programming language for defining constraint networks and solving constraint satisfaction problems (CSP). While 'programming by constraints' is recognized as a powerful way of expressing problems, it is seldom used because of the complexity of the algorithms needed[1]. Problems that have been formalized as CSPs include image interpretation, scheduling, temporal reasoning and graphic simulation, to name only a few.

To define those problems we may use an extensional style (enumerating all the instances) or an intensional style (through patterns). The unary Walt-Disney-Character relation can be defined through the list (Mickey Mouse, Bambi, Winnie Poo, etc.) or by the pattern (created by Walt Disney & cartoon-character)

While constraint-satisfaction techniques in AI have been appearing since the 1970s, only recently has it been recognized that constraints can be the basis of programming languages. Sussman[2] and Guesgen[3] have provided the first examples of such languages.

Our work is aimed at integrating CSP in the LISP environment as a general-purpose tool. We are interested in a simple and practical implementation of a CSP on finite or infinite domains, with n-ary relations. Basic design criteria have been: problem-independence, efficient integration of the CSP into LISP, and a good debugger.

The idea of combining the constraint programming with other paradigms has become very attractive and interesting. In recent years, the paradigm of Constraint Logic Programming (CLP)[4-6] has appeared, and a new generation of logic languages that incorporate techniques for CSP are now available.

Our approach is different because we want to make available some programming tools for constraint solution while programming in a conventional language such as LISP. Moreover, we can define constraints on $n$ variables while CLPs are limited to unary and binary constraints, for which quite efficient algorithms are available

## 2. BACKGROUND

Constraint satisfaction can be illustrated by the simple example of addition. Given the relation $x + y = z$, and two values for a couple of variables, we can determine the value of the third variable. A constraint is often represented as a box with input and output arrows. Many constraint-boxes can be connected to form a constraint network. In the following:

| | |
|---|---|
| $V_1, \ldots, V_n$ | denote the $n$ variables occurring in CSP, |
| $D_1, \ldots, D_n$ | the domains associated with them, and |
| $C_1, \ldots, C_k$ | the n-ary constraints among the variables. |

Finding the solutions of the network[1] means finding all the combinations of values that satisfy all the constraints. In other words, to find all the sequences $(b_1, ..., b_n)$ of values for the $n$ variables that are global or local solutions, according to the basic definitions:

**Definition 1**
*Given a constraint network $C = \{C_1, C_2, ... C_k\}$, on the variables $V = \{V_1, V_2, ... V_n\}$, a* ***global solution*** *of the network is a combination of values $S = \{s_1, s_2, ... s_n\}$ such that their ordered assignment to the variables satisfies all the constraints.*

**Definition 2**
*Given a constraint network $C = \{C_1, C_2, ... C_k\}$, on the variables $V = \{V_1, V_2, ... V_n\}$, a* ***local solution*** *is a set of domains $D = \{D_1, D_2, ... D_n\}$ such that for each $C_i$ and for each of its variables $V_j$, if $V_j$ is assigned any value in the domain $D_j$ it is possible to find in the domains of the other variables instances that satisfy $C_i$.*

Not all the combinations obtained from a local solution are also global solutions. The set $S$ of all the global solutions of a network is a subset of the Cartesian product of the domains: $S \leq \mathbf{X}\, D_i$. It is worth knowing whether local and global solutions are the same[7] so that $S = \mathbf{X}\, D_i$. In this case, all the global solutions can be found through local propagation, a simple technique that does not require a complete tree search[8]. However it has been proved that local propagation is not enough to find global solutions when $S \leq \mathbf{X}\, D_i$.

Many advanced algorithms[9] have been developed for constraint satisfaction as an evolution of the obvious backtracking. They are based on look-ahead and look-back strategies. All anticipate some tests on constraints to avoid many more tests later on, and save auxiliary information. The performance in reducing the number of constraint tests increases by a factor of 2.5 from the best to the worst algorithm, while the number of look-ups increases. For instance, the number of look-ups for the 8-queens problem is from 104 to 105. Remember that for pure backtracking it is 0.

For a general programming language the more advanced algorithms are quite inappropriate. If they use auxiliary memory they are limited, in practice, to unary and binary constraints because n-dimensional arrays are too expensive to manage. Moreover, all those algorithms work on instantiated values; it is easy to instantiate variables if they have finite associated domains, it is difficult for intensional constraints that usually start with unknown domains.

Another class of possibly significant algorithms is consistency checking[10]. Those algorithms check node consistency and path consistency so that constraint satisfaction operates on a reduced graph. Also, those algorithms work on unary and binary constraints. So the need for a solution strategy that is able to work efficiently for general-purpose applications, and on n-ary constraints, is obvious.

The Constrictor project emerged when we approached different problems in manufacturing and in assembly and we found the recurrent use of constraint satisfaction techniques. We developed a general-purpose tool to be used in all these different problems, small enough to require a very limited amount of memory and to run on a personal computer. Since many parts of the applications we had in mind can be programmed in algorithmic fashion, we wanted CSP techniques integrated into a traditional language. We chose LISP because it allows dynamic memory allocation and the best treatment of symbols.

We are much indebted to Consat[3]. We have adopted a very similar syntax and worked on a few new objectives: to obtain a portable system simply integrated into CommonLisp, to adopt only one strategy for satisfying the network, and to make some improvement by deleting constraints as soon as possible. Along the way we discovered that programming a constraint network is often hard. For this reason, we developed an integrated debugger that allows the animation through icons of the solution steps.

Pecos also is a similar system[11]. It is not a new language, rather it is a constraint library added to an object-oriented Le-Lisp environment. Pecos thus benefits directly from the entire base environment which, in this case, supports object-oriented, functional, structured, non-deterministic and graphics features. The solving mechanism for finite domain constraints is essentially based on arc consistency checking and generation[10], the one used for floating point constraints is based on interval calculus.

We have instead, as Consat, chosen to realize a language.

## 3. CONSTRAINT DEFINITION AND SOLUTION IN CONSTRICTOR

We managed definitions of constraints as 'extensional' and 'intensional', we added a solution procedure, and functions that allows some optimization by rearranging the order of evaluation of the constraints.

Constraint relations are described by the primitive defconstraint. All finite relations can be represented by enumerating their extensions, so listing all the tuples of the admissible values, under the *:tuples* keyword. Infinite relations are only described in an intensional style in a compound constraint.

Let us introduce the language through examples. A simple intensional constraint is the addition, where we have three interface variables (under the *:interface* keyword), and a *:relation* field, containing some *:pattern*, which are LISP expressions:

```
(defconstraint addition
  (:interface add1 add2 sum)
  (:relation
     (:pattern(add1 add2(+ add1 add2))
              (:if(constrained-p add1 add2))
  (:pattern (add1(- sum add1) sum)(:if (constrained-p add1 sum))
  (:pattern ((- sum add2) add2 sum)(:if (constrained-p add2 sum))))
```

The expression following :pattern are evaluated as expressions into a COND according to which variable in the interface is undefined (:unconstrained). If no *:pattern* is evaluated, the constraint is not satisfied and NIL is returned.

Here we see how to assign values to the interface variables

```
(satisfy addition :WITH '((add1 1)(add2 2)(sum 4))) => NIL
(satisfy addition :WITH '((add1 (1 2 3))(add2 (2 4))))
                    => ((1 2 3)(2 2 4)(3 2 5)(1 4 5)(2 4 6)(3 4 7))
```

It is impossible to evaluate a constraint if no variable has been instantiated. The keyword *:condition* is used to add conditions, for instance to delay the evaluation of a constraint. Those conditions can result in augmenting the efficiency of the constraint resolution.

Using our definition of addition

```
(satisfy addition :with '((add1 4)) => NIL
```

This result is not what expected. In fact it is impossible to compute the two values for add2 and sum, because they are infinitly many, but NIL (= no solutions) is not a good answer. We need to distinguish between NIL and UNCOMPUTABLE. To get those different answers we add a global computability condition in the definition of the constraint, after the :relation, to state that at least two of the variables should be assigned:

```
(:condition (< 1 (constrained-count add1 add2 sum)))
```

The condition is evaluated before the patterns; if a negative answer is produced *:UNCOMPUTABLE* is returned, as in the case

```
(satisfy addition :with'((add1 4)) => :UNCOMPUTABLE
```

Suppose now we want to describe the constraint double-sum, to work on five variables, so that C=A+B, E=C+D:

```
(defconstraint double-sum
  (:type compound)
  (:interface A B C D E)
  (:constraint-expressions (addition A B C) (addition C D E)))
```

The assignments of any three variables among A, B, D and E make the constraint computable. For instance:

```
(satisfy double-sum :with '((A 1)(B 2)(E 4))) => ((1.2.3.1.4))
(satisfy double-sum :with '((A 1)(D 1)(E 4))) => NIL
```

because the addition on A and B cannot be computed if both B and C are *:undefined*. If we add the same *:condition* as before:

```
(satisfy double-sum :with '((A 1)(D 1)(E 4))) => ((1.2.3.1.4))
```

Intensional constraints present some problems. For instance, errors in the expressions of the *:pattern* can have unpredictable effects. The reason is that errors can create non-monotonic constraints. In fact, two main requirements should be posted on our constraint system, as hereafter defined.

Let $C$ be a constraint on n variables, $D = \{D_1, D_2, \ldots D_n\}$ be the set of initial domains, and $S = \{S_1, S_2, \ldots S_k\} = C(D)$ be the set of solutions.

**Definition 3**
*$C$ is a* ***monotonic*** *constraint if, when evaluated for any $D_r = (D_{1r}, D_{2r}..D_{nr})$. $D_{jr}$ being a subset of $D_j$ for $1 \leq j \leq n$, the set of corresponding solutions $S_r$ is a subset of $S$ for every $D$.*

**Req. 1** $S_i = C(S_i)$ for each $S_i$ in $S$.
**Req. 2** All the constraints in the network must be monotonic.

All the enumerated constraints are monotonic. A compound constraint is monotonic if all the constraints inside are monotonic. For non-monotonic constraints the results are unpredictable. Reqs. 1 and 2 are sufficient conditions.

## 4. THE RESOLUTION ALGORITHM IN CONSTRICTOR

The use of constraints in Waltz[12] opens up another view of a constraint. A constraint so far is a relation on instantiated variables used to accept or reject some values. A constraint is also a *filter*, a relation on domains (sets of values) that filters the domains reducing their extension, under the monotonic assumption. In this sense, all the constraints are equivalent; also the 'qualitative' constraints that operate on infinite domains because they reduce those domains (initially represented by the keyword *:unconstrained*) to finite dimensions. This idea of reducing the domains while reducing the constraints to be satisfied is the basis of our constraint satisfaction method. The resolution algorithm of Constrictor had to be the kernel of a general-purpose and efficient system with minimum hardware requirements. For the first reason we could not use domain knowledge, and for the second it is important to save memory to reduce the calls to garbage collection. Moreover, a general-purpose language can use neither heuristics[13] nor domain knowledge to guide the search.

We started from LPB (Local Propagation and Backtracking) developed for Consat[3] and we added to it an elimination phase. So we named our algorithm LPBE (Local Propagation, Backtracking, and Elimination). LPB is a compromise between local propagation, used to propagate set of values, and backtracking, necessary to find all the global solutions. The steps are:

A: local propagation: all the constraints are activated and the domains are reduced;

B: the constraints whose domains have been modified are executed again, until the domains cannot be reduced any more;

C: If the result is not unique, choice points are sequentially set and local propagation continues until a solution is found. If one solution has been found save it and continue to D for backtracking to the last choice until no more branches are unexplored. If no choices are left the network is inconsistent.

D: backtracking: values are changed and local propagation continues from B.

We have observed that the condition in C is often too strict. Suppose that during local propagation a constraint $P$ is satisfied for any element of $C_t$, where $C_t$ is the Cartesian product of the sets of the domains of the $n$ variables at that time in the solution. Because all the constraints are monotonic, any other constraint can only reduce the domains of its variables. In practice, the constraint $P$ has terminated its influence on the solution, because it will be satisfied by any subset of its solution set. In fact, the Cartesian product of the domains reduced by the application of other constraints is a subset of $C_t$. In the previous algorithm, instead $P$ will be called again. *To reduce the number of constraints* in the network before exploring other branches is the idea of LPBE. The test on the cartesian product is too expensive, so we have found a simple condition to test to eliminate a constraint:

*TEST: After evaluation of the constraint P, if all the domains of its variables but one contain a single value, eliminate P.*

Motivation: Suppose $P$ is a n-ary relation. We have $n - 1$ variables instantiated and one variable associated with a domain of $k$ values. The solutions of that constraint are exactly the Cartesian product of the domains, which is $k$ combinations. If we eliminate $P$, all the other constraints in the network can only reduce the number of the $k$ combinations. In fact another constraint, $P'$, can possibly reduce the elements in the multiple set, so that some combinations are eliminated. If $P'$ reduces a domain containing a single value to a null set, the subtree is no longer explored anyway, because it cannot satisfy $P'$. This condition, that is similar to the look-ahead inference rule of Van Hentenryck[6], is useful because it reduces the number of tests near the leaves of the tree, and the number of leaves grows quickly with the number of variables and the looseness of constraints.

The main steps of LPBE in the solution are:

A: local propagation: initial domains are assigned to the variables of the network;

B: all the constraints are evaluated and the domains reduced;

C: the constraints whose domains have been modified are re-executed until no constraint can reduce the domains any more. Any constraint that satisfies the TEST is eliminated;

D: if all the constraints have been eliminated, the Cartesian product of the domains reduced by local propagation gives the global solutions. The execution terminates.

E: if the constraint list is not null, backtracking is used to instantiate in sequence all the values of the domains, and LPBE is recursively called from B.

In practice, the effort required to evaluate constraints is different for enumerated and for intensional constraints. For enumerated constraints, each tuple of the definition is checked against the initial domains of the variables. So the complexity of computation is dominated by a factor due to the number of tuples in the definition of the constraint, while the dimension of the initial domain is not so important. For intensional constraints, the evaluation is based on the evaluation of the pattern. Patterns can operate on single values, not on domains. The filter is applied on all the tuples obtained from the initial domains, and only the admissible tuples are returned. The complexity of this computation grows with the dimensions of the domains, and tends to grow with the Cartesian product of the domains.

To improve the efficiency in the case of intensional constraints, pre-compilation is used to compile the functions implicitly defined by the patterns. The evaluation of a pattern is done by the call to its corresponding compiled function.

The programmer may also improve the speed of execution by changing the order of evaluation of constraints. As discussed by Nudel[14], this order can have a significant impact on the complexity of the backtracking. Usually LPBE explores the list of constraints to be activated in sequence. If a cyclic strategy has been defined, when a constraint is successfully evaluated the search to satisfy the constraints on the modified variables is restarted from the beginning of the list, not from the next element of that list. Changing the activation order of constraints is useful mainly when constraints are sorted. In this case, the most complex constraints are delayed after the domains have been considerably reduced by the application of the simplest constraints.

## 5. NOTES ON IMPLEMENTATION AND DEBUGGING

Constrictor is functional so that constraint solutions are returned as values of the function SATISFY, the user interface to call LPBE. A full version of the interpreter and the debugger runs on the Apple Macintosh in Allegro CommonLisp. It takes 230 Kb of RAM (200 in Lisp Heap and 30 in Mac Heap). For more details, a short manual of Constrictor is given in Appendix 1.

Usually, we define constraints in the global context of the LISP interpreter. We can also define local constraints through the primitive constraint-let. The *:pattern* expressions that appear in the intensive constraints are translated into COND expressions and compiled during the execution of the defconstraint.

Satisfy is implemented as a macro. It is called with the name of the constraint to be satisfied and with optional parameters to indicate initial values, and to give instructions for how many solutions to find and how to print them. It is possible to define recursive constraints by inserting a call to satisfy in the definition of an intensional constraint.

Constrictor has been provided with a debugging facility oriented to the problems of constraint satisfaction techniques. In fact, it is difficult to understand how the domains are reduced along with the application of constraints and where possible bugs are. The debugger is fully integrated with the menus. When Constrictor is installed a new option appears in the sixth position on the top menu line of Allegro. Different menus pop-up from its opening.

We can get help on Constrictor; we can have a menu to set the special variables, and we can ask for a constraint trace of the chosen constraints.

When the constraint is activated, directly or through a compound constraint in which it is called, we see how the process evolves in an interactive window. Step and Autostep keys appears to start each step of execution of simple or compound constraints. In the upper part of the tracing window we see the icons describing the domains associated with the variable; in the lower part we see the icons representing the elementary constraints that belong to the compound constraint.

Three different icons indicate enumerated, intensional or compound constraints; three icons describe the associated domains(infinite (*:unconstrained* in Constrictor), finite, or with an assigned value). The animation shows the icons changing during the solution. When a domain is modified its icon appears with an inverted background. When a constraint is eliminated from the activation list, its icon is crossed. All the icons are active elements: we can select one to get all its information in the text field

## 6. EVALUATIONS AND CONCLUSIONS

Here we have approached CSP on the base of formal results on sequential algorithms. Work is under way to define a parallel algorithm to be implemented on a transputer architecture.

In recent years, much work has been done in CSP in the AI community. Our system aims to be a useful tool for developing solutions to such problems. It can be tailored from local propagation, that reduces the depth of the search tree, to backtracking, and it can be used in any conventional LISP program. Moreover, it has been tested on many classical combinatorial problems.

We have made a test to assess the real efficiency of LPBE on the classical n-queens problem (see the definitions in Constrictor for $n = 4$ in Appendix 2). In Figure 1 we see a comparison between different algorithms. The parameter we have chosen is the number of tests to get the solution, as indicated in Mackworth and Freuder[1]; in the same paper, a comparison between many algorithms, shows that backtracking requires the maximum number of tests from $n = 7$, while forward checking (with backmarking) is the best.

We found out that for $n \leq 6$ (for little domains of the variables) LPBE has a greater number of tests than other

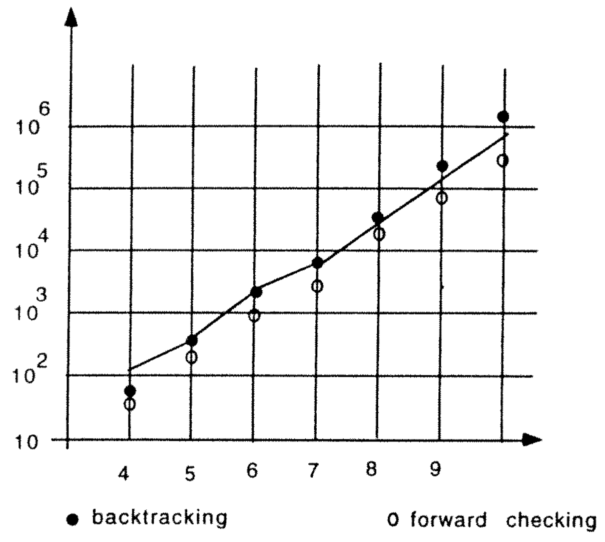

**Figure 1** The number of constraint checks for the N-queens problem (the line indicates the values of LPBE)

algorithms, while for large domains it shows performance better than looking ahead algorithms. If we consider the number of table lookups, that are of the same order of magnitude of constraints checks, we can conclude that the performance can be competitive with both forward checking and backmarking. In fact, LPBE does no table lookups at all.

## ACKNOWLEDGEMENTS

We thank the anonymous referees for their useful observations. This work has been partially supported by CNR through 'Progetto Finalizzato Robotica'.

## APPENDIX 1. CONSTRICTOR: A SHORT MANUAL

While loading Constrictor into LISP, two packages are created: the language interpreter, and the pre-compiled procedures corresponding to the definitions of constraints. A simple version of Constrictor runs on MS-DOS machines while the full version of the interpreter and the debugger runs on the Apple Macintosh Plus (or more), in Allegro CommonLisp. It takes 230 Kb of RAM (200 in Lisp Heap and 30 in Mac Heap).

***Defconstraint*** is the procedure used to define constraints. *(defconstraint const-decl$_1$, ... const-decl$_n$)* defines a constraint in the global context.

The basic keyword declaration is:

*(**:interface** var$_1$, var$_2$, ... var$_n$)*

Optional declarations are:

*(**:type** c-type)*, to declare the type either primitive (the default value), or compound

*(**:relation** rel-sequence)*, where rel-sequence is a sequence of ***:tuples*** for extensional constraints, a sequence of ***:pattern*** for intensional constraints;

*(**:constraint-expressions** constraint-calls)*, enumerates the

constraints in a compound constraint;

*(:condition global-condition)*, associates to intensional or compound constrains a condition for evaluation. The global condition is evaluated before the constraint is solved. Those conditions are essentially based on the predicates constrained-p, constrained-count, instanced-p, instanced-count.

We can also define local constraints:

***(constraint-let** (init-$l_1$, init-$l_2$, ... init-$l_n$) form$_1$ form$_2$ ... form$_k$)* defines the n constraints described by the init-lists and then sequentially evaluates the forms. Outside the lexical closure determined by the constraint-let the constraints are no more accessible.

***Satisfy*** is a macro. It is called with the name of the constraint to be satisfied and with the optional parameters.

*:with domain-list*, to associate initial values to variables. The value of domain-list is a kind of association list between names and values. Variables not included in :with are initially assigned :unconstrained.

***:assoc-solutions** boolean*, T to print the solutions as an association list.

***:count** n*, with n integer, to stop after finding the first n global solutions

***:extended-result** boolean*, T to print the solutions in the form of an association list of couples (variable-name.list of values). In case no global solution has been found it is possible to see the restrictions on the initial domains computed by the applied constraints.

***:locally** boolean*, T to find only locally consistent solutions. Note that it is possible to define recursive constraints by inserting a call to satisfy in the definition of an intensional constraint. In this case it is difficult to understand the solution and the debugger is not guaranteed to handle correctly any situation. The four predicates used in the :condition part are:

***(constrained-p** var$_1$, var$_2$, .. var$_n$)* ; returns T if all the variables have finite domains (no :unconstrained is found)

***(constrained-count** var$_1$, var$_2$, .. var$_n$)*; returns the number of the variables with finite domains

***(instanced-p** var$_1$, var$_2$, .. var$_n$)*; returns T if all the variables are instantiated (their domain contains one element)

***(instanced-count** var$_1$, var$_2$, .. var$_n$)*; returns the number of the instantiated variables.

Other primitives allow extracting information from the constraints. Among other utilities we mention:

***(constraint-sorted** constraint)*, returns T if sorted by autosort;

The overall behaviour of Constrictor is controlled by five special variables; in particular:

****compound-auto-sort****, sorts the constraints from unary, binary, to n-ary, and from extensive to intensive to compound. It is based on the idea that the initial application of simple constraints can considerably reduce the solution time by reducing the domains to be checked for more complex constraints.

Constraints can be protected from auto-sorting by the procedures ***compound-lock*** and ***compound-unlock***.

## APPENDIX 2: THE SOLUTION OF THE 4 QUEENS PROBLEM IN CONSTRICTOR

This is a simple case of the famous N-queens problem. Here we use the classical constraints definition, that is the best for N > 4, as discussed in Nadel[15]. Since we know that only one queen can be placed on a row, we define four variables:

$$Z = \{row1, row2, row3, row4\},$$

with domains

$$D_{zi} = \{1, 2, 3, 4\} \quad \text{for } 1 \le i \le 4$$

and we want to find for each row the position of the queen (i.e. the column of the chess board). Queens placed on the same row, the same column, or the same diagonal can attack each other. The first constraint is implicitly satisfied by our representation of the chess board, the last two constraints are:

$$C_{ij} = (z_i \neq z_j) \text{ and } (|z_i - z_j| \neq |i - j|) \qquad \text{for } 1 \le i < j \le 4$$

We can express the constraints by enumerating the admissible tuples, or by simbolic relations. We develop in detail three programs and discuss their performance. In the example, for n = 4, we have only the two solutions indicated in Figure 2.

**CASE 1: only enumerated constraints**

In this case we define the constraints between couples of rows: i and i + 1, i and i + 2, i and i + 3.

```
(defconstraint next-row
   (:interface row1 row2)
   (:relation (:tuple(1 3))(:tuple(1 4))(:tuple(2 4))
            (:tuple(3 1))(:tuple(4 1))(:tuple(4 2)) ))
 (defconstraint next2-row
   (:interface row1 row2)
   (:relation (:tuple(1 2))(:tuple(1 4))(:tuple(2 1))(:tuple (2 3))
            (:tuple(3 2))(:tuple (3 4))(:tuple(4 1))(:tuple (4 3)) ))
(defconstraint next3-row
   (:interface row1 row2)
   (:relation (:tuple(1 2))(:tuple(1 3))(:tuple(2 1))(:tuple(2 3))
            (:tuple(2 4))(:tuple(3 1))(:tuple(3 2))(:tuple (3 4))
            (:tuple (4 2)) (:tuple (4 3))  ))
```

These constraints are compounded in Four-queens:

```
(defconstraint Four-queens
   (:type compound)
   (:interface row1 row2 row3 row4 )
   (:constraint-expressions
         (next-row row1 row2) (next-row row2 row3) (next-
         row row3 row4)
         (next2-row row1 row3) (next2-row row2 row4)
         (next3-row row1 row4)))
```

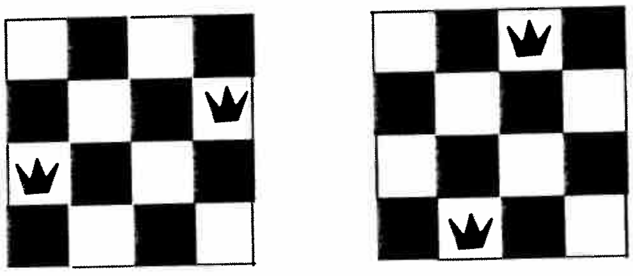

**Figure 2** (satisfy Four-queens) => ((2.4.1.3)(3.1.4.2))

The time taken by the solution in this case will be the reference time unit, 1.

**CASE 2: intensional constraints**
In this case we define the symbolic expressions that should be satisfied between couples of rows. We need also a unary constraint, row, to define the possible values for the rows.

```
(defconstraint row
   (:interface row)
   (:relation (:tuple (1)) (:tuple (2)) (:tuple (3)) (:tuple (4)) ))
```

The relations between couples of rows are so defined:

```
(defconstraint next-row
   (:interface row1 row2)
   (:relation (:pattern (row1 row2)
         :if(and(not(= row2 row1))
                  (not (= row2 (- row1 1)))
                  (not(= row2 (+ row1 1))))))
   (:condition (constrained-p row1 row2)))
 (defconstraint next2-row
   (:interface row1 row2)
   (:relation (:pattern (row1 row2)
         :if(and(not(= row2 row1))
                  (not (= row2 (- row1 2)))
                  (not(= row2 (+ row1 2))))))
   (:condition (constrained-p row1 row2)))
(defconstraint next3-row
   (:interface row1 row2)
   (:relation (:pattern (row1 row2)
         :if(and(not(= row2 row1))
                  (not (= row2 (- row1 3)))
                  (not(= row2 (+ row1 3))))))
   (:condition (constrained-p row1 row2)))
```

These constraints can be applied only to assigned values of the interface variables. They are organized together in a compound constraint:

```
(defconstraint Four-queens
   (:type compound)
   (:interface row1 row2 row3 row4 )
   (:constraint-expressions
    (row row1)(row row2)(row row3)(row row4)
    (next-row row1 row2) (next-row row2 row3) (next-row
    row3 row4)
    (next2-row row1 row3) (next2-row row2 row4) (next3-row
    row1 row4)))
```

The constraints are evaluated when both the variables are assigned a domain not :undefined. Here the execution time is reduced to 93% of the time in the previous example.

**CASE 3: intensional and delayed constraints**
We want to impose an order on the evaluation of constraints. The constraint 'row' is the same as in CASE 2. The other constraints are so modified:

```
(defconstraint next-row-blk
   (:interface row1 row2)
   (:relation (:pattern (row1 row2)
         :if(and(not(=row2 row1))
                  (not (=row2 (- row1 1)))
                  (not(=row2 (+ row1 1))))))
   (:condition (or (instanced-p row1) (instanced-p row2))))
```

The same :condition part is substituted in the relative constraints to get next2-row-blk and next3-row-blk; the new constraints are used in the expressions of the Four-queens-blk that has the same structure of the old one. The constraints are evaluated when at least one of the variables is instantiated. We get a time that is 83% of the CASE1.

If we activate the debugger, we can see that in CASE 2 the intensional constraints are evaluated from the first time: they are activated on two domains of four elements each, with 16 tests, and obtain six valid tuples. The backtracking is called to instantiate the domains and two global solutions are found. The intensional constraints have been activated 28 times., with 170 tests, of which only 78 are positive. In CASE 3 LPBE calls the backtracking because the relations are all uncomputable. After that any constraint is called on domains with at most four values and one value instantiated. Intensional constraints are activated 26 times, with 76 tests, of which 36 are positive. The same technique applied to the problem of 8 queens has saved 50% of time with respect to the CASE 2 solution.

This technique of delaying the application of intensional constraints can be developed using the debugger. In fact, we should always start with a general definition to find all the global solutions, because the operation of delaying some constraints can make uncomputable the network.

## REFERENCES

1 **Mackworth, A K and Freuder, E C** 'The complexity of some polynomial network consistency algorithms for constraint satisfaction problems', *Artificial Intelligence*, Vol 25 (1985) pp 65-74

2 **Sussman, G J and Steele, G L** 'CONSTRAINTS - a language for expressing almost-hierarchical descriptions', *Artificial Intelligence*, Vol 14 (1980) pp 1-39

3 **Guesgen, H W** 'A universal constraint programming language', *Proc. IJCAI-89*, Morgan Kaufman, San Mateo, CA (1989) pp 60-65

4 **Cohen, J** 'Constraint Logic Programming', *Commun. ACM*, Vol 33 N 7 (July 1990)

5 **Dincbas, M, Van Hentenryck, P, Simonis, H, Aggoun, A, Graf, T and Berthier, F** 'The constraint logic programming language CHIP', *Proc. Int. Conf. on Fifth Generation Computer Systems*, Tokyo, Japan (1988) pp 693-702

6 **Van Hentenryck, P** *Constraint satisfaction in logic programming*, MIT Press, Cambridge, MA (1989)

7 **Freuder, E C** 'A sufficient condition for backtrack-free search', *J. ACM*, Vol 29 (1982) pp 24-32

8 **Guesgen, H W and Hertzberg, J** 'Some fundamental properties of local constraint propagation', *Artificial Intelligence*, Vol 36 (1988) pp 237- 247

9 **Haralick R M and Elliot G L** 'Increasing tree search efficiency for constraints satisfaction problems', *Artificial Intelligence*, Vol 14 (1980) pp 263-313

10 **Mackworth, A K** 'Consistency in networks of relations', *Artificial Intelligence*, Vol 8 (1977) pp 99-118

11 **Puget, J F** 'Programmation par constraintes orienteé objects', *Proc. Avignon '92*, Avignon, France (1992)

12 **Waltz, D L** *Generating semantic descriptions from drawings of scenes with shadows*, AI-TR-271, MIT, Cambridge, MA (1972)

13 **Dechter, R** 'Network-based heuristics for constraint-satisfaction problems', *Artificial Intelligence*, Vol 34 (1988)

14 **Nudel, B** 'Solving the general consistent labeling (or constraint satisfaction) problem: two algorithms and their expected complexities', *Proc. AAAI83*, Washington DC (1983) pp 292-296

15 **Nadel, B A** 'Representation selection for constraint satisfaction: a case study using n-queens', *IEEE Expert* (June 1990) pp 16-23