# Integrating Constraints into an Imperative Programming Language

Krzysztof R. Apt
*CWI*
*P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*
and
*Dept. of Mathematics, Computer Science, Physics & Astronomy*
*University of Amsterdam, The Netherlands*

Andrea Schaerf
*Università di Roma "La Sapienza"*
*Dipartimento di Informatica e Sistemistica*
*via Salaria 113, 00198 Roma, Italy*

**Abstract**

We investigate here how to integrate constraints directly into the imperative programming paradigm, as "first class citizens" and not by means of an external library. To this end we provide a systematic account of the issues that arise. In particular, we discuss the role of the logical and customary variables, the interaction between the constraint store and the program, and the need for lists.

These considerations lead to an extension of an imperative programming language that supports declarative programming, called Alma-0, introduced in Apt, Brunekreef, Partington & Schaerf (1998). Constraint programs written in this style are simple to understand and analyze.

## 1 Introduction

Constraint programming in a nutshell consists of generating constraints (requirements) and solving them by general and domain specific methods. This approach to programming was successfully realized in a number of programming languages, notably constraint logic programming languages. As most of programming is being done in imperative style, it is natural to analyse in what way constraint programming can be realized within the imperative programming paradigm.

Up to now, the most successful approach to imperative constraint programming is the object-oriented approach taken by ILOG Solver (Puget & Leconte 1995, ILOG 1998). In this system constraints and variables are treated as objects and are defined within a C++ class library. Thanks to the class encapsulation mechanism and the operator overloading capability of C++, the user can see constraints almost as if they were a part of the language. A similar approach was independently taken in the *NeMo+* programming environment (Shvetsov, Telerman & Ushakov 1997).

In this paper we consider a different approach and investigate how to integrate constraints directly into the imperative programming paradigm, as "first class citizens" of the language. We believe that in this approach the interaction between the constraint engine and the program becomes more transparent and conceptually simpler. This leads to a very natural style of programming in which the constraint generation part of the program is often almost identical to the problem specification. In fact, the constraint programs presented here are, in our opinion, simpler and more natural than their counterparts written in the constraint logic programming style or in the imperative languages augmented with constraint libraries.

One of the difficulties is that constraints employ variables in the mathematical sense of the word (so *unknowns*) while the imperative programming paradigm is based on the computer science concept of a variable, so a *known*, but varying entity.

In contrast, the logic programming paradigm is based on the notion of a variable in the mathematical sense (usually called in this context a *logical* variable). This partly explains why the integration of constraints into logic programming such as in the case of CHIP (Van Hentenryck, Simonis & Dincbas 1992), Prolog III (Colmerauer 1990) and CLP($\mathcal{R}$) (see (Jaffar, Michaylov, Stuckey & Yap 1992)), to name just three examples, is very smooth and elegant.

Other difficulties have to do with the presence of local and global declarations and various parameter passing mechanisms, issues that are absent in the logic programming framework.

One of the basic tenets of programming with constraints is reliance on various search techniques. The presence of automatic backtracking in the logic programming framework is yet another reason why the addition of constraints to logic programming becomes so natural.

On the other hand, logic programming lacks the assignment, which is in our opinion needed in a number of common situations, as we illustrate by means of several examples. In our view, built-in facilities of logic programming, such as automatic backtracking, that simplify dealing with algorithmic problems that involve search are also most helpful when programming with constraints within the imperative programming framework. This was one of the motivations behind our work on the implementation of the imperative programming language Alma-0. The initial work on the design of this language was reported in Apt & Schaerf (1997); the final description of the language, its implementation and semantics is presented in Apt et al. (1998).

Alma-0 can be viewed as an intermediate stage in an attempt of incorporating constraints into an imperative programming language. It extends a subset of Modula-2 by just nine features the aim of which is to facilitate writing declarative programs, in particular programs that involve search. We extend here this line of research by proposing how to integrate constraints into Alma-0.

In Apt et al. (1998) we stated that our language proposal should be viewed as "an instance of a *generic method* for extending (essentially) any imperative programming language with facilities that encourage declarative programming." That is why we think that the proposal here discussed should be viewed not only as a suggestion how to integrate constraints into Alma-0, but more generally how to integrate constraints into any imperative language.

The paper is organized as follows. In Section 2 we summarize the new features of the language Alma-0. Then, in Section 3 we illustrate the basics of our proposal for adding constraints to Alma-0 and in Section 4 we explain how constraints interact with procedures. In turn in Section 5 we discuss language extensions for expressing complex constraints and for facilitating search in presence of constraints. Finally, in Section 6 we discuss related work and in Section 7 we draw some conclusions and discuss the future work.

## 2 Summary of Alma-0 Features

Alma-0 is an extension of a subset of Modula-2 that includes nine new features inspired by the logic programming paradigm. We briefly recall most of them here and refer to Apt et al. (1998) for a detailed presentation.

- Boolean expressions can be used as statements and vice versa. A boolean expression that is used as a statement and evaluates to `FALSE` is identified with a *failure*.

- *Choice points* can be created by the non-deterministic statements `ORELSE` and `SOME`. The former is a dual of the statement composition and the latter is a dual of the `FOR` statement. Upon failure the control returns to the most recent choice point, possibly within a procedure body, and the computation resumes with the next branch in the state in which the previous branch was entered.

- The created choice points can be erased or iterated over by means of the `COMMIT` and `FORALL` statements. `COMMIT S END` removes the choice points created during a successful execution of `S`. `FORALL S DO T END` iterates over all choice points created by `S`. Each time `S` succeeds, `T` is executed.

- The notion of *initialized* variable is introduced: A variable is uninitialized until the first time a value is assigned to it; from that point on, it is initilized. The `KNOWN` relation tests whether a variable of a simple type is initialized.

- The equality test is generalized to an assignment statement in case one side is an uninitialized variable and the other side an expression with known value.

We now introduce a search problem that we shall discuss throughout the paper. First we provide a solution in Alma-0, to illustrate its programming style.

**Problem 1** *Frequency Assignment* (Hale 1980). Given is a set of $n$ *cells*, $C := \{c_1, c_2, \ldots, c_n\}$ and a set of $m$ frequencies (or channels) $F := \{f_1, f_2, \ldots, f_m\}$. An *assignment* is a function which associates with each cell $c_i$ a frequency $x_i \in F$. The problem consists in finding an assignment that satisfies the following constraints.

**Separations:** Given $h$ and $k$ we call the value $d(f_h, f_k) = \ |h-k|$ the *distance* between two channels $f_h$ and $f_k$. (The assumption is that consecutive frequencies lie one unit apart.) Given is an $n \times n$ non-negative integer symmetric matrix $S$, called a *separation matrix*, such that each $s_{ij}$ represents the minimum distance between the frequencies assigned to the cells $c_i$ and $c_j$. That is, for all $i \in [1..n]$ and $j \in [1..n]$ it holds that $d(x_i, x_j) \geq s_{ij}$.

**Illegal channels:** Given is an $n \times m$ boolean matrix $F$ such that if $F_{ij} = true$, then the frequency $f_j$ cannot be assigned to the cell $i$, i.e., $x_i \neq f_j$.

Separation constraints prevent interference between cells which are located geographically closed and broadcast in each other's area of service. Illegal channels account for channels reserved for external uses (e.g., for military bases).

Here is the Alma-0 program that computes the number of solutions to this problem and prints all of them (in fact we tested this program on various data). We assume here for simplicity that each $c_i$ equals $i$ and each $f_i$ equals $i$, so $C = \{1, \ldots, n\}$ and $F = \{1, \ldots, m\}$.

```
MODULE FrequencyAssignment;
CONST N = 30;  (* number of cells *)
      M = 27;  (* number of frequencies *)

TYPE SeparationMatrix = ARRAY [1..N],[1..N] OF INTEGER;
     IllegalFrequencies = ARRAY [1..N],[1..M] OF BOOLEAN;
     Assignment = ARRAY [1..N] OF [1..M];  (* solution vector *)

VAR S: SeparationMatrix;
    F: IllegalFrequencies;
```

```
    A: Assignment;
    noSol: INTEGER;

PROCEDURE AssignFrequencies(S: SeparationMatrix; F: IllegalFrequencies;
                            VAR A: Assignment);
VAR i, j, k: INTEGER;
BEGIN
  FOR i := 1 TO N DO
    SOME j := 1 TO M DO (* j is a candidate frequency for cell i *)
      NOT F[i,j];
      FOR k := 1 TO i-1 DO
        abs(A[k] - j) >= S[k,i]
      END;
      A[i] = j
    END
  END
END AssignFrequencies;

BEGIN
  noSol := 0;
  InitializeData(S,F);
  FORALL
    AssignFrequencies(S,F,A);
  DO
    PrintSolution(A);
    noSol := noSol + 1
  END;
  WRITELN('Number of solutions is: ',noSol)
END FrequencyAssignment.
```

The simple code of the procedures `InitializeData` and `PrintSolution` is omitted.

# 3 Introducing Constraints

In what follows we discuss a proposal for adding constraints to Alma-0. We proceed in a number of steps.

In Subsection 3.1 we discuss the addition of constrained types and unknowns to the language and in Subsections 3.2 and 3.3 we define the constraint store and illustrate its interaction with the program execution. Then, in Subsection 3.4 we illustrate uses of constraints by means of two example programs.

To simplify our considerations we ignore in this section the presence of procedures. In particular, we assume for a while that all declarations are at one level.

## 3.1 Adding Constrained Types, Unknowns and Constraints

We start by adding a new kind of variables of simple types, called unknowns. This is done by using the qualifier `CONSTRAINED` in declarations of *simple types*, that is `INTEGER`, `BOOLEAN`, `REAL`, enumeration and subrange types.

**Definition 3.1**

- A type qualified with the keyword `CONSTRAINED` is called a *constrained type*.
- A variable whose type is a constrained type is called an *unknown*. □

We shall see in Subsection 4.1 that this way of defining unknowns simplifies the treatment of parameter passing in presence of unknowns. From now on we distinguish between variables and unknowns. In the discussion below we assume the following declarations.

```
CONST N = 8;
TYPE Board = ARRAY [1..N] OF CONSTRAINED [1..N];
     Colour = (blue, green, red, yellow);
     Info = RECORD
       co: Colour;
       No: CONSTRAINED INTEGER;
     END;
VAR i, j: INTEGER;
    a: ARRAY [1..N] of INTEGER;
    C: CONSTRAINED [1..N];
    X, Y: Board;
    Z: Info;
```

So `a`, `i` and `j` are variables while `C` is an unknown. In turn, `X` and `Y` are arrays of unknowns and `Z` is a record the first component of which is a variable and the second an unknown.

Because of the syntax of Alma-0, boolean expressions can appear both in the position of a statement and inside a condition.

**Definition 3.2** A *constraint* is a boolean expression that involves some unknowns and does not appear inside a condition. □

For example, in the program fragment

```
IF FOR i := 1 TO N-1 DO X[i] <= X[i+1] END
THEN Y[1] <> Y[N]
END
```

only `Y[1] <> Y[N]` is a constraint.

We postulate that the values of unknowns are determined only by means of constraints that are placed on them. This means that it is not allowed to use assignment to assign a value to an unknown. For instance, in presence of the above declarations the statements `X[1] := 0` and `C := 1` are illegal. In contrast, the constraints `X[1] = 0` and `C = 1` are legal.

There are no other restrictions concerning the occurrences of unknowns. In particular, the assignment `i := X[1] + X[2]` is a legal statement (which is not a constraint) and so is the statement `IF X[C] > 0 THEN k := 3 END`. A bit more subtly, the assignment `a[X[1]] := X[2]` is also syntactically legal and so is `a[C] := X[2]`. The point is that the last two assignments do not assign a value to an unknown.

Initially each unknown has an undetermined value that belongs to the domain associated with the type. By placing constraints on an unknown its domain can *shrink*. The unknown continues to have an undetermined value until the domain gets reduced to a singleton.

If the program control reaches an occurrence of an unknown outside of a constraint, this unknown is evaluated. If its value is at this moment undetermined, this evaluation yields a run-time error. If the value is determined (that is, the domain is a singleton), then it is substituted for the occurrence of the unknown. So the occurrences of an unknown outside of a constraint are treated as usual variables.

Note that during the program execution the domain of an unknown *monotonically* decreases with respect to the subset ordering. This is in stark contrast with the case of variables. Initially, the value of a variable of a simple type is not known but after the first assignment to it its value is determined though can *non-monotonically* change to any other value from its type.

Intuitively, a program is viewed as an "engine" that generates constraints. These constraints are gradually solved by means of the constraint solving process that we shall explain now.

Note that we do not allow constraints to appear within conditions. This restriction might be relaxed in some specific cases, leading to constructs such as the implication constraint of `cc(FD)` of Hentenryck, Saraswat & Deville (1995) or use of constraints in implications as in 2LP of McAloon & Tretkoff (1995).

### 3.2 Adding the Constraint Store

We now introduce the central notion of *constraint store*. This is done in a similar way as in the constraint logic programming systems, though we need to take into account here the presence of variables and constants.

**Definition 3.3** We call a constraint $C$ *evaluated* if each constant that occurs in it is replaced by its value and each variable (not unknown) that occurs in it is replaced by its current value. If some variable that occurs in $C$ is uninitialized, we say that the evaluation of $C$ *yields an error*. Otherwise we call the resulting boolean expression the *evaluated form* of $C$. □

So no variables occur in the evaluated form of a constraint. For technical reasons we also consider a *false constraint*, denoted by $\perp$, that can be generated only by a constraint solver to indicate contradiction.

**Definition 3.4** A *constraint store*, in short a *store*, is a set of evaluated forms of constraints. We say that an unknown is *present in the store* if it occurs in a constraint that belongs to the store.

We call a store *failed* if $\perp$ is present in it or if the domain of one of the unknowns present in it is empty. By a *solution* to the store we mean an assignment of values from the current domains to all unknowns present in it.

Further, we say that a constraint is *solved* if its evaluated form is satisfied by all combinations of values from the current domains of its unknowns. □

For example, in the program fragment

```
i := 1;
j := 2;
X[i] <= j;
Y[X[i+2]] <> Y[N];
```

we have two constraints, `X[i] <= j` and `Y[X[i+2]] <> Y[N]`. Here `X[1] <= 2` is the evaluated form of the first one, while `Y[X[3]] <> Y[8]` is the evaluated form of the second one. If we deleted the assignment `i := 1` the evaluations of both constraints would yield an error.

The notion of a failed store is a computationally tractable approximation of that of an inconsistent store, i.e., a store that has no solutions. Indeed, a failed store is inconsistent but an inconsistent store does not have to be failed: just consider {`X[1] = X[2], X[1] < X[2]`}.

### 3.3 Interaction between the Program and the Constraint Store

The program interacts with the store in the following two ways:

- By adding to it the evaluated forms of encountered constraints. If the evaluation of such a constraint yields an error, a run-time error arises.

- By generating possible values for unknowns that are present in the store by means of some built-in primitives to be introduced in Subsection 5.2.

The store is equipped with a number of procedures called *constraint solvers*. Their form depends on the applications. One or more of them can become activated upon addition of (an evaluated form of) a constraint to the store. An activation of constraint solvers, in the sequel called *constraint solving*, can reduce the domains of the unknowns, determine the values of some unknowns by reducing the corresponding domains to singletons, delete some constraints that are solved, or discover that the store is failed either by generating the false constraint $\perp$ or by reducing the domain of an unknown to the empty set.

We assume that constraint solving is a further unspecified process that depending of application may be some form of constraint propagation or a decision procedure. We require that the result of constraint solving maintains equivalence, which means that the set of all solutions to the store does not change by applying to it constraint solvers.

The store interacts with the program as follows.

**Definition 3.5** Upon addition of a constraint to the store, constraint solving takes place.

- If as a result of the constraint solving the store remains non-failed, the control returns to the program and the execution proceeds in the usual way.
- Otherwise the store becomes failed and a *failure* arises. This means that the control returns to the last choice point created in the program. Upon backtracking all the constraints added after the last choice point are retracted and the values of the variables and the domains of the unknowns are restored to their values at the moment that the last choice point was created. □

This means that we extend the notion of failure, originally introduced in Section 2, to deal with the presence of the store.

### 3.4 Examples

To illustrate use of the introduced concepts consider the following classical problem.

**Problem 2** *Eight Queens.* Place 8 queens on the chess board so that they do not attack each other.

We present here a solution that uses constraints. We only write the part of the program that generates constraints.

```
CONST N = 8;
TYPE Board = ARRAY [1..N] OF CONSTRAINED [1..N];
VAR i, j: [1..N];
    X: Board;

BEGIN
  FOR i := 1 TO N-1 DO
    FOR j := i+1 TO N DO
      X[i] <> X[j];
      X[i] <> X[j]+j-i;
      X[i] <> X[j]+i-j
    END
  END
END;
```

Each generated constraint is thus of the form `X[i] <> X[j]` or `X[i] <> X[j] + k` for some values `i,j` $\in$ `[1..N]` such that `i` $<$ `j` and `k` being either the value of `j-i` or of `i-j`.

Next, we present a solution to the *Frequency Assignment* problem (Problem 1) that uses constraints. Again, we only write the part of the program that generates constraints. We assume here that the variables `S` and `F` are properly initialized.

```
TYPE SeparationMatrix = ARRAY [1..N],[1..N] OF INTEGER;
     IllegalFrequencies = ARRAY [1..N],[1..M] OF BOOLEAN;
     Assignment = ARRAY [1..N] OF CONSTRAINED [1..M];
VAR S: SeparationMatrix;
    F: IllegalFrequencies;
    X: Assignment;
    i, j: INTEGER;

BEGIN
  FOR i := 1 TO N DO
    FOR j := 1 TO M DO
      IF F[i,j] THEN X[i] <> j END
    END
  END;
  FOR i := 1 TO N DO
    FOR j := 1 TO i-1 DO
      EITHER X[i] - X[j] >= S[i,j]
      ORELSE X[j] - X[i] >= S[i,j]
      END
    END
  END
END;
```

The use of the `ORELSE` statement creates here choice points to which the control can return if in the part of the program that deals with solving constraints a failed store is produced.

Alternatively, one could use here a disjunction and replace the `ORELSE` statement by

`(X[i] - X[j] >= S[j,i]) OR (X[j] - X[i] >= S[j,i]).`

In this case no choice points are created but the problem of solving (disjunctive) constraints is now "relegated" to the store.

The latter solution is preferred if the constraint solver in use is able to perform some form of preprocessing on disjunctive constraints, such as the *constructive disjunction* of Hentenryck et al. (1995). On the other hand, the former solution allows the programmer to retain control upon the choice generated by the system. For example, she/he can associate different actions to the two branches of the `ORELSE` statement.

## 4 Constraints and Procedures

So far we explained how the program interacts with the store in absence of procedures. In Alma-0 one level procedures are allowed. In presence of procedures we need to explain a number of issues: handling of constraints that involve local unknowns, handling of constraints that involve both local and global unknowns, and parameter passing for unknowns. Various choices arise here. Our solution aims at finding a balance between simplicity and effectiveness.

## 4.1 Procedures and Constraint Store

The scope rules that we enforce for unknowns are the same as for customary variables. In particular, an unknown declared as a local in a procedure is not visible outside the body of the procedure and its life cycle is confined to the execution of the procedure.

After the call of the procedure the local unknowns go out of scope. If no choice points are left after the execution of the procedure, all constraints that involve only local unknowns are deleted from the store. In contrast, each constraint that involves both local and global unknowns is transformed into a constraint on the global unknowns only by existentially quantifying over the local unknowns. Each quantification ranges over the final domain of the local unknown.

For example, consider the constraint `X = 2 * Z` in the following program fragment

```
VAR X: CONSTRAINED [1..N];

PROCEDURE P;
VAR Z: CONSTRAINED [1..N];
BEGIN
  ...
  X = 2 * Z;
END P;
```

When a call of the procedure `P` terminates, the local unknown `Z` goes out of scope and the constraint is transformed into the existential constraint $\exists$ `Z (X = Z * 2)` that involves only `X`. The quantification ranges over the final domain of `Z`. The domain of `X` is the one found at the exit of the call to `P`.

## 4.2 Parameter Passing

In presence of unknowns we need to explain how they can be passed as actual parameters. Formal parameters of constrained types are considered as unknowns. This means that it is not allowed to use assignment to assign a value to a formal parameter of a constrained type and analogously for the components of formal parameters of compound types which are of a constrained type.

We discuss first call by variable. An unknown (or a compound variable containing an unknown) passed as an actual variable parameter is handled in the same way as the customary variables, by means of the usual reference mechanism.

For example consider the following problem.

**Problem 3** Given is an array which assigns to each pixel on an $M \times N$ board a colour. A *region* is a maximal set of adjacent pixels that have the same colour. Determine the number of regions.

To solve it we represent each pixel as a record, one field of which holds the colour of the pixel and the other is an unknown integer. Then we assign to each pixel a number in such a way that pixels in the same region get the same number. These assignments are performed by means of constraint solving. For instance, in the case of Figure 1 the constraint solving takes care that the value 1 is assigned to all but two pixels once it is assigned to the leftmost uppermost pixel.

To achieve this effect in the program below we assume that the constraint solving process is able to reduce the domain of `y` to {`a`} given the constraint `x = y` and the fact that the domain of `x` equals {`a`}. The program uses both constraints and an assignment.

```
TYPE Colour = (blue, green, red, yellow);
     Info =  RECORD
       co: Colour;
```

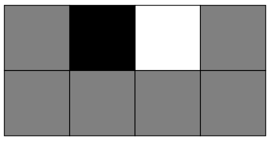

Figure 1: Constraint Solving and Pixels

```
      No: CONSTRAINED INTEGER;
    END;
    Board = ARRAY [1..M],[1..N] OF Info;

PROCEDURE Region(VAR X: Board; VAR number: INTEGER);
  VAR i, j, k: INTEGER;
BEGIN
  FOR i := 1 TO M DO
    FOR j := 1 TO N DO
      IF i < M AND X[i,j].co = X[i+1,j].co
      THEN X[i,j].No = X[i+1,j].No
      END;
      IF j < N AND X[i,j].co = X[i,j+1].co
      THEN X[i,j].No = X[i,j+1].No
      END
    END
  END;
  k := 0;
  FOR i := 1 TO M DO
    FOR j := 1 TO N DO
      IF NOT KNOWN(X[i,j].No)
      THEN k := k+1; X[i,j].No = k
      END
    END
  END;
  number = k
END Region;
```

Note that for any `i` in `[1..M]` and `j` in `[1..N]`, the record component `X[i,j].No` is of a constrained type. Here the first double `FOR` statement generates the constraints while the second double `FOR` statement solves them by assigning to the pixels that belong to the same region the same number.

Due to the call by variable mechanism, the actual parameter corresponding the formal one, `X`, is modified by the procedure. In particular, the second component, `No`, of each array element is instantiated after the procedure call.

The call by value mechanism in presence of unknown is more involved. We stipulate the following.

**Definition 4.1** An unknown passed as an actual value parameter (or as a field or an element of a compound actual value parameter) is *not* evaluated. Instead, its domain and the constraints attached to this unknown are *duplicated* for the formal parameter. □

For example suppose that in the `Region` procedure the parameter `X` is declared as a call by value parameter:

```
PROCEDURE Region(X: Board; VAR number: INTEGER);
```

In this case the full data structure `Board` of the actual parameter is copied into the *local* variable `X`, the constraints are applied to `X` and the assignment does not affect the original unknowns of the actual parameter.

For example, suppose we wish to check whether the number of regions is increased by changing the color of a given pixel. Assuming the call by value mechanism for the board `b` this can be done by the following program fragment

```
Region(b,r);
b[i,j].co := red;
Region(b,r1);
IF r1 > r
THEN ...
END;
```

As another example of the use of the call by value mechanism, consider a variant of the Frequency Assignment problem according to which we additionally wish to check whether there exists a part of the area with a frequency that can be dedicated for other uses. More specifically, we wish check whether there exists a region `k` in a given array `R` of regions and a frequency $f \in F$ such that for some assignment no cell in the region `k` uses $f$.

In the program fragment that deals with this problem we represent as before an assignment by an array of type `Assignment` declared by

```
TYPE Assignment = ARRAY [1..N] OF CONSTRAINED [1..M].
```

In turn, a region is defined as a set of cells represented by an array of `N` booleans. The original problem constraints are now stated using a procedure `AssignFrequency` the body of which consists of the program fragment presented in Subsection 3.4. This procedure generates the constraints on an array `X` of type `Assignment` that is passed by *variable*.

In contrast, the additional constraints are generated and solved by means of the procedure `ConstraintRegion` to which the assignment `X` is passed by *value*.

```
AssignFrequency(S,F,X);
SOME k := 1 TO Q DO
  ConstraintRegion(R[k],X)
END;
```

Here the variable `R` and the procedure `ConstraintRegion` are defined as follows:

```
TYPE Region = ARRAY [1..N] OF BOOLEAN;
     ArrayOfRegion = ARRAY [1..Q] OF Region;
VAR R: ArrayOfRegion;

PROCEDURE ConstraintRegion(R: Region; Y: Assignment);
VAR i: INTEGER;
    Z: CONSTRAINED [1..M];
BEGIN
  COMMIT
    FOR i := 1 TO N DO
      IF R[i] THEN Y[i] <> Z END
    END;
    INDOMAIN(Y)
  END
END ConstraintRegion;
```

`INDOMAIN` is a built-in that generates successive values for the unknowns in `Y` and is explained in Subsection 5.2.

Notice that thanks to the fact that in the call of `ConstraintRegion` the argument `X` is passed by value, the new constraints are stated on the local variable `Y` and no additional constraints are imposed on the original variable `X`.

# 5 Language Extensions

In this section we discuss some built-in procedures of the proposed language that make it easier for the user to program with constraints. In particular, in Subsection 5.1 we discuss built-ins for stating constraints, and in Subsection 5.2 we present built-ins for assigning values to unknowns.

## 5.1 Built-ins for Expressing Constraints

The practice of constraint programming requires inclusion in the programming language of a certain number of language built-ins that facilitate constraint formulation.

For example, if we wish to state that the unknowns of the array `X` must have pairwise different values, we write

```
ALL_DIFFERENT(X);
```

This call results in a constraint which is equivalent to the set of all the corresponding constraints of the form `X[i] <> X[j]`, for `i` $\in$`[1..N-1]` and `j` $\in$`[i+1..N]`. [1]

Similarly, if we wish to state that at most `k` among the unknowns belonging to the array `X` can have the value `v`, we write

```
AT_MOST(k,X,v);
```

This sort of built-ins on arrays are present in other imperative constraint languages. We do not list all of them here, but we envision their presence in the language.

Such built-ins on arrays are the counterparts in imperative languages of the corresponding built-ins on lists provided by constraint logic programming systems such as CHIP (Van Hentenryck et al. 1992). These languages also support symbolic manipulation of terms which makes it easy to generate arithmetic constraints. The traditional imperative programming languages lack this power and exclusive reliance on arrays can lead to artificial and inefficient solutions.

For example, suppose we are given an $n \times n$ matrix $A$ of integer unknowns and we wish to state the constraint that the sum of the elements of the main diagonal must be equal to a given value $b$. A customary solution would involve resorting to an auxiliary array of unknowns in the following way.

```
VAR A: ARRAY [1..N], [1..N] OF CONSTRAINED INTEGER;
    V: ARRAY [1..N] OF CONSTRAINED INTEGER;
    b: INTEGER;

   V[1] = A[1,1];
   FOR i := 2 to N DO
      V[i] = A[i,i] + V[i-1];
   END;
   V[N] = b;
```

[1] In some systems, such a constraint is kept in its original form in order to exploit constraint propagation techniques that deal specifically with constraints of this kind (Regin 1994).

This solution, which one would write for example in ILOG Solver, has the obvious drawback of creating N new unknowns for stating one single constraint.

Therefore we propose to use of lists of unknowns (as done for example in the ICON programming language of Griswold & Griswold (1983) for the case of variables), identified by the keyword LIST, upon which constraints of various forms can be stated by means of built-ins. The above program fragment would then be replaced by

```
VAR A: ARRAY [1..N], [1..N] OF CONSTRAINED INTEGER;
    L: LIST OF CONSTRAINED INTEGER;
    b: INTEGER;

   Empty(L);
   FOR i := 1 to N DO
     Insert(L, A[i,i])
   END;
   Sum(L,'=',b);
```

where Sum is a built-in with the expected meaning. Once the constraint Sum(L,'=',b) has been added to the store, the variable L can be used again for a different purpose. Note that in this solution no additional unknowns are created. In order to obtain a similar behaviour in ILOG Solver one needs either to add a similar built-in to it or to make explicit use of pointers to objects representing unknowns.

Consider now again the Frequency Assignment problem. We discuss here the formulation of an additional constraint for this problem which requires the use of lists. Suppose that we wish to state that in a particular region (i.e., a set of cells) a given frequency is used no more than a given number of times.

This type of constraint is useful in real cases. In fact, in some situations even though the pairwise interference among cells is below a given threshold and no separation is required, the simultaneous use of a given frequency in many cells can create a interference phenomenon, called *cumulative interference*.

The following procedure states the constraints for preventing cumulative interference in region R. Here max is the maximum number of cells in the region that can use the same frequency.

```
PROCEDURE RegionConstraint(R: Region; max: INTEGER; VAR X: Assignment);
VAR i, k: INTEGER;
    L: LIST OF CONSTRAINED [1..M];

BEGIN
  FOR k := 1 TO M DO
    Empty(L);
    FOR i := 1 TO N DO
      IF R[i] THEN Insert(L,X[i]) END
    END;
    AT_MOST(max,L,k)
  END
END RegionConstraint;
```

## 5.2 Built-ins for Assigning Values

In order to search for a solution of a set of constraints, values must be assigned to unknowns. We define the built-in procedure INDOMAIN which gets an unknown of a finite type (so BOOLEAN, enumeration or a subrange type) as a parameter, and assigns to it *one* among the elements of its domain. The procedure also creates a choice point and all other elements of the domain are successively assigned to the unknown upon backtracking.

The choice of the value to assign to the unknown is taken by the system depending on the current state of the store, based on predefined *value selection* strategies. We do not discuss the issue of which is the best value selection strategy. We only assume that all consistent values are eventually generated, and that the choice point is erased after the last value has been generated.

The procedure `INDOMAIN` can be also used on arrays and on lists. For example, the call `INDOMAIN(A)`, where `A` is a matrix of integer unknowns, generates (upon backtracking) all possible assignments for all elements of `A`.

The order of instantiation of the elements of `A` is taken care of by the store, which applies built-in strategies to optimize the retrieval of the first instantiation of the unknowns. As in the case of value selection, we do not discuss the issue of the *variable ordering* in this paper.

## 6 Related Work

As already mentioned in the introduction, the most successful imperative constraint language is the C++ library ILOG Solver (ILOG 1998). The main difference between our proposal and ILOG Solver is that the latter is based on the conventional imperative language C++ and consequently it does not support automatic backtracking. Therefore the interaction with the store cannot be based on failures issued by the store constraint solvers while evaluating the statements. In ILOG Solver such an interaction is always explicit, whereas in our proposal we aim at making it transparent to the user.

We are aware of two other language proposals in which constraints are integrated into an imperative language — the commercial language CHARME of Oplobedu, Marcovitch & Tourbier (1989) and 2LP of McAloon & Tretkoff (1995). In each language some of the issues here discussed have been addressed, but not all of them.

More specifically, in CHARME unknowns (called logical variables) and linear constraints on them are allowed. The language supports use of Prolog-like terms, arrays and sequences of logical variables and a number of features (like demons and the `element` primitive, an equivalent of `INDOMAIN`) adopted from the CHIP language. Also, it provides a nondeterministic `or` statement and iterations over finite domains, arrays and sequences of logical variables.

The C like syntax creates an impression that CHARME supports the imperative programming. However, from the paper it is not clear whether it is actually the case. If it is, then it is not clear how the logical variables, constraints and nondeterministic statements interact with the usual features of the underlying imperative language. CHARME does provide bidirectional connection with C.

In 2LP unknowns (called *continuous* variables) are global. They vary over the real interval $[0, +\infty)$ and can be either simple ones or arrays. The only way these variables can be modified is by imposing linear constraints on them. Constraints can also appear in conditions. This leads to a conditional way of adding them to the store.

Whenever a constraint is added to the store, its feasibility w.r.t. the old constraints is tested by means of an internal simplex-based algorithm. This algorithm maintains the current feasible region, which is a polyhedron, together with a *witness point* which is a distinguished vertex.

The continuous variables can appear outside of the constraints as arguments of any procedure whose signature has a continuous variable, and as arguments to some predeclared functions like `wp` that returns the value of a witness point. In the latter case when a continuous variable is passed as a parameter, the witness point value is used.

2LP provides the nondeterministic statements analogous to the `ORELSE` and `SOME` statements of Alma-0 and a limited form for the FORALL statement. Automatic backtracking over assignment and combination of continuous and customary variables in compound variables is not supported.

## 7 Conclusions and Future Work

In this paper we proposed a way to integrate constraint programming features into an imperative language. To facilitate dealing with search techniques we envisage to use as the underlying language a language that supports declarative programming, such as the language Alma-0 described in Apt et al. (1998), rather than a conventional imperative language.

We discussed some of the issues related to the proposed integration, such as the interaction between the program and the constraint store, the local and global unknowns, and the parameter passing mechanisms. Finally, we presented some examples that illustrate the resulting style of programming.

In our future work we plan to extend the work carried out in Apt et al. (1998) to the language proposal here outlined. More specifically, we envisage to

- extend the executable, operational semantics based on the ASF+SDF Meta-Environment of Klint (1993);
- extend both the Alma-0 compiler and its underlying abstract machine AAA;
- implement a set of constraint solvers or provide an interface between the language and existing constraint solvers.

In the absence of procedures in the language the first item can be easily dealt with by adding to the executable semantics of Alma-0 given in Apt et al. (1998) a few rules that formalize the interaction between the program and the store stipulated in Subsection 3.3. These rules are parameterized by the constraint solvers attached to the store. Also, we do not envisage any difficulties in extending the Alma-0 compiler to deal with constraints that involve global unknowns only.

The presence of procedures makes these two tasks more difficult. The most complex issues involve the enforcement of the scope rules for unknowns, and the management of the parameter passing mechanism.

Regarding the last item, we plan to develop a simple solver for finite domain constraints to be use for prototyping and testing purposes. We also plan to exploit more powerful external solvers already available for subsequent releases of the system.

Due to lack of space we did not deal here with some of the issues related to the design of the language. Specifically, we omitted discussion of

- a full set of built-ins,
- primitives for selecting variable and value selection strategies,
- the language support for the dynamic creation of unknowns.

These can be taken care of in a systematic way and lead to a complete and rigorous definition of an imperative constraint programming language.

## Acknowledgements

We would like to thank Eric Monfroy and Vincent Partington for useful discussions on the subject of this paper.

## References


Apt, K. R. & Schaerf, A. (1997), Search and imperative programming, *in* 'Proc. 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)', ACM Press, pp. 67–79.

Apt, K. R., Brunekreef, J., Partington, V. & Schaerf, A. (1998), 'Alma-0: An imperative language that supports declarative programming', *ACM Toplas*. In press. Available via `http://www.cwi.nl/~apt`.

Colmerauer, A. (1990), 'An introduction to Prolog III', *Communications of ACM* **33**(7), 69–90.

Griswold, R. E. & Griswold, M. T. (1983), *The Icon Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.

Hale, W. K. (1980), Frequency assignment: Theory and applications, *in* 'Proc. of IEEE', pp. 1497–1514.

Hentenryck, P. V., Saraswat, V. & Deville, Y. (1995), Design, implementation, and evaluation of the constraint language cc(FD), *in* A. Podelski, ed., 'Constraint Programming: Basics and Trends', LNCS 910, Springer-Verlag. (Châtillon-sur-Seine Spring School, France, May 1994).

ILOG (1998), 'ILOG optimization suite — white paper', Available via `http://www.ilog.com`.

Jaffar, J., Michaylov, S., Stuckey, P. J. & Yap, R. H. C. (1992), 'The CLP($\mathcal{R}$) language and system', *ACM Transactions on Programming Languages and Systems (TOPLAS)* **14**(3), 339–395.

Klint, P. (1993), 'A meta–environment for generating programming environments', *ACM Transactions on Software Engineering and Methodology* **2**(2), 176–201.

McAloon, K. & Tretkoff, C. (1995), 2LP: Linear programming and logic programming, *in* P. Van Hentenryck & V. Saraswat, eds, 'Principles and Practice of Constraint Programming', MIT Press, pp. 101–116.

Oplobedu, A., Marcovitch, J. & Tourbier, Y. (1989), CHARME: Un langage industriel de programmation par contraintes, illustré par une application chez Renault, *in* 'Ninth International Workshop on Expert Systems and their Applications: General Conference, Volume 1', EC2, Avignon, France, pp. 55–70.

Puget, J.-F. & Leconte, M. (1995), Beyond the glass box: Constraints as objects, *in* 'Proc. of the 1995 International Symposium on Logic Programming', pp. 513–527.

Regin, J.-C. (1994), A filtering algorithm for constraints of difference in CSPs, *in* 'AAAI-94: Proceedings of the 12th National Conference on Artificial Intelligence', pp. 362–367.

Shvetsov, I., Telerman, V. & Ushakov, D. (1997), *NeMo+* : Object-oriented constraint programming environment based on subdefinite models, *in* G. Smolka, ed., 'Artificial Intelligence and Symbolic Mathematical Computations', Lecture Notes in Computer Science, vol. 1330, Springer-Verlag, Berlin, pp. 534–548.

Van Hentenryck, P., Simonis, H. & Dincbas, M. (1992), 'Constraint satisfaction using constraint logic programming', *Artificial Intelligence* **58**, 113–159.