The Alma Project, or How First-Order Logic Can Help Us in Imperative Programming

Krzysztof R. Apt[1,2] and Andrea Schaerf[3]

[1] CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
K.R.Apt@cwi.nl
[2] Dept. of Mathematics, Computer Science, Physics & Astronomy
University of Amsterdam, The Netherlands
[3] Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica
Università di Udine
via delle Scienze 208, I-33100 Udine, Italy
schaerf@uniud.it

Abstract. The aim of the Alma project is the design of a strongly typed constraint programming language that combines the advantages of logic and imperative programming.
The first stage of the project was the design and implementation of Alma-0, a small programming language that provides a support for declarative programming within the imperative programming framework. It is obtained by extending a subset of Modula-2 by a small number of features inspired by the logic programming paradigm.
In this paper we discuss the rationale for the design of Alma-0, the benefits of the resulting hybrid programming framework, and the current work on adding constraint processing capabilities to the language. 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.

1 Introduction

1.1 Background on Designing Programming Languages

The design of programming languages is one of the most hotly debated topics in computer science. Such debates are often pretty chaotic because of the lack of universally approved criteria for evaluating programming languages. In fact, the success or failure of a language proposal often does not say much about the language itself but rather about such accompanying factors as: the quality and portability of the implementation, the possibility of linking the language with the currently reigning programming language standard (for instance, C), the existing support within the industry, presence of an attractive development environment, the availability on the most popular platforms, etc.

The presence of these factors often blurs the situation because in evaluating a language proposal one often employs, usually implicitly, an argument that the "market" will eventually pick up the best product. Such a reasoning would be correct if the market forces in computing were driven by the desire to improve the quality of programming. But from an economic point of view such aspects as compatibility and universal availability are far more important than quality.

Having this in mind we would like to put the above factors in a proper perspective and instead concentrate on the criteria that have been used in academia and which appeal directly to one of the primary purposes for which a programming language is created, namely, to support an implementation of the algorithms. In what follows we concentrate on the subject of "general purpose" programming languages, so the ones that are supposed to be used for developing software, and for teaching programming.

Ever since Algol-60 it became clear that such programming languages should be "high-level" in that they should have a sufficiently rich repertoire of control structures. Ever since C and Pascal it became clear that such programming languages should also have a sufficiently rich repertoire of data structures.

But even these seemingly obvious opinions are not universally accepted as can be witnessed by the continuing debate between the followers of imperative programming and of declarative programming. In fact, in logic programming languages, such as Prolog, a support for just one data type, the lists, is provided and the essence of declarative programming as embodied in logic and functional programming lies in not using assignment.

Another two criteria often advanced in the academia are that the programming language should have a "simple" semantics and that the programs should be "easy" to write, read and verify. What is "simple" and what is "easy" is in the eyes of the beholder, but both criteria can be used to compare simplicity of various programming constructs and can be used for example to argue against the `goto` statement or pointers.

In this paper we argue that these last two criteria can be realized by basing a programming language on first-order logic. The point is that first-order logic is a simple and elegant formalism with a clear semantics. From all introduced formalisms (apart from the propositional logic that is too simplistic for programming purposes) it is the one that we understand best, both in terms of its syntax and its semantics. Consequently, its use should facilitate program development, verification and understanding.

One could argue that logic programming has realized this approach to computing as it is based on Horn clauses that are special types of first-order formulas. However, in logic programming in its original setting computing (implicitly) takes place over the domain of terms. This domain is not sufficient for programming purposes. Therefore in Prolog, the most widely used logic programming language, programs are augmented with some support for arithmetic. This leads to a framework in which the logical basis is partly lost due to the possibility of errors. For instance, Prolog's assignment statement `X is t` yields a run-time error if `t` is not a ground arithmetic expression.

This and other deficiencies of Prolog led to the rise of constraint logic programming languages that overcome some of Prolog's shortcomings. These programming languages depend in essential way on some features as the presence of constraint solvers (for example a package for linear programming) and constraint propagation. So this extension of logic programming goes beyond first-order logic.

It is also useful to reflect on other limitations of these two formalisms. Both logic programming and constraint logic programming languages rely heavily on recursion and the more elementary and easier to understand concept of iteration is not available as a primitive. Further, types are absent. They can be added to the logic programming paradigm and in fact a number of successful proposals have been made, see, e.g., [15]. But to our knowledge no successful proposal dealing with addition of types to constraint logic programs is available.

Another, admittedly debatable, issue is assignment, shunned in logic programming and constraint logic programming because its use destroys the declarative interpretation of a program as a formula. However, we find that assignment *is* a useful construct. Some uses of it, such as recording the initial value of a variable or counting the number of bounded iterations, can be replaced by conceptually simpler constructs but some other uses of it such as for counting or for recording purposes are much less natural when simulated using logic formulas.

1.2 Design Decisions

These considerations have led us to a design of a programming language Alma-0. The initial work on the design of this language was reported in [3]; the final description of the language, its implementation and semantics is presented in [2].

In a nutshell, Alma-0 has the following characteristics:

- it is an extension of a subset of Modula-2 that includes assignment, so it is a strongly typed imperative language;
- to record the initial value of a variable the equality can be used;
- it supports so-called "don't know" nondeterminism by providing a possibility of a creation of choice points and automatic backtracking;
- it provides two forms of bounded iterations.

The last two features allow us to dispense with many uses of recursion that are in our opinion difficult to understand and to reason about.

As we shall see, the resulting language proposal makes programming in an imperative style easier and it facilitates (possibly automated) program verification. Additionally, for several algorithmic problems the solutions offered by Alma-0 is substantially simpler than the one offered by the logic programming paradigm.

The following simple example can help to understand what we mean by saying that Alma-0 is based on first-order logic and that some Alma-0 programs are simpler than their imperative and logic programming counterparts.

Consider the procedure that tests whether an array `a[1..n]` is ordered. The customary way to write it in Modula-2 is:

```
i:= 1;
ordered := TRUE;
WHILE i < n AND ordered DO
  ordered := ordered AND (a[i] <= a[i+1])
  i := i+1
END;
```

In Alma-0 we can just write:

```
ordered := FOR i:= 1 TO n-1 DO a[i] <= a[i+1] END
```

This is much simpler and as efficient. In fact, this use of the `FOR` statement corresponds to the bounded universal quantification and the above one line program equals the problem specification.

In the logic programming framework there are no arrays. But the related problem of finding whether a list `L` is ordered is solved by the following program which is certainly more involved than the above one line of Alma-0 code:

```
ordered([]).
ordered([X]).
ordered([X, Y | Xs]) :- X =< Y, ordered([Y| Xs]).
```

1.3 Towards an Imperative Constraint Programming Language

In Alma-0 each variable is originally *uninitialized* and needs to be initialized before being used. Otherwise a run-time error arises. The use of uninitialized variables makes it possible to use a single program for a number of purposes, such as computing a solution, completing a partial solution, and testing a candidate solution. On the other hand, it also provides a limitation on the resulting programming style as several first-order formulas, when translated to Alma-0 syntax, yield programs that terminate in a run-time error.

With the addition of constraints this complication would be overcome. The idea is that the constraints encountered during the program execution are moved to the constraint store and evaluated later, when more information is available. Then the above restriction that each variable has to be initialized before being used can be lifted, at least for the variables that are manipulated by means of constraints. Additionally, more programs can be written in a declarative way. In fact, as we shall see, an addition of constraints to Alma-0 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.

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.

Up to now, the most successful approach to imperative constraint programming is the object-oriented approach taken by ILOG Solver (see [16], [9]). 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 of [18].

In our approach constraints are integrated into the imperative programming paradigm, as "first class citizens" of the language. The interaction between the constraint store and the program becomes then more transparent and conceptually simpler and the resulting constraint programs are in our opinion more natural than their counterparts written in the constraint logic programming style or in the imperative languages augmented with constraint libraries.

The reason for this in the case of constraint logic programming is that many uses of recursion and lists can be replaced by the more basic concepts of bounded iteration and arrays. In the case of the imperative languages with constraint libraries, due to the absence of non-determinism in the language, failure situations (arising due to inconsistent constraints) must be dealt with explicitly by the programmer, whereas in Alma-0 they are managed implicitly by the backtracking mechanism.

When adding constraints to a strongly typed imperative programming language one needs to resolve a number of issues. First, 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. We wish to separate between these two uses of variables because we want to manipulate unknowns only by means of constraints imposed on them. This precludes the modelling of unknowns by means of uninitalized variables since the latter can be modified by means of an assignment.

Second, one needs to integrate the constraints in such a way that various features of the underlying language such as use of local and global declarations and of various parameter passing mechanisms retain their coherence.

Additionally, one has to maintain the strong typing discipline according to which each variable has a type associated with it in such a way that throughout the program execution only values from its type can be assigned to the variable. Finally, one needs to provide an adequate support for search, one of the main aspects of constraint programming.

So the situation is quite different than in the case of the logic programming framework. Namely, 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 greatly facilitates the addition of constraints and partly explains why the integration of constraints into logic programming such as in the case of CHIP (see [19]), Prolog III (see [4]) and CLP($\mathcal{R}$) (see [11]), to name just three examples, has been so smooth and elegant. Further, logic programming languages provide support for automatic backtracking.

However, as already mentioned, in constraint logic programming languages types are not available. Moreover, there is a very limited support for scoping and only one parameter mechanism is available.

Let us return now to Alma-0. The language already provides a support for search by means of automatic backtracking. This support is further enhanced in our proposal by providing a built-in constraint propagation. In [2] 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 strongly typed imperative language. In fact, Alma-0 can be viewed as an intermediate stage in such an integration.

The remainder of the paper is organized as follows. In Section 2 we summarize the new features of Alma-0 and in Section 3 we illustrate the resulting programming style by two examples. Then, in Section 4 we discuss the basics of our proposal for adding constraints to Alma-0 and in Section 5 we explain how constraints interact with procedures. In turn in Section 6 we discuss language extensions for expressing complex constraints and for facilitating search in presence of constraints. Finally, in Section 7 we discuss related work and in Section 8 we draw some conclusions and discuss the future work.

2 A Short Overview of Alma-0

Alma-0 is an extension of a subset of Modula-2 by nine new features inspired by the logic programming paradigm. We briefly recall most of them here and refer to [2] for a detailed presentation.

- Boolean expressions can be used as statements and vice versa. This feature of Alma-0 is illustrated by the above one line program of Subsection 1.2.
  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 the first 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.

– In Alma-0 three types of parameter mechanisms are allowed: call by value, call by variable and *call by mixed form*. The first two are those of Pascal and Modula-2; the third one is an amalgamation of the first two (see [2]). Parameters passed by mixed form can be used both for testing and for computing.

Let us summarize these features of Alma-0 by clarifying which of them are based on first-order logic.

In the logical reading of the programming language constructs the program composition `S; T` is viewed as the conjunction $\mathtt{S} \wedge \mathtt{T}$. A dual of ";", the `EITHER S ORELSE T END` statement, corresponds then to the disjunction $\mathtt{S} \vee \mathtt{T}$.

Further, the `FOR i:= s TO t DO S END` statement is viewed as the bounded universal quantification, $\forall \mathtt{i} \in [\mathtt{s}..\mathtt{t}]\, \mathtt{S}$, and its dual, the `SOME i:= s TO t DO S END` statement is viewed as the bounded existential quantification, $\exists \mathtt{i} \in [\mathtt{s}..\mathtt{t}]\, \mathtt{S}$.

In turn, the `FORALL S DO T END` statement can be viewed as the restricted quantification $\forall \bar{\mathtt{x}}(\mathtt{S} \rightarrow \mathtt{T})$, where $\bar{\mathtt{x}}$ are all the variables of `S`.

Because the boolean expressions are identified with the statements, we can apply the negation connective, `NOT`, to the statements. Finally, the equality can be interpreted both as a test and as an one-time assignment, depending on whether the variable in question is initialized or not.

3 Programming in Alma-0

To illustrate the above features of Alma-0 and the resulting programming style we now consider two examples.

3.1 The Frequency Assignment Problem

The first problem we discuss is a combinatorial problem from telecommunication.

Problem 1. Frequency Assignment ([7]). 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 $F_{ij} = \mathit{true}$, 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 close and which broadcast in each other's area of service. Illegal channels account for channels reserved for external uses (e.g., for military bases).

The Alma-0 solution to this problem does not use an assignment and has a dual interpretation as a formula. 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
  InitializeData(S,F);
  AssignFrequencies(S,F,A);
  PrintSolution(A)
END FrequencyAssignment.
```

The simple code of the procedures `InitializeData` and `PrintSolution` is omitted. The generalized equality `A[i] = j` serves here as an assignment and the `SOME` statement takes care of automatic backtracking in the search for the right frequency `j`.

In the second part of the paper we shall discuss an alternative solution to this problem using constraints.

3.2 Job Shop Scheduling

The second problem we discuss is a classical scheduling problem, namely the *job shop scheduling* problem. We refer to [5, page 242] for its precise description. Roughly speaking, the problem consists of scheduling over time a set of jobs, each consisting of a set of consecutive tasks, on a set of processors.

The input data is represented by an array of jobs, each element of which is a record that stores the number of the tasks and the array of tasks. In turn, each task is represented by the machine it uses and by its length. The output is delivered as an integer matrix that (like a so-called Gantt chart) for each time point k and each processor p stores the job number that p is serving at the time point k.

The constraint that each processor can perform only one job at a time is enforced by using generalized equality on the elements of the output matrix. More precisely, whenever job i requires processor j for a given time window $[d_1, d_2]$, the program attempts for some k to initialize the elements of the matrix $(j, k + d_1), (j, k + d_1 + 1), \ldots, (j, k + d_2)$ to the value i. If this initialization succeeds, the program continues with the next task. Otherwise some element in this segment is already initialized, i.e., in this segment processor j is already used by another job. In this case the execution fails and through backtracking the next value for k is chosen.

The constraint that the tasks of the same job must be executed in the provided order and cannot overlap in time is enforced by the use of the variable `min_start_time` which, for each job, initially equals 1 and then is set to the end time of the last considered task of the job. To perform this update we exploit the fact that when the `SOME` statement is exited its index variable `k` equals the smallest value in the considered range for which the computation does not fail (as explained in [2]).

We provide here the procedure that performs the scheduling. For the sake of brevity the rest of the program is omitted.

```
TYPE
   TaskType      = RECORD
                      machine : INTEGER;
                      length  : INTEGER;
                   END;
   JobType       = RECORD
                      tasks : INTEGER;
                      task  : ARRAY [1..MAX_TASKS] OF TaskType
                   END;
   JobVectorType = ARRAY [1..MAX_JOBS] OF JobType;
   GanttType     = ARRAY [1..MAX_MACHINES],[1..MAX_DEADLINE] OF INTEGER;

PROCEDURE JobShopScheduling(VAR job: JobVectorType; deadline:INTEGER;
                            jobs :INTEGER; VAR gantt: GanttType);
VAR
   i, j, k, h     : INTEGER;
   min_start_time : INTEGER;
```

```
BEGIN
  FOR i := 1 TO jobs DO
    min_start_time := 1;
    FOR j := 1 TO job[i].tasks DO
      SOME k := min_start_time TO deadline - job[i].task[j].length + 1
DO
        (* job i engages the processor needed for task j from time k to
           k + (length of task j) - 1.
           If the processor is already engaged, the program backtracks.
*)
        FOR h := k TO k + job[i].task[j].length - 1 DO
          gantt[job[i].task[j].processor,h] = i;
        END
      END;
      min_start_time := k + job[i].task[j].length;
        (* set the minimum start time for the next task
           to the end of the current task *)
    END;
  END
END JobShopScheduling;
```

In this program the "don't know" nondeterminism provided by the use of the `SOME` statement is combined with the use of assignment.

Furthermore, as already mentioned, for each value of `i` and `j` the equality `gantt[job[i].task[j].processor,h] = i` acts both as an assignment and as a test.

The array `gantt` should be uninitialized when the procedure is called. At the end of the execution the variable `gantt` contains the first feasible schedule it finds.

Preinitialized values can be used to enforce some preassignments of jobs to processors, or to impose a constraint that a processor is not available during some periods of time. For example, if processor 2 is not available at time 5, we just use the assignment `gantt[2,5] := 0` (where 0 is a dummy value) before invoking the procedure `JobShopSchedule`.

As an example, suppose we have 3 jobs, 3 processors (p_1, p_2, and p_3), the deadline is 20, and the jobs are composed as follows:

		task 1		task 2		task 3		task 4	
job	tasks	proc	len	proc	len	proc	len	proc	len
1	4	p_1	5	p_2	5	p_3	5	p_2	3
2	3	p_2	6	p_1	6	p_3	4		
3	4	p_3	6	p_2	4	p_1	4	p_2	1

The first solution (out of the existing 48) for the array `gantt` that the program finds is the following one, where the symbol `'-'` means that the value is uninitialized, i.e., the processor is idle in the corresponding time point.

```
1 1 1 1 1 - 2 2 2 2 2 2 - - - 3 3 3 3 -
2 2 2 2 2 2 1 1 1 1 1 3 3 3 3 - 1 1 1 3
3 3 3 3 3 3 - - - - - 1 1 1 1 1 2 2 2 2
```

For some applications, it is necessary to make the schedule as short as possible. To this aim, we can use the following program fragment.

```
COMMIT
  SOME deadline := 1 TO max_deadline DO
    JobShopScheduling(JobVector,deadline,jobs,Gantt)
  END
END
```

It computes the shortest schedule by *guessing*, in ascending order, the first deadline that can be met by a feasible assignment. The use of the `COMMIT` statement ensures that once a solution is found, the alternatives, with larger `deadline` values, are discarded.

4 Introducing Constraints

In what follows we discuss a proposal for adding constraints to Alma-0.

This Section is organized as follows. In Subsection 4.1 we discuss the addition of constrained types and unknowns to the language and in Subsections 4.2 and 4.3 we define the constraint store and illustrate its interaction with the program execution.

To illustrate how the proposed addition of constraints to Alma-0 provides a better support for declarative programming we illustrate in Subsection 4.4 their use by means of three 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.

4.1 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 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 Section 5 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 2. *A* constraint *is a boolean expression that involves some unknowns.*

We postulate that the unknowns can appear only within constraints or within the right hand side of an assignment.

The values of unknowns are determined only by means of constraints that are placed on them. In particular, by the just introduced syntactic restriction, one cannot use assignment to assign a value to an unknown. So 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. Further, the assignments `i := X[1] + X[2]` and `i := Y[X[2]]` are also legal statements.

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, so within the right hand side of an assignment, 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.

4.2 Adding the Constraint Store

We now introduce the central notion of a *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. *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 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]`.

4.3 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 the 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 6.2.

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 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.

Note that constraints are interpreted in the same way independently of the fact whether they appear as a statement or inside a condition. For example, the following program fragment

```
IF X[1] > 0 THEN S ELSE T END
```

is executed as follows: The constraint `X[1] > 0` is added to the store. If the store does not fail `S` is executed, otherwise `T` is executed. So we do not check whether `X[1] > 0` is *entailed* by the store and execute `S` or `T` accordingly, as one might intuitively expect. This means that constraints are always intepreted as so-called *tell* operations in the store, and never as so-called *ask* operations, which check for entailment (see Section 8 for a discussion on this point).

4.4 Examples

To illustrate use of the introduced concepts we now consider three examples. We begin with 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. The code that actually solves the generated constraints would make use of the built-in `INDOMAIN` as explained in Subsection 6.2.

```
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`.

Note that the above program text coincides with the problem formulation.

Next, consider the following problem that deals with the equations arising when studying the flow of heat.

Problem 3. Laplace Equations. Given is a two dimensional grid with given values for all the exterior points. The value of each interior points equals the average of the values of its four neighbours. Compute the value of all interior points.

The solution using constraints again truly coincides with the problem specification. It is conceptually much simpler than the solution based on constraint logic programming and given in [10].

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

BEGIN
  FOR i := 2 TO M-1 DO
    FOR j := 2 TO N-1 DO
      X[i,j] = (X[i+1,j] + X[i-1,j] + X[i,j+1] + X[i,j-1])/4
    END
  END
END;
```

We assume here that the constraint solver that deals with linear equations over reals is sufficiently powerful to solve the generated equations.

Finally, 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 constraints solving 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 [8]. 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.

It is important to realize that the integration of constraints to Alma-0 as outlined in this section is possible only because the unknowns are initially uninitialized.

5 Constraints and Procedures

So far we explained how the program interacts with the store in absence of procedures. In Alma-0 one level (i.e., not nested) procedures are allowed. In presence of procedures we need to explain a number of issues.

First, to keep matters simple, we disallow local unknowns. This means that the constrained types can be only introduced at the outer level. However, unknowns can be used within the procedure bodies provided the restrictions introduced in Definition 2 are respected.

Next, we need to explain how unknowns can be passed as parameters. Formal parameters of constrained types are considered as unknowns. This means that in the procedure body such formal parameters can occur only within the constraints or within the right hand side of an assignment.

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 4. 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.

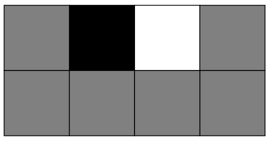

Fig. 1. Constraint Solving and Pixels

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. In addition, the program uses the built-in `KNOWN` that, when used on unknowns, checks whether the domain of the argument is a singleton.

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

Next, we explain the call by value mechanism in presence of unknowns. An unknown passed as an actual value parameter is treated as a customary variable: it is evaluated and its value is assigned to a local variable associated with the formal parameter. If the value of this unknown is at this moment undetermined, this evaluation yields a run-time error. This evaluation process also applies if a field or an element of a compound actual value parameter is an unknown.

6 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 6.1 we discuss built-ins for stating constraints, and in Subsection 6.2 we present built-ins for assigning values to unknowns.

6.1 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. 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];
```

[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, see [17].

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

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 the use of lists of unknowns (as done for example in the ICON programming language of [6] 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 of constraining the sum of the unknowns in L to be equal to b. 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 (where the type Region is an array of booleans representing a subset of the set of cells). 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;
```

6.2 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 are the best value selection strategies. 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 here the issue of the *variable ordering*.

7 Related Work

We concentrate here on the related work involving addition of constraints to imperative languages. For an overview of related work pertaining to the Alma-0 language we refer the reader to [2].

As already mentioned in the introduction, the most successful imperative constraint language is the C++ library ILOG Solver [9]. 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 [14] and 2LP of [13]. 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 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. In particular, the use of logical variables outside of constraints, the impact of backtracking on the assignment statements and the status of choice points created within procedure bodies is not explained (probably due to space limitations). CHARME does provide bidirectional connection with C.

2LP was designed for linear programming applications. 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.

8 Conclusions and Future Work

In this paper we discussed the programming language Alma-0 that integrates the imperative and logic programming paradigm and illustrated the resulting programming style by a number of examples. Alma-0 is based on first-order logic in the sense that it provides a computational interpretation for the standard connectives, so negation, disjunction and conjunction, and for various forms of quantification. In fact, many first-order formulas and their extensions by bounded quantifiers, sorts (i.e., types), and arrays, can be interpreted and executed as Alma-0 programs. The precise logical nature of this computational interpretation of first-order logic was worked out in [1].

Then we discussed a proposal how to integrate constraint programming features into the language. In this regard we believe that the use of an underlying language based on first-order logic, such as Alma-0, rather than a conventional imperative language, makes the integration of constraints more natural and conceptually simpler.

We analyzed here a number of issues related to the proposed integration, such as the use of constrained types and the unknowns, interaction between the program and the constraint store, 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 [2] to the language proposal here outlined. More specifically, we envisage to

- extend the executable, operational semantics based on the ASF+SDF Meta-Environment of [12];
- 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.

The first item can be dealt with by adding to the executable semantics of Alma-0 given in [2] a few rules that formalize the interaction between the program and the store stipulated in Subsection 4.3. These rules are parameterized by the constraint solvers attached to the store.

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

As already mentioned in Section 4.3, we do not allow so-called *ask* operations in the store. This is a deliberate design decision which allows us to keep the language design simple and the underlying execution model easy to implement.

Nevertheless, in future versions of the language, we plan to investigate the possibility of equipping the store with an *entailment* procedure. This procedure should check whether an evaluated form of a constraint is logically implied (or *entailed*) by the store. Upon encounter of an `ask` constraint, the entailment procedure would check whether the evaluated form is entailed by the store. If it is the case, the constraint evaluates to `TRUE`. Otherwise the constraint evaluates to `FALSE`. We would require that the entailment procedure returns correct results but would not assume that it is complete.

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, in particular the ones appropriate for constraint optimization,
- 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 Jan Holleman, Eric Monfroy and Vincent Partington for useful discussions on the subject of this paper. Helpful comments by Tony Hoare and other two, anonymous, referees allowed us to improve the presentation.

References

1. K. R. Apt and M. A. Bezem. Formulas as programs. In K.R. Apt, V.W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: A 25 Year Perspective*, pages 75–107, 1999.
2. K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM Toplas*, 20(5):1014–1066, 1998.
3. K. R. Apt and A. Schaerf. Search and imperative programming. In *Proc. 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 67–79. ACM Press, 1997.
4. A. Colmerauer. An introduction to Prolog III. *Communications of ACM*, 33(7):69–90, 1990.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability—A guide to NP-completeness*. W.H. Freeman and Company, San Francisco, 1979.
6. R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1983.
7. W. K. Hale. Frequency assignment: Theory and applications. In *Proc. of IEEE*, pages 1497–1514, 1980.
8. P. Van Hentenryck, Vijay Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer-Verlag, 1995. (Châtillon-sur-Seine Spring School, France, May 1994).
9. ILOG. ILOG optimization suite — white paper. Available via `http://www.ilog.com`, 1998.
10. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *14th ACM Principles of Programming Languages Conference*, pages 111–119, Munich, F.R.G., 1987. ACM, New York.
11. Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, July 1992.
12. P. Klint. A meta–environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
13. K. McAloon and C. Tretkoff. 2LP: Linear programming and logic programming. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 101–116. MIT Press, 1995.
14. A. Oplobedu, J. Marcovitch, and Y. Tourbier. 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*, pages 55–70, Avignon, France, 1989. EC2.
15. F. Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

16. J.-F. Puget and M. Leconte. Beyond the glass box: Constraints as objects. In *Proc. of the 1995 International Symposium on Logic Programming*, pages 513–527, 1995.
17. J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-94: Proceedings of the 12th National Conference on Artificial Intelligence*, pages 362–367, 1994.
18. I. Shvetsov, V. Telerman, and D. Ushakov. *NeMo+* : Object-oriented constraint programming environment based on subdefinite models. In G. Smolka, editor, *Artificial Intelligence and Symbolic Mathematical Computations*, Lecture Notes in Computer Science, vol. 1330, pages 534–548, Berlin, 1997. Springer-Verlag.
19. P. Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.