A Spreadsheet Based on Constraints

Marc Stadelmann

1. Motivation

In 1986, Weichang Du together with Bill Wadge worked out "an intensional 3D spreadsheet" based on Lucid. Briefly, Du's spreadsheet can be explained as a single (intensional) variable whose values vary in two spacial dimensions and one time dimension. The two spatial dimensions are represented by a conventional grid of rows and columns (called a sheet) and time is taken to be the third dimension. Intensional operators, such as 'Up', 'Down', 'Left', 'Right', etc. are added to switch spatial context while traditional temporal Lucid operators are used for time.

While Du's spreadsheet has several advantages over conventional spreadsheets it also shares a shortcoming: cells can only be calculated in one direction, given values for the cells they depend on. Even though Lucid formulas are declarative, eduction (as well as the evaluation of a conventional spreadsheet) has a fixed flow of calculation.

This insight led us to toy with the idea of new spreadsheet based on the notion of a *constraint*. In one word, we replace formulas (whether intensional or conventional) with constraints. The key idea of a constraint, in general, is that it allows the user to declare a relationship among objects and let the system worry about maintaining and satisfying this relationship. In the case of a spreadsheet, we let the user declare numerical constraints between the real values of cells. Examples of such constraints include:

- let two cells always be equal,
- let a cell be greater or equal to zero,
- let one cell be the sum of several other cells
- let one cell be twice as big as its neighbor
- etc.

Recalculating the spreadsheet then means (1) checking whether the given values in cells satisfy the constraints over these cells and (2) finding all solutions for empty cells that satisfy the constraints. In other words: given a set of constraints over a set of cells, and given values for some of these cells, the spreadsheet is able to mathematically derive all solutions to all remaining cells with respect to the given constraints. This process is referred to as *constraint satisfaction* or *constraint solving*.

The elaboration of this idea is the author's Masters thesis. It consists of a formalization of the concept and a (partial) implementation. In this paper we provide an overview and illustrate, in gen-

eral, the advantages of a constraint-based spreadsheet over a conventional spreadsheet. We describe the design of the interface of our new spreadsheet and its constraint language. At the same time, we give several small example applications for which our new spreadsheet is better, or even uniquely suited. Finally, we refer to related work and draw some conclusions.

2. Advantages

The small shift from formulas to constraints has a rather significant impact on a spreadsheet's problem-solving power. As already mentioned, the flow of calculation is not hard-wired anymore into the spreadsheet's formulas. Instead, whichever cell is empty (or somehow marked as not containing an input value) can be calculated according to the constraints that reign. Consider the example in *Figure 1* where we declare a constraint between a cell named Fahrenheit:C1 and a cell named Celsius:C1. Any value put into either of the two cells gets converted into the other temperature measurement.

Figure 1 Simple constraint

The same is not possible in a conventional spreadsheet: if we wanted to convert Celsius into Fahrenheit, we would have to write the formula **1.8 * Celsius + 32** into the Fahrenheit-cell and if, on the other hand, we wanted to convert Fahrenheit into Celsius we would have to algebraically transform the formula ourselves into **5/9 * (Fahrenheit - 32)** and write it into the Celsius-cell.[1] A constraint, on the other hand, acts like a true equation that can be solved for any unknown cell and can be written in any algebraic form.

A second improvement is that a cell can be constrained by several constraints simultaneously instead of being calculated by one and only one formula. We can, for example, extend the above spreadsheet with a second constraint that requires both the Celsius-cell and the Fahrenheit-cell to be equal. This is illustrated in *Figure 2*.

1. Note also that we could not have the formula calculating Fahrenheit in the Fahrenheit-cell and at the same time have the formula calculating Celsius in the Celsius-cell. This would introduce a circular dependency between the two cells.

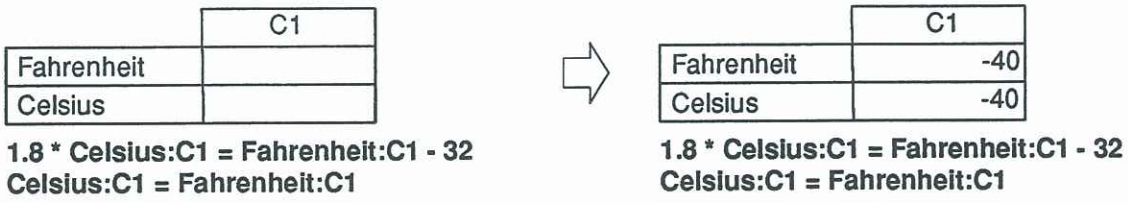

Figure 2 Two simultaneous constraints (equations)

This problem is in fact a set of two linear equations with two unknowns. Conventional spreadsheets only offer *iteration* to solve such kinds of problems. Iteration repeatedly calculates all formulas with the current cell values and eventually converges to a single result. However, the user has to first recognize the nature of his problem, then turn on the iteration feature and specify a condition for termination and a maximum number of iterations. However, even when the problem is known to have a solution, there is no guarantee that iteration finds it. Consider the example in *Figure 3*. The conventional spreadsheet in the top row converges towards the correct result while the same problem below, formulated differently, diverges. For solving simultaneous equations reliably, one would have to resort to programming a specific algorithm in a spreadsheet's macro language.

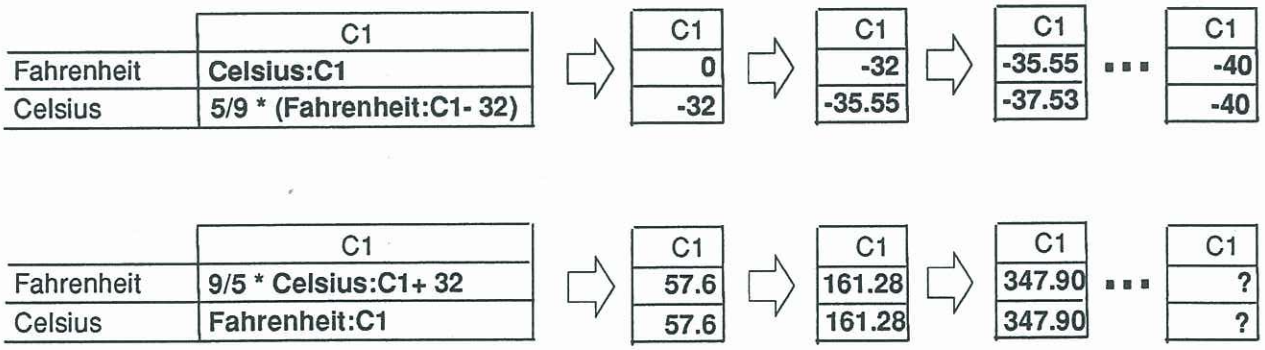

Figure 3 Convergence and divergence of iteration in conventional spreadsheets

A third improvement over conventional spreadsheets is that we always get *all* solutions to a particular problem. We can go even further and compute symbolic answers: if a problem has infinitely many solutions, we can get a more "simple" constraint as an answer that captures the relationship between the under-constrained cells. Or, alternatively, we can compute one "best" solution out of infinitely many by maximizing the value of a particular cell under the given constraints.

3. The Constraint-Based Spreadsheet

3.1 User Interface

The new spreadsheet's user-interface is very similar to a conventional spreadsheet. We have a two-dimensional grid of cells which can hold text or real numbers, but no formulas. Since a constraint can possibly calculate any cell it references we chose not to write a constraint into a particular cell but to collect all constraints together in a separate window, outside the grid of cells. This is what we did in the two motivating examples in *Figure 1* and in *Figure 2*. To make it easier for the user to see which constraint(s) affect(s) which cell(s), we can highlight the cells when the user selects a constraint and highlight the associated constraints when the user selects a cell.

Recalculating the constraint-based spreadsheet is not as straightforward as recalculating a conventional spreadsheet. There are choices involved. Consider the example where A, B and C are cells with values 2, 3 and 5 respectively, satisfying the constraint A + B = C. If the user changes any of these cells the spreadsheet does not know which of the other cells it has to adapt in order to satisfy the constraint. There are a variety of possibilities how the given constraint can be maintained. Let us assume the user has changed the value in cell C to 6, the spreadsheet can either:

- add 1 to cell A or add 1 to B or
- add 1/2 to both or
- add the same percentage to A and B so that the total amount added is 1
- or ...

depending on the problem we are trying to solve. There are several strategies of how the user can tell the spreadsheet which cell to change and how. One could, for example, "freeze" certain cells and make their value read-only, which means they can only be updated by the user, but not be changed by the constraint-solver in order to satisfy a constraint. Or as an alternative, one could ask the user which cell to adapt in the satisfaction process. We opted for a very simple solution: we interpret a value in a cell as an simple equality constraint for that cell to be equal to that value. These cells are called input cells. Any cell that is empty and otherwise constrained will be calculated. To distinguish values input by the user from values calculated by the spreadsheet we display the latter in bold.

3.2 Constraints

Assembling constraints outside of the grid of cells coalesces all information in one single area. However, even a simple spreadsheet application can easily have more than a dozen constraints, probably about as many constraints a similar conventional spreadsheet would have formulas. Like formulas, many of these constraints calculate cells in the same manner. We introduce the concept of a constraint over a *vector* of cells that takes advantage of the spreadsheet's grid to write sets of constraints more efficiently. We define a vector as two or more adjacent cells in one row or one column. There are several types of operations that we can perform on vectors. Let us take for example addition.

1. A single cell (scalar) can be added to (every cell of) a vector.
2. Two vectors can be added together. All operations on vectors are defined point-wise, that is, cell by cell. Therefore, the size of two vectors, i.e. the number of cells, must be the same.
3. We can extract cells or subvectors of a vector
4. We can add together all elements of a vector.

Similar operations can be defined on matrices as well, or cubes, etc. This leads to an APL-like constraint-language. We will introduce and illustrate some of the most useful operations in the following.

3.2.1 Adding, multiplying, etc. two vectors

Let us first introduce a naming scheme for vectors. The simplest scheme would be to have as vectors only the entire row or column of cells and then use the name (or number) of this row or column. For most but the simplest applications such a scheme appears not to be flexible enough. We propose to group a number of adjacent rows or columns together into a *group* and give the group a separate name. A group, basically, allows to shorten an entire row or column vector to a desired length. In *Figure 3*, for example, Resistors groups together columns R1 to R4.

Figure 3 shows a spreadsheet that calculates current and resistance in a simple electric circuit. Ohm's law, for example, can be expressed by one single vector constraint between the voltage, the current and the resistivity of all Resistors:

V:Resistors = R:Resistors * I:Resistors

By assuming a direction for the current (see the two arrows in *Figure 3*) Kirchhoff's current law can be formulated as a constraint for each node[1] which states that the sum of the currents entering and leaving any node is algebraically zero. This leads to the five constraints shown. Similarly, Kirchhoff's voltage law states that the sum of the voltages around any closed circuit or loop in the network is algebraically zero, which gives raise to two additional constraints. As we can see, the basic electrical laws can be entered directly as constraints without transforming them to solve the problem for one particular variable.

3.2.2 Extracting cells and subvectors of a vector

We introduce four structural operations on vectors, **FIRST()**, **LAST()**, **ABF()** and **ABL()**. **FIRST(***vectorName***)** and **LAST(***vectorName***)** extract the first and last cell of a vector respectively. The first cell of a vector is its leftmost or topmost cell, depending on whether the vector is horizontal or vertical. The last cell is defined correspondingly. **ABF(***vectorName***)** gives "all but the first" cell of a vector and **ABL(***vectorName***)** returns "all but the last" cell of a vector. Together, these four operations allow to extract any subvector of a given vector. *Figure 4* gives an example of how these operations can be used. The spreadsheet calculates the depreciation of an investment according to the year's digits method. This method is used when the depreciation is the greatest during the first few years of use.

1. A *node* in the network is any point to which two or more wires are connected.

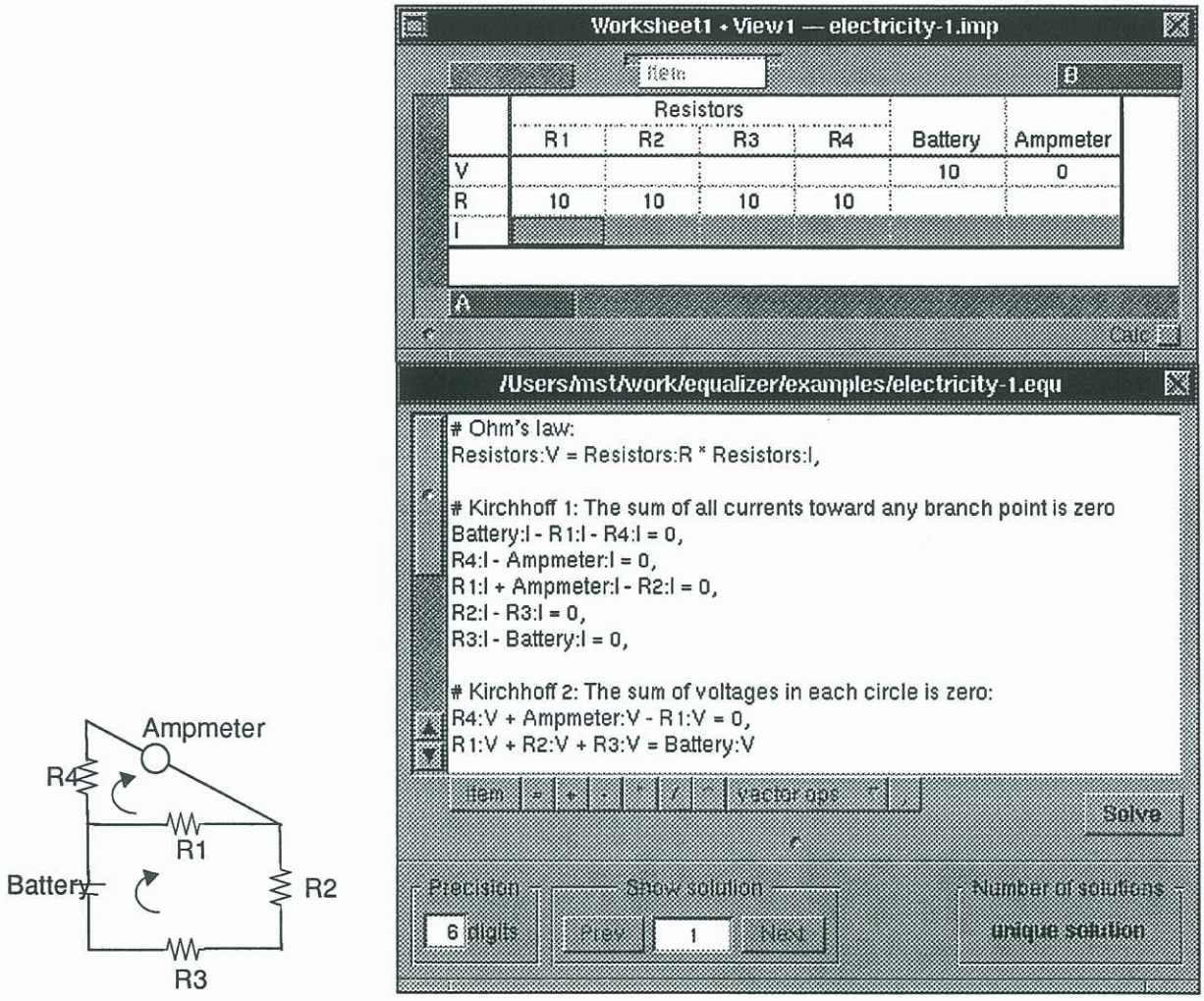

Figure 4 Electric circuit with corresponding spreadsheet model

For example, suppose a drilling machine costs $15,000 and can be resold after five years for $5,000. The total depreciation is $10,000. The sum of the years 1 through 5 is 1+2+3+4+5=15. Thus, the depreciation for the first year is 5/15, for the second 4/15, etc. The digits for each year can be calculated as follows:

LAST(digits:years) = 1

ABL(digits:years) = ABF(digits:years) + 1

Notice that the two vectors on the left and right hand side of the equal sign in the second constraint overlap. This technique allows us to calculate sequences of values within a vector. This would normally be achieved by as many formulas with relative references as there are cells in that

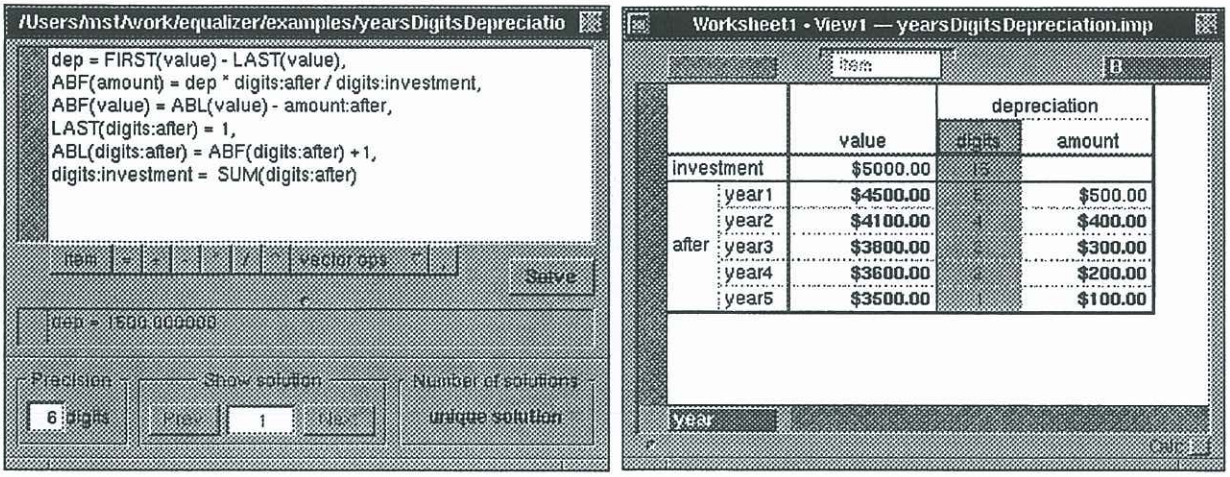

Figure 5 Depreciation. Year's digits method

vector. Not only is our language shorter but it also allows the user to add rows (for example for a sixth year) without extending the constraints.

3.2.3 Adding, multiplying, etc. all elements of a vector

These operations are called reduction operations on vectors. Reduction is equivalent to placing a dyadic operator between each element of the vector and evaluating the resulting expression. For example, the sum of the year's digits is computed by:

SUM(years:digits) = digits:investment

Then we calculate the depreciation for each year. It is sometimes convenient to have a new variable that is not part of the sheet itself, like **dep**, for the total depreciation of the article:

dep = FIRST(value) - LAST(value)

ABF(depreciation) = dep * digits:years / digits:investment

3.2.4 Matrix constraints

Consider the flow of heat in a rectangular metallic plate when a heat source is applied to the edge(s) of this plate. The heat from a hot edge will gradually diffuse into the plate and heat up colder regions of the plate until a stationary state is reached. We would like to find the heat at each point of the plate at this stationary state. This classical engineering problem is solved with Laplace's equation. Briefly, in the case of our example Laplace's equation is satisfied when the heat of the plate at each point is the average of the heat at its four closest neighbor points. We use the spreadsheet grid itself to represent the plate and interpret the value of each cell as the heat at that point on the plate. *Figure 5* shows the constraint-based solution to this problem. **ABF(***matrixName***)** and **ABL(***ma-*

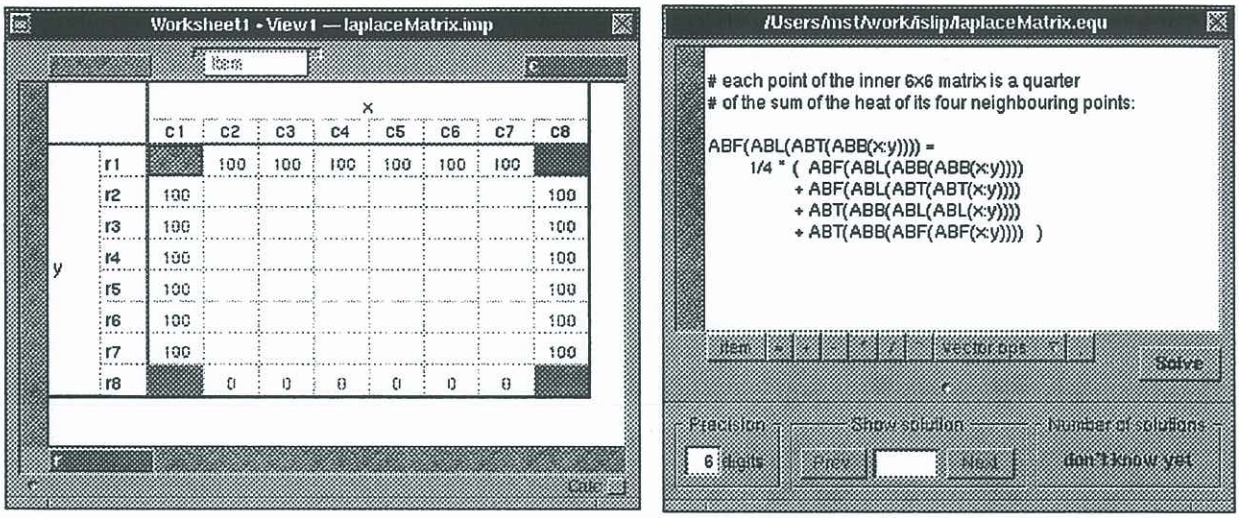

Figure 6 Steady-state two-dimensional heat flow

trixName) take the first row (column) off a matrix. **ABT(***matrixName***)** and **ABB(***matrixName***)** take off the top and bottom row correspondingly. Therefore, the left-hand-side of the equality constraint in *Figure 5* refers to the inner 6x6 matrix with rows r2 to r7 and column c2 to c7. Each point of this sub-matrix then is a quarter of its four neighbor points to the east, west, north and south of it. The whole system of equation is formulated as one single constraint on several overlapping matrices.

4. Related Work and Conclusion

We are not the first toying with the idea of a spreadsheet based on constraints. In the constraint-programming research community, a spreadsheet is often used as a vehicle to introduce and illustrate the basic concept of constraint programming. However, to the author's knowledge, no one from either research or industry has ever explored the idea to its full extent or published about it or implemented anything. As a matter of fact, beyond the vast number of very general books describing tips, tricks and traps of any commercial spreadsheet, there is very little published about spreadsheets. Besides Du's spreadsheet that we mentioned in the beginning, the following work is closely enough related to ours and should be mentioned and given credit.

Spenke and Beilken [2][3] describe a spreadsheet interface for logic programming and mention the integration of constraints into spreadsheets. Their system, called PERPLEX, is based on an earlier idea of Van Emden et al. [4] and Krivaszek [5] who propose the raster layout of spreadsheets as a flexible user-interface to an interactive programming tool based on Prolog. Spreadsheet

cells can be seen as Prolog variables and can be related through Prolog predicates. Prolog variables can (besides many other things) hold real numbers while predicates are like constraints. However, the emphasis of PERPLEX was not to be a more powerful *calculation tool* but, on the contrary, a more general *programming tool* whose applications go beyond numerical problems. Therefore, the solving capabilities for numerical constraints are fairly limited and "neither symbolic transformations of constraints nor iterative approximation algorithms are currently implemented" ([3] page 10).

Recently, some of the logic programming languages, such as Prolog III [7] and CLP($\Re$) [6], include constraints over limited domains and extend logic programming into constraint logic programming. To our knowledge, no one has worked on connecting a spreadsheet as an interface to any of these languages.

In a few words, all of the above proposals take an existing programming language and use a spreadsheet as its user-interface. We, on the other hand, introduce a shift in the way we look at a spreadsheet and obtain a more general and more powerful calculation tool.

References

[1] W. Du, W. W. Wadge, "A 3-D Spreadsheet Based on Intensional Logic", IEEE Software, May 1990

[2] M. Spenke, C. Beilken, "A Spreadsheet Interface For Logic Programming", CHI '89 Proceedings, 1989

[3] M. Spenke, C. Beilken, "PERPLEX: A Spreadsheet Interface for Logic Programming by Example", Research Report, FB-GMD-88-29, Gesellschaft für Mathematik und Datenverarbeitung, November 1988

[4] M. H. van Emden, M. Ohki, A. Takeuchi, "Spreadsheets with Incremental Queries as a User Interface for Logic Programming", ICOT Tech. Rep. TR-144, Oct 1985

[5] F. Kriwaszek, "LogiCalc - A PROLOG Spreadsheet", in B. Kowalski, F. Kriwaszek, "Logic Programming", pp. 105-117, also in: D. Michie and J. Hayes, Machine Intelligence ll, 1987

[6] J. Jaffar, S. Michaylov, P. J. Stuckey, R. H. C. Yap, "The CLP(R) Language and System", Proceedings of the 4th ICLP, 1987

[7] A. Colmerauer, "An Introduction to Prolog III", Communications of the ACM, Vol. 33, No. 7, July 1990, pp. 69-90