A Spreadsheet Based on Constraints

Marc Stadelmann

mst@sanjuan.uvic.ca
University of Victoria
Department of Computer Science
Engineering Office Wing
V8W 3P2 Canada

ABSTRACT

Constraints allow the user to declare relationships among objects and let the system maintain and satisfy these relationships. This paper is concerned with the design of a spreadsheet based on constraints. Instead of formulas, we let the user enter numerical constraints, such as >, < and = over the real values in the cells of the spreadsheet. Recalculating the spreadsheet then means (1) checking whether the given cell-values satisfy all constraints and (2) finding values for cells that satisfy the constraints.

After a brief presentation of the advantages of a constraint-based spreadsheet, we describe the design of its user-interface. The concept of constraints over cells is extended to constraints over vectors and matrices of cells. We introduce a language for constraints over vectors and matrices of a two-dimensional spreadsheet and define several operations on these vectors.

The described new spreadsheet has been implemented and we provide a brief overview of this effort before discussing future work and giving some concluding remarks.

KEYWORDS

spreadsheets, user-interface, constraints, constraint satisfaction, vectors, matrices, APL

INTRODUCTION

In this paper, we describe a new spreadsheet based on the notion of a *constraint*. Simply stated, we replace formulas 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. Constraints are successfully used to maintain relationships of graphical objects in drawing programs, typesetting and many other domains [1][2][3]. 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.

This small shift from formulas to constraints has a rather significant impact on a (1) spreadsheet's problem solving power and also on (2) how the user interacts with the spreadsheet. In this paper, we focus on the second aspect. Section 2 introduces the constraint-based spreadsheet by demonstrating its advantages over the conventional ("formula-based") spreadsheet. In section 3, we will focus on a number of user-interface problems and describe solutions. We have implemented a subset of this new spreadsheet restricted to equality-constraints. Throughout this paper we provide a collection of examples for which the idea of a constraint-based spreadsheet is well, or even uniquely, suited.

ADVANTAGES

A first advantage of the constraint-based spreadsheet is that the flow of calculation is not hard-wired into the spreadsheet's formulas. Instead, whichever cell-value we are interested in 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.

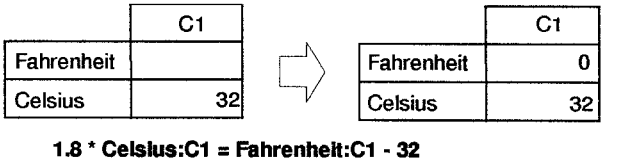

Figure 1: Simple constraint

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. On the other hand, if 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. Moreover, the user is freed from writing the constraint in any specific algebraic form.

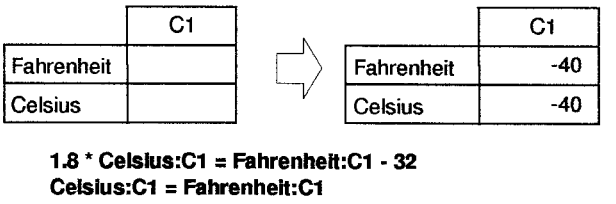

Figure 2: Two simultaneous constraints

A second improvement is that a cell can be constrained by several constraints simultaneously instead of being calculated by one and only one formula. *Figure 2*, for example, extends the spreadsheet of *Figure 1* with a second constraint that requires both the Celsius-cell and the Fahrenheit-cell to be equal. This spreadsheets asks the question: "At what temperature do both the Celsius scale and the Fahrenheit scale give the same measurement".

This problem is, in fact, two simultaneous linear equations with two unknowns. Conventional spreadsheets typically offer *iteration* to solve simultaneous equations. Iteration repeatedly calculates all formulas with the current cell values and eventually converges to a single result. Iteration corresponds to the Gauss-Seidel numerical method for solving simultaneous equations. However, even when the problem is known to have a solution, there is no guarantee that iteration finds it because the condition for convergence of the Gauss-Seidel algorithm is quite arbitrary. Without going into detail, consider the two conventional spreadsheet versions in *Figure 3* of example presented in *Figure 2*. The conventional spreadsheet in the top row converges towards the correct result while the equivalent problem below, formulated differently, diverges.

Therefore, in a conventional spreadsheet, the user has to (1) recognize the nature of his problem, then (2) formulate the correct set of formulas so that the iteration converges, (3) turn on the iteration feature and (4) specify a condition for termination and a maximum number of iterations in case the

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.

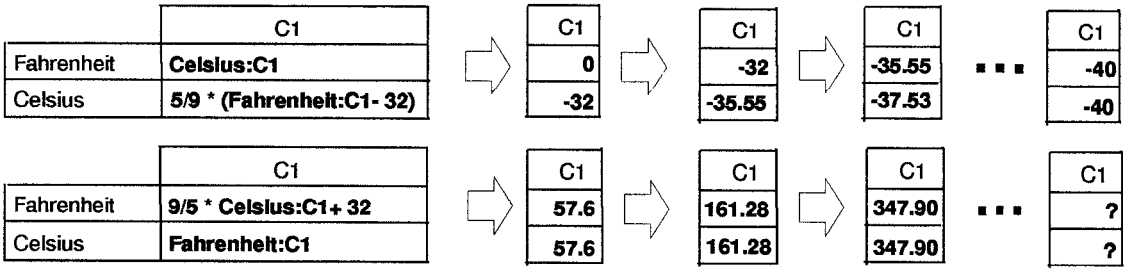

Figure 3: Convergence and divergence of iteration in conventional spreadsheets

condition is never met. Or, for solving simultaneous equations reliably, one would have to resort to programming a specific algorithm in a spreadsheet's macro language. Either solution is a burden to the user while, in the constraint-based spreadsheet, simultaneous equations are not a special case.

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 "simplified" 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 or minimizing the value of a particular cell under the given inequality constraints.

There are, of course, limitations to what a constraint-based spreadsheet can solve. However, since the focus of this paper is the design of the interactive aspects of this new spreadsheet we will not further discuss any computational issues. The only point we would like to emphasize is that the new spreadsheet's problem-solving power is no longer limited by the solving power of one particular algorithm (Gauss-Seidel) but by the limitation of numerical and symbolic mathematics.

INTERACTION

Since the same constraint can be used to calculate any of the cells 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 affects which cell(s), we can highlight the cell(s) when the user selects a constraint and highlight the associated constraint(s) when the user selects a cell. A similar approach has been taken by Improv, a commercial spreadsheet from Lotus for the NeXT, where the spreadsheet's formulas are written as a set of assignments, outside the grid of cells.

Recalculation

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. There are several strategies of how the user can tell the spreadsheet which cell to change and how. One could, for example, *freeze* the values of certain cells and make them read-only, which means they can only be updated by the user, but not be changed by the constraint-solver. Or as an alternative, one could ask the user which cell to adapt as soon as there is a choice in the constraint satisfaction process. Or hierarchies of constraints [1] with different strengths could be used to satisfy constraints that are absolutely required first and then proceed with less important constraints.

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 constrained will be calculated. To distinguish values input by the user from calculated values in the resulting sheet we display the former in normal font and the latter in bold.

Recalculating can be summarized as follows: (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. Note that as a consequence we do not automatically recalculate when any single cell is changed but wait for the user to erase the values he is interested and then trigger the solving process manually.

Constraint Language

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 as a similar conventional spreadsheet would have formulas. Like formulas that have been copied, many of the constraints constrain sets of cells in a similar way. 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 compactly. 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.

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 all 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. In Figure 5 ,for example, Resistors groups together columns R1 to R4. Groups simply allow us to refer to a subvector of an entire row or column by a name.

Figure 5 shows a spreadsheet that calculates current and resistance of the simple electric circuit shown in *Figure 4*. Given a resistance for each resistor and the voltage of the battery as input, the spreadsheet is able to derive all other values of each electrical component according to the electrical laws entered as constraints. 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, 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 says 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. Again, values input into different cells would calculate the problem backwards without changing the constraints.

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

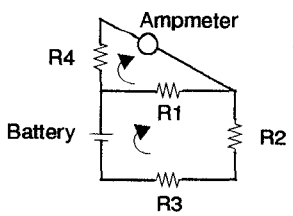

Figure 4: Electric circuit with corresponding spreadsheet model

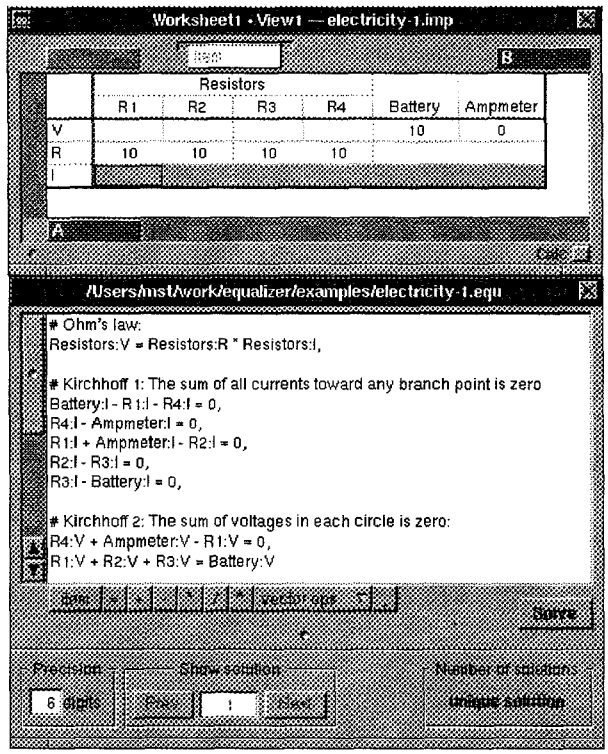

Figure 5: Electric circuit with corresponding spreadsheet model

Extracting cells and subvectors. 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 6* 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. 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 vector. Not only is our language shorter but it also allows the user to add rows (for example for a sixth year) without modifying the constraints.

Vector reduction. Reduction of a vector is equivalent to placing a binary 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

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 [8]. 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 7* shows the constraint-based solution to this problem. **ABF(***matrixName***)** and **ABL(***matrixName***)** 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 7* 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 the sum of the heat at its four neighbor points. The whole system of 36 equations is formulated as one single constraint on several overlapping matrices.

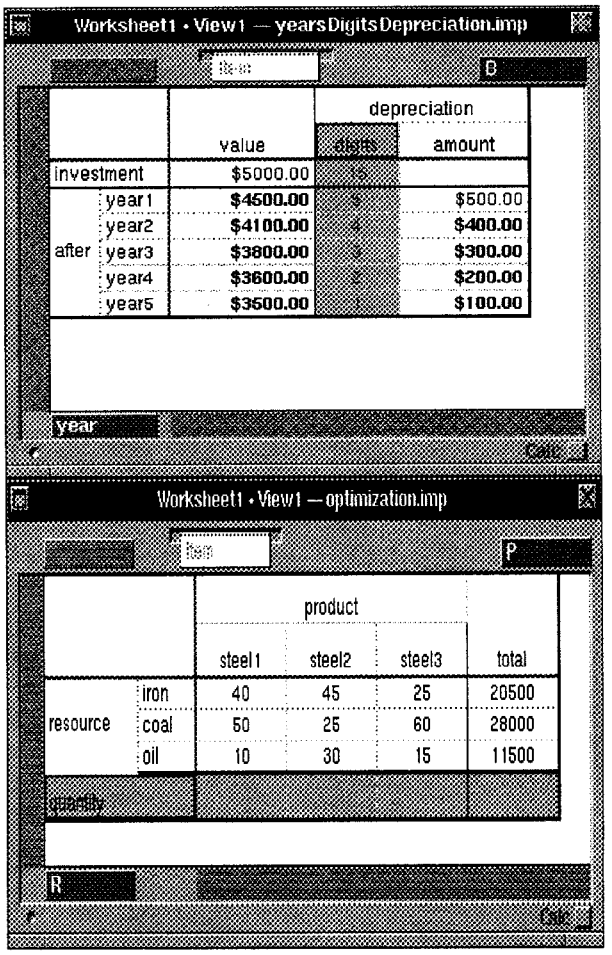

Figure 6: Depreciation. Year's digits method

Answer constraints

This leads us to the more general case when a problem is *under-constrained*, that is it has infinitely many solutions, none of which is any better than any of the others. Instead of arbitrarily choosing one solution over the other it is more helpful for the user to get back one or several more *simple* constraints that capture the relationship between the under-constrained cells. This means to symbolically transform and *simplify* the given constraints. Notice that the term *simplify* is subjective and depends on the problem at hand into which a symbolic solver has no insight. However, in general we can say that an answer constraint formulates a condition that every solution to the given input constraints must satisfy. Answer constraints can be seen as the result of a partial evaluation of the spreadsheet.

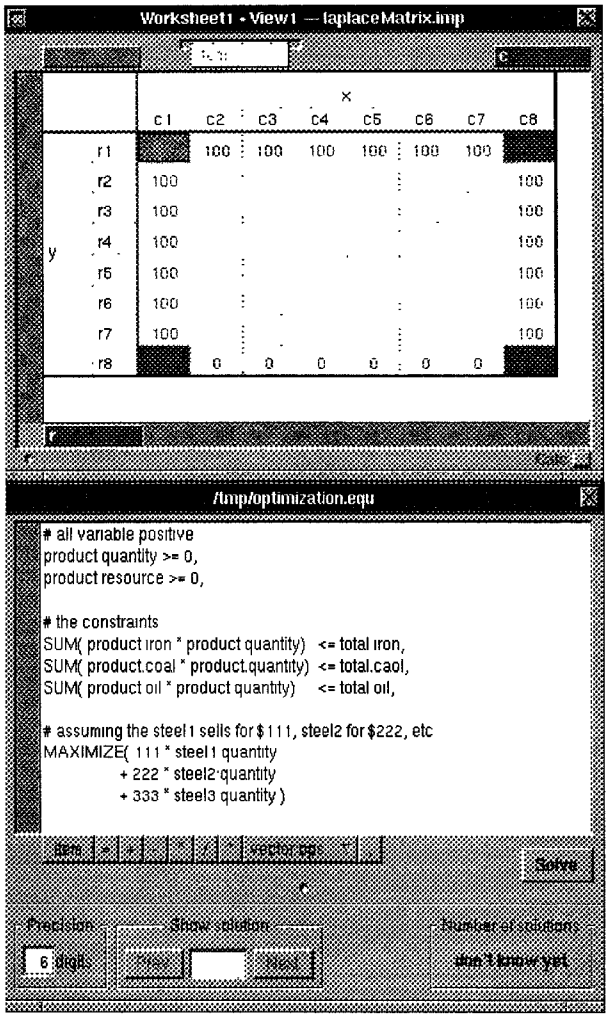

Figure 7: Steady-state two-dimensional heat flow

The following example shows that an answer constraint itself can be the solution to the problem. Consider the spreadsheet in *Figure 8*[1] which calculates payments at the end of each year to pay back a loan over ten years at a given interest rate. If we only provide the interest rate and ask for the loan and the yearly payment we get back as an answer the constraint:

loan:scenrio1 = 5.650223 * pmt:scenario1

The interpretation of this answer is quite interesting. It says that for a ten-year loan at a 12% interest-rate 5.65 payments out of ten (56.5%) are for amortizing the loan while the rest is for the interest.

Optimization

In most practical problems that have multiple solutions the user is interested in a "best" solution, which implies optimization. Optimization fits in perfectly with constraints: we optimize a function under a given set of constraints (inequalities). Consider the linear programming problem in *Figure 8*. We have three different qualities of steel. Each kind of steel takes needs the given proportions of raw material. Given a finite amount of raw material and the selling price for each kind of steel, we would like to maximize the profit. Let us assume **steel1**, **steel2** and **steel3** sell for $111, $222 and $333 respectively. If we extended our constraint language with a command like **MAXIMIZE()** we could write something like:

```
MAXIMIZE(   111 * steel1:quantity
          + 222 * steel2:quantity
          + 333 * steel3:quantity)
```

under the constraints shown in *Figure 9*.[2]

1. A similar example is presented in [8]

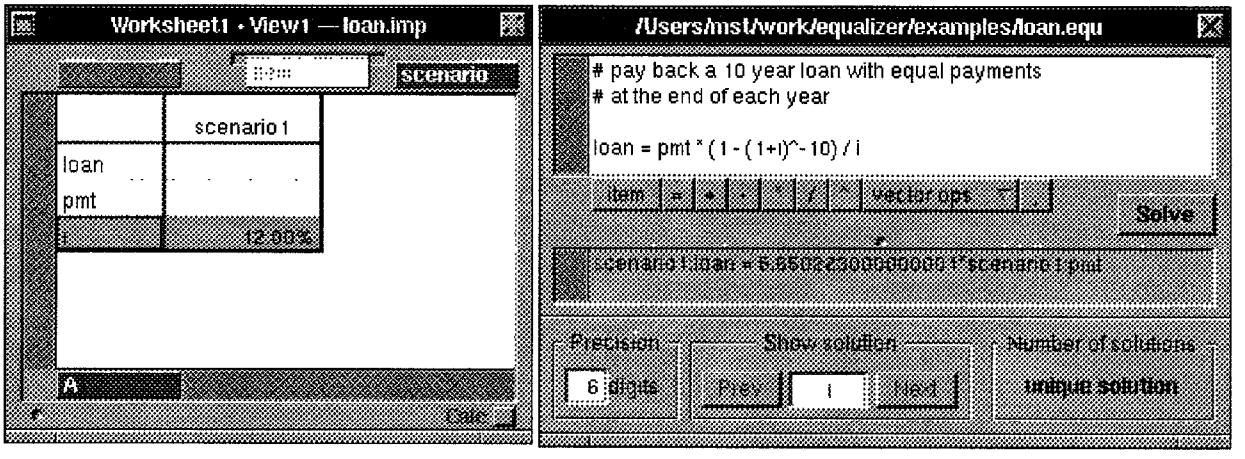

Figure 8: Calculating equal payments for a loan

IMPLEMENTATION

We have implemented a subset of this new spreadsheet restricted to equality constraints (equations) as part of the author's Masters thesis. The goal of this implementation was to give evidence of the practical potential of the idea. All examples given, except for the very last in *Figure 9*, do work and can be explored on-line. Because of its limitation of equality constraints, we named the resulting spreadsheet *The Equalizer.*

The Equalizer has been developed in Objective-C on a NeXT computer with NextStep 2.1. Instead of implementing a full spreadsheet interface with all its editing and manipulation capabilities we make use of the interface of an existing spreadsheet, Improv from Lotus. This allows us to have a full-blown spreadsheet in one window which we extend with our constraints in a second window. From the user's point of view the interaction with these two windows is a little awkward in some situations, but this was a very small price to pay for a number of man-months, if not years, of programming.

On the constraint solving side too we made use of an existing software package, Mathematica from Mathematica Inc. Finally, the code we have written glues all parts together: we parse the constraints and retrieve the structure and values from the spreadsheet. We check the input for consistency and then translate it into a suitable form for Mathematica to solve. Mathematica returns either a numerical or a symbolic solution, which we analyze, translate and display to the user. It follows that the constraint solving power of our implementation is that of Mathematica, or any other package for solving equations.

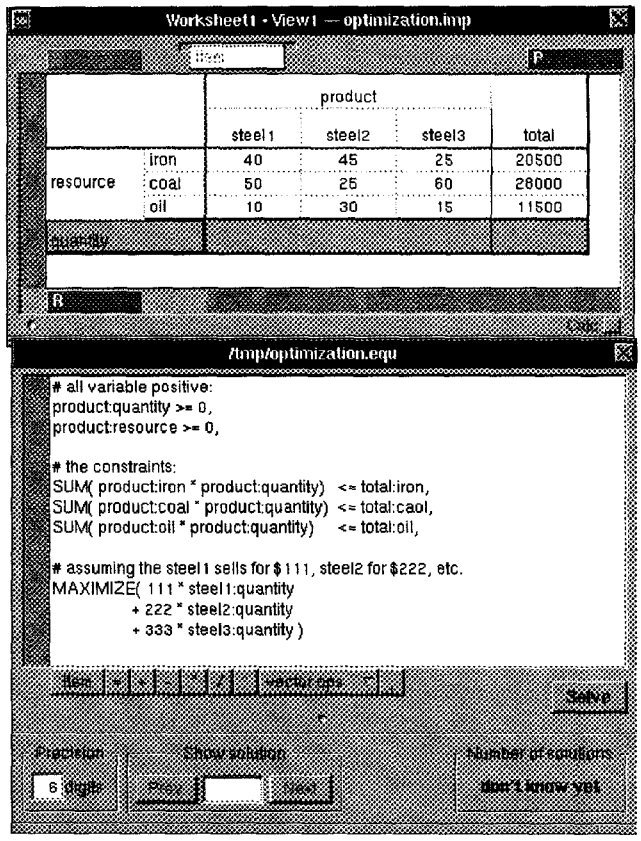

Figure 9: Linear programming

RELATED WORK

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. The following work is closely enough related to ours and should be mentioned and given credit.

Spenke and Beilken [4][5] 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. [6] and Krivaszek [7] who propose the grid 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" ([5] page 10).

Recently, some of the logic programming languages, such as Prolog III [9] and CLP($\Re$) [8], include constraints over limited domains and extend logic programming into

2. **MAXIMIZE()** is not implemented yet.

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.

CONCLUSION AND FUTURE WORK

Spreadsheets are one of the most widely used calculation tools, yet they cannot solve problems of elementary mathematics. In this paper, we have proposed a simple shift in the spreadsheet paradigm, replacing formulas (assignments) with constraints (declarations), and we have described the impact of this shift on a spreadsheet's user-interface.

On the user-*input* side, we have shown that the use of constraints (1) significantly simplifies the user's task when dealing with circular problems or simultaneous equations. There is no need for iteration or tedious macro programming anymore. Second, (2) constraints give the user more freedom in expressing his problem. The user can declaratively state his problem without worrying how it is solved. While the idea of writing formulas outside the grid of cells has been implemented by Improv we have (3) extended the concept to constraints. We argue that constraints, unlike formulas, should not be tied to a particular cell and it is more natural to grouped them together. In addition, this provides the user with a vue d'ensemble instead of being able to see one and only one formula at a time. We then (4) presented a textual language for constraints based on the vectors and matrices of a 2-dimensional spreadsheet grid. This significantly compacts constraint writing.

From the *output* point of view, the constraint-based spreadsheet is always able to gives all solutions to a given problem. We have shown that (5) answer constraints can deal with infinitely many solutions by capturing the relationship between the under-constrained cells. Often an answer constraint can be interpreted as the solution to a more subtle problem.

This work is far from being completed. Beyond computational issues and efficiency, there are a number of user-interface issues to consider. As the problem-solving power increases, so does the complexity of the applications. Better tools for spreadsheet data analysis are needed. For example a tool that explains how a certain result has been calculated or tools to better explore multiple solutions.

Despite some unsolved problems, we strongly believe that the next generation of commercial spreadsheets will be based on constraints. For three reasons. First, because a constraint-based spreadsheet can solve a superset of problems a conventional spreadsheet can solve. Second, because many features that have subsequently been added to conventional spreadsheets, such as goal-seeking and optimization, are seamlessly integrated into a spreadsheet based on constraints. And last, but not least, because a constraint-based spreadsheet can be implemented as a compatible extension of existing commercial products. Our implementation is just one example.

REFERENCES

[1] B. Freeman-Benson, J. Maloney, A. Borning, "An Incremental Constraint Solver", Communications of the ACM, 33(1):54-63, January 1990

[2] B. Vander Zanden, "Incremental Constraint Satisfaction and Its Applications", Ph.D. dissertation, Cornell University, October 1988, published as Technical Report 88-941, Department of Computer Science, Cornell University

[3] W. Leler, "Constraint Programming Languages", Addison-Wesley, 1987

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

[5] 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

[6] 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

[7] 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 II, 1987

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

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