Knowledgesheet: A Graphical Spreadsheet Interface for Interactively Developing Constraint Programs

Gopal Gupta

Laboratory for Logic, Databases, and Advanced Programming
Department of Computer Science
New Mexico State University
Las Cruces, NM, USA

Shameem F. Akhter

Technology CAD Group
Intel Corporation
Portland, OR, USA

Abstract

We introduce a generalization of the spreadsheet paradigm, called Knowledgesheet, for solving a class of constraint satisfaction problems. The traditional spreadsheet paradigm is based on attaching expression to individual cells and then evaluating them; our Knowledgesheet interface instead allows *finite domain constraints* to be attached to individual cells that are then solved to obtain a solution. This extension provides an easy-to-use interface for solving a large class of constraint satisfaction problems—those whose specification and solution conforms to a 2-dimensional structure, e.g., scheduling problems, time-tabling problems, etc. A prototype for the Knowledgesheet has been developed and applied to solve many different types of problems. This prototype includes the spreadsheet-like interface as well as the protocol between the interface and the back-end constraint-solving engine. The constraint solving engine is based on a finite domain constraint logic programming system (CLP(FD)). The protocol maps the constraints attached to the various cells of the Knowledgesheet to a CLP(FD) program. The auto-generated program is executed by the back-end CLP(FD) engine and the computed results communicated to the Knowledgesheet interface for display. A major advantage of Knowledgesheet is that novice users can solve certain types of practical constraint satisfaction problems without being an expert in constraint logic programming or constraint solving.

1 Introduction

Many problems that arise in daily life are NP-hard in nature. For instance, course scheduling at Universities, resource allocation, task scheduling, examination timetabling, etc. Most NP-hard problems can be elegantly expressed as constraints. A solution to these problems can then be found by constraint solving. However, formulating the problem as a set of constraints is not an easy task. Very often a lot of experimentation is involved and different constraints need to be tried. In this paper, we present a spreadsheet based interface, called Knowledgesheet, for finite-domain constraint programming. We show that this interface is highly suitable for interactively developing constraint-based solutions to a large class of practical NP-hard problems. A prototype implementation of Knowledgesheet, implemented in Java, is operational [9, 5].

The Knowledgesheet interface is based on the observation that most combinatorial problems are naturally modeled as a set of variables and a set of constraints on these variables. While the facilities provided by constraint languages (such as logic programming languages and constraint programming languages) precisely match the requirements for modeling combinatorial problems, the way in which the solution has to be programmed requires deep knowledge of the language. The Knowledgesheet programming interface that we present in this paper bridges this gap. The Knowledgesheet interface allows novice users to program a large class of constraint satisfaction problems (CSPs), just in the same manner as an ordinary spreadsheet allows a novice user to program a large class of problems involving arithmetic computations. The major advantage of the Knowledgesheet approach is that it uses the spreadsheet paradigm, which is familiar to most people, for solving certain class of CSPs.

In traditional spreadsheets only atomic values (numbers and constants) and arithmetic expressions can be attached to individual cells. In a Knowledgesheet, finite domain constraints can also be attached to individual cells. This generalization of spreadsheet, or Knowledgesheet, is inspired by the idea that the solution to many constraint problems, such as course scheduling, examination timetabling, resource scheduling, etc., can be expressed via constraints entered in a 2-dimensional table. These constraint satisfaction problems can be programmed using the spreadsheet paradigm if constraints are also allowed to be attached to individual cells of the spreadsheet.

Consider the problem of course scheduling at an academic department in a University. The course schedule consists of a table with a number of rows (one row per course) and a number of columns. For each course, the table specifies the instructor who is going to teach the course, the room in which the class is going to be taught, etc., under appropriate columns. The person who designs this course schedule essentially uses a *pencil-and-paper* approach. He/she starts out with a blank table on a piece of paper that has one empty row per course. He/she then gradually schedules the instructors and the classrooms for each course following certain constraints. These constraints are imposed by certain facts, such as: which instructors have the expertise to teach which courses, which room is big enough to hold the maximum number of students allowed in that course, etc. The designer will make some guesses, and then use trial and error to manually come up with a correct schedule. Experience shows that this process of trial and error is quite hard, and mistakes are invariably made. A tool such as Knowledgesheet can automate this process for the designer. The idea behind the Knowledgesheet approach is to let the designer just state the constraints regarding instructors' expertise, room capacities, etc., in the various empty cells of the table. The possible ways in which the boxes in the table can be filled is also stated (domain or set of values that a particular slot or box can take). This domain information as well as the constraints are collected in a program, which is then executed to compute the results. The computed results are then displayed in the table. So given the Knowledgesheet tool, a designer's task boils down to stating the constraints and domains. An automatic constraint solver does the task of computing exact values that constitute a schedule of courses. If the designer doesn't like the solution, he/she can interactively change the constraints to obtain more acceptable solutions.

The Knowledgesheet tool consists of three parts: (i) a spreadsheet like interface for entering the constraint and domain information, (ii) an off-the-shelf back-end CLP(FD) engine that solves constraints, and, (iii) a protocol for communicating between the interface and the back-end engine. We use the ECLiPSe CLP(FD) System [1] as the back-end engine for solving the constraints. The Knowledgesheet interface provides facilities to enter constraints interactively. It also provides facilities for copying constraints entered in one box to another box with appropriate transformations (much in the fashion of a traditional spreadsheet). Thus, repetitive computations are not programmed as loops (or recursions), rather an instance of the computation is entered in a Knowledgesheet cell, and then this computation is copied on remaining cells.

The major advantage of the Knowledgesheet interface is that it facilitates problem modeling and interactive problem solving. Essentially, it forces the programmer (or problem solver) to think in an intuitive and structured (tabular) way. The problem solver still has to come up with all the constraints, and a model of the solution based on constraints—the interactive nature of the Knowledgesheet and its enforcement of the spreadsheet discipline make the task of the programmer, we believe, considerably easier. It also allows novice users to be able to interactively solve CSPs. The disadvantage is that it can be used for solving only a class of constraint satisfaction problems.

It should be emphasized that the Knowledgesheet Interface is an interactive tool and involves the programmer in finding a solution that is acceptable. Essentially, the programmer has to keep experimenting until a satisfactory solution is found. All the Knowledgesheet does is that it makes the process of interactive development of the solution easier. The responsibility of modeling the solution still rests with the programmer.

2 Constraint Logic Programming

The back-end engine of the Knowledgesheet is a constraint logic programming engine. Therefore, we give a brief overview constraint logic programming first.

Traditional logic programming languages, such as Prolog [7], are inefficient for solving complex search problems. These programming languages use the *generate and test* paradigm. In the generate and test paradigm, constraints (or tests) are applied *a posteriori* to detect failures after values have been chosen for the unknown variables in the problem model. Constraint languages remove this pitfall of LP by using consistency techniques. The idea behind these techniques is to actively use constraints to prune the search space *a priori* by removing combinations of values that cannot appear together in a solution. Consistency techniques divide the task of searching for a solution into two steps: (1) propagate the constraints as much as possible, and (2) choose values for some of the variables. Step (1) and (2) are applied until the problem is solved. Essentially, instead of using the generate and test paradigm, as done in traditional logic programming, constraint languages use *test and generate*. Constraints considered *a priori* that cannot be solved are suspended until enough values are generated that they can be solved.

CLP is a result of integration of consistency techniques into logic programming. The integration is done in such a way that the semantic properties of Logic Programming are preserved. CLP is a result of merging of two declarative paradigms: logic programming and constraint solving. The combination helps make CLP programs both more expressive and flexible. CLP can be viewed as a general and very efficient tool for solving NP-hard problems. More details on CLP can be found in many of the books and articles available [1, 2, 3]. We do assume for this paper that the reader has some familiarity with CLP(FD).

An important class of CLP languages is finite domain based constraint languages, referred to by the generic name CLP(FD), e.g., CHIP [1] and ECLiPSe [8]. In CLP(FD) languages the possible values that a variable can take are restricted to a finite set. Finite domain constraint are useful for solving many real life problems such as scheduling, time tabling, etc.

3 The Knowledgesheet System

3.1 The Knowledgesheet Paradigm

The Knowledgesheet paradigm for building application programs is inspired by the spreadsheet paradigm. The spreadsheet paradigm can be thought of as a pencil-and-paper solution technique. In the spreadsheet paradigm a user enters data and functions in a 2-dimensional grid of cells. The functions may take other cell values as inputs. Once the data and functions have been entered, they are evaluated and the resulting values displayed on the grid. For particular types of applications, especially those that involve 2-dimensional structures (e.g., financial accounts), the spreadsheet paradigm is perhaps the most convenient tool available. The spreadsheet paradigm, however, is limited because only data or arithmetic expressions can be entered in the cells (though most modern spreadsheet packages permit the user to write Macros, which are programs written in powerful Turing-machine equivalent programming language).

For many applications that have a 2-d structure, the spreadsheet paradigm is not enough, because of its limitations to functions only. For example, a schedule of classes at a University is a 2-dimensional structure, where data in various cells is related in complex ways. However, this relationship is not functional in nature, therefore, the spreadsheet paradigm is not adequate for developing schedules. However, the development of such applications on a spreadsheet becomes possible if we generalize functions to predicates or constraints. This generalization of spreadsheets to include constraints is termed Knowledgesheet by us. We chose the name Knowledgesheet because certain applications that fall in the realm of AI can now be programmed using the spreadsheet paradigm.

There are a large number of applications that can be structured as a 2-d grid of cells, where the values of different cells are related via complex constraints. Class scheduling at Universities, resource allocation, task scheduling, examination timetabling, work assignment, intelligent cost-accounting, many types of puzzles etc. fall in this category. The solution of each one of these problems involves building a 2-d table. The different data-items that constitute the solution table must satisfy certain constraints.

In the Knowledgesheet paradigm, constraints on a particular data item will be attached to the cell where that data-item is to appear. A cell corresponds to a variable (an empty cell corresponds to an unbound variable). Once all the constraints are laid out (analogous to attaching expressions to cells in the spreadsheet paradigm), the constraints are collected and solved (using a constraint solver). The values calculated as a result of this constraint solving are then displayed. One can choose any type of constraints (for example, constraints over reals, finite domain constraints, etc.), and once the type of constraints allowed is decided, the appropriate constraint solver has to be selected. Indeed, spreadsheet interface that allows constraints over reals [16], as well as spreadsheet interfaces for specific problems [12], have been designed in the past. For our project, we consider constraints over finite domains, as developing tools for interactively solving scheduling problems was one of our primary goals.

3.2 The Knowledgesheet Interface

Based on the Knowledgesheet paradigm, we have developed a spreadsheet based interface for solving a class of constraint satisfaction problems. The tool that has been developed is just a prototype. However, it has enough functionality to solve many interesting CSP problems. The prototype interface has all the major windows, dialog boxes and the required set of functions. The visual layout gives the user a good idea regarding how to use the system. In order to use the system, the user first develops a mental model of the problem in terms of 2-dimensional tables. The user needs to have some idea of the basic concepts of constraint programming, such as domain, constant, and constraints (just as in the traditional spreadsheet, the user must be familiar with arithmetic expression, data values, etc.). However, no detailed knowledge of any constraint programming language is needed.

The interface has two major parts: (i) a data area and, (ii) the scrolling view window. The data area is used for inserting data into the cell objects and the scrolling view window displays a part of the whole worksheet. The actual appearance of the interface is shown in Figure 1 and is not much different from an ordinary spreadsheet. The Knowledgesheet interface is implemented in Java and supports multiple tables. Using the interface the user specifies relations (or facts) and queries in a tabular form. The constraints involved in the query are next entered in the cells of the query table (the user chooses which table is the query table). Once the tables are filled, the system will assemble the CLP(FD) program and communicate it to the back-end CLP(FD) engine, which will compute values for the empty cells in the query table. The system will then display these results in the query table. This process is akin to ordinary spreadsheet, where arithmetic expressions are attached to cells of various tables, and the values of these expressions are displayed once they have been computed. Note that if there are multiple solutions to a given problem, then these solutions can be displayed one after the other when the user clicks on the `next solution` icon.

In the Knowledgesheet interface the columns are labeled `A, B, C, D, E, etc.`, and the rows are labeled `1, 2, 3, ...` (Figure 1). Thus, the cells are referred to in a manner identical to ordinary spreadsheets. There is also a data area on top where the value of the current active cell (the cell that was clicked on most recently) is displayed, just as in an ordinary spreadsheet. This area is also used for entering data in the current active cell. The Knowledgesheet can be quite large, one can scroll around the Knowledgesheet using the mouse. The Knowledgesheet window can be increased or decreased in size just like any ordinary window. The various other operations such as selecting a cell or a range of cells, entering data, text, a constant, a domain, a constraint or resetting a cell are performed by selecting the appropriate command from a floating pop-up menu that appears when the left mouse button is clicked. Menus are also available for loading data (constraints, domain values, etc.) from a file, saving data, etc.

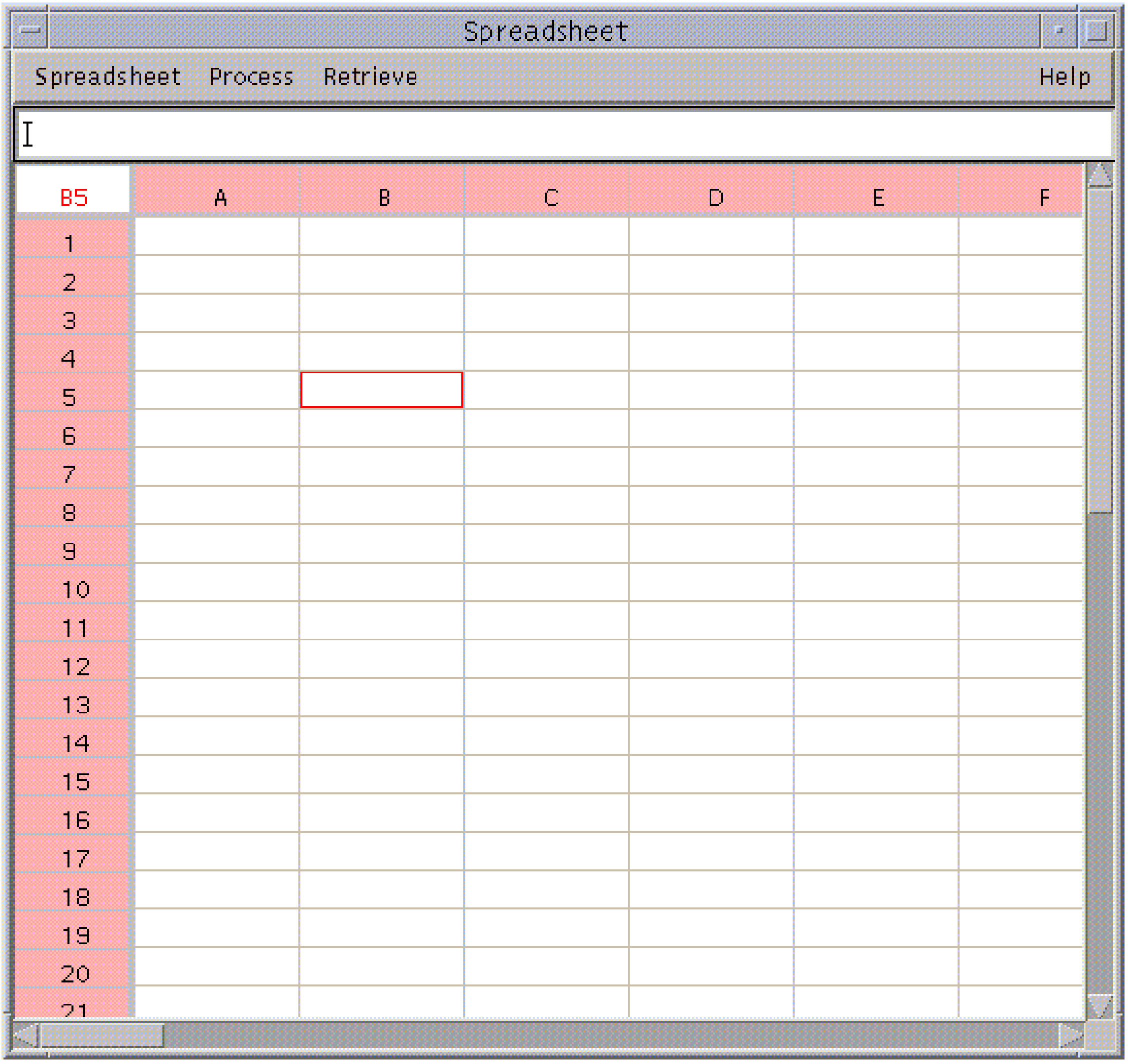

Figure 1: The Knowledgesheet Interface

The primary operation that the user will be performing using the Knowledgesheet Interface are the following: (i) entering auxiliary tables, (ii) entering the query table, (iii) entering constants, domain values, and constraints in cells of the query table. The user may further copy the constraints and domains in a cell to other cells. These are copied with appropriate transformation applied (just as in an ordinary spreadsheet). For example, the constraint `A1 + B1 #= 2` in cell `C1` when copied to cell `C2` will become `A2 + B2 #= 2`. (Note that `#=` is the *forward checkable* CLP(FD) operator [1, 2] for checking equality over domain variables.) Both absolute reference as well as relative reference can be made to a cell, just as in an ordinary spreadsheet. The copying feature gives the user the power to copy the active-cell to another cell or to an array of cells (selected with the mouse). The references between cells may be either absolute or relative in either their horizontal (A, B, C, ...) or vertical index (1, 2, 3, ..). All copies of an absolute reference will refer to the same row, column or cell whereas a relative reference refers to a cell with a given offset from the current cell.

Most of the operations use mouse for input and event control, keyboard is used to type data (constants, domains, constraint). Most of the events associated with the Knowledgesheet need to be performed with the help of the mouse. The range selection operation is also supported.

At present, the constraints that use builtins (e.g., `alldifferent`, `element`, `cumulative`, etc.)

have to be manually entered. Work is in progress to to provide these in the interface as buttons. Thus, given the button for the `alldifferent` constraint, the user will select a range of cells, then click on the `alldifferent` button. This will auto-generate the `alldifferent` constraint for variables corresponding to the selected cells.

3.3 Autogeneration of the CLP(FD) Program

Once the user has created the auxiliary tables, the query tables, and entered the data, domains, and constraints in these tables, he/she selects the `solve` option in one of the drop-down menus. This selection causes a CLP(FD) program to be autogenerated from the constraints and domain values in the cells of the various tables. This program is then executed by a CLP(FD) system. The process of program execution is transparent to the user. The computed results are then displayed in the query table.

The autogenerated program contains a series of facts for each of the auxiliary tables (one fact per row of the table). The auxiliary tables are simple maps and are mostly used for mapping symbolic values to integers, since domain values in CLP(FD) system can only be integral. The query table is translated into a query, and contains all the domain declarations and the constraints posted in each of this cell. Each cell in the query table is treated as an unbound variable whose value is sought. The query along with the facts is passed to a CLP(FD) engine, that solves the query. The engine writes the value of the unknown variables in a file, from where the front-end interface reads the values and displays them in the table.

The appropriate labeling predicates are added automatically to the query in the autogeneration process. The user could be given a range of choices regarding what labeling strategy to use [1, 2]. This could be done by providing appropriate buttons, one button for each strategy. The user will then just click on the appropriate button to choose the desired strategy. However, the inclusion of labeling strategy buttons presupposes a lot of knowledge on users' part regarding how a CLP program is executed, and thus will be useful only for advanced users. In the current version of the system, a default labeling strategy is used which is `deleteffc` [1].

3.4 Structure of the Generated Program

The structure of the constraint logic program generated by the Knowledgesheet interface is based on the constraint (or test) and generate methodology of CLP. The Knowledgesheet interface will always produce a non-recursive program for the constraint part, because of the non-recursive nature of the spreadsheet paradigm. The information contained in each cell of the spreadsheet is mapped into a constraint. The generate (labeling) section of the synthesized program (which is automatically added) is recursive, since it mainly consists of labeling predicates, which are recursive.

There is only one main rule involved in the program, which is generated from the query-table (the table in which the user is trying to represent a solution). The autogenerated program file thus contains a section containing CLP(FD) facts corresponding to the auxiliary tables, and a main rule corresponding to the query table. The main rule contains a domain declaration part (all domain declarations from the query table are placed here), a constraint declaration part (all constraints entered in the cells of the query table are placed here), and the labeling section which is automatically added (default labeling strategy is deleteffc). The final part of the main rule is autogenerated CLP(FD) code that deals with extraction of the result so that it can be passed to the front-end interface for display. Example auto-generated programs (e.g., for the class-scheduling problem) can be found in [9].

4 Examples

We next give a number of examples to illustrate the Knowledgesheet approach.

4.1 Scheduling Resources

Resource scheduling is frequently needed in various situations. Consider a large retail store that is open 14 hours a day (e.g., Walmart); such stores typically hire a number of managers. The schedule of these managers has to be drawn in such a way that, for example, at least one manager is present at any given time. Also, the schedule has to be fair to all the managers. Thus, some of the constraints under which the schedule has to be drawn are: every manager should get 2 days off during the week; every manager should be assigned the same number of morning, afternoon, and evening shifts over the long run; a manager who is present during the evening shift to supervise the closing of the store should not be assigned to a morning shift the following day to supervise the opening of the store; etc. In most of these store, this assignment is done manually, primarily because the constraints can vary in time (e.g., store hours may change, a manager may call sick), frequently resulting in unfair and/or wrong schedules.

Let's try to solve this scheduling problem using the Knowledgesheet approach. Assume that the retail store opens from 8AM to 11PM, 7 days a week. Assume that there are 5 managers (Bill, Mary, John, Gary, and Linda) in the store each of whom is supposed to work 40 hours per week. There must be one manager present at any given time when the store is in operation. Each manager is supposed to get 2 days off every week (obviously, these 2 off days cannot always be Saturday and Sunday for every manager, since the store is open on Saturday and Sunday too). Each manager works no more than 8.5 hours a day (which includes half hour for lunch), either in the morning shift (8:00 PM to 4:30 PM), or the midday shift (10:00 AM to 6:30 PM), or the evening shift (2:30PM to 11PM). The schedule should obey the following constraint: a manager doing an evening shift should not be scheduled for a morning shift for the immediately following day. The schedule should also take manager preferences into account to the extent possible (e.g., a manager may want specific days off during a particular week). The schedule should be fair to all the managers: in the long run everybody should have an equal number of days of morning mid-day and night shifts. Most managers prefer the two off-days to fall on Saturday and Sunday. The scheduler should be fair in assigning Saturdays and Sundays as off-days to all the managers. In the long run, everybody should get the same number of Saturdays and Sundays off.

	A	B	C	D	E	F	G	H	I
1		MON	TUE	WED	THU	FRI	SAT	SUN	
2	Bill								
3	Mary								
4	John								
5	Gary								
6	Linda								
7									

S O L V E

Figure 2: An Empty Table

Suppose we want to draw a schedule for 1 week (we choose one week to keep the example simple). To solve this problem, the user will invoke the tool, which will display a table just like most spreadsheets do when they are invoked. The user will then label the columns and rows appropriately (with the names of the managers and days of the week). This is shown in Figure 2. Because we are drawing a schedule for only one week, the constraint for ensuring fairness of allocating Saturdays and Sunday as the two days off cannot be enforced, so we'll ignore it. We'll try to enforce all the other constraints. At this point the user would have defined an empty table with 5 rows (one for each manager) and 7 columns (one for each day).

We will assume that there are three shifts: morning, midday and evening. These will be denoted by numbers 5, 2, and 4 respectively (see later for justification for this peculiar choice). 0 will be used to indicate a manager's day off. The cells in the table will finally display either the shift that a manager is suppose to be present for during that day, or 0, which means (s)he has the day off. Note that each cell in the table can only take values from the domain $\{0, 2, 4, 5\}$. The user will indicate this finite domain by entering this set in each cell (of course, the user has to enter this domain in one cell and then copy it in the remaining cells just as in a regular spreadsheet). This is shown in Figure 3. For example, in the cell C2, the user will enter, `<D>[0,2,4,5]<D>` to indicate the domain of cell `C2`. This will be copied in all other cells. However, to keep figure uncluttered, the domain values of all the cells are not shown in Figure 3. The use of finite domains is very important in constraint solving, because otherwise the search space may be very large. In fact, the search space is quite large even now (with 35 total cells each of which can be assigned any one of the 4 values from the set $\{0, 2, 4, 5\}$ there are 4^{35} possible candidates to check for an acceptable solution), but when we enter further constraints, these constraints will be used to *actively prune* this massive search space and produce a solution.

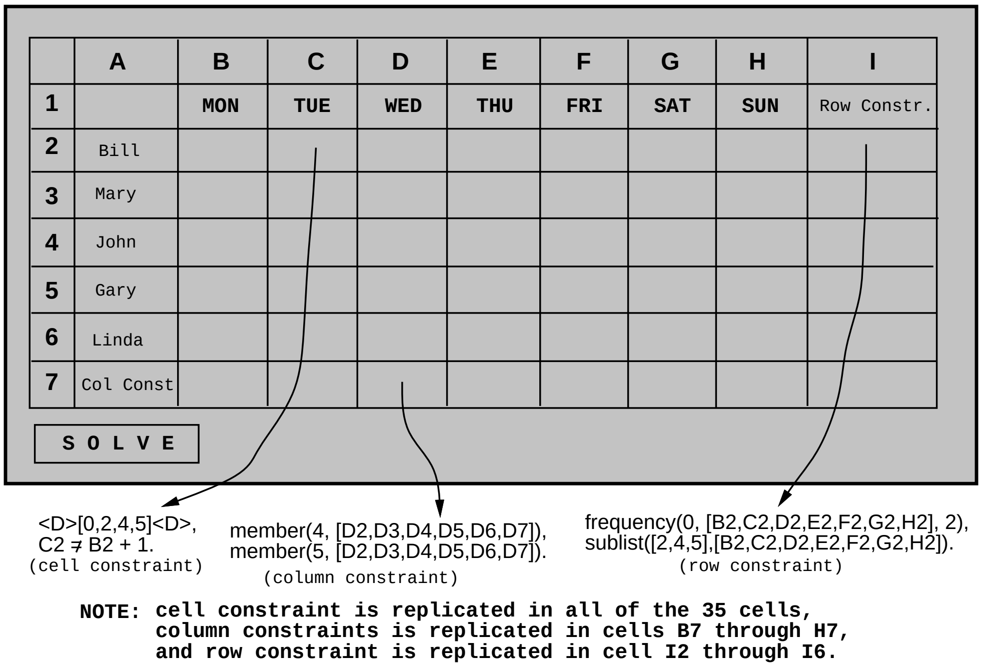

Figure 3: Table after Adding Further Constraints

Once the domain value has been entered, the user will indicate the other constraints that each cell must obey. Some of these constraints are local (i.e., apply to each cell), while some have to be applied to an entire row or entire column. The first constraint that we need to attach to each cell (except those in the first column) is that a person cannot be assigned to the morning shift if (s)he has worked on an evening shift the previous day. This constraint can be entered by adding, for example, in cell C2 the constraint: $\texttt{C2} \neq \texttt{B2 + 1}$. The constraint is shown in figure 3, and is labeled *cell constraint*, since it is associated with each cell. This constraint can then be copied everywhere in the table except in column `B` (while copying appropriate transformations are applied just as in a regular spreadsheet, e.g., when copied in cell `D2`, this cell constraint will be transformed to D2 in [0,2,4,5], D2 $\neq$ C2 + 1). One can now see why we chose the values 2, 4, and 5 to denote midday, evening, and morning shifts respectively, and 0 to indicate the day off. We wanted to make sure that only the value assigned to morning shift is 1 more than that for the evening shift, so that this constraint can be expressed numerically.

Now we have to ensure the global constraints, namely: (i) at least one manager is present at any time during the day, (ii) no manager works more than 5 days a week, and (iii) every manager has more or less the same proportion of morning, midday and evening shifts. These are global constraints because they involve more than one cells, and are to be enforced over each column or row in the table. We'll add an extra column to the right of the table, and an extra row at the bottom of the table where these constraints over rows and columns will be placed.

Let's consider constraint (i) first. The person defining the constraints can use his/her knowledge (domain-specific knowledge) of the fact that if there is at least one person present in the morning shift and at least one person present in the evening shift, then we are assured that there is at least one manager present at any given time when the store is in operation. Thus, the column constraint that we need, for example for column D is member(4, [D2, D3, D4, D5, D6]) and member(5, [D2, D3, D4, D5, D6]). This constraint is placed in row 7 in column D. Note that [D2, D3, D4, D5, D6] refers to a list of values consisting of those computed for cells D2, D3, D4, D5, and D6. This constraint will then be copied across the extra row at the bottom (row 7). The member constraint is a builtin constraint provided as a button.

Let's now consider row constraints (ii), namely, every manager has exactly 2 days off, and constraint (iii), namely, no manager works more than 5 days a week and that the shifts are fair. To check for (ii) we have to ensure that in each row there are exactly two 0s (2 days off). Constraint (iii) can be more or less achieved by ensuring that every manager is working at least one morning, one midday, and one evening shift during the week. Constraint (ii) can be checked by defining an auxiliary table that maps days-off to 1, and morning, midday, and evening shifts to 0, and then checking that there are exactly two 1s in each row resulting from the mapping. To each of the cells in the extra column to the right (Column I), we'll add the appropriate constraint. For example, in the top most cell in the column I (cell I2), we'll add the constraint: frequency(0, [B2, C2, D2, E2, F2, G2, H2], 2). The constraint frequency is provided as a builtin; frequency(N, L, K) checks that the value N appears in the list L exactly K times. It can be implemented using the element constraint provided in most clp(FD) systems [1]. Constraint (iii) can then be checked by using the sublist builtin predicate. For the first row, this constraint will be: sublist([2,4,5], [B2, C2, D2, E2, F2, G2, H2]), where the sublist(S, R) builtin checks that the list S is a completely contained in the list R.

The final table decorated with all the different types of constraints is shown in the Figure 3 (all the constraints in each cell are not shown to avoid clutter). Once the constraints have been defined, the user will click on the SOLVE button, and tool will calculate the values for all the 35 variables using a constraint solving engine. These values will be then displayed. Figure 4 shows one solution.

4.2 The 3x3 Grid Puzzle

Consider the well-known 9-squares puzzle, where all the squares of a 3x3 grid have to be filled with integer values from 1 to 9 with no two squares having the same value. The sum of the values in the squares across all rows, columns and major diagonals must be 15.

To solve this problem using the Knowledgesheet paradigm, we will define a 3x3 query table in which we will attach appropriate constraints (Figure 5). Suppose the table is defined in columns B, C, and D, and rows 3, 4, and 5. Then in the cell B3 we should add the constraint B3 + C3 + D3 #= 15. This constraint should then be copied in the cells B4 and B5. The constrains C3+C4+C5 #= 15 should be next attached to the cell C3, and copied to the cell D3. The cell B3 should be augmented with the constraint B3+B4+B5 #= 15. The diagonal constraints should be next specified,

	A	B	C	D	E	F	G	H	I
1		MON	TUE	WED	THU	FRI	SAT	SUN	Row Constr.
2	Bill	5	5	5	2	0	4	0	
3	Mary	4	2	0	5	5	5	0	
4	John	0	5	2	4	2	0	2	
5	Gary	0	2	4	0	4	2	5	
6	Linda	0	4	2	2	0	2	4	
7	Col Const								

S O L V E

Figure 4: Displaying a Solution

and, finally, the domain of each cell should be set to `1..9`. The constraints in each cell of the grid are shown below.

```
Cell    Constraints
B3      B3 + C3 + D3 #= 15,    B3 + B4 + B5 #= 15
C3      C3 + C4 + C5 #= 15
D3      D3 + D4 + D5 #= 15
B4      B4 + C4 + D4 #= 15
B5      B5 + C5 + D5 #= 15,    B5 + C4 + D3 #= 15
D5      B3 + C4 + D5 #= 15,    alldifferent([B3, B4, B5, C3, C4, C5, D3, D4, D5])
```

	A	B	C	D	E	F	G	H	I
1									
2		2	6	7					
3		4	8	3					
4		9	1	5					
5									
6									

S O L V E

Figure 5: The 3x3 Puzzle

In the above table, all the cells do not contain constraints. The cells without any constraints cells can have constraints, but that may introduce duplicate or redundant constraints. Introduction of redundant or duplicate constraints does not seriously compromise performance, however, the user should be careful while using them. To model the problem and to display the solution we need a mapping table (auxiliary table, not shown), which maps symbolic names attached to the cells in the query table to actual integer values. Once the mapped table and query tables have been specified, the back-end CLP(FD) engine is invoked. The solutions are computed by the engine and displayed on the Knowledgesheet interface as shown below.

The Knowledgesheet interface can also be used to graphically solve cryptarithmetic puzzles, such as `SEND + MORE = MONEY` (Figure 6). The constraints are specified for just the rightmost column, and then copied to other columns. The constraints can then be solved, and the solution displayed. In the figure, we have a auxiliary table that associates the letters `s, e, n, d, m, o, r` and `y` with empty cells (the constant letters are shown in column G and the empty cells in the column `H`). The query table consists of the rectangle with diagonal corners `B2, E2, B5,` and `E5` plus the cell `A5`. The user enters the constraint for computing the sum in cell `E5,` and the carry in cell `D2`. The expression in `E5` is copied to cells `B5, C5`, and `D5`, the expression for carry in cell `D2` is copied into cells `C2` and `B2`. In the figure, we have shown variable names on the cells, in reality they will be blank. To make the association between multiple occurrence of the same letter (e.g., `O` that occurs in `C4` and `D5`), we set equality constraints between each letter and its position in the column `H`. Thus, the constraints `C4 #= H7` and `D5 #= H7` will have to be explicitly generated (this is done by clicking with the mouse on the two cells to be equated and then clicking on the button corresponding to the equality operator. Finally, the `alldifferent([H2,H3,H4,H5,H6,H7,H8,H9])` constraint is generated and placed in one of the cells of the query table. Now the specification is complete and these constraints can be solved to display a solution.

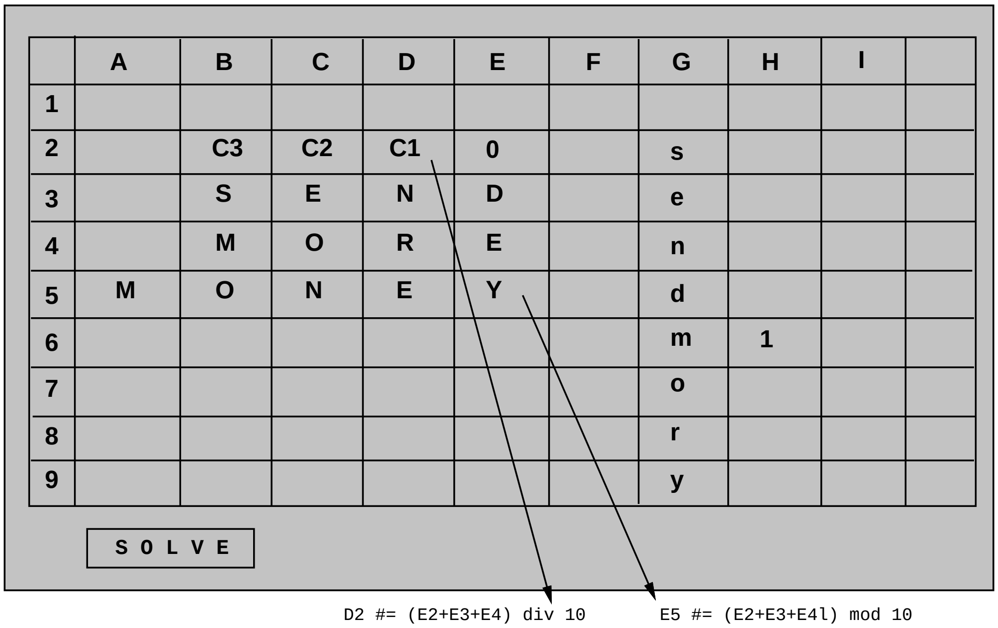

Figure 6: Cryptarithmetic Puzzle

4.3 Course Scheduling

Course scheduling is another CSP that has a 2-d structure, and that can be solved with the help of the Knowledgesheet Interface. Consider the problem of designing the course schedule in the Computer Science Department at New Mexico State University. There are 12 research faculty members each of who teach exactly two courses per semester. There are 2 teaching faculty members who teach exactly 4 courses per semester. There are 4 class rooms where all the courses are to be scheduled. A course has either three 50 minute lectures which are to be scheduled Monday, Wednesday, and Friday (same time on all three days) or two 75 minute lectures, which means it is scheduled Tuesday and Thursday (same time on both days). The time slots on Monday, Wednesday, Friday as well as Tuesday, Thursday are fixed (i.e, 50 minute classes start 30 minutes past the hour every hour starting from 7:30am on M, W, F, and the 75 minute classes are held 7:30–8:45, 8:55–

10:10, 10:20–11:35, 11:45–1:00, 1:10–2:25, 2:35–3:50, 4:0–5:15; there is a 10 minute interval between any two consecutive lectures).

There are many constraints to be considered in designing the schedule: each instructor has the expertise to teach only some of the courses offered. An instructor cannot teach two courses at the same time. A room cannot hold two courses at the same time. Instructors may have preferences for the day and time when they want to teach a course (e.g., many instructors prefer a Tuesday-Thursday schedule). Two freshman courses should not be scheduled in the same slot (because a student taking one of them, may also take the other). Likewise for sophomore, junior, and senior level courses. For each course, we know the maximum number of students who can take this course. The room assigned to that course should have at least that many seats.

To solve the problem using the Knowledgesheet system, a blank table is created and all the course names entered in the course column. For each course, under the instructor column, the domain containing instructor name is entered for each cell (the domain may be different for different courses). Constraints are entered next. In most cases, the constraints have to be entered in one row or column and then copied across to other rows and columns. There are several other auxiliary tables as well that map lecture time periods to integers, instructor names to integers, class rooms available to integers, etc (in most CLP(FD) systems domains can only contain integers). More details of this problem can be found in [9]. The Knowledgesheet system allows a non-expert user to design working course-schedules, and perhaps, we believe, is the single most significant application of the Knowledgesheet system.

5 Discussion

5.1 Advantages of the Knowledgesheet Approach

For each problem considered, a specialized software system can be designed using standard operations research techniques, such as integer programming, that will generate the optimal schedule. However, the constraints are fixed in advance in these specialized software systems, thus they are too rigid to be useful in dynamic environments where constraint may arise dynamically on the fly. For instance, in the retail store example, the policy that the store uses for assigning managers might change (e.g., for some reason the company may decide that for the two of the days in the week, each manager should split his/her eight hour shift into two four hour shifts, one in the morning and one in the evening for certain parts of the year). Thus, ideally there is a need for a software tool in which a non-expert user can interactively enter the various constraints and preferences and obtain a satisfactory schedule that satisfies these constraints. This non-expert user should also be able to enter as constraints specialized knowledge that he/she has acquired about that particular situation to obtain a customized schedule. The Knowledgesheet approach makes all this possible.

Let's consider the course scheduling problem once again: the constraint under which the schedule has to be developed may change dramatically from one semester to another. The set of instructors might change, their preference for time and the courses they wish to teach may change. The students might demand a new course, that has to be added to the course catalog and scheduled. Thus, for the course scheduling problem as well, one would wish to have a tool that one can use to interactively enter the constraints, and automatically obtain a schedule that obeys those constraints. In most universities and colleges, course scheduling is done by hand, resulting in under-use of resources and many dissatisfied instructors and students, especially if the number of courses to be scheduled is large.

Likewise, in the retail store example, we would like the senior-most manager to draw up the schedule, given that the constraints under which the schedule is to be developed may change from one week to another (for example, in a given week if one or more managers are on leave, then the scheduling will completely change). Indeed, this is what is practiced in most retail stores, where a senior manager manually draws up the schedule of managers who are under him. There are many factors that might change from time to time, affecting the constraints under which the schedule has to be drawn. For example, during the Christmas Holiday season many stores have extended hours, or wish to have more managers on duty at all times. A scheduling tool built using standard Operations Research (OR) technology will have to anticipate all the constraints under which the schedule must be drawn, which is clearly not possible, given the dynamic nature of the business. An interactive tool that enables the non OR-expert store manager to enter all the constraints that he wishes to be considered in drawing up of the schedule, and then obtain a schedule from these constraint is highly desirable. Such a tool will not only greatly improve the efficiency with which the schedules can be drawn, it will also produce fairer schedules.

5.2 Disadvantages of the Knowledgesheet Approach

The major disadvantage of the Knowledgesheet approach is that it is applicable to only a small class of CSPs—those that fit the tabular structure. Also, in the case of many problems, e.g., course scheduling, the search space may be too large, if a naive approach is used. In fact, using the simple approach outlined in the example in section 4.3 above, Knowledgesheet cannot be used for scheduling more than 20 courses. To schedule larger number of courses, one has to use builtin constraints such as `cumulative` [2, 6, 4, 10, 11]. For really hard or very large problems, ultimately, the user has to think about how the constraints are being solved, and has to choose appropriate heuristics as well as appropriate builtins for the labeling phase, to get a solution.

Another disadvantage of the approach is that if there is no solution (the users' specification is overconstrainted), the system will not give any help regarding how to get out of this failure. Of course, the user could remove the constraints one by one and see if he/she can obtain a solution. The interactive nature of the Knowledgesheet system will be helpful in this process.

5.3 Interactive Problem Solving with Knowledgesheet

For very large problems, e.g., course scheduling for 50 courses, the Knowledgesheet tool can still be useful. For most scheduling problems, the person designing the schedule can manually come up with a partial solution (e.g., coming up with a schedule for, say, 40 of the 50 courses), for the rest of the schedule, the Knowledgesheet system can be used. Essentially, the user will fill out all the constraints and the domains in the various cells of the query table, then he/she will enter constants for those values that he/she can determine manually. For computing the rest of the values the back-end constraint solver of the Knowledgesheet system will be called. Given the user's initial choice, there may not be any solution feasible, however, the user can experiment by incrementally finding the solution row by row, or by slightly changing the manually determined solution, or both. Thus, the interactive nature of Knowledgesheet can be very helpful in solving large problems. Essentially, the Knowledgesheet system permits cooperative problem solving—the user and the CLP(FD) system cooperate via the user interface to solve a problem. The same situation applies to an overconstrained system when the system cannot produce a solution. In this case as well, the user can incrementally remove constraints and finally find a solution, as discussed in the previous subsection.

5.4 Previous Work

The concept of a spreadsheet interface for general purpose programming has been proposed by, among others, Yoder and Cohen [14]. They show how spreadsheets can be utilized as a programming tool via their spreadsheet-based tool called Mini-SP. However, the application of Mini-SP system is limited. It can be used for solving some well-known problems, such as sorting, and temperature gradient simulation.

Most of the spreadsheet based programming tools are for specific task. Lai, Malone and Yu developed *Object Lens* system [13], a spreadsheet for cooperative work. They claim that the Object Lens system allows unsophisticated computer users to create their own cooperative work applications using a set of simple, but powerful building blocks. The Object Lens system integrates facilities for hypertext, object-oriented databases, electronic messaging, and rule-based agents. It is a semi-formal system, which represents knowledge in a way that both people and their computational agents can process intelligently. Hofe has developed ConPlan [15] interface specifically for scheduling nurses. The ConPlan interface uses tabular input. It is designed for a specific application and is far less general than the Knowledgesheet sytsem.

The SD-spreadsheet, a constraint-spreadsheet based application for solving financial planning problems has also been developed by Shvetsov, Kornienko, and Preis [16]. The system is based on a variant of interval constraint programming. SD-spreadsheet is implemented in C++ in the Windows environment. SD-spreadsheet is a dedicated system for dealing with only financial data. The computational capabilities are based on solving system of constraints. In the SD-spreadsheet relations between cells are defined via constraints which look like ordinary arithmetic expression but have different semantics. The value of any cell occurring in the constraint can be calculated, if the values of the other cells are known. In Knowledgesheet, the constraint and data operations in cells are similar to the SD-spreadsheet, however, the Knowledgesheet is based on CLP(FD) while the SD-spreadsheet system uses interval constraints and constraints over reals. It is also perhaps not as general as our system.

Recently, Configuration Spreadsheet has been developed by Renschler [12] for configuring the mobile phone network transceivers. The configuration spreadsheet uses CLP(FD), however, it is not as general as our Knowledgesheet system.

6 Conclusions And Future Work

In this paper we presented the design and implementation of the Knowledgesheet Interface. The Knowledgesheet interface is a generalization of the traditional spreadsheets that enables interactive solving of a class of constraint satisfaction problems. A large class of CSPs can be modeled as consisting of values arranged in a 2-d table, where the values are related to each other via constraints. In the Knowledgesheet interface, the user states only these constraints, which are then solved to obtain a solution. The Knowledgesheet interface enables non-experts to solve CSP problems.

We believe that the Knowledgesheet interface has many potential applications and that it can be used by novice users for solving many practical problems, such as course scheduling, resource allocation, task scheduling, examination time-tabling, work assignment, intelligent cost-accounting, etc. A software patent for the Interface is pending [5]. The Knowledgesheet tool also has pedagogical value, as we believe that it is a good aid for teaching solving of CSPs and puzzles to non-expert users, since its graphical interactive nature allows for experimentation.

A Knowledgesheet-like approach can also be applied for building interfaces for solving engineering and other types of decision support problems. Engineering design problems can also be modeled in terms of a set of variables related via constraints. One can envisage a graphical tool in which the various design constraints are attached to the drawing of the part being designed; the constraints are next solved to obtain concrete values for the various design parameters. Likewise, decision trees can be graphically programmed: the nodes of the decision tree that contain constraints are mapped to the various cells of the Knowledgesheet. Thus, a class of expert systems can be quickly designed and implemented by non-expert users. These extensions to the Knowledgesheet tool to support engineering graphics and decision tree specification capabilities are planned in the future.

The current interface for Knowledgesheet is specialized for problems involving finite domain constraint solving. In the future, the Knowledgesheet will be extended to allow for constraint solving over other domains (e.g., reals, intervals). It should be noted that such extensions require only a trivial amount of work.

We plan to extend the Knowledgesheet system further in the future, to make it more sophisticated: e.g., supporting multiple sheets, buttons for more powerful builtin constraints, buttons for choosing more sophisticated labeling strategies, etc. We also plan to support hierarchical constraints, as this is crucial for solving certain type of practical problems. Many features that are available in commercial spreadsheets can be included in the Knowledgesheet to make it more easy to use (e.g., ability to change sizes of rows and columns, allow use of keyboard for operations that are currently only possible through the mouse, etc.). Work on incorporating these extensions is in progress.

References

[1] Van Hentenryck, P., Constraint Satisfaction in Logic Programming, The MIT Press, Cambridge, MA, England, 1989.

[2] Marriott, K., and Stuckey, P. J., Programming with Constraints: An Introduction, The MIT Press, Cambridge, MA, England, 1998.

[3] Cohen, J., Logic Programming and Constraint Logic Programming, The Computer Science and Engineering Handbook, pp. 2066-2093, CRC Press, Inc., Florida, US, 1996.

[4] H.-J. Goltz and D. Matzke. University Timetabling Using Constraint Logic Programming, Proc. Practical Aspects of Delcarative Languages, Springer Verlag Lecture Notes 1551, 1999.

[5] G. Gupta and S. F. Akhter. Knowledgesheet: A Spreadsheet-based Interface for Interactively Solving Scheduling Problems. Patent pending. Feb. 1999.

[6] Boizumault, P., Delon, Y., and Peridy, L., Constraint Logic Programming for Examination Timetabling, The Journal of Logic Programming, pp. 217-233, 1995.

[7] Sterling, L., and Shapiro, E., The Art of Prolog, The MIT Press, Cambridge, MA, 1994.

[8] Meier, M., ECLiPSe User's Manual, International Computers Limited and IC-Parc, 1997.

[9] S. F. Akhter. Knowledgesheet: A User-Interface for Solving Constraint Satisfaction Problems. Master's thesis. Laboratory for Logic, Databases, and Advanced Programming. New Mexico State University, Las Cruces, NM, 88011. July 1998.

[10] Henz, M., and Würtz, J., Using Oz for College Timetabling, In proceedings of the 1995 International Conference of the Practice and Theory of Automated Timetabling, Edinburgh, Scotland.

[11] Colorni, A., Dorigo, M., and Maniezzo, V., Metaheuristics for High-School Timetabling, Computational Optimization and Applications, 9(3): 277-298, 1998.

[12] Renschler, M., Configuration Spreadsheet for Interactive Constraint Problem Solving, Conference Proceeding, Proc. Practical Applications of Constraint Technology (PACT), 98.

[13] Lai, K., Malone, T., W., and Yu, K., Object-Lens: A "Spreadsheet" for Cooperative Work, ACM Transactions on Office Information Systems, 6(4): 332-353, 1998.

[14] Yoder, A., G., and Cohn, D., L., Real Spreadsheets for Real Programmers Proc. International Conference on Computer Languages, IEEE, pp. 20-30, 1994.

[15] Hofe, H., M., ConPlan/SIEDAplan: Personnel Assignment as a Problem of Hierarchical Constraint Satisfaction, Conference Proceeding, PACT 97, pp. 257-271.

[16] Shvetsov, I., Kornienko, V., and Preis, S., Interval Spreadsheet for problems of financial planning, Conference Proceeding, PACT 97, pp. 373-385.