# SPREADSPACES: Mathematically-Intelligent Graphical Spreadsheets

Nachum Dershowitz*[1] and Claude Kirchner[2]

[1] School of Computer Science, Tel Aviv University, Ramat Aviv 69978, Israel
`Nachum.Dershowitz@cs.tau.ac.il`
[2] INRIA Bordeaux – Sud-Ouest, Talence, France
`Claude.Kirchner@inria.fr`

**Abstract.** Starting from existing spreadsheet software, like Lotus 1-2-3®, Excel®, or Spreadsheet 2000®, we propose a sequence of enhancements to fully integrate constraint-based reasoning, culminating in a system for reactive, graphical, mathematical constructions. This is driven by our view of constraints as the essence of (spreadsheet) computation, rather than as an add-on tool for expert users. We call this extended computational metaphor, *spreadspaces*.

*We believe that research towards more general and realistic constraint solving frameworks has to go on in parallel with the effort to make fewer and fewer requests to the user. In other words, users should be asked only for as much as they want to give the system. This amount of information (decided by users but with a minimum set by the system below which most precision is lost) is then used by the system to construct the whole constraint problem.*

*—Ugo Montanari and Francesca Rossi [18]*

## 1 Overview

Our ultimate goal in this work is the design of a graphical environment for spreadsheet-like computations, including solving and optimization, wherein the graphical interface serves as an input medium, in addition to its traditional output rôle. Changing a displayed value, be it graphical or textual, results immediately in the appropriate changes to values it depends on. This integrated system, with its transparent graphical mode of interaction, will dramatically extend the capabilities of existing commercial products, providing sophisticated mathematical intelligence for the computationally naïve.

Spreadspaces do not have the look or feel of spreadsheets [20], or even of graphical spreadsheets, but rather that of a graphical user interface. At the design level, the system serves as a graphical design environment with definable and extensible graphical objects. Thus, educators, for example, can design

* This author's research was performed while on leave.

spreadspaces within which schoolchildren can solve problems and investigate variants.

Target users are everyday users of personal computers. Examples of spreadspace applications include:

– Exploring mathematical relations (see the baguette example below).
– Simulating physical devices (such as a pendulum).
– High school problem solving.
– Contingent ("what-if") financial calculations (for example, home loan planning, tax computations).
– Logical and mathematical puzzles (for example, map coloring, crypto-arithmetic).

Today's spreadsheets provide ad-hoc constraint solving [8, 16, 7], mainly via linear programming, and incorporate sophisticated graphical output. But these features are patched on top of the basic spreadsheet, making the interface difficult and limiting its general use.

The following sections lead us from minor cosmetic enhancements of current spreadsheets, through sophisticated tools for the incorporation of mathematical intelligence, to user-friendly graphical spreadspaces.

## 2 Cosmetic Constraints

*The goal was to give the user a conceptual model which was unsurprising – it was called the principle of least surprise. We were illusionists synthesizing an experience. Our model was the spreadsheet – a simple paper grid that would be laid out on a table. The paper grid provided an organizing metaphor for a working with series of numbers. While the spreadsheet is organized we also had the back-of-envelope model which treated any surface as a scratch pad for working out ideas.*

*—Bob Frankston (coinventor of Visicalc)*

We begin with some simple "cosmetic" improvements to modern-day spreadsheets. The central notion is that of *constraints*, which are boolean (true/false) formulæ, involving comparisons and conditionals, that are required to evaluate to `true`. Constraints extend ordinary formulæ by allowing the user to specify more general relations between variables. Guaranteeing the truth of a constraint forces the variables it involves to take on appropriate values. Values that make a constraint true are called a *solution*. Typical constraints involve inequalities (such as `Years` < `80`), type information (`Years: Integer`), and logical combinations (`(Years=62 and Gender=Female) or (Years=65 and Gender=Male)`). To satisfy a constraint, the variables (like `Years`) appearing in it are set by the system to appropriate values by computation and solving mechanisms.

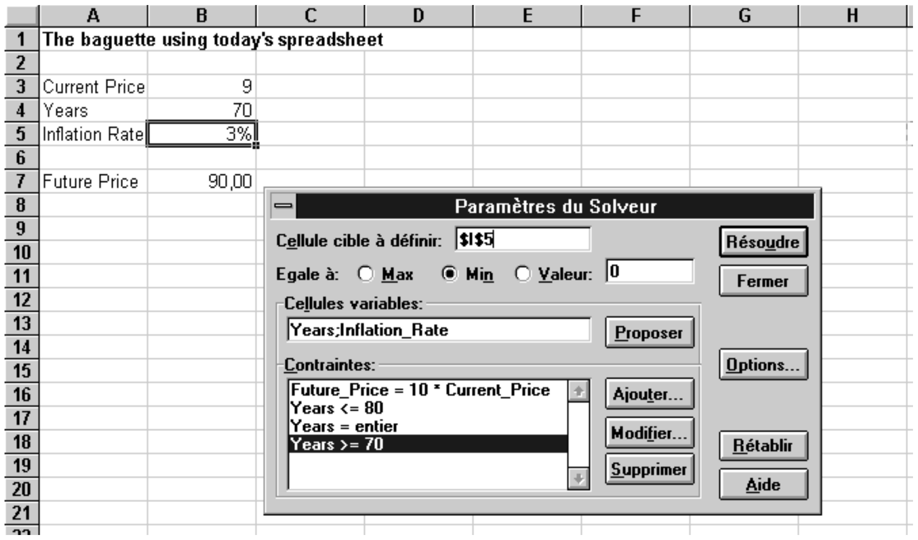
To incorporate constraints into the spreadsheet paradigm, we do the following:

1. Add a new kind of cell for constraints. One should be able to switch the type of a cell from boolean to constraint and back again easily. This facilitates debugging a set of constraints.

2. Only cells with an empty value are considered to be variables. Special flags can be used to indicate preference for maximal or minimal possible solutions. Cells that contain user-supplied values, like `X3` having the value 4, would be interpreted as an implicit constraint, viz. `X3` = 4. This is in contrast with current systems which allow the solver to modify cells containing user-supplied values.
3. As in today's spreadsheets, symbolic names (like `Current_Price`) can be given to cells instead of their Cartesian name (e.g. `G13`), and these could be used in expressing constraints.
4. It would be nice to allow cells to contain interval values (like 0..100) to express ranges of possible inputs or outputs.

As a simple example, consider the problem of graphing the price of a (nice, fresh and crusty) baguette under variable inflation rates.

Using today's spreadsheets, one can answer the question, "What is the lowest inflation rate such that the price of a baguette will increase tenfold in the span of a person's lifetime?", by using the following spreadsheet, where the constraints are specified and solved via the integrated solver:



| | A | B |
|---|---|---|
| 1 | **The baguette using today's spreadsheet** | |
| 2 | | |
| 3 | Current Price | 9 |
| 4 | Years | 70 |
| 5 | Inflation Rate | 3% |
| 6 | | |
| 7 | Future Price | 90,00 |

**Paramètres du Solveur**

Cellule cible à définir: $I$5

Egale à: Max [✓ Min] Valeur: 0

Cellules variables: Years;Inflation_Rate

Contraintes:
- Future_Price = 10 * Current_Price
- Years <= 80
- Years = entier
- Years >= 70

With the same example, using a constraint spreadsheet, a user will input the following spreadsheet containing two constraints: a type constraint in `C2` and an inequality constraint in `C3`. The variable to be maximized is `B2`:

| | A | B | C | D |
|---|---|---|---|---|
| 1 | Current_Price | 9 | | |
| 2 | Years | | :70≤B2≤80 | :B2 integer |
| 3 | Inflation_Rate | !min | | |
| 4 | | | | |
| 5 | Future_Price | B1*(1+B3)^B2 | :B5=10*B1 | |

The system is designed to automatically solve the constraint and display the following values:

| | A | B | C | D |
|---|---|---|---|---|
| 1 | Current_Price | 9 | | |
| 2 | Years | 70 | *solved* | *solved* |
| 3 | Inflation_Rate | 3.34% | | |
| 4 | | | | |
| 5 | Future_Price | 90 | *solved* | |

The value 3.34% is assigned to the cell `B3` (better known as `Inflation_Rate`) since it is the smaller number that allows the constraints given in the spreadsheet to be satisfied. Constraints are written in cells like `C2` and `D2` to specify that `Years` should be an integer in between 70 and 80. In its solved form the constraint spreadsheet displays the result of the computation in cells that contain formulæ (like `B5`), and either *solved*, when the constraint is satisfied, or `false`, if no solution has been found, in cells that contain constraints (like `C5`) .

Formally, a *spreadspace* is a finite set of constraints (which are finite or infinite relations) on values of cells. At any given moment, each cell is either "protected" (user-supplied input or derived therefrom) or "variable". Relations defined in constraints are not directional: whether a constraint `X3*3=2*Y3` would cause `X3` to be calculated from a known value of `Y3` or vice-versa would depend on the context. A variable can be determined, not only by fixing related rigid values and calculating functional dependencies (as in backsolving a value for `X3` from the equation `Y3=3*X3` and a fixed value for `Y3`), but also by solving several inequalities (like `Y3=X3*X3`, `X3>0`, and `10<Y3<20`, for *integer* `X3` and `Y3`), depending on the sophistication of the available solver routines.

Thus, processing spreadsheets with constraint cells involves the following steps:

1. Extract the set of constraints from the spreadsheet.
2. Choose which variable cell(s) to solve for.
3. Attempt to solve the constraints using constraint solvers.
4. If the set of solutions is non-empty, determine which solution should be fed back to the appropriate cells.

Today's systems have capabilities to "backsolve" single constraints, optimize by linear programming, and solve some non-linear equations using Newton's method and the like. Only rudimentary solving capabilities for integers are available. As we outline in the next section, more powerful tools are in fact available.

## 3 Constrained Spreadsheets

*What was important were the features we'd left. We'd already discussed wall-sized interactive displays with live graphics but the systems weren't up to it. More important, the grid provided the simplifying structure that made it a spreadsheet as a opposed to a more general surface.*

*—Bob Frankston (letter to D. J. Power, 15 April 1999)*

Starting from the cosmetically improved spreadsheet of the prior section, we aim to add more sophisticated solvers. The driving engine is a cooperating set of numerical and symbolic constraint-solving modules. They transform the extracted set of constraints into a "solved form".

The system should incorporate as much mathematical ability as possible. These could include facilities for:

– interval arithmetic,
– finite domains,
– finite sets,
– propositional calculus,
– algebraic identities (associativity, commutativity, etc.),
– polynomials.

Such capabilities exist in computer algebra systems designed to solve elaborated constraints like: Axiom, Maple, Mathematica®, MuPAD®, Numerica. Finite domain solvers can be solved using ILOG® Solver ([11]) or GNU Prolog ([21]) or specialized solvers written in general-purpose or rule based languages like ELAN ([5, 2]). In some cases, searching for solutions might be necessary. Several works stemming from the declarative programming community extend classical spreadsheets with constraints, including, among others, instance [15, 10, 23, 3, 12].

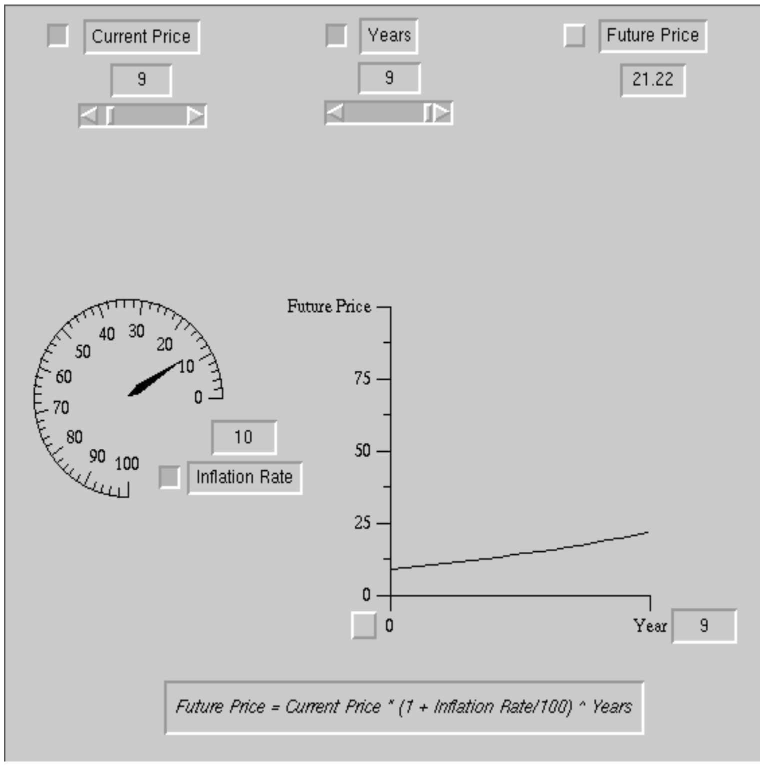
To achieve the kind of capabilities we envision, a blackboard architecture, with component solvers contributing partial solutions to the listed constraints, is indicated.

## 4 Constrained Graphics

Modern spreadsheets provide tools for generating graphical representation of spreadsheet data. The resultant graphs can be sized, placed, and annotated, as desired. For instance, TK!solver [13, 14], Spreadsheet 2000® [24] and iWork® Numbers [4] for the Mac provide nice interfaces that are more graphical and less tabular than standard spreadsheets. The relationship between graphics and constraints has a very long history to which Ugo Montanari has contributed greatly [17]. The relationship with spreadsheets has also been developed by numerous authors, including, for example, [6, 9].

The parameters of the graphics (position, color, spacing, etc.) should be linkable to the spreadsheet itself. Furthermore, constraint solving could be employed to determine their value.

Graphical objects should include dials, meters, switches, etc. The baguette spreadsheet could be portrayed in the following manner:



Future Price = Current Price * (1 + Inflation Rate/100) ^ Years

Importantly, it is not hard to express graphical objects themselves as sets of relatively simple constraints. Thus constraint-solving could be used to calculate the graphical representations. This adds a lot more expressivity.

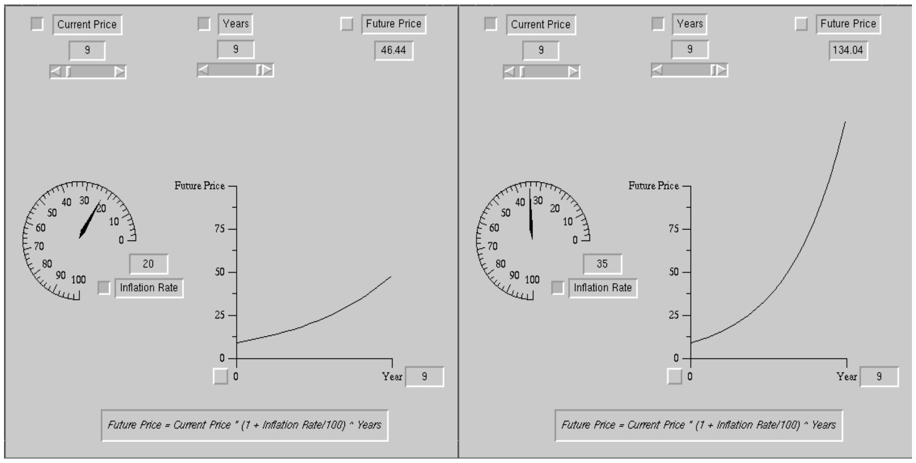
## 5 Active Graphics

Once we have graphics expressed as first-class constraints, the only difference being that results are displayed on a screen, it is possible to allow the user to directly manipulate the graphical objects, causing constraints to be solved and other displays to change accordingly. At this stage, the system would no longer bear any external resemblance to spreadsheets.

Virtually all interaction becomes graphical. Graphical output would not be an add-on that sits atop arithmetic computations, as in today's systems, but would be fully integrated with the calculations and constraint solving. By representing graphical objects and their properties (value, size, color, etc.) in this way, changes the user makes to the graphical objects will immediately result in new values that drive other parts of the spreadspace.

Returning to the baguette example, the same spreadspace as above can be used to gracefully solve all the following queries:

1. *What will the price of a baguette be in 9 years if the current price is 9 pesos, and the inflation rate is 10%?*
   To indicate what the input values are, the user pushes the buttons alongside `Current_Price`, `Years` and `Inflation_Rate`. Then the user sets the current price to 9 and the rate to 10. The answer is graphed and also displayed in the `Future_Price` cell as shown in the figure just above.
2. *What happens if the inflation rate rises to 20% (35%)?*
   The user just uses the mouse to move the dial to the appropriate values.

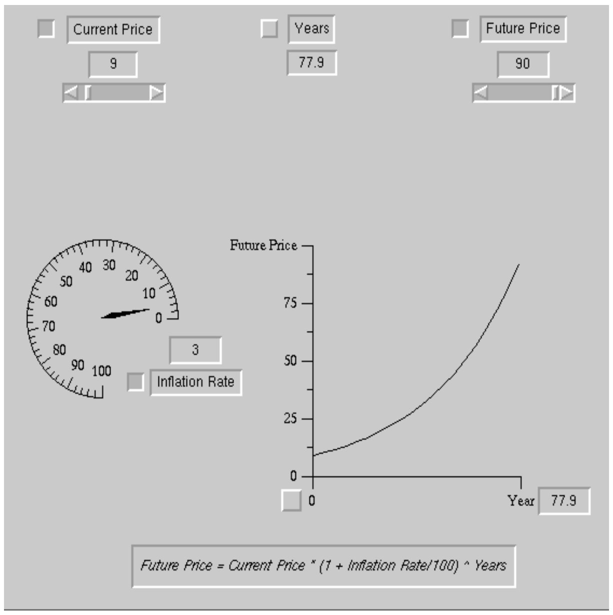


Future Price = Current Price * (1 + Inflation Rate/100) ^ Years

Future Price = Current Price * (1 + Inflation Rate/100) ^ Years

3. *What is the lowest inflation rate such that the price of a baguette will increase tenfold in the span of a person's lifetime?*
   This time, the user pushes `Current_Price`, `Future_Price` and `Inflation_Rate`, and sets the final price to 10 times as much (90). Lastly he/she turns the inflation rate dial until a satisfactory value appears in the `Years` cell: graphics become active.
4. *What inflation rate causes the price to increase tenfold in only 4 years?*
   Starting with the previous state, the user reverses the statuses of `Years` and `Inflation_Rate` by toggling their buttons, and then sets `Years` to `4`. The inflation rate is displayed and the graph is updated.

This spreadspace is simply constructed by choosing the graphical elements from menus, placing and sizing them with the mouse, changing some of the default values to better ones. Entering the formula either textually or in a menu-driven manner relates the various entities mathematically. Dragging cell names or values can streamline the construction of the formula.

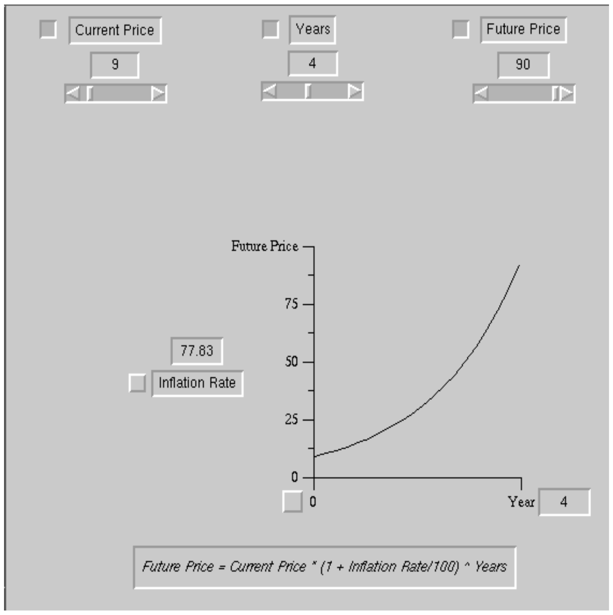


Future Price = Current Price * (1 + Inflation Rate/100) ^ Years

To give a deeper intuition of the way it works, let us write a script for some of the actions needed to create the baguette spreadspace:

1. I pull down the `device` menu and choose a `dial`. I place it where I want on the screen and stretch it to the desired size.
   An instance of the object dial is created and therefore the following constraints are added to the currently empty constraint store. We assume the dial to have its center at coordinates $(a, b)$ and to be of radius $r$ (all specified indeed graphically by the action of the user who drags the dial on the working space):
   $\texttt{dial.value} = 0$ the default value presented by the dial
   $\texttt{dial.center} = (a, b)$
   $\texttt{dial.radius} = r$
   $\texttt{dial.shape} = ((x-a)^2 + (y-b)^2 = r^2)$
   $\texttt{dial.min} = 0$
   $\texttt{dial.max} = \texttt{dial.min} + 100$
   $\texttt{dial.marking}[\texttt{dial.min} : \texttt{dial.max}].\texttt{color} = \texttt{black}$
   $\texttt{dial.foreground} = \texttt{black}$

   Finally, since by default the value of this object can be either set by the user interactively (in which case `dial.in` is `true`, value by default) or set by the constraint solver (in which case `dial.out` is `true`), the following constraints are added:
   $\texttt{dial.readWrite} = (\texttt{dial.in} \neq \texttt{dial.out})$
   $\texttt{dial.in} = \texttt{true}$



Future Price = Current Price * (1 + Inflation Rate/100) ^ Years

2. I change the *high value* from the default 100 to 200, and all the intermediate values change to match.
   The constraint store now contains:
   $\texttt{dial.value} = 0$
   $\texttt{dial.center} = (a, b)$
   $\texttt{dial.radius} = r$
   $\texttt{dial.shape} = ((x - a)^2 + (y - b)^2 = r^2)$
   $\texttt{dial.min} = 0$
   $\texttt{dial.max} = 200$
   $\texttt{dial.marking}[\texttt{dial.min} : \texttt{dial.max}].\texttt{color} = \texttt{black}$
   $\texttt{dial.foreground} = \texttt{black}$
3. I change the color of the markings in the range [100 : 200] to red, by catching all of them, holding the mouse button down to get the list of attributes, and then choosing the foreground color item, which gives a palette from which I choose a dark blue. I leave the default low value of 0 and default interval markings.
   So now we have the following:
   $\texttt{dial.value} = 0$
   $\texttt{dial.center} = (a, b)$
   $\texttt{dial.radius} = r$
   $\texttt{dial.shape} = ((x - a)^2 + (y - b)^2 = r^2)$
   $\texttt{dial.min} = 0$
   $\texttt{dial.max} = 200$
   $\texttt{dial.marking}[\texttt{dial.min} : \texttt{100}[.\texttt{color} = \texttt{black}$
   $\texttt{dial.marking}[\texttt{100} : \texttt{dial.max}].\texttt{color} = \texttt{red}$

`dial.foreground` = `darkBlue`

4. Then I choose a simple rectangular display from the menu, placing it near the dial.
   This has the effect to add to the previous constraint store the following:
   `rectDisplay1.position` = $(c, d)$ determined by the user action
   `rectDisplay1.height` = 5 default value
   `rectDisplay1.length` = `rectDisplay1.height` ∗ `5` default value
   `rectDisplay1.type` = `real`
   `rectDisplay1.value` = 0
   `rectDisplay1.min` = 0
   `rectDisplay1.max` = `rectDisplay1.min` + 100
   `rectDisplay1.foreground` = `black`
   *etc.*

As one can see, an explicit set of constraints is built using a graphical interactive interface. It represents exactly all the behavioral knowledge the user wants to put into his or her model.

## 6 Examples

Here are a few simple examples highlighting some of the original features of spreadspaces.

### 6.1 Color-Changing Rectangle

*Context:* The user is resizing a rectangle by dragging one of the corners or sides of the rectangle with her mouse.

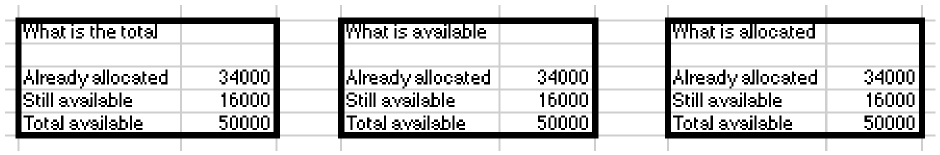
*Constraint:* The designer of the current spreadspace has written the following constraints, where $P$ identifies the perimeter of the rectangle.

$$\begin{aligned} P > 3 \ &\Rightarrow\ \texttt{rectangle.backGroundColor} = \texttt{red} \\ P \leq 3 \ &\Rightarrow\ \texttt{rectangle.backGroundColor} = \texttt{blue} \end{aligned}$$

*Behavior:* When the user is in-playing the size of the rectangle with her mouse, the color of the rectangle is displayed in red when the perimeter of the rectangle is larger than 3 in the current length unit. Otherwise, it is shown in blue.

### 6.2 Standard Spreadsheet

*Context:* The user uses a spreadsheet to understand the relationship between the total amount of money "available", the amount "allocated" (earmarked) and the amount still available. With a standard spreadsheet, depending on the quantity one wants to compute, one has to make three different computations expressed in three different spreadsheets as illustrated in Fig. 1.



| | |
|---|---|
| What is the total | |
| | |
| Already allocated | 34000 |
| Still available | 16000 |
| Total available | 50000 |

| | |
|---|---|
| What is available | |
| | |
| Already allocated | 34000 |
| Still available | 16000 |
| Total available | 50000 |

| | |
|---|---|
| What is allocated | |
| | |
| Already allocated | 34000 |
| Still available | 16000 |
| Total available | 50000 |

**Fig. 1.** Spreadsheet examples

*Constraint:* The designer of that spreadspace simply writes the following constraint:

$$\texttt{Already\_allocated} + \texttt{Still\_available} = \texttt{Total\_available}$$

*Behavior:* As soon as the value of 2 of the above variables are known, the third is fulfilled automatically.

### 6.3 Red First

*Context:* The user sees three circles and can in-play the color of them using a menu poping up when (s)he clicks right on one of the circles. At the beginning the circles have the same background color than the overall background (which is assumed not to be red!).

*Constraint:* The designer of that spreadspace has written the following constraint:

$$\begin{aligned}&(\texttt{C1.backGroundColor} = \texttt{red})\\ &\qquad \oplus (\texttt{C2.backGroundColor} = \texttt{red})\\ &\qquad\quad \oplus (\texttt{C3.backGroundColor} = \texttt{red})\\ &\qquad\qquad = 1\end{aligned}$$

where $\oplus$ denotes exclusive or.

*Behavior:* The only valid in-play of the user will be to enter the first one to be red, and the other not to be red. This will therefore force the user to behaves accordingly.

## 7 $\mathbf{S_p^2}$ System Organization

The diagram in Fig. 2 exemplifies how a user of the $\mathbf{S}_\mathrm{p}^2$ spreadspace system we are describing fills in values for the fields of the cell just positioned on the spreadspace.

1. The user clicks on the `Name` field, types "Already allocated", and hits the return key.
2. The Interaction Manager
   (a) identifies `C1.Name` as the field modified, and

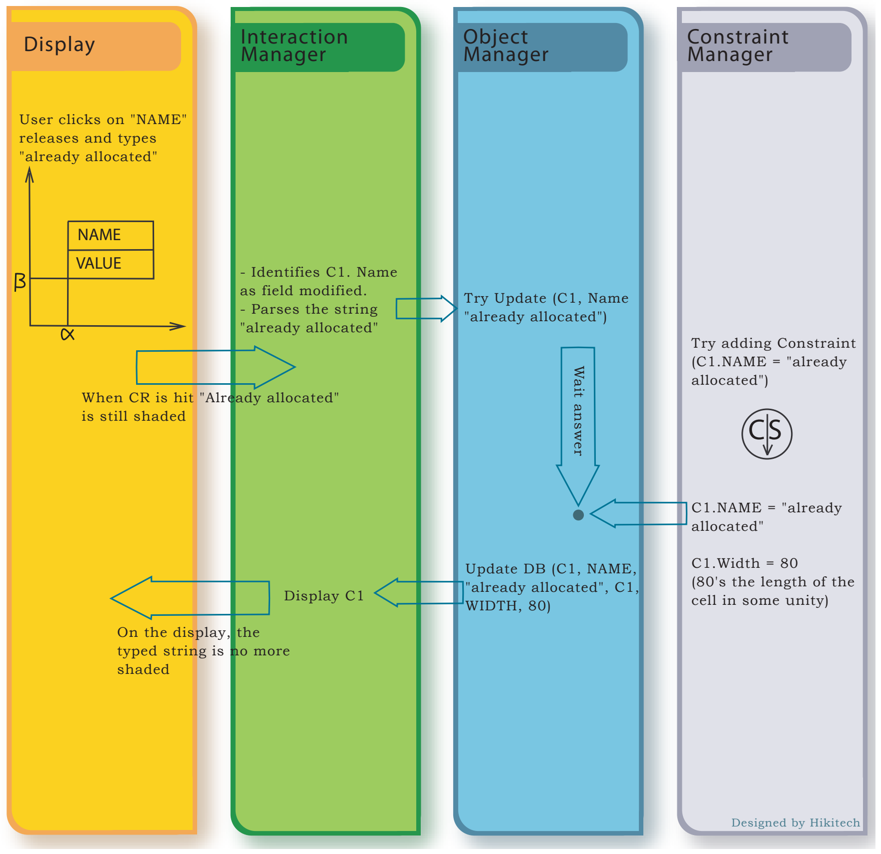

Display
Interaction Manager
Object Manager
Constraint Manager
User clicks on "NAME" releases and types "already allocated"
NAME
VALUE
β
α
- Identifies C1. Name as field modified.
- Parses the string "already allocated"
Try Update (C1, Name "already allocated")
When CR is hit "Already allocated" is still shaded
Wait answer
Try adding Constraint (C1.NAME = "already allocated")
C|S
C1.NAME = "already allocated"
C1.Width = 80 (80's the length of the cell in some unity)
Update DB (C1, NAME, "already allocated", C1, WIDTH, 80)
Display C1
On the display, the typed string is no more shaded
Designed by Hikitech

**Fig. 2.** Information flow from user to constraints

(b) parses the string "Already allocated".

3. The Object Manager tries now to update the cell and – for this purpose – initiates the solver and waits for an answer.
4. The Constraint Manager tries to add the new constraint `C1.Name` = "Already allocated".
   The solver detects no contradiction, but has computed (by constraint propagation, based on the name and default font) that the width of the cell should be 80, and thus `C1.Width = 80` is added as a constraint, and that value is also passed to the Object Manager.
5. The Object Manager updates the `Name` and `Width` fields of cell `C1` in the database.
6. The Interaction Manager updates the display with the new values. No values are shaded at this point on the display.

In this context, one can see the importance of the constraint satisfier and/or solver. Of course, all the work on constraint solving, combination, propagation and clever handling of constraint stores shall be reused and possibly adapted to handle huge numbers of heterogeneous constraints.

## 8 Conclusion

It is clear that people use computers to do many computations that can be expressed as mathematical problem solving. However, many of these tasks are difficult or inconvenient with current software.

- Spreadsheets may provide constraint-solving, but only as an afterthought; though relatively powerful, the interface is not intuitive, and the computational meaning of constraints is obscured. Most aspects of the layout and graphics are not integrated into the spreadsheet, even in graphical spreadsheets, and the graphics certainly have no connection with the solver.
- Commercial constraint solvers are designed for programmers, and cannot be used by spreadsheet users and their kin. Symbolic systems may have powerful graphing capabilities, but solving requires mathematical and programming sophistication, since solving usually necessitates heavy user-interaction. Existing solvers cannot cooperate, and future improvements cannot be added modularly.
- Graphical interface design systems allow one to construct the kind of objects in our baguette example, but all calculations must be hard-wired.

The beauty and value of spreadspaces lie in their seamless integration of spreadsheet computations, constraint solving, and optimization, in an active and appealing graphical environment. As such, it contributes to the large research interest in spreadsheets, both with regard to their deductive extensions [1, 22] and from the risk point of view [19].

**Acknowledgements**

Many thanks to Carlos Castro who, a long time ago, coded the baguette spreadspace into the ILOG view and solver environments and to Franz Kirchner for his graphical design of the information flow presented in Fig. 2. Thanks also to the ILOG company for letting us freely use their software.

## References

1. Margaret Burnett, Sherry Yang, and Jay Summet. A scalable method for deductive generalization in the spreadsheet paradigm. *Interactions*, 9(5):9–11, 2002.
2. Carlos Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998.
3. Siddharth Chitnis, Madhu Yennamani, and Gopal Gupta. Exsched: Solving constraint satisfaction problems with the spreadsheet paradigm. *The Computing Research Repository (CoRR)*, `abs/cs/0701109`, 2007.
4. Apple Corp. Numbers, 2008. (`http://www.apple.com/iwork/numbers`).

5. ELAN. (`http://elan.loria.fr`).
6. Gopal Gupta and Shameem F. Akhter. Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs. In *Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages (PADL 2000)*, volume 1753, pages 308–323, 2000.
7. Bruce Horn. Constraint patterns as a basis for object-oriented programming. In *Proceedings of the SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 218–233. ACM, 1992.
8. Walter Hower and Winfried Graf. A bibliographical survey of constraint-based approaches to CAD, graphics, layout, visualization, and related topics. *Knowl.-Based Syst.*, 9(7):449–464, 1996.
9. Ed Huai hsin Chi, John Riedl, Phillip Barry, and Joseph Konstan. Principles for information visualization spreadsheets. *IEEE Comput. Graph. Appl.*, 18(4):30–38, 1998.
10. Eero Hyvönen and Stefano De Pascale. A new basis for spreadsheet computing: Interval solver for Microsoft Excel. In *Proceedings of the Sixteenth National Conference on Artificial intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference (AAAI '99/IAAI '99)*, pages 799–806, Menlo Park, CA, 1999. American Association for Artificial Intelligence.
11. ILOG. (`http://www.ilog.fr`).
12. Bharat Jayaraman and Pallavi Tambay. Modeling engineering structures with constrained objects. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *Lecture Notes in Computer Science*, pages 28–46. Springer, 2002.
13. M. Konopasek and S. Jayaraman. *The TK! Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education*. Osborne/McGraw-Hill, 1984.
14. M. Konopasek and S. Jayaraman. Constraint and declarative languages for engineering applications: The TK!Solver contribution. *Proceedings of the IEEE*, 73(12):1791–1806, December 1985.
15. T. Kunstmann, M. Frisch, and R. Muller. A declarative programming environment based on constraints. In *Proceedings of the 11th International IEEE Symposium on Visual Languages (VL '95)*, pages 120–121, Washington, DC, September 1995. IEEE Computer Society.
16. Ghassan Kwaiter, Véronique Gaildrat, and René Caubet. Modelling with constraints: A bibliographical survey. In *Proceedings of the Second International Conference on Information Visualisation (IV'98)*, pages 211–220. IEEE, 1998.
17. Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.
18. Ugo Montanari and Francesca Rossi. Constraint solving and programming: What's next? *ACM Comput. Surv.*, page 70, 1996.
19. Raymond R. Panko and Richard P. Halverson Jr. Spreadsheets on trial: A survey of research on spreadsheet risks. In *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS), Volume 2: Decision Support and Knowledge-Based Systems*, pages 326–335. IEEE Computer Society, January 1996.
20. D. J. Power. A brief history of spreadsheets, August 2004. (`http://dssresources.com/history/sshistory.html`).
21. GNU Prolog. (`http://www.gprolog.org`).

22. C. R. Ramakrishnan, I. V. Ramakrishnan, and David Scott Warren. Deductive spreadsheets using tabled logic programming. In Sandro Etalle and Miroslaw Truszczynski, editors, *Proceedings of the 22nd International Conference on Logic Programming (ICLP)*, volume 4079 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 2006.
23. Marc Stadelmann. A spreadsheet based on constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 217–224, 1993.
24. Steve Wilson. Visual programming: Building a visual programming language. *MacTech*, 13(4), 2000.