Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 282 (2012) 35–45

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# A Tool for the Integration of Constraint Solving in Spreadsheets

Ana María Fernández-Soriano[1] Julio Mariño[1] Ángel Herranz[1]

{amfernandez,jmarino,aherranz}@fi.upm.es
Babel Research Group
Universidad Politécnica de Madrid
Madrid, Spain

**Abstract**

Spreadsheets have become widely used tools, but they are applied to increasingly complex problems, far beyond the kind of tasks for which they were originally conceived. This often results in large, hard to maintain sheets, with little guarantee about their correctness. Potential errors are due in part to unskilled users and also to the spreadsheet systems' own limitations. This contribution presents a tool that aims at improving the usual working flow when filling in a spreadsheet. The tool integrates a constraint solver based on transformations of the cell content into equality and inequality systems over rationals. The transformed systems are then solved using constraint logic programs and the obtained solutions are presented to the user in an understandable way. One of the practical benefits of this solution based on constraint logic programming is backwards execution: our tool is able to find out the required input values to reach the desired outputs depending on aggregation formulae written in the cells. Also, the constraint model offers a simple and sound solution to the problem of circular references in cell formulas.

*Keywords:* Spreadsheet, Constraint Logic Programming, Constraint Solving, Linear Constraints, Programming Environments, End-User Development

## 1 Introduction

The term *End-User Development* (EUD) [5] has been coined as an umbrella to refer to those environments which allow nonprofessional developers to create or modify software without significant knowledge of a programming language. Mash-ups, *Programming by Example* (PbE), or graphical scripting languages

[1] Work partially supported by MICINN grant TIN2009-14599-C03-03 (DESAFIOS10) and grant S2009TIC-1465 (PROMETIDOS-CM) from the regional government of Madrid.

doi:10.1016/j.entcs.2011.12.004

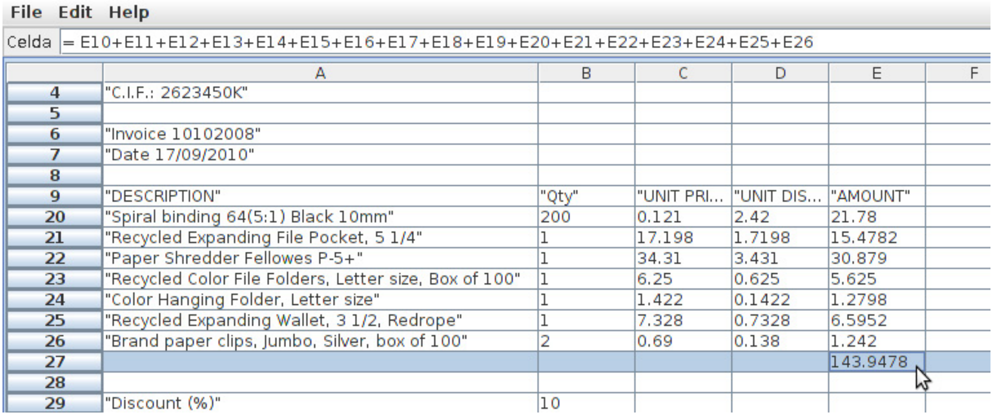
are among the instances that are receiving more attention today but the oldest example and, definitely, the most successful of these tools is the spreadsheet. Some of the reasons for its success are paradigmatic: 1.- *Availability*: they are present in the office suites on most computers. 2.- *Immediate feedback*: changes made in one part of the spreadsheet are reflected instantly in the results without any restructuring, recompilation or testing that a *conventional* software system would require, and 3.- *Cheap reuse*: fragments from a spreadsheet can be easily combined with other ones, and users can apply the skills learnt on one application to another one, very often just extending some existing sheet.

Of course, the EUD model is not free from criticism. The results of repeated composition, cut & paste and analogy can be unpredictable, specially given that there is seldom a previous specification of the problem that has to be solved and tests are nonexistent. In the case of spreadsheets, this results in huge, hard to maintain and understand sheets but that are often responsible for key functions for the business logic of certain parts of an organisation. In fact, studies from consulting firms such as PWC and KPMG reveal that 95% of spreadsheets in certain areas might contain some kind of error [6].

Typical sources of errors include arithmetical operations between incompatible data (e.g. adding yards to pounds), operating with undefined cells (erroneously taken as zero) or circular references[2] in cell sets (that are treated in various ways by different spreadsheet systems). While some of these problems could be attacked using conventional programming technology – e.g. type systems – that possibility has usually been discarded in favour of features such as the aforementioned immediate feedback.

The tool we are introducing extends the traditional spreadsheet model, in which cell contents are either constant or the result of operating with the contents of other cells, with the possibility of stating relations between cell contents. This can alleviate some of the problems above. For example, if we want to add a series of quantities and there is one which we do not yet know, it is very likely that we end up forgetting about it and eventually taken as zero (even if it stands for the price of a Ferrari). With our proposal, the cell can be assigned a range (e.g. $150000 \leq A5 \leq 240000$) that will be reflected in the summation with a corresponding range. Mutual relations between cells, that often lead to unexpected results in many systems, can also be used in combination with partial information. Imagine that now we want to add the prices of several items, but some come in dollars and others in euros. We could have each item in a different row and reserve one column for the price in dollars and the next one for the price in euros. That means that, initially,

[2] Expression in a cell that refers back to itself, either directly or indirectly.



Celda = E10+E11+E12+E13+E14+E15+E16+E17+E18+E19+E20+E21+E22+E23+E24+E25+E26

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 4 | "C.I.F.: 2623450K" | | | | | |
| 5 | | | | | | |
| 6 | "Invoice 10102008" | | | | | |
| 7 | "Date 17/09/2010" | | | | | |
| 8 | | | | | | |
| 9 | "DESCRIPTION" | "Qty" | "UNIT PRI... | "UNIT DIS... | "AMOUNT" | |
| 20 | "Spiral binding 64(5:1) Black 10mm" | 200 | 0.121 | 2.42 | 21.78 | |
| 21 | "Recycled Expanding File Pocket, 5 1/4" | 1 | 17.198 | 1.7198 | 15.4782 | |
| 22 | "Paper Shredder Fellowes P-5+" | 1 | 34.31 | 3.431 | 30.879 | |
| 23 | "Recycled Color File Folders, Letter size, Box of 100" | 1 | 6.25 | 0.625 | 5.625 | |
| 24 | "Color Hanging Folder, Letter size" | 1 | 1.422 | 0.1422 | 1.2798 | |
| 25 | "Recycled Expanding Wallet, 3 1/2, Redrope" | 1 | 7.328 | 0.7328 | 6.5952 | |
| 26 | "Brand paper clips, Jumbo, Silver, box of 100" | 2 | 0.69 | 0.138 | 1.242 | |
| 27 | | | | | 143.9478 | |
| 28 | | | | | | |
| 29 | "Discount (%)" | 10 | | | | |

Fig. 1. A typical spreadsheet for an invoice, including amount, purchase price and discount rate for several items.

the only information in these cells reflect the currency exchange rate. If we receive a price in euros then we can update the cell in the right column and the quantity in the left one will be recalculated, reducing the possibility of mixing currencies inadvertently.

A more elaborate example is shown in Figure 1. The discount rate applied is 10% for all the products. The total price for the invoice is 143,94€. Imagine that we have a constraint of total price 140,00€, for example, due to marketing reasons. The problem is to recalculate the discount rate to apply to each product. Even more, imagine that we cannot apply more than 10% to one of the products. How to recalculate then the discount rate for the rest of the items? In a traditional spreadsheet system we can only do iteratively changes in different values of the invoice in order to approximate the solution but we cannot "ask" the spreadsheet for a solution that fits the constraint. The application of our tool to this example will be shown in Section 5.

Our system works by translating the cell contents into a logic program written in a Prolog dialect capable of reasoning about linear inequalities of rationals. A formal description of the cell language and its translation is given in Section 3. An advantage of the constraint logic programming (CLP) approach is that it allows for a uniform treatment of several application domains, so the solution presented here for rationals could be adapted to solve combinatorial problems instead. The framework that allows for this genericity is discussed in Section 2. An overview of the tool's architecture is provided in Section 4.

There have been previous proposals for the extension of spreadsheets with deductive capabilities. Due to lack of space, the reader is referred to [4] for a quick survey of the field. While building our own system implied reimplementing some of the functionality in earlier tools, we consider the effort worthwhile as it has provided us with an extensible platform for applying the ideas in this paper to more complex domains.

## 2 Preliminaries: Constraints and Logic Programming

The reader is referred to [3] for a formal, detailed presentation of the parametric Constraint Logic Programming framework (CLP(X)). In what follows we present an informal account of some features of the framework relevant to our work. CLP(X) stands for an extension of logic programming in which atomic formulae can be either the application of *user-defined* predicates to tuples of terms or a new category of formulae (the constraints, $\mathcal{L}$) written in a separate first-order signature $\Sigma$ and provided with a first-order structure $\mathcal{D}$ as standard interpretation. $\mathcal{L}$ is closed under conjunction and existential quantification. The pair $(\mathcal{D}, \mathcal{L})$ is called a *constraint domain* and is used to enrich a *host* logic programming language with some domain-specific intelligence that would be inefficient to implement using the standard deduction mechanisms of the host. CLP($\mathcal{D}$) denotes a particular instance of the generic scheme for the constraint domain given. Traditional Prolog can be seen as an instance CLP(Herbrand) where the constraints are just syntactic equalities of terms. The instance used in our prototype is called CLP(Q), where the language of constraints allows linear inequalities interpreted over the rational field. Another successful instance of the framework is CLP(FD), where the constraints include arithmetic inequalities interpreted over a finite subset of the integers. Constraint domains are expected to provide the following operations and checks on constraints in order to be used in a CLP language effectively: i) testing for consistency, $\mathcal{D} \models \exists\, c$; ii) checking the implication of one constraint by another, $\mathcal{D} \models c \rightarrow c'$; iii) projecting a constraint $c$ onto a set of variables $\hat{x}$ in order to obtain a simpler, equivalent $c'$: $\mathcal{D} \models c' \leftrightarrow \exists\, \hat{x}.c$; and iv) detecting that a constraint $c$ *forces* a single value for one of its free variables: $\mathcal{D} \models \exists\, z. \forall\, x, \hat{y}.c(x, \hat{y}) \rightarrow x = z$. Availability of these operations is used in our spreadsheet model to ensure the consistency of the set of cells and also to compute the visualisation of the cells that is shown back to the user.

## 3 Formal Specification of the Translation Function

The main characteristics of the translation process are that cell references are represented as logic variables in the CLP program and that cell content is represented as constraints in the CLP program. Figure 2 illustrates the essence of the translation: we show a pair of spreadsheets, their translation into CLP programs and the result of their execution.

The first example, **ssA**, is a *standard* spreadsheet, each cell *generates* an

<table>
<tr><th>Spreadsheet</th><th>CLP program and solution</th></tr>
<tr><td>
<table>
<tr><th>ssA</th><th>A</th><th>B</th><th>C</th></tr>
<tr><td>1</td><td>14</td><td>25</td><td>$A1 + B1$</td></tr>
</table>
</td><td>
<pre>
?- { A1 = 14, B1 = 25, C1 = A1
   + B1 }.
A1 = 14,
B1 = 25,
C1 = 39.

?-
</pre>
</td></tr>
<tr><td>
<table>
<tr><th>ssB</th><th>D</th><th>E</th></tr>
<tr><td>3</td><td>100 | $D3 = 1.45 * E3$</td><td>$E3$</td></tr>
</table>
</td><td>
<pre>
?- { D3 = 100, D3 = 1.45 * E3 }.
D3 = 100.0,
E3 = 68.9655

?-
</pre>
</td></tr>
</table>

Fig. 2. Translation of spreadsheets into CLP programs

equality constraint and the CLP solver finds the solution. In the second example, **ssB**, cell $D3$ contains an expression (100) but also establishes a constraint on cell $E3$: $D3 = 1.45 * E3$ (perfectly could be a change rate euro-dollar). The value for the cell $E3$ is, directly, $E3$, a way to establish no other constraint. In this case, the cells generate two constraints: `D3 = 100` and `D3 = 1.45 * E3` and the solver finds the value for $E3$.

We formally describe the translation process as a mathematical function. Previous to its formal definition we need a formal model for spreadsheets with constraints, the domain of the translation function, and a formal model for CLP programs, its codomain.

### 3.1 Abstract Syntax of Spreadsheets with Constraints

Conceptually, a spreadsheet is a partial function from cell references, pairs of natural numbers representing row and column, into cell content, valid expressions involving numbers and operations on cells. Our example **ssA** in Figure 2 could be represented by a mathematical object similar to this one:

$$\mathbf{ssA} = \{(1,1) \mapsto 14, (1,2) \mapsto 25, (1,3) \mapsto (1,1) + (1,2)\}$$

where $(1,1) + (1,2)$ is an informal representation of the cell content $A1 + B1$.

Our formal representation of spreadsheets is a partial function from *Address*, the Cartesian product of natural numbers, into cell content:

$$\begin{aligned} \mathit{Spreadsheet} &= \mathit{Address} \nrightarrow \mathit{Cell} \\ \mathit{Address} &= \mathbb{N} \times \mathbb{N} \end{aligned}$$

Cell contents have two components: an expression and a constraint. Example **ssB** follows this grammar:

$$\mathit{Cell} ::= \mathit{Expr}\,[\ |\ \mathit{Constr}]$$

where square brackets represents optionality of the constraint. Without any

$$\begin{aligned}
\mathit{Cell} &= \mathit{Expr} \times \mathit{Constr} \\
\mathit{Expr} &::= \mathit{Expr}_{\mathbb{Q}} \\
\mathit{Constr} &::= \top \mid \mathit{Constr} \wedge \mathit{Constr} \mid \mathit{Constr}_{\mathbb{Q}}
\end{aligned}$$

$$\begin{aligned}
\mathit{Expr}_{\mathbb{Q}} &::= \mathit{Address} \mid \mathbb{Q} \mid \mathit{Unaexpr}_{\mathbb{Q}} \mid \mathit{Binexpr}_{\mathbb{Q}} \\
\mathit{Unaexpr}_{\mathbb{Q}} &::= \mathit{Unaop}_{\mathbb{Q}}\ \mathit{Expr}_{\mathbb{Q}} \\
\mathit{Unaop}_{\mathbb{Q}} &::= + \mid - \\
\mathit{Binexpr}_{\mathbb{Q}} &::= \mathit{Expr}_{\mathbb{Q}}\ \mathit{Binop}_{\mathbb{Q}}\ \mathit{Expr}_{\mathbb{Q}} \\
\mathit{Binop}_{\mathbb{Q}} &::= + \mid - \mid * \mid /
\end{aligned}$$

$$\begin{aligned}
\mathit{Constr}_{\mathbb{Q}} &::= \mathit{Expr}_{\mathbb{Q}}\ \mathit{Relop}_{\mathbb{Q}}\ \mathit{Expr}_{\mathbb{Q}} \\
\mathit{Relop}_{\mathbb{Q}} &::= = \mid < \mid > \mid \leq \mid \geq
\end{aligned}$$

Fig. 3. Abstract syntax of spreadsheets with constraints

loss of generality, constraint omission is equivalent to a trivial constraint ($\top$), so we assume that the constraint always exists. Figure 3 shows the complete abstract syntax[3] for cell content.

Since we are going to focus on linear constraints over rationals, the formal model for constraints in Figure 3 has been defined as

$$\begin{aligned}
\mathit{Expr} &::= \mathit{Expr}_{\mathbb{Q}} \\
\mathit{Constr} &::= \ldots \mid \mathit{Constr}_{\mathbb{Q}}
\end{aligned}$$

where $\mathit{Expr}_{\mathbb{Q}}$ are arithmetic expressions on rational numbers and *Constr* are built by mean of atomic constraints, application of predefined constraint predicates to CLP(Q) terms ($\mathit{Expr}_{\mathbb{Q}}$), or conjunction of another constraints.

Nevertheless, our formalisation enables one to easily extend the set of domains adding, for example, constraints on finite domains:

$$\begin{aligned}
\mathit{Expr} &::= \mathit{Expr}_{\mathbb{Q}} \mid \mathit{Expr}_{\mathbb{Z}} \\
\mathit{Constr} &::= \ldots \mid \mathit{Constr}_{\mathbb{Q}} \mid \mathit{FDConstr}_{\mathbb{Z}}
\end{aligned}$$

The spreadsheet **ssB** presented at the begin of the section would be represented by the following mathematical object:

$$\mathbf{ssB} = \{(3,4) \mapsto (100, (3,4) = 1.45 * (3,5)), (3,5) \mapsto ((3,5), \top)\}$$

### 3.2 Translation Function

We explain in this section the syntax-directed translation of a spreadsheet into a CLP program. The abstract syntax of CLP Programs that our tool

[3] For readability reasons, we present the components in the form of an ambiguous grammar.

$$
\begin{aligned}
\mathit{Prog} \;&=\; \mathbb{P}\,\mathit{Clause}\\
\mathit{Clause} &::= \mathit{Atom} :- \mathit{Goal}.\\
\mathit{Goal} &::= \mathit{Lit}, \ldots, \mathit{Lit}\\
\mathit{Lit} &::= \mathit{Atom} \mid \neg \mathit{Atom}\\
\mathit{Atom} &::= \mathit{PS}(\mathit{Term}, \ldots, \mathit{Term})\\
\mathit{PS} &::= \texttt{globalcon} \mid \texttt{=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{...}\\
\mathit{Term} &::= \mathit{Var} \mid \mathit{FS}(\mathit{Term}, \ldots, \mathit{Term})\\
\mathit{Var} &::= \texttt{A1} \mid \texttt{A2} \mid \texttt{B1} \mid \texttt{...}\\
\mathit{FS} &::= \mathbb{Q} \mid \texttt{+/1} \mid \texttt{-/1} \mid \texttt{+/2} \mid \texttt{-/2} \mid \texttt{*/2} \mid \texttt{//2}
\end{aligned}
$$

Fig. 4. Abstract syntax of CLP programs

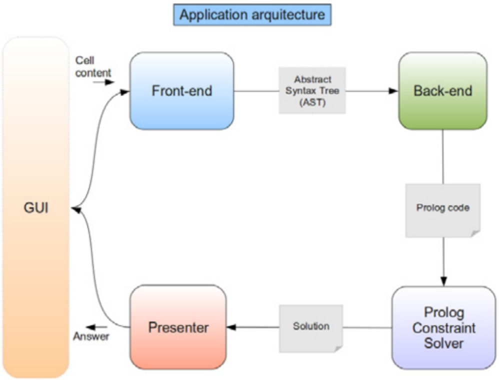


Fig. 5. Tool architecture

generates is defined in Figure 4. The CLP program generated has only one clause and establishes a global constraint on all the cells simultaneously. We generate a predicate called globalcon, that associates non null cells with a unique variable corresponding with the solution for the cell, taking into account the information contained in all the cells globally. The full definition of the translation function can be found in [1]. Let us just show the resulting CLP programs for the previous examples:

| **Example** | **Automatically Generated CLP program** |
|---|---|
| **ssA** | `globalcon([(1,1,A1),(1,2,B1),(1,3,C1)]) :-`<br>`    { A1 = 14, B1 = 25, C1 = A1 + B1 }.` |
| **ssB** | `globalcon([(3,4,D3),(3,5,E3)]) :-`<br>`    { D3 = 100, D3 = 1.45 * E3 }.` |

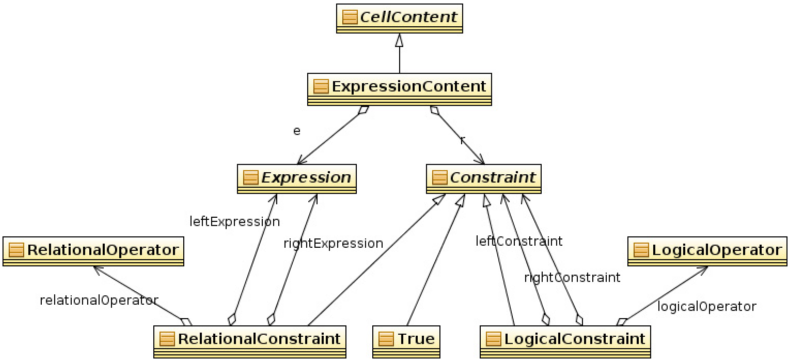


Fig. 6. AST UML class diagram.

## 4 Tool Architecture

We can see in Figure 5 that our application has five main components. Let us briefly describe the functionality of each one. The Graphic User Interface (GUI), a spreadsheet appearance, with cells arranged in a matrix of rows and columns. To fill in a cell the user clicks on it and just writes down its content following a quite standard concrete syntax for cell content of spreadsheets augmented with constraints. Once the user fills in a cell the "Front-end" component processes each cell content in the spreadsheet: it checks the syntax is correct and constructs an abstract syntax tree that implements the mathematical objects defined in Section 3.1 (see Figure 6 for the UML model of the abstract syntax trees). The "Back-end" component generates a CLP program that is the implementation of the translation function in Section 3.2. It follows the layout of a Visitor pattern [2]. The "Prolog Constraint Solver" component starts a Prolog engine with the CLP extension, loads the generated program and returns a solution following the same concrete syntax of cell content. Finally, the solution is parsed by the "Presenter" component that shows it in the GUI component to the user.

The application is written in Java, the GUI uses Java Swing components, the Front-end has been implemented by using JavaCC and the Prolog engine used is Ciao Prolog.[4] The prototype can be freely downloaded (`http://babel.ls.fi.upm.es/software/anatomicsolver`) and tested.[5]

## 5 Using the Tool

We illustrate the interaction with our tool by solving two of the examples stated in Section 1. In the invoice example (Fig. 1), the user would follow

[4] Available from `http://www.ciaohome.org`.

[5] Additional examples can be found under directory `scenarios/`.

these steps:

(i) Instead of iteratively changing the discount rate until 140.00€ is achieved, the user types a constraint into $E27$:

| | E |
|---|---|
| 27 | $E10 + \ldots + E26 \mid E27 == 140$ |

(ii) Then, the user removes the discount rate of 10% from cell $B29$ or types the following into it:

| | B |
|---|---|
| 29 | $B29 \mid \mathit{TRUE}$ |

(iii) The solver works out a solution to the constraint system

| | B |
|---|---|
| 29 | 12.47 |

$$\begin{aligned}(\mathit{UnitDiscount})D20 &= B20 * C20 * B29/100,\\ (\mathit{Amount})E20 &= B20 * C20 - D20,\\ &\vdots\\ E27 &= E10 + \ldots + E26,\\ E27 &= 140,\\ B29 &= B29\end{aligned}$$

The “dollars vs. euros” example was about mutual dependency of cells in two adjacent columns, say D and E:

(i) The user doesn’t know if prices for items are expressed in dollars (D) or euros (E). It is enough to say how each currency depends on the value of the other one (circular references are allowed provided no inconsistencies are introduced):

| | Dollars | Euros |
|---|---|---|
| 3 | $= 1.4501 * E3$ | $= D3/1.4501$ |

(ii) Now it is possible to change the value in one column and get the appropriate currency in the other just by adding the constraint that establishes it:

| | Dollars | Euros |
|---|---|---|
| 3 | $= 1.4501 * E3$ | $= D3/1.4501 \mid E3 = 2$ |

(iii) obtaining the following answer:

| | Dollars | Euros |
|---|---|---|
| 3 | 2.9002 | 2 |

## 6 Conclusions and Future Work

In this paper we have presented a tool that explores the integration of constraint solvers in spreadsheet applications. A minor extension to the standard language of spreadsheets and the coordination with an existing CLP environment allows our tool to effectively compute in the presence of incomplete information. We have applied this capability to overcome three limitations in existing spreadsheet systems. On one hand, we propose a simple and sound solution to the problem of circular references in cell formulas as shown in Section 5. Also, constraints can be used to alleviate some of the inconveniences caused by null values. Finally, our approach allows to solve target optimization problems in spreadsheets without the usual resort to trial-and-error as the example in Fig. 1 shows.

We can mention some extensions planned for our prototype:

- Other constraint domains and their integration. On one hand, this improvement requires to extend the abstract syntax and concrete grammars. For the introduction of *finite domains* (CLP(FD)), for instance, we need to extend the syntax to express intervals and membership. Such an extension would be useful, for instance, to solve schedule problems, although user interaction would be slightly more complex given that some mechanism to iterate through different solutions would be necessary. Also, a type system would be mandatory in the case of mixing different domains under the same expression syntax, at least for ensuring that constraints are not used in a potentially inconsistent way.
- Improving usability. For the sake of simplicity, our prototype allows to write a constraint that affects any cell in the content of any other cell, and we have not answered to some important questions on usability: should we allow a set of constraints for every cell? should we allow just a global set of constraints? should the system answer with any constraint that affect a cell in the resulting content of that cell? should the application keep a separate content for the user input and another for the answer of the solver?

  A proper answer to these questions is crucial for the end-user adoption of our proposal.

## References

[1] Fernández-Soriano, A. M., "Una propuesta para la integración de restricciones en hojas de cálculo," Master's thesis, Universidad Politécnica de Madrid (2010).

[2] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns - Elements of Reusable Object Oriented Software," Addison-Wesley, 1995.

[3] Jaffar, J. and M. J. Maher, *Constraint logic programming: A survey*, Journal of Logic Programming **19/20** (1994), pp. 503–581.

[4] Kassoff, M. and A. Valente, *An introduction to logical spreadsheets.*, Knowledge Eng. Review (2007), pp. 213–219.

[5] Lieberman, H., F. Paternò and W. Volker, editors, "End User Development," Human-Computer Interaction **9**, Springer Verlag, 2006.

[6] Powell, S. G., K. R. Baker and B. Lawson, *A critical review of the literature on spreadsheet errors*, Decis. Support Syst. **46** (2008), pp. 128–138.