ABSTRACT
The popularity of the spreadsheet attests to its success at providing a usable programming interface to users with no programming experience. This success prompts the question of how a spreadsheet could be extended to be more powerful while retaining its ease of use. Adding the ability to express and satisfy constraints within the spreadsheet would enable it to be used to solve more complex problems. In this paper, a model for adding constraint satisfaction to a spreadsheet is given. It is designed to be spreadsheet-centric, in that the means to define and solve constraint networks is designed to be familiar to spreadsheet users. The model is implemented using Microsoft Excel and is contrasted with other models of adding constraint satisfaction to spreadsheets.

1. INTRODUCTION
It is common knowledge that spreadsheet interfaces do well to allow users with little or no formal programming experience to write simple but useful programs. They allow a user to set up a chain of functions to be evaluated, using primarily arithmetic and branching. Because the values of the functions are displayed in a grid and updated dynamically, and the possible computations are all fairly simple, it is easy for users to spot and correct mistakes. This has made the spreadsheet ubiquitous in domains where non-programmers have a need for numerical computation.

The success of the spreadsheet paradigm warrants a look at the ways in which it can be extended and adapted to solve more complex problems while remaining simple and usable. Because the language to describe formulae in a spreadsheet is a limited functional programming language, spreadsheet extensions sometimes try to mimic successful designs from other programming languages. These include adding a type system [1] and a method of defining reusable subroutines [2]. Others have incorporated concepts from constraint logic programming [3].

Because spreadsheet programming is often thought of as “creating a model,” it seems that constraint satisfaction would be a natural fit. A constraint network is a problem modelled using variables and constraints. The constraints outline the relationships between variables. Hence, in a spreadsheet with constraint satisfaction, the normal process of defining relationships between cells would remain, but the relationships would be more expressive. Instead of declaring a cell to be the result of a deterministic calculation, it could be specified as a value that satisfies a complex relationship between two or more cells. This would allow the simplicity of the spreadsheet to be used in domains where non-programmers have a need for automated reasoning or constraint solving.

In practise, there are restrictions placed on spreadsheets that make defining these relationships impossible. The restrictions maintain the ease of programming spreadsheets, and therefore cannot be lifted without hindering usability. This paper presents a model for augmenting a spreadsheet with a means to define constraint relationships between cells. The model is intended to minimize the loss of usability that is a consequence of relaxing the restrictions. An implementation of the model for Microsoft Excel is also discussed, as well as the approach of other logical spreadsheets [4, 5, 6, 7, 8, 9, 10].

2. THE SPREADSHEET PARADIGM
A spreadsheet is a rectangular array of cells usually addressed by a string consisting of a column index represented by letters followed by a row index, e.g., ‘A1’ or ‘FG417’. The value of each cell is determined by a formula associated with the cell. The formulae are governed by what has come to be called the “value rule” [11], that is, a cell is determined solely by its own formula. A formula may reference other cells, but there is no mechanism by which a formula can alter a cell other than its own cell. This ensures that each cell can be verified by only verifying its formula and the formulae of the cells it references. No other factors play a part.

Another important concept is the recalculation order of the spreadsheet. The original spreadsheet, VisiCalc, calculated the entire spreadsheet in a single pass, either going row-by-row or column-by-column. In order to make sure that after each calculation the value of every cell was correct and final, users were discouraged from writing formulae that contained forward references (references to cells that would be calculated after the referencing cell), and circular references (references to the referencing cell) [12]. Since then, natural order
recalculation has solved the problem of forward-references by calculating cells using a dependency graph. However, it has created the potential to turn circular references into infinite loops. Because of this, modern spreadsheets either generate an error when faced with a circular reference, or place an upper bound on the number of times a formula can recurse.

3. MODELLING CONSTRAINTS IN A SPREADSHEET

If the spreadsheet generates an error on encountering a circular reference, there is no natural way to define a constraint relationship between two cells. For example, if we want to specify the constraint that cell A1 is not equal to cell B1, then either A1, B1 or both must contain a formula equivalent to A1 ≠ B1. All of these cases require a circular reference.

There are a two ways to circumvent this limitation. Relationships between a set of cells could be specified externally, either in other cells, or some interface outside of the spreadsheet. The first suggestion violates the value rule, and thus increases the difficulty of programming spreadsheets by introducing side effects. A problem common to both is what to do when inconsistency is introduced by the user. Either a warning is generated, or the constraint is relaxed to allow the inconsistency. Neither option is correct for all situations.

Another approach would be to allow circular references in some fashion. The circular reference does not have to be evaluated, only interpreted as symbol that represents a variable in a constraint network. However, this means that all formulae may contain circular references, and that the method of determining whether a circular reference is intended or accidental becomes more complex, as they cannot be forbidden per se.

Here, the second approach is used. A constraint relation is defined by a function, CONSTRAIN, that accepts a list of constraints as parameters. Each of these constraints is a relation between the calling cell, other cells, and constants. For example, cell A1 could contain the formula CONSTRAIN(A1 ≠ B1). Since the value rule should be preserved, it is important to note that the value of cell B1 will not change as a result of this constraint.

Hence, if B1 is set to a constant, A1 will select a value from its domain that satisfies the constraint A1 ≠ B1. If, however, B1 also uses the CONSTRAIN function to specify the same constraint, both cells will have values selected from their domains that satisfy the constraint.

In order to associate cells with domains, a second function is introduced for convenience. The DOMAIN function either accepts a range of numbers or a set of values that constitute the cell’s domain. It is used inside of the CONSTRAIN function as a constraint. For example, a cell’s formula could be CONSTRAIN(DOMAIN([0,9])) for range 0 through 9, or CONSTRAIN(DOMAIN([3,5,7])) for the values 3, 5, 7. Of course, it could use cell references for domain values as well.

Constraints defined by CONSTRAIN each need to have at least one circular reference. Any constraint that does not have a circular reference is (almost) guaranteed to be unsatisfiable. This is because the only variable that can be changed, the calling cell, has no effect on the constraint. It is only satisfied if it happens to be satisfied. Since this feature can be duplicated using an if-then-else construct, it is not necessary. This allows for constraints without at least one circular reference to be forbidden, as they serve no purpose within a spreadsheet.

This, in theory, makes determining whether a circular reference is deliberate or accidental quite simple: if a cell is constrained, each constraint must contain a circular reference, otherwise, if a circular reference is encountered, the spreadsheet is free to handle it as it normally would.

There is one other caveat for the CONSTRAIN function. While it may be possible to place it anywhere within a formula, some difficulties arise when it does not represent the final value of the cell it is called from. The difficulties are due to the fact that there is no way to reference the component parts of a formula. Because of this, it may not be able to simply return a solution to the constraints it specifies, rather, it might have to perform some sort of manipulation on the formula in order to convert it into a form in which a single CONSTRAIN function represents the final value of the cell. Therefore, because of both the ease of implementation and the conceptual simplicity of a cohesive relationship between cells and constraint variables, the result of the CONSTRAIN function must represent the final value of the cell.

4. IMPLEMENTATION

This model has been implemented for Microsoft Excel. It is written as a C# add-in that exposes the CONSTRAIN and DOMAIN functions, as well as monitoring events to detect when cells are modified. The add-in maintains information about which cells are constrained, and whether, upon a recalculation request, it needs to solve the constraint network or simply provide a cached solution. Currently, it only provides the first solution found, but it is not inconceivable to provide a means of selecting other solutions.

When Excel calls the CONSTRAIN function, the list of constrained cells is used to build a constraint network. Each cell gets a corresponding variable, and the cells’ formulae are parsed in order to determine the constraints on the variables. The network is then solved using a solver based on CSharpCream, a C# translation of the original Cream [13]. This solution is stored and used to provide values to cells until the constraint network is changed on the spreadsheet, and the network must be solved again.

Excel, by default, generates an error message when there is a circular reference in a formula. One way to avoid this is to let the arguments of the CONSTRAIN function be strings, which are not evaluated. This way, if a circular reference is introduced in a non-constrained cell, the user is suitably warned. Another option is to disable circular reference check-
ing by enabling bounded recursion. Of course, with this option, circular references could potentially go unnoticed. However, because Excel provides many conveniences for formula entry that are disabled when entering strings, this option is preferred. It is also quite possible to check cells for circular references and alert the user from within the add-in.

5. EXAMPLE

As an example, the model will be used to solve a $3 \times 3$ magic square. A magic square is simply an arrangement of distinct numbers on a grid such that the sum of each row, column, and diagonal is some constant. It can be modelled in Excel by reserving space for the constant, the grid, and by building checks for each of the sums (see Fig. 1). The checks are simple boolean expressions. For example, $F4$ is $C1 + D2 + E3 = A2$.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Magic Constant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>4</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

Fig. 1. An empty magic square modelled in Excel

To solve the magic square, the `CONSTRAIN` function can be used in each of the 9 magic square cells, $C1$ through $E3$. The first constraint on each cell will be the domain, which should be between 1 and the constant. The second constraint will be that each cell is distinct. The remaining constraints are the sums, and will be defined using the checks already specified. Using the model given in Fig. 1, cell $C2$ would have the formula

\[
\text{CONSTRAIN}(\text{DOMAIN}(1, A2), \text{NEQ}(C1:E3), C4, F2).
\]

Here, `NEQ` (not equals) is not an Excel function, but is defined for use with `CONSTRAIN`. Filling out all the cells in a similar manner will yield the result shown in Fig. 2.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Magic Constant</td>
<td>2</td>
<td>11</td>
<td>8</td>
<td></td>
<td>TRUE</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>13</td>
<td>7</td>
<td>1</td>
<td></td>
<td>TRUE</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td>3</td>
<td>12</td>
<td></td>
<td>TRUE</td>
</tr>
<tr>
<td>4</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td></td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Fig. 2. A solved magic square in Excel

6. EVALUATION

This model seems to integrate well with the normal use of spreadsheets. That is, on their own, the `CONSTRAIN` and `DOMAIN` functions are rather blunt tools for specifying a constraint network. It is the use of references and intelligent copy-and-paste that make them useful for more complicated tasks. This is akin to typical spreadsheet functions [14].

The fact that the constrained cells are dynamically updated allows for rapid trial-and-error development of spreadsheet programs. However, due to the nature of constraint solving, this process is bound to become unusably slow when the constraint network becomes large. In Excel, it is possible to turn off dynamic recalculation, a feature that is designed to be used when dealing with a slow recalculation process.

In order to fully integrate with Excel or any other modern spreadsheet, the constraint solver would have to be able to solve a wide range of problems involving a myriad of statistical, financial, and engineering functions. In its current state, the implementation’s solver handles only simple, finite-domain constraints.

7. COMPARISON TO OTHER LOGICAL SPREADSHEETS

There have been a number of logical spreadsheets that have been built and presented. (A short introduction is given in [3].) Many of them specify constraints using one cell to describe the relationship between other cells [10, 6, 5, 4]. As stated previously, because of the inconsistency with the value rule, this approach is undesirable. One of the problems that may occur is losing constraints. If a constraint is put on an obscure cell, say ZZ1024, it may be forgotten and subsequently hard to find. It also makes it difficult to learn exactly how a cell gets its value.

The other solvers store constraints in an external list [7, 8, 9]. This makes it easy to keep track of the constraints placed on the spreadsheet. It technically does not violate the value rule, either, since the constraints are external to the spreadsheet. The way the values are added to cells is analogous to the running of a macro.

Both approaches, however, have the problem of dealing with inconsistency introduced by the user. A decision has to be made as to whether the constraint should be upheld, and the inconsistent value removed, or vice versa. This is discussed at length in [7].

With the model proposed in this paper, as soon as a value is entered in a constrained cell, the cell is no longer considered part of the constraint network, except as a constant where it is referenced. The new value might render the constraint network unsolvable, however. In that case, it is conceptually similar to a situation where $A1 = 1/B1$, and the user sets $B1$ to 0. Essentially, the decision of what to do with inconsistent input is determined automatically due to conformance
with the spreadsheet paradigm—the input remains, as the old constraint was lost upon new input.

While overwriting constraints in this manner is natural to spreadsheets, it does not allow for maintaining a relationship between two cells. A basic example is storing a temperature in both Celsius and Fahrenheit [9]. If an external constraint specifies the relationship between two cells, either one can be determined by the other, and if both are specified, the input can be validated by the constraints. While this can largely be emulated using logical tests, as soon as both values are overwritten, the formulae have to be reentered in order to determine one value from the other, whereas, in other models the relationship is preserved during editing.

8. CONCLUSION AND FUTURE WORK

The spreadsheet paradigm can be made to accommodate constraint solving while (mostly) upholding traditional restrictions on what kind of formule cells can contain. The implementation is basic, but is expressive enough to describe and solve useful problems such as scheduling employees.

There is no objective test that can be performed in order to learn which model of mixing spreadsheets and constraint satisfaction is the best; different situations may call for different models. The argument presented in this paper is that this model provides a uniquely gentle learning curve for utilizing constraints from within a spreadsheet. As such, it is decidedly spreadsheet-centric, but offers the full gamut of constraint solving capability.

There are a number of ways in which the implementation may be improved. Support for the more ubiquitous functions like \texttt{SUM} and \texttt{AVERAGE} could be added to the solver. Along with this, the solver could be extended to solve continuous domain constraints, as they would be useful for handling the many statistical functions that spreadsheets offer. Finally, the solver could make use of dynamic constraint propagation algorithms to improve performance [15].

9. REFERENCES


