Display-Based Problems in Spreadsheets: A Critical Incident and a Design Remedy

David G. Hendry
School of Computer and Mathematical Sciences
The Robert Gordon University
Aberdeen, United Kingdom, AB1 1HG

Abstract

A critical incident is reported where a discretionary user needed to partition the spreadsheet grid into an arithmetic series of blocks and apply a function to each, but was unable to devise a solution. The incident is first scrutinized to explicate the reasons for the failure. Then, a new method for progressional computation on tabular displays is described and reviewed for usability. The method increases the expressive power of cell referencing but also simplifies the notation. The spreadsheet is shown to be a surprisingly poor artefact for three classes of display-based problem.

Keywords

visual programming, spreadsheet, interaction style, human factors, cognitive dimensions, display-based problem solving, programming by demonstration

1 Introduction

The mundane spreadsheet is a spectacularly popular visual programming environment where people structure their problems on a grid and carry out computations by copying and pasting formulae. It, better than many programming languages, bridges a person's tasks in a problem-domain with solutions in the computer-domain. The spreadsheet is a success story for visual programming, and it has justifiably acquired exemplary status [1].

There are, however, display-based problems that are extremely difficult to solve with the spreadsheet. By 'display-based' I mean a problem where a few pencil jottings can specify a user's goals, once some initial numbers and headings have been arranged on the grid(see figure 1). Perceptual cues, such as the use of white space and changes in font style, can be coded on the display and thereafter help people 'see' what needs to be done. The difficulty is that mapping *some* display-based problems into the formulae needed by the spreadsheet is extremely hard, at least compared to how easily the problem can be stated and understood. Surprisingly, not all computations to tabular data are easily solved with spreadsheets. Why?

In this paper a 'critical incident' is examined in some depth. It is a problem which can be stated in one or two short sentences and understood in seconds but which defies easy solution in the spreadsheet. The incident was first described by a lecturer in management economics when asked to describe his use of spreadsheets and what he considered to be their strengths and weaknesses [2]. John Carroll and colleagues note the value of analysing critical incidents for improving interface design and enriching the Human-Computer Interaction design lore [3].

	A	B	C	D	E	F	G
1	Income for Widgets and Rings by Week						
2		**US $**		**UK Stirling**		**Sum (UK)**	
3	**Week**	**Widget**	**Ring**	**Widget**	**Ring**	**Widget**	**Ring**
4	1	134	312	*89.78*		*89.78*	
5	2	257	244	*172.19*		*261.87*	
6	3	157	267				
7	4	269	311				
8	5	224	482				
9	Exch. Rate (US $ -->UK Stirling)				0.67		

Figure 1: Spreadsheet for converting currencies and calculating cumulative sums. In this idealised case, the problem is structured with headings and raw data. The italic numbers and arrows are 'jottings' for what next needs to be done.

The goal of this paper is not to criticise the spreadsheet nor to quibble about its features, but rather to better understand the limits of its tabular display and formula language. This is important given its status and that its pioneering features are often reworked in new designs(for reviews see [2, 4]).

The integration of a task-oriented formula language and tabular display is the major reason for the success of the spreadsheet [5]. The grid is a familiar and flexible display for structuring a wide range of problems. The formula language provides a set of mathematical, business, and statistical functions, which compute new results via a notation for drawing data from cells on the grid.

Figure 1 illustrates the power of these two simple ideas when combined with copy and paste. An accountant's goal is to convert US dollars into UK pounds and to compute cumulative sums for Widgets and Rings. Headings and raw data have been entered onto the surface of the sheet and the accountant is about to enter two formulae. The formula for currency conversion:

```
=B4 * $E$9
```

will be entered into cell D4 and copied down-and-left to E8. The formula to compute the cumulative sums:

```
=Sum(D$4..D4)
```

will be entered into cell F4 and copied down-and-left to G8. The dollar signs, which are used to fix a reference to a particular column or row no matter where the formula is copied, are crucial. An omission of any single dollar sign(and possibly the inadvertent insertion of an extra one) would cause data to be drawn from incorrect cells.

Once the raw data and headings are entered on the sheet the problem is structured and the accountant can sketch what is required with a few pencil jottings—this makes the problem trivial. In this idealisation the accountant must learn how to translate his or her pencil marks into formulae that when copied produce the correct results. For this problem that is not difficult, but even here a precise understanding of cell referencing techniques is required, that is, where to put the dollar signs. (See any spreadsheet reference guide for a full description of cell referencing—a programming feature that the reader is assumed to be familiar with hereafter.)

2 Critical Incident: Difficulty Summing Blocks

A list of numbers has been placed into the first row of a spreadsheet. The data is grouped into blocks of 4, and the goal is to compute the sum of each of the blocks and display the results in a nearby row(see figure 2). To solve this problem requires that various system-oriented functions be combined, the result being a very complex formula:

```
=Sum(  Range(  MakeRange(
       ColOf($B$1)+4*(Column()-ColOf($B$1)),
        RowOf($B$1),
       ColOf($B$1)+4*(Column()-ColOf($B$1)+1)-1,
        RowOf($B$1)
     )))
```

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Input:	3	3	5	4	5	5	8	9	9	4	4	1	. . .	
2															
3	Output:	15	27	18	. . .										
4															

Figure 2: The critical incident. In sum-blocks, the problem is to compute the sum of successive blocks of numbers. Here, the blocks are oriented horizontally, abut each other, and four numbers are contained in each block.

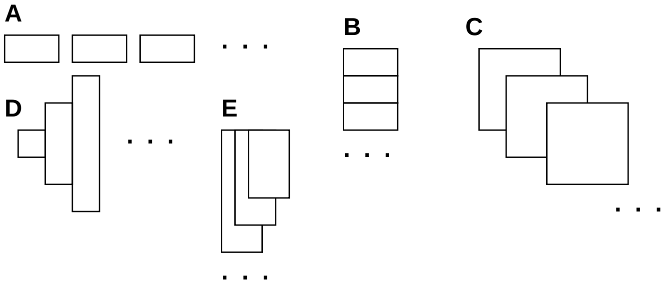

Figure 3: Patterns for partitioning the spreadsheet grid in horizontal, vertical, and diagonal directions. In A the distance between blocks is greater that 0, in B it is 0, and in C it is less than 0. In D the blocks grow at each end, but in E at only one end, which is the same pattern as the cumulative sums in figure 1. Though not shown, geometric progressions might also be required.

Though this solution is particular to Claris Resolve, one brand of spreadsheet, similar solutions are required in other brands—below we examine why this problem cannot be solved with the standard machinery for cell referencing.

Creating this formula is nothing less than programming: one has to know about primitive functions which lack names that concord with the problem description, function composition, casting types(e.g., Range() converts a string to a range type), self-reference(e.g., Column() returns the column number in which Column()'s formula is found), and so on. Hendry & Green discuss one discretionary user's unsuccessful attempt to solve this problem, and his conclusion : "I've kind of got a problem that's too difficult for the standard tools you are given" [2].

The sum-blocks problem requires that the spreadsheet grid to be divided into horizontal blocks, but as illustrated in figure 3, people might wish to partition the grid according a variety of arithmetic progressions. Each of these patterns are easily sketched on a tabular display, yet translating such sketches into formulae requires programming like that shown above. The difficulty is representing the pattern in the system, not with applying a task-oriented function to the partitions.

3 Remedy: Partitioning the Grid without Formulas

Why is it that sum-blocks cannot be solved with the standard machinery for cell referencing and formula copying and what enhancements are required so that it can? To answer these questions we shall first specify exactly how the references in a formula change when the formula is copied. The specification will reveal what limits the standard notation and suggest how its expressive power can be raised.

When one keys in a formula, copies it, then pastes it into an output region, several entities are tracked by the system. In figure 4, these entities are represented,

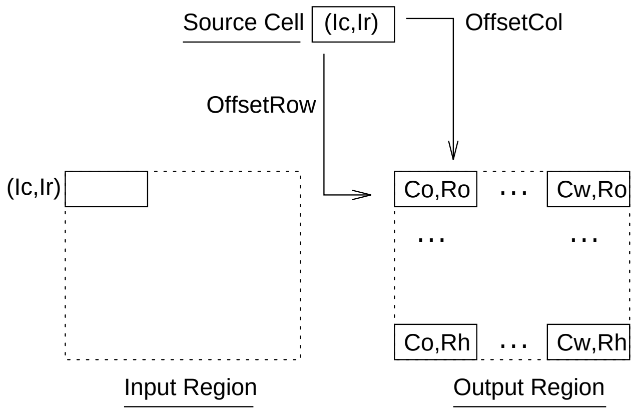

Figure 4: Representation of the entities that participate in formula copying. When a formula in a source cell is copied to an output region it draws data from an input region. (Precisely how is described in the text.)

where a formula consisting of a single reference to an input region, (I_c, I_r) is copied. I_c and I_r are the column and row components respectively (e.g., if A1 is a cell reference, then $A1 \equiv (I_c, I_r) \equiv (A, 1)$). These components are also typed as either absolute or relative(i.e., dollar sign or not) so that the three standard referencing types can be represented. Though a formula made of a single reference is a simple case, any conclusions derived from it are perfectly general since all references in a formula, no matter how simple or complex, change only according to their types and the distance copied. The distance of the copy gesture is given by vertical and horizontal offsets, namely OffsetRow and OffsetCol, and the size of the output region is given by its width and height. When the formula is pasted, it is placed into each cell in the output region but the reference changes to reflect the distance copied and the referencing type employed. If the new references in each of the cells in the output region are represented by the ordered pair (C_j, R_k), then the semantics of the copy operation can be specified by equations that compute the correct values for $C_0, C_1, \ldots C_w$ and for $R_0, R_1, \ldots R_h$.

To derive the equations, first consider the columns. If the column component of the reference is relative then:

$$C_0 = I_c + \text{OffsetCol}$$

and the column reference to the right is exactly the same except that it one cell farther away. Thus:

$$C_1 = C_0 + 1$$

and since the cell in the top right is w cells to the right:

$$C_w = C_0 + (w - 1)$$

However, if the column component of the reference is absolute, the copying distance (OffsetCol) has no effect; thus:

$$C_0 = I_c \qquad C_1 = I_c \qquad C_w = I_c$$

The exact same pattern occurs for the rows. Thus, given the entities in figure 4, the two equations below specify how the references in a formula change when it is copied from a source cell and pasted into an output region of given width and height:

$$C_w = \begin{cases} I_c + \text{OffsetCol} + w & \text{if relative}(I_c) \\ I_c & \text{otherwise} \end{cases}$$

$$R_h = \begin{cases} I_r + \text{OffsetRow} + h & \text{if relative}(I_r) \\ I_r & \text{otherwise} \end{cases}$$

where $0 \leq w \leq (\text{Width}-1)$ and $0 \leq h \leq (\text{Height}-1)$.

Given this specification it is possible to state precisely the expressive power of cell referencing. These equations are arithmetic recurrence relations; thus, they can be written in this general form:

$$C_w = I_c + \sum_{k=1}^{\text{OffsetCol}} d_c + \sum_{k=1}^{w} d_c \tag{1}$$

$$R_h = I_r + \sum_{k=1}^{\text{OffsetRow}} d_r + \sum_{k=1}^{h} d_r \tag{2}$$

where $0 \leq w \leq (Width-1)$ and $0 \leq h \leq (Height-1)$.

Now to make this reverse-engineering payoff. The first point is that changes to the column component of a reference, C_w, will occur only when w changes, and similarly the row component, R_h, will change only when h changes. What this means is that for a formula to gather new data by column, the formula must be pasted horizontally, and similarly for a formula to gather new data by row it much be pasted vertically. These two constraints prevent a person from summarising columned-data in a row or rowed-data in a column. Neither task is possible with the basic notational machinery—an unexpected result of this analysis that predicts a new type of 'critical incident'.

The more important point is that the common difference terms, d_c and d_r, are restricted to either 1 or 0 for relative and absolute referencing, respectively. It is this either-or restriction through the omission or insertion of dollar signs that accounts for the complex programming required to solve the sum-blocks problem. To raise the expressive power of the referencing notation so any of the arithmetic progressions in figure 3 can be generated, a mechanism for attaching any integral step value to the row and column components of a reference is required. The problem, clarified by deriving the equations, is to find a convenient and comprehensive way of doing just that.

Several approaches were considered before settling on a simple style of programming by demonstration, where a person:

1. keys in the first two formulae of the progression,

2. selects the region over which it should be generated, and

```
A      =Sum(D5..D5)
       =Sum(D5..D6)
       =Sum(D5..D7)

B      =B5 * E11      =C5 * E11

       =B6 * E11      =C6 * E11

C    =Sum(B3..E3) =Sum(F3..I3) =Sum(J3..M3)
```

Figure 5: Three formula patterns to continue. The formula within each box is computed based on the examples outside. The arrows indicate the direction the computation is to be extended. In A, cumulative sums are computed, by filling downward. The difference terms are: term-1, $(d_c, d_r) = (D - D, 5 - 5) = (0, 0)$; term-2, $(d_c, d_r) = (D - D, 6 - 5) = (0, 1)$. In B, a block of data is multiplied by a constant, by filling down-to-right. The difference terms are: term-1, $(d_c, d_r) = (C - B, 6 - 5) = (1, 1)$; term-2, $(d_c, d_r) = (E - E, 11 - 11) = (0, 0)$. In C, the sum-blocks problem is solved, by filling right. The difference terms are: term-1, $(d_c, d_r) = (F - B, 3 - 3) = (4, 0)$; term-2, $(d_c, d_r) = (I - E, 3 - 3) = (4, 0)$.

3. chooses the appropriate menu option, say Continue-Pattern.

Then, the system fills in the selected region. This approach requires a method for computing the values for d_r and d_c, and fortunately this simply requires two adjacent terms from the sequence. Thus, given the two formulae (C_0, R_0) and (C_1, R_1), for example, it is possible to solve for d_r and d_c by rewriting equations Eq. 1 and Eq. 2:

$$d_c = C_1 - C_0$$

$$d_r = R_1 - R_0$$

and all the references in the pattern can be computed.

Figure 5 shows three examples where, based on two initial calculations, a series of computations is produced. There the accountant's problems, as well as sum-blocks, are solved with the Continue-Pattern referencing style. Indeed, any problem which requires the grid to be partitioned into an arithmetic series can be solved with this technique. Sum-blocks is no longer a difficult programming exercise.

4 Evaluation: Cognitive Review of Continue-Pattern

What are the cognitive costs and benefits of Continue-Pattern? While hardly polished, its broad strengths and weaknesses can be reviewed with the 'cognitive dimensions framework' [6]. That framework provides an general but well defined vocabulary for discussing the aspects of a notation that determine its usability. The most pertinent dimensions are: *closeness of mapping*, *hard mental operations*, and *role expressiveness*. Below these dimensions are used to discuss the virtues of Continue-Pattern. (Italics is used when referring to a cognitive dimension.)

Continue-Pattern offers a *close mapping* between the problem- and computer-domain. This dimension refers to 'primitiveness' of the commands needed to complete tasks. In the solution to sum-blocks, found in section 2, we saw that many primitive commands, entirely unrelated to the problem, had to be composed so to compute a series of block-sizes on the grid. Continue-Pattern eliminates these 'programming games'[7]. Computations can be carried out over any arithmetic progression of blocks. All that is required is that the person enter the first two formulae and select the range over which the pattern is to be continued. This allows people to rely on the perceptual structure of the display, bypassing the explicit calculation of block sizes with computer code.

Importantly, the technique makes the three referencing types unnecessary. Certainly, for some people learning the mapping between problem and referencing type, that is, where to put the dollar signs, is a source of difficulty[8, 2]. Besides these reports from the class room and office, there are other grounds for thinking that the dollar signs are problematic. What is the difference between `=A$1 + $M2` and `=$A1 + M$2` when copied diagonally down-and-right? The meaning of these two formulae are not easily discerned, compared to, for example, `=$A$1 + M2`, which adds a constant to each cell in a block. The dollar signs may very well be a *hard mental operation*, a notational feature which makes the expression of a meaning in the problem-domain unnecessarily difficult and becomes particularly problematic when two or more of the troublesome features must be combined.

Consider, for example, this formula:

```
= __A__1 * __M__10
```

where dollar signs are either inserted in or omitted from the slots. Suppose that the formula is copied either down, across to the right, or diagonally down-and-right, and the derived formula is:

```
= __?__? * __?__?
```

Now, where do the question marks refer, that is, where does the formula draw its data from? In theory there are 48 possible paths from base to derived formula to consider ($2 \times 2 \times 2 \times 2$ for the dollar signs, multiplied by 3 for the direction). In practice some of the paths are easy to ignore and in time people probably learn a few standard configurations for placing the dollars signs and thus circumvent the full problem space. However, Hendry and Green describe one economist's spreadsheet for modelling global warming where this is certainly not the case [9].

For the advantage that Continue-Pattern makes the referencing types unnecessary people pay a price: they must enter formulae for the first two results they wish to compute. Under some conditions this repetitiveness may be a weakness, where it would be more efficient to create a single formula which employs the referencing types. However, for people who are using the spreadsheet for incremental idea capture those

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		Input Region					Calculation Region						
2		X	Y	Y	Z		1	10	10	100			
3		Z	Z	X	Z		100	100	1	100			
4		Y	Z	X	Y		10	100	1	10			
5							Number of Zs, Ys, and Xs:				543		

Figure 6: Tally-items. The problem is to tally the number of Xs, Ys, and Zs in the input region. The solution is to re-represent the items as numbers and to then compute their sum.

conditions may be rare [2]. Determining if there is an efficiency crossover and when it occurs will require empirical work.

Continue-Pattern renders the dollar signs extraneous and thus individual formulae become less *role expressive*, that is, one cannot determine the pattern of referencing for a set of derived formulae by inspection of the base formula alone. The reference `$M$10`, for example, immediately signals that a cell is being used as a constant data store, but with Continue-Pattern to discriminate 'absolute' from 'relative' cell references all corresponding references in the two adjacent formulae would have to be compared. This is a difficult task in spreadsheets where the *visibility* of formulae is poor. Continue-Pattern effectively creates *hidden dependencies* between corresponding references in two adjacent formulae. In practice this may be a minor consequence because when people reverse engineer formulae they seem to rely on headings and the visual rhetoric for how data is structured on the grid.

Continue-Pattern appears to be neutral with respect to the other dimensions, such as *viscosity*(the ease which the sheet can be changed), and *secondary notation*(extra information not carried by the formal syntax).

This usability review of Continue-Pattern suggests that, if suitably refined, it would be an effective method for partitioning the grid according to arithmetic progressions. It offers a very *close mapping* between problems and solutions and renders the referencing types unnecessary; thus, simplifying what needs to be learnt to propagate formulae. For these benefits, it makes formulae less *role expressive* and requires users to enter two examples of what is required.

5 Discussion

5.1 Other Display-Based Problems

Sum-blocks is not the only critical incident where a display-based problem is difficult to solve with the spreadsheet. The tally-items and sort-blocks problems, shown in figures 6 and 7, exemplify two additional classes of problem. Both problems confronted PhD students in psychology for whom the spreadsheet was but one data analysis tool. (I learnt of these problems through informal conversation.)

With tally-items, each cell in an input region contains an item, either an X, Y, or Z, and the problem is to compute tallies for each type of item. One solution to this problem requires two formulae. The first maps the letters into numeric values so that they can be added; it is:

```
=if( B2='X',1,
    if(B2='Y',10,
       if(B2='Z',100,0)
    ))
```

If B2 is the top-left cell in the input region and the formula is copied over a rectangular region, call it the calculation region, of the same size of the input region, then the display will appear as shown in figure 6. The second formula:

```
=Sum(G2..J4)
```

computes the sum of the calculation region, giving in this example the number 543 from which the tallies can be read off. A precondition for this formula is that there are fewer than 10 Xs, Ys, and Zs.

The student who reported this problem had programming experience in industry, yet, even with this experience, he was unable to devise a solution. Of course, he could have written a macro but did not want to. His thinking was that there must be an easy solution. Like sum-blocks the difficulty is that the mapping between problem and solution is too wide. Here, though, the trouble is in re-representing the problem so that it is amenable in the spreadsheet, not with learning the details of the system-oriented functions for accessing the grid. Both `Sum()` and `If()` were familiar to the student. The manoeuvre shown in figure 6 is far from obvious, yet without functions for matching patterns on the grid, programming games of that sort are required.

Another display-based problem is sort-blocks. In this problem, there are a hundred blocks of data, arrayed in columns and beginning in column B. The goal is simply to sort each block in place. Two of the blocks have been sorted and you would like to continue the pattern for the remaining 98 blocks(see figure 7).

It seems that the only non-repetitive solution is to write a macro, and although this is not a difficult program, it requires skills in imperative programming. But, even if you are fluent with the macro language, why should you need to rely on it for such a seemingly straight-forward problem?

The sort-blocks problem requires that an environmental command, or for that matter, any function be

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		10	4	7	14	...							
2		12	7	3	87	...		Continue sorting blocks ...					
3		45	10	5	12	...							

Figure 7: Sort-blocks. The problem is to sort a pattern of data blocks. Here, the blocks are arrayed in columns, and consist of three numbers.

repeated over a series of regions. This problem is similar to sum-blocks in that one needs to partition the grid and visit each block in turn. The distinguishing characteristic, though, is that the data needs to be changed in place. However, traditional formulae do not do that; rather, they only draw data from other cells and leave behind a result in their residing cell.

Sum-blocks, tally-items, and sort-blocks are display-based problems from three respective classes. Given a formula language and tabular display, people may wish to:

- Partition the grid according to simple arithmetic sequences
- Match certain patterns of data, and count the number of instances
- Visit a pattern of partitions and change the data in place

Each problem is a critical incident because the programming solution is unnecessarily complex, breaking Alan Kay's adage that "simply problems should have simply solutions".

5.2 What is a spreadsheet?

For Fischer and colleagues it is an excellent problem-domain environment, a computerised ledger, where the visual and notational abstractions concord with what book-keepers, lawyers, or even school children already know [10, 11]. For Nardi and colleagues it is this too, but their seminal research on spreadsheet use *in situ* describes the intricate web of activity that really does exist and they clarify the reasons for it [12, 4]. But more, for them it is an outstanding example of how a textual notation can be combined with a tabular display—an idea they generalise in ACE where a variety of general display types, called application frameworks, can be tailored to problem-domains [13]. Many other views are possible: for an engineer it might be ideal for solving differential equations, for an architect its utility may lie in computing the volumes of tetrahedrons.

Sum-blocks, tally-items, and sort-blocks present a different view, a new answer for what a spreadsheet is. In these display-based problems people can represent the basic structure for what is required, eye-ball some data, and effectively, jot down a plan to follow. This grid-cum-calculator view of the spreadsheet is compelling if only because of its simplicity. It is not an accountant's ledger, and while some mileage can be gained from this most conventional view these display-based problems reveal some severe limitations. Donald Norman would say that these 'experiential' tasks become deeply 'reflective' [14].

One could argue that spreadsheets were never intended to be used to solve these sorts of display-based problem. That is almost certainly true. Nevertheless, people do seem to think that the spreadsheet should offer support, and how can you blame them for representing their problems on a tabular display and using its structure to 'see' what needs to be done next. Using Donald Schön's words the display 'talks back'[15], which is perhaps the spreadsheet's greatest strength. Unfortunately, the cell referencing machinery, does not provide the expressive power to access the grid with the flexibility required; thus, the recourse to macro programming and complex function composition. Moreover, even for the expert programmer, it seems that the spreadsheet is not open to the notational redesign that is required so that general solutions to these problems could be provided.

6 Conclusion

The spreadsheet is an outstanding visual programming environment. However, comparatively little is known about how people actually use a tabular display and formula language [2]. In this paper a person's problem was treated seriously, as a critical incident, to better understand the limits of the spreadsheet. This user-centred stance resulted in Continue-Pattern, a method that utilizes a tabular display and a simple cell referencing notation to allow users to carry out computations to an arithmetic series of blocks. It has higher expressive power than the spreadsheet's standard referencing machinery, and is simplier too, though it requires that two example formulae be entered.

The programming effort to solve some display-based problems is out of line with their evident simplicity. Experiential problems should not become reflective programming exercises. When they do, we should try to understand why.

Let us not become complacent with the spreadsheet model. It embodies many niches of function and more critical incidents are bound to turn up, suggesting new answers for what a spreadsheet really is. It is an exemplary tool, but the tabular display and formula language can be made into an even better problem-domain environment.

Acknowledgements

Many thanks to Thomas Green who inspired and supported this work, and to David Harper who allowed me to finish it. This work was funded, in part, by the Principal's Research Fund at The Robert Gordon University and UBILAB of the Union Bank of Switzerland. Thanks also go to the informants for bringing their spreadsheet problems to my attention and to Martin Stacey for his criticism of an early draft.

References

[1] B. A. Myers, D. C. Smith, and B. Horn, *Languages for Developing User Interfaces*, chapter Report of the "End-User Programming" working group, pp. 343–366, Jones & Bartlett, 1989.

[2] David G. Hendry and Thomas R. G. Green, "Creating, comprehending, and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model", *International Journal of Human-Computer Studies*, vol. 40, pp. 1033–1065, 1994.

[3] John M. Carroll, Jurgen Koenemann-Belliveau, Mary Beth Rossen, and Mark Kevin Singley,

"Critical incidents and critical themes in empirical usability evaluation", in *Proceedings of the HCI'93 Conference on People and Computers VIII*, 1993, pp. 279–292.

[4] Bonnie A. Nardi and James R. Miller, "Twinkling lights and nested loops - distributed problem solving and spreadsheet development", *International Journal of Man-Machine Studies*, vol. 34, pp. 161–184, 1991.

[5] Bonnie A. Nardi and James R. Miller, "The spreadsheet interface: a basis for end user programming", in *Proceedings of IFIP INTERACT'90: Human-Computer Interaction*. 1990, pp. 977–983, Elsevier.

[6] Thomas R. G. Green, "Describing information artifacts with cognitive dimensions and structure maps", in *Proceedings of the HCI'91 Conference on People and Computers VI*, 1991, pp. 297-315.

[7] Clayton Lewis and Gary M. Olson, "Can principles of cognition lower the barriers to programming?", in *Empirical Studies of Programmers: Second Workshop*. 1987, pp. 248–263, Ablex.

[8] J. R. Doyle, "Naive user and the lotus interface: a field study", *Behaviour & Information Technology*, vol. 9, no. 81-89, 1990.

[9] David G. Hendry and Thomas R. G. Green, "CogMap: a visual description language for spreadsheets", *Journal of Visual Languages and Computing*, vol. 4, no. 1, pp. 35–54, 1993.

[10] Gerhard Fischer, "Beyond human computer interaction: Designing useful and usable computational environments", in *Proceedings of the HCI'93 Conference on People and Computers VIII*, 1993, pp. 17–31.

[11] Gerhard Fischer and Andreas Lemke, "Construction kits and design environments: Steps toward human problem-domain communication", *Human Computer Interaction*, vol. 3, pp. 179–222, 1988.

[12] Bonnie A. Nardi, *A Small Matter of Programming: Perspectives on End-User Computing*, MIT Press, 1993.

[13] Jeff A. Johnson, Bonnie A. Nardi, Craig L. Zarmer, and James R. Miller, "Ace: Building interactive graphical appliations", *Communications of the ACM*, vol. 36, no. 4, pp. 41–55, 1993.

[14] Donald A. Norman, *Things that Make Us Smart*, Addison Wesley, Reading, MA, 1993.

[15] Donald A. Schön *The Reflective Practitioner*, Temple Smith, London, 1983.