Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model

D. G. Hendry and T. R. G. Green

MRC Applied Psychology Unit, 15 Chaucer Road, Cambridge CB2 2EF, UK

(Received 16 December 1993 and accepted 11 January 1994)

Ten discretionary users were asked to recount their experiences with spreadsheets and to explain how one of their own sheets worked. The transcripts of the interviews are summarized to reveal the key strengths and weaknesses of the spreadsheet model. There are significant discrepancies between these findings and the opinions of experts expressed in the HCI literature, which have tended to emphasize the strengths of spreadsheets and to overlook the weaknesses. In general, the strengths are such as allow quick gratification of immediate needs, while the weaknesses are such as make subsequent debugging and interpretation difficult, suggesting a situated view of spreadsheet usage in which present needs outweigh future needs. We conclude with an attempt to characterize three extreme positions in the design space of information systems: the incremental addition system, the explanation system and the transcription system. The spreadsheet partakes of the first two. We discuss how to improve its explanation facilities.

1. Introduction

The spreadsheet is often lauded in the HCI community for giving task-oriented users high computational power through effective displays and operations. Kay (1984), who foresaw that the spreadsheet style of programming would be integrated into many future systems, argued that the most powerful feature of spreadsheets is that people can see what is to be done and simply do it. Norman (1986) says that spreadsheets merge “the computational power of the computer with a clean, useful conceptual model, allowing the interface to drive the entire system, providing just the right tools for a surprising variety of applications” (p. 50). Designers often credit spreadsheets with inspiring their new designs, and the marketers of multimedia can often be heard crying for the invention of the next “spreadsheet” to drive sales in hardware.

Given their exemplary status, one might expect that much is known about how spreadsheets are used. Yet, although there are many studies of learning other programming languages, such as Pascal (Détienne, 1990), spreadsheets have received much less attention.

As part of ongoing work into the comprehensibility of programs (Green, Gilmore, Blumenthal, Davies & Winder, 1992), we asked users to explain how their own spreadsheets worked. Originally, the accounts contained in the resulting transcripts were to be used mainly to appraise a method for improving the comprehensibility of
program code (Hendry & Green, 1993). However, the informants provided views of
the pros and cons of spreadsheets that deserve to be set against the received wisdom
which views the spreadsheet as a legend in its own lifetime.

In the next section we summarize the small empirical literature on spreadsheets.
In Section 3, we present data from the informants dealing with the problems of
creating, comprehending and explaining spreadsheets. In Section 4, we use this
material to discuss the usability of spreadsheets from several vantage points. We
conclude in Section 5 by distinguishing different types of information systems: for
incremental growth, for transcription, and for explanation; the spreadsheet appears
to occupy a nearly extreme position in this design space as a device supporting
incremental growth at the expense of other types of usage.

2. Review of research on spreadsheets

It is certainly possible that more people "program" with spreadsheets than with any
other programming environment. Yet its literature is tiny. Three years ago, Nardi
and Miller (1990) remarked on the thinness of literature on spreadsheets, and the
situation has changed little. Of the approximately 900 papers published between
1984 and 1991 in the two leading conference series in the HCI community (the IFIP
"Interact" and the ACM "CHI" series), there are only seven entries in the keyword
indexes to papers on spreadsheets. (For comparison, there are 57 references to
papers on User Interface Management Systems.) Of the seven papers, two use
spreadsheets as an incidental part of research (e.g. as a task-environment for
research on speech recognition), four take inspiration from the spreadsheet model in
developing systems for new problem domains, and one analyses the advantages of
spreadsheets as a general model for end-user programming. Although not summma-
tional of the whole field, these numbers suggest the meagreness of the extant
research. The present state of research will be reviewed in following sections.

Given their astounding success, one would like to know about spreadsheets in
great detail, and one would like to know not just about the low-level details (e.g.
speed of entering formulae) but about "spreadsheet ecology": the place of the
spreadsheet in its users' lives, its relationship to competing tools and to collaborating
tools. Little has been reported about spreadsheet users; their roles in task-oriented
work settings; their life-cycle of creation, comprehension, modification, sharing, and
dissemination; their range of size and complexity. Likewise, little has been reported
on what features are regularly (or rarely) used and what features are difficult to use;
what "programming plans" or other mental representations are common; how
spreadsheets are made comprehensible; how they are explained to co-workers; and
so on. Indeed, apart from the series of papers by Nardi and her colleagues
(summarized below), almost nothing has appeared on these wide-aspects.

The existing literature on spreadsheets, covering hardly any of the above, can be
divided into reviews of the model and design innovations, controlled empirical
studies, and studies of spreadsheet users at work. The literature in each of these
categories is summarized below. In sum, the reviewers have emphasized the
strengths of the model and the evaluators have uncovered weaknesses, but these
have gone unnoticed by the innovators. The field studies show that spreadsheets are an effective communication tool and that the computations performed are generally simple.

2.1. REVIEWS OF THE SPREADSHEET MODEL AND DESIGN INNOVATIONS

Several researchers have reviewed spreadsheets from a theoretical or intuitive perspective, generally by taking a simple scenario and comparing the effort to complete it with a spreadsheet and with a traditional imperative programming language. In any scenario where data must be entered, computations performed and results displayed, the spreadsheet scores well. The malleable two-dimensional sheet eliminates the troublesome coding of input and output; iteration and the need to assign values to variables is disguised in cell referencing techniques and formula copying; and task-oriented statistical and business functions eliminate algorithm design. These reviews have tended to emphasize the strengths of spreadsheets, and generally ignore their weaknesses.

Norman (1986) examines the design of VisiCalc, the first electronic spreadsheet, and concludes that its success rested on its design model: "It let users work on their own terms, putting together a 'spreadsheet' of figures, readily changing the numbers and watching the implications appear in the relevant spots" (p. 57). Importantly, although the command structure for VisiCalc was hard to learn, this didn't matter because the first wave of users had no experience with other systems, were well practised, and truly needed a system that automatically updated row and column totals. Proposing that the spreadsheet be the basis for end-user programming, Nardi and Miller (1990) emphasize the strengths of its formula language (because it offers high-level, task-specific functions) and tabular format (because it offers great flexibility for representing problems). Myers, Smith and Horn (1992) summarize the opinions of a panel of experts. They claim, for example, that using cell formulae is easy, that computations are visible and local to cells, and that it is easy to spot mistakes because computed data is always up-to-date.

In perhaps the most balanced and insightful analysis, Lewis and Olson (1987) mention both strengths and weaknesses, but they consider their analysis speculative because they do not base it on the systematic collection of opinions or observations of task-oriented users. Their goal was to formulate principles that can "lower the barriers in programming". On the one hand, they note that an advantage of the spreadsheet is that it "suppresses the inner world" of variables and flow of control and eliminates many of the complexities and "programming games" associated with conceptually simple operations like adding a list of numbers or handling input and output. On the other hand, they note that debugging and reusing spreadsheets is difficult, weaknesses which are confirmed in the present paper.

Another analysis, the fruit of several years' research, can be found in Nardi (1993). Her book describes research on spreadsheets in the workplace and concluded that their success rests partly on their use of a visual formalism (see below), which led her and her colleagues to propose a more general approach, ACEKit, in which the visual formalism inherent in the spreadsheet was made into a framework for application development (Nardi & Zarmer, 1993; Johnson, Nardi, Zarmer & Miller, 1993). ACEKit generalizes interestingly from the lessons learned
from their analyses of the success of spreadsheets, but at the present stage there is no evaluative data to test the success of their design.

Others have enthusiastically endorsed the spreadsheet model without pausing to analyse it explicitly, and have concentrated on extending it by building new systems. Piersol (1986) describes a spreadsheet interface that was implemented in Smalltalk-80 and illustrates a surprising variety of applications based on the spreadsheet model; the formula language is Smalltalk and the cells can contain any Smalltalk-80 object (e.g. bitmaps, dictionaries, etc.). Spenke and Beilken (1989) describe PERPLEX, a system that extends the standard formula language to allow constraints to be imposed among sets of cells; thus, like Prolog, results can be computed either forwards (e.g. you provide the input values) or backwards (e.g. you provide the output value and some of the input values), and multiple solutions can also be displayed, among other effects afforded by logic programming languages. Wilde and Lewis (1990) describe NoPumpG II, a system which automatically updates not just cell values, but graphic plots of sets of data points; thus, for example, users can create formulae that generate curves but are not required to write complicated code to display them. Wilde and Lewis (1991) sketch several innovative representations of the spreadsheet which better reveal the causal connections between cells and allow for more explanatory information to be included on the sheet while keeping the display compact. Myers (1991) describes C32, a spreadsheet-like system for creating user interfaces where constraints among objects in a user interface can be viewed, created, debugged, and so on; the formula language is LISP. As a final example, Davenport and Harber (1991) describe Hypercalc, an enhanced version of the commercial spreadsheet Wingz in which formulae can be constructed to display messages in a variety of media (e.g. IF(C12/D12 > 50%, PLAYVIDEOSEGL, 0)); here, the spreadsheet model is extended to multimedia presentations, which may be valuable in education or where persuading an audience is the goal.

2.2. CONTROLLED EMPIRICAL STUDIES

Laboratory studies have generally examined the cognition involved when creating spreadsheets or learning how to use them—we have found no experimental work on how they are comprehended or modified. Nor are there any reports on spreadsheet programming with macro languages. Baxter and Oatley (1991) studied the learnability and usability of two brands of spreadsheets, finding that both spreadsheets were about as easy to learn for new users and were equally usable by experienced users. This suggests that, in general, results based on one brand of spreadsheet easily generalize to other brands because the defining characteristics of spreadsheets (e.g. a two-dimensional surface of cells and an underlying formula language for accessing and manipulating their contents) remain constant across brands. Olson and Nilsen (1987–1988), in a study on operator efficiency, showed, among other things, that apparently inconsequential differences in two spreadsheet formula languages can, in fact, have a significant effect on time to enter the formulae: languages that provide several methods to accomplish the same goal take longer to use because users have to make decisions on which method is appropriate. They recommend that users be trained to recognize the conditions under which one method is superior to the
alternatives so that decisions on which to use can be made more efficiently. [For further data on operator efficiency in spreadsheet formula entry, see Young and Maclean (1988) and Lane, Napier, Batsell and Naman (1993).] Brown and Gould (1987) studied the errors made by experienced users when creating three test spreadsheets, finding that 44% of the spreadsheets contained errors despite subjects' reports that they were "quite confident" that they were error free. The errors were manifested mostly as mistakes in formulae and they suggest that debugging tools to highlight formula structure would be helpful, as would a facility for displaying formulae from multiple cells simultaneously. Saariluoma and Sajaniemi (1989, 1991) studied the time to learn formulae, showing that subjects learned them faster when the "surface" appearance was congruent with the computational structure beneath, than when the two representations differed—the more they differed the longer it took to learn the underlying formulae. They recommend that spreadsheets should offer features that allow users to see how the surface structure maps onto the computational structure, thereby reducing the memory load associated with learning and remembering how the formula works.

2.3. STUDIES OF USERS AT WORK

Only a few studies have been published that examine how task-oriented users of spreadsheets actually use them to carry out work. None of those studies has compared usage patterns and "ecological niches" across the many areas where spreadsheets are used. In fact, we know little about what purposes they are used for, beyond the obvious applications in business—although we do know spreadsheets are widely used in scientific and engineering disciplines where the computations may be more complicated than those of business. (An on-line search of a database of the major journals in science generated hundreds of references for the keyword spreadsheet including: Spreadsheet inversion of the Laplace transform, Spreadsheet for Fourier-series, Spreadsheet solution of partial-differential equations.)

Sajaniemi and Pekkanen (1988) summarized the computational properties of a sample of 135 spreadsheets that were gathered from users in business and government in Finland. Among other summary statistics, they report that on average only 5% of the cells contained formulae: that a small number of different operators (average 4; range 0-10) and functions (average 2; range 0-8) were used; that IF, SUM and ROUND were by far the most popular functions; and that there was little nesting of functions within formulae. They conclude that most spreadsheets are computationally very simple; they did not consider the visual structure, ease of use or other aspects of the spreadsheets that may bear on complexity. Napier, Batsell, Lane and Guadagno (1992) unobtrusively collected command usage data from 40 experienced users of Lotus 1-2-3 who worked in a variety of large organizations. Analysis of almost half a million keystrokes showed that 27 commands (about 5% of those available in Lotus) accounted for 85% of the total commands issued. Commands for copying and moving data, and for changing the appearance of the sheet, accounted for about 45% of the commands issued.

Doyle (1990) outlines 10 common errors that novices make when learning Lotus. Most of the errors concern inconsistencies with the menu system, but the author does mention that novices experience difficulties learning about relative vs. absolute cell referencing, a feature of all spreadsheets.
Nardi and Miller (1991; also see Nardi & Miller, 1990) interviewed spreadsheet users at their places of work to uncover how spreadsheets are actually used. They observed a variety of cooperative activities. As an application development tool, it was found to be a superb communication bridge for proposing and refining program requirements among the stake holders. When verifying the correctness of spreadsheets, workers with quite different skills and levels of expertise were found to work together. Users very gradually learnt new techniques from other users and programmers, not from manuals. It seems that only when macros are required do users experience a big jump in the amount that must be learned [for supporting evidence see Myers et al. (1992), who compare the learning curves of several systems].

Nardi (1993) draws on this work to suggest that the ACEKit system, mentioned above, should be used in an organizational structure that explicitly distinguishes between the roles of workers who mainly use spreadsheets developed by others or who make minor modifications, workers who develop solutions for other people’s problems ("gardeners", in her terminology), and workers at the system level. This interesting approach keeps the end-users in control while giving professional programmers the infrastructure to reuse large software components in application-specific development.

2.4. WHERE DO WE STAND?

The present state of the research literature appears to have been dominated by bedazzlement. Even the most neutral of analysts (Lewis & Olson, 1987) present their work in terms of inquiry into the reasons for success. Yet there remain problematic areas where it is not clear that the spreadsheet genuinely is a success. We shall pick out three:

(i) How usable is the formula language? Differing opinions have been expressed in the literature, with Brown and Gould (1987) and Saariluoma and Sajaniemi (1989, 1991) suggesting difficulties and Nardi and Miller (1990, 1991) and Nardi (1993) claiming that the formula language is on the whole clear. But the extant literature has dealt with only very simple examples of formulae, such as simple summations (e.g. see Nardi, 1993: chapter 3). Do discretionary users develop more complex formulae, and if so, do they find them comprehensible?

(ii) There are also questions remaining about the construction of programs as entities, the comprehension of spreadsheet programs, and the like, on which no research has yet been published comparable to the series carried out on Pascal. There is now a large literature on the comprehension of standard assignment-based programming languages (reviewed by Détienne, 1990; Davies, 1993, and others), of which a standard finding is that programs are conceived not as individual statements but at a higher level of organization. Not surprisingly, therefore, programs are comprehended best when the whole text is available for inspection, rather than an individual line at a time (Robertson, Davis, Okabe & Fitz-Randolf, 1990). Yet that is exactly how a spreadsheet must normally be viewed, which may suggest that the users have problems that have not yet been reported in the literature.
(iii) As a last area where further research is indicated, we turn to the question of spreadsheets as interpersonal communication tools. The pioneering work of Nardi and Miller on this aspect of their use, leading them to propose the spreadsheet as the dominant model for information-based devices in an organization, needs to be replicated in other settings, so that we can have a firm basis for such far-reaching conclusions.

We decided to address only the first two areas. In particular, we wanted to test the accuracy of popular beliefs such as the claims advanced by Myers et al. (1992), and overall we wanted to discover whether there was a place for techniques designed to improve the comprehensibility of spreadsheets, such as the tool for providing a "cognitive mapping" of task-oriented structure described in Hendry and Green (1993).

3. Views from task-oriented users of spreadsheets

The methodology followed by Nardi and Miller is exemplary because it allows one to uncover how the spreadsheet is actually used by task-driven users. Here, we follow their methodology, but in our modest contribution concentrate on usage by individuals rather than the cooperative aspects of spreadsheets. This section summarizes 10 interviews with task-oriented users where opinions about creating, understanding and explaining spreadsheets were elicited. The informants discussed a variety of experiences where they encountered difficulties in creating or comprehending spreadsheets. Moreover, analyses of their explanation for how their spreadsheets worked revealed that several general types of information were conveyed.

3.1. ELICITATION METHODS AND INFORMANTS

The first author recruited 10 spreadsheet users by word of mouth and then interviewed them in their workplaces. The interview consisted of two parts. In the first part, informants were asked some general questions, similar to those used by Nardi and Miller (1991), on what was easy and hard about spreadsheeting, how they checked for errors, how they went about understanding a spreadsheet, and how they prepared easy-to-understand spreadsheets. In the second part, informants were asked to choose a spreadsheet and then explain it, pretending, as best as possible, that the interviewer was a colleague who needed to understand or reuse it. Tape recordings were made (with consent).

There were differences in discourse expectations between the interviewer and informant in the two parts of the interview. In the first part, control over what topics were addressed was shared equally between the interviewer and informant. Moreover, in the first part, the informants probably perceived the interviewer as an expert; whereas, in the second, the interviewer was most likely perceived as a novice, as he was not familiar with the domain. Consequently, the material collected in the first part of the interview concerns both a wide variety of experiences with
spreadsheets and more general experience with computer systems. In the second part, the material concerns the details of one particular spreadsheet. The analysis of the transcript reflects this difference: the material from the first part of the interview is used to illustrate some difficulties when comprehending and creating spreadsheets, the material from the second part is catalogued to summarize what sort of information users believe to be important when explaining their spreadsheets.

Data from only five of the informants will be discussed because this data covers what was learned from the other five. All informants were discretionary users: "professionals who do not work in the computer field itself but have integrated personal computers and personal computer software into their everyday work" (Santhanam & Wiedenbeck, 1993: p. 202). These informants, fictitiously named,† worked in a variety of settings:

- Bella is a part-time secretary who maintains the data integrity of a set of spreadsheets for financial reporting and occasionally reuses spreadsheets (implemented by a more experienced person) for slightly different presentations of data.
- Jude, Tess and Sue are on the staff at a university department of Management Economics and use spreadsheets to create complex models of global warming, in teaching and administration of student grades, and to analyse data collected from a computer simulation, respectively.
- Angel is a manager at a high-technology firm who models the software and hardware costs of a speech recognition product which is under development.

The spreadsheets they explained varied in size, complexity (e.g. length of formulae, type of referencing, dispersion of references, etc.), importance, and time to create: one was a single page, another was 14 pages long; one only contained functions for addition and averages, another employed formulae containing up to 15 cell references; some were basically scratch-pads, others were used in business decision making; some were prepared in a couple of hours, others were developed over several years. The users and spreadsheets reflect a fairly broad spectrum of usage.

3.2. DIFFICULTIES IN CREATION

In this section we concentrate on entering and copying formulae. Some familiarity with cell referencing techniques is assumed—see the appendix for a thumbnail account.

3.2.1. Solutions to simple problems can require much formula gamesmanship

The interviews revealed that creating cell formulae is "easy" only when the functions needed to complete a task are tailored to it. One informant, Bella, for whom this was the case, claimed that she had no trouble understanding most of the spreadsheets she worked with (they were created by her supervisor). Moreover,

† These names refer to the same informants as in Hendry and Green (1993): e.g. "Jude" refers to the same person in both papers.
that reusing them, by copying and altering, was usually straightforward. Bella's
description supports the expert opinion that was summarized in Section 2.1. It is
noteworthy, though, that the functions used in the spreadsheet formulae closely
matched the requirements of the problem domain: given a common set of data for
ten or so departments, compute descriptive statistics for each department. The
spreadsheet consisted exclusively of domain-relevant functions, such as SUM,
AVERAGE and ROUND.

Sue explained a task that, on the surface at least, appears equally uncomplicated
but, in fact, she was unable to solve it. The problem, slightly simplified here, was to
calculate the averages of blocks of numbers, contained in a list of 500 records. As
shown in Figure 1, each record consisted of two data fields and they were organized
into 10 blocks, where each block contained 50 records. Starting at row 1, the records
were imported so that the 10 blocks were found in rows 1–50, 51–100, ..., 451–500,
with data in columns A and B. Her goal was to compute the average of column B
for each of the 10 blocks, and list the averages in column C. She explained:

... say that [in cell F1] I want the averages of [the first block] then it's nice and easy
because all I have to do is [use the average function and create a range for it by selecting
the first 50 rows of column B]... I always think at this point, right, I've done it for one cell
I ought to be able to copy [that formula down 10 rows and get the averages of the other
blocks]... but of course if you do that... you wouldn't get what you wanted because [you
would get the average of rows 1–50 in C1, 2–51 in C2, 3–52, in C3, and so on]. And that's
the sort of thing I mean when I talk about the facilities to copy formulae going wrong. Or,
I've kind of got a problem that's too difficult for the standard tools that you are given. And
what I normally do at this point is to scratch my head and try and work out some way of
doing this quickly or neatly or elegantly. [Sue-1]
The following formula, written for Claris Resolve,† is one solution to this problem:

\[
\text{=Average(Range(MakeRange(}
\text{ColOf(B$1), RowOf(B$1)+H$5*(Row()-RowOf(B$1))},
\text{ColOf(B$1), RowOf(B$1)+H$5*(Row()-RowOf(B$1)+1)-1}))}
\]

It is assumed that the column of raw input data begins at cell B1; that the block size is a constant held in cell H5, which in this example contains 50; that the formula is copied downwards, in any column; and that the first row of the formula is aligned with the first row of the input data (in this example row 1).

The formula works by computing row and column offsets and using the MakeRange command to create a string representation of the range, then the string is cast to a range data type (with Range), and finally the average of the numbers in that range is computed. Computing the offsets with the ColOf, RowOf and Row functions requires even more gamesmanship. The ColOf and RowOf functions return the columns and row components of a reference respectively (e.g. "=ColOf(B$1)" returns 2; "=RowOf(B$1)" returns 1), and the Row function returns the row of the cell reference within which it is found (e.g. "=Row()" returns 11 when it is keyed into cell F11). Use of the ColOf and RowOf functions makes the formula fairly general, the only constraint being that the first row of input data and the beginning row of the formulae are aligned, but other less general solutions are, of course, possible.

Solving this seemingly simple problem is difficult because primitive system-oriented functions are required, including lookup functions, and the range building functions. These functions have no place in the user’s knowledge of how to solve the problem within the domain of work; they are solely sheet-oriented. So if the user tries to create a solution by solving the problem on paper and recording each step, these functions will not be part of the solution process because they do not solve problem tasks. They are unmappable into the domain. In the terminology of other writers, they solve “inherent” goals rather than “planning” goals, or they are said to achieve “enabling” tasks rather than goal tasks.

The distinction between “inherent” goals and “planning” goals has been made many times in many ways. Anderson, Farrell and Sauers (1984: p. 110), describing their simulation of a Lisp novice, write:

There are two types of goals for the purposes of composition [a form of learning]: \textit{inherent goals} and \textit{planning goals}. Inherent goals are intrinsic parts of the programming task. For current purposes inherent goals are all variants of writing code… On the other hand, planning goals produce results that are used to guide solution of the original problem but the results themselves are not part of the final solution.

May, Byerley, Denley, Hill, Adamson, Patterson and Hedman (1993)

† In Excel the OFFSET function is required for constructing the range, as in:

\[
\text{=AVERAGE(OFFSET(B$1,H$5*(ROW()-ROW(B$1)),0):}
\text{OFFSET(B$1,H$5*(ROW()-ROW(B$1)+1)-1,0))}
\]
distinguish similarly between “the tasks that an end-user wants to do with a design and the tasks that the design forces them to do.” They continue (p. 178):

From the user’s point of view any tasks that they must perform that involve telling the system what to do, configuring it, making choices about courses of action that it suggests, or even telling it that the goals have been achieved and that it can stop, are not goal tasks. They are tasks that must be carried out to enable the goal tasks to be performed, and it is because of this that we must make the distinction between goal tasks and enabling tasks.

Essentially, an enabling task is one whose goal is to get the device into a suitable state (an “enabling state”) to accomplish a goal task. They develop the concept of enabling states at some length, and state an important usability criterion: “A usable system reduces to a minimum the costs to the user in reaching the appropriate enabling states for their goal tasks.” (p. 183)

Current evidence indicates that planning goals present problems of both learnability and usability, so that it is not surprising that these tasks are difficult ones for the informants.

3.2.2. Cell referencing requires much care
There are several techniques for cell referencing, distinguished by how the cell references change when a formula is copied to a new location in the spreadsheet. Cell referencing techniques when combined with copying formulae give users many options for implementing computational structures. For example, the formulae “=A1 + F1” when copied over a block of cells, from the top-left to the bottom-right, adds corresponding cells in two matrices and shows the results in a third, which in an imperative programming language would require arrays and nested loops. Our informants revealed that a common source of error was using the incorrect referencing technique (e.g. most simply, for formulae that use a constant, an absolute reference is required).

To the question, “what do you find hard about spreadsheets compared to other systems you’ve used”, Jude responded:

... when you’ve got a very long spreadsheet equation which may be calculating stuff in a particular cell it’s not particularly intuitive... whether you’ve put in an absolute address where you should have put in a relative address or vice-versa, to actually impose the discipline on yourself to go and check all of that, I find quite hard. [Jude-1]

The most experienced spreadsheet user indicated that initially he had trouble with referencing techniques but now is vigilant to check the references very carefully. When the interviewer noted that he hadn’t mentioned any problems with cell referencing, Angel responded:

I think that’s because I’ve been doing it for so long that I tend to be very careful when I cut and paste or move functions. And therefore I tend to avoid I think a lot of the obvious ones, just by the way I actually manipulate it. Certainly, I remember when I started, yes there were a lot of problems there... but I think over the years I have just evolved a way of doing it where those kinds of problems don’t really crop up, as often as they used to. [Angel-1]

Now, if the spreadsheet is used primarily as a presentation device, then users may
never need to know about the complexities of cell referencing. That is, "relative" referencing, the default technique, will be sufficient. However, for even moderately complicated problems, like a simple forecasting model (see Section 3.3.2), various cell referencing techniques are required. Choosing which one to use requires careful consideration.

3.2.3. Handling errors requires planning

The majority of informants mentioned that they plan for mistakes and use specific techniques for spotting mistakes, such as systematic testing of formula chains; using carefully prepared test data; computing results by two different methods and then checking for consistency; breaking large, but conceptually coherent, formulae into smaller chunks so intermediate results can be easily inspected; and using consistent design practices so similar techniques for tracking down the causes of errors can be used when similar errors occur. Perhaps because of the difficulty in comprehending formulae (see next section), experienced users plan for debugging when creating a spreadsheet.

To the question, "In general what do you find easy about spreadsheets compared to other systems that you've used in modelling, and on the other hand what do you find hard?", Jude responded:

Well what's easy is that you can have something up and running in a day, program the stuff yourself and do the systems analysis yourself... I think the great disadvantage is that because it's so easy, instead of laboriously checking each stage that it is doing what you wanted to do before going on to the next bit, the temptation is to knock something up quickly and then it's quite difficult to trace errors and so on. I find that a rather more difficult problem. It's imposing a discipline on myself to make sure bits of the spreadsheet are actually doing what I think they're doing... [Jude-2]

Jude seems to believe a good test strategy is stage-by-stage component testing; nevertheless, he indicates a tendency to "grow computations" rapidly, but once they are large, testing becomes difficult. Following this theme, Tess alludes to the fact that errors are often compounded and these can be the most difficult to track down:

Other problems come from the compounding of errors in other parts of the spreadsheet so you make references to things which are wrong... You only notice it when it comes out in a critical calculation. [Tess-1]

Finally, Angel points out the importance of decomposing similar problems in similar ways and of making intermediate computational steps visible:

Alternatively, the other kind of errors you get are the ones that are glaringly obvious when they actually occur, suddenly the whole spreadsheet goes bang or whatever!... because I tend to design them all the same way, I just go and find the lowest occurrence of where [the error] occurred, and that's usually where the fault is. [Angel-2]

In sum, it seems that the informants often have trouble spotting mistakes and debugging formulae. Hence, they often employ design strategies that aid error checking and debugging. For further discussion of the typical strategies employed for debugging see Nardi and Miller (1991).
3.3. DIFFICULTIES IN COMPREHENSION

Comprehension, that is, understanding how a spreadsheet works, is a core task necessary for a variety of activities, including debugging, modifying, and reusing spreadsheets. In practice, how easy is it to learn from spreadsheets? The interviews revealed that the informants encountered a variety of difficulties when trying to recall how their own spreadsheets worked. Presumably, the problems would be magnified when trying to understand someone else's.

3.3.1. The computational steps in a spreadsheet are not always sufficiently visible

To the question "Do you ever have any trouble recalling how you structured a spreadsheet", some of the informants gave the impression that they did. Tess and Jude, for example, remarked that it is difficult to predict how changes to one part of the spreadsheet will affect other parts:

... knowing what relates to what bits and working out if you make changes in one part of it, how it relates to other parts of it. And depending on the spreadsheet you've got, if there are multiple spreadsheets linked into that, how those change as well. So I certainly recognise that problem in large spreadsheets of maintaining an overview of what's happening, particularly because you're operating through a small window onto it, and even if you can get several windows open at once, being able to manage what's on those different areas is difficult. [Tess-2]

Jude's view is similar but he goes further and describes his technique for helping to keep the "computation visible", yet despite his efforts, he seems to be less than satisfied:

I don't think it's that easy to see [spreadsheet structure]. I mean I try and block it all diagonally, or put block, calculation block, results, blocked diagram down the spreadsheet, so then you could see the name of the blocks and so on. Hopefully that helps, but even with that I find it quite hard to have the entire picture in my mind at one time. [I find it quite hard] in each bit of the spreadsheet to know where this particular thing is drawing its inputs from. Now what happens if I actually dropped out a few rows here, am I sure I haven't done something disastrous elsewhere?... So yeah, I mean I find it really quite hard to keep all that in mind. [Jude-2]

To the question "what are the main advantages and disadvantages of spreadsheets", Angel responded:

The disadvantage of using the spreadsheet is there's a lot of it when you print it out, getting at the workings...[and] the documentation of the workings of it, can be quite obscure sometimes... It's very difficult to actually document why something is being done in one particular way. Some of the spreadsheet packages that are around at the moment have features where you can attach notes to cells, but again it's very difficult to get at that information readily and easily. Whereas, in a computer program...the logic would be much more visible. [Angel-3]

These comments reveal that the informants often find the structure of the computations being carried out in a spreadsheet rather obscure—they are not as visible as one might expect. The informants confirm one of Nardi's (1993) conclusions: "It is difficult to get a global sense of the structure of an individual formula that may have dependencies spread out all over the spreadsheet" (p. 89).
3.3.2. Mapping cell and range references into problem domain interpretations is difficult

The results of copying a formula depend on a rather complex interaction between the direction of the copy-gesture and how the cells in the originating formula are referenced. When trying to understand a formula, mapping the referencing techniques into meanings in the domain can be difficult—spreadsheet regions must be probed and referenced cells must be tracked down and inspected. For example, Tess exposed one of her spreadsheet formulae and, in her words, "reverse engineered" it.

Tess mapped the range references in this formulae:

\[
\text{If}(\text{Sum}($B$11..$B$13) > \text{Sum}(N$10..N$11), \text{Sum}($B$11..$B$14) - \text{Sum}(N$10..N$11) + E15 + O$4, 0)
\]

into the following domain interpretations

- `Sum($B$11..$B$13)` the demand up to one week ahead
- `Sum(N$10..N$11)` the amount that has been ordered
- `Sum($B$11..$B$14)` the demand up to two weeks ahead
- `E15` the forecasted amount for two weeks ahead
- `O$4` the safety stock

Even after explaining several other similar formulae, understanding this one seemed quite hard for Tess: during her explanation she paused and backtracked several times, and used her fingers to keep track of cell references on the display as she probed the spreadsheet. Using named ranges in the formula would certainly have helped but in some spreadsheets mixed references (e.g. `$B$11..$B$13`) cannot be named, and for those systems that allow it, a new set of difficulties seems to arise. Moreover, using named ranges requires planning and some start-up effort is required before they can be used.

Part of the difficulty in Tess’s formula may also come from the presence of the conditional. In some circumstances, nested conditionals are needed, which are likely to add much greater complexity. Consider the problem of tallying the number of Xs, Ys, and Zs in the range of A1..E10. Here’s one solution. Copy this formula into the range G1..K10:

\[
=\text{If}(A1=’X’, 1, \text{If}(A1=’Y’, 100, \text{If}(A1=’Z’, 10000, 0)))
\]

and enter this formula into some cell:

\[
=\text{Sum}(G1..K10)
\]

What is computed? A number, such as 52213, which stands for the number of Zs, Ys, and Xs. After a night spent dreaming up a solution, one gardener made one discretionary user very happy when this code was provided. Partly because of the nested conditionals, this formula is neither easy to create nor easy to understand, yet the problem is trivial.
3.3.3. Comprehension requires much non-localized probing of cells and integration of information at the presentation and formula layers

A detailed understanding of the underlying formula tree is necessary when, for example, a spreadsheet is to be explained in sufficient detail for someone else to employ similar solution techniques, when extensive modifications are to be carried out, or when malfunctioning parts need repair. Understanding the structure of the formula tree is a detailed comprehension task—an outcome of it is knowing what the formulae mean, where they gather their data from, how they are copied, and so on. Understanding the formula tree contrasts to general comprehension tasks, like determining the gross computational structure of a spreadsheet or how headings and blocks of data relate to the problem domain. While all of the informants addressed these latter topics when explaining their sheets, only two gave detailed descriptions of the formula tree.

Here is an idealized description of how the informants explained the formula tree that illustrates the nature of the probing and information integration that is required. First, they chose a cell, often one at the top of a column or at the top-left of a block of cells, but not always because, at times, these “beginning” cells contain formulae that are special cases. Then, if the cell was part of a two-dimensional block of data, they often inspected neighbouring cells to see whether the formula was copied and, if so, how—that is important, if the implications of changing the formula are to be understood.

Next, they studied the formula. The operators and functions (e.g. IF, SUM, +, etc.) gave the informants no difficulty, but the cell references did because their meanings were often unknown (e.g. G13). When the meaning of a reference was not known, the informants either guessed it, or more typically, decided to search for it.

Tracking down the meaning of a cell reference requires two steps. If, for example, reference G13 were in the formula, they would first scroll the sheet so that the cell was visible; second, they would scan up the column and across the row to inspect the headings, thereby determining what the reference represented (e.g. product demand for week #11). If this referenced cell also contained a formula, often its cell references would also need to be tracked down. As the depth of this cell-to-cell recursion increases, it becomes more difficult to remember what information is being sought. Moreover, cognitive overheads, such as scrolling and cross-referring from cells to data headings, can exacerbate this goal management problem, especially for large spreadsheets. Of course, formulae usually consist of several references, which look much the same (e.g. H13 versus J13..J15), and if each has to be tracked down, it is easy to forget what each of them mean (we observed informants search for the meaning of a reference repeatedly).

Some of the informants took precautions during spreadsheet creation to minimize the difficulties of comprehending formula trees, while others did not. For example, in this quotation, Angel mentions that he uses the drawing features of Excel in order to improve the comprehensibility of his sheet, which was designed for easy modification:

... there's a number of features that help make the spreadsheet more understandable, especially if you are describing it to somebody else. I'll give you an example on this one here, where you can start using bolder boxes, and so you can start saying 'Right, all the bold boxes are input'. ... [Angel-4]
Jude reveals a policy which may be more common, especially when the spreadsheet is transitory or for personal use:

I don’t keep a lot of annotations about it… I don’t tend to write a user manual or anything like that. I just rely on the fact that when I get back in there and look at the equations again and with the titles to help me I will know what I’ve done [Jude-3].

Importantly, when modifying formulae the demands on working memory would be expected to be even higher than for formula comprehension. Scrolling around a large two-dimensional surface, scanning for headings so that arcane cell references can be mapped into domain interpretations, and remembering which subtasks have been done and which still need to be done, severely taxes working memory.

3.4. EXPLAINING A SPREADSHEET TO A COLLEAGUE: WHAT’S IMPORTANT?

Explaining what a spreadsheet does and how it does it is an activity that is closely allied to comprehending, creating, and especially to modifying spreadsheets, although none of the informants addressed this last topic. There are two questions: (1) how do users share their knowledge about the problem domain and know-how about spreadsheets, and (2) what information do they consider important about their spreadsheets. Nardi (1993) set out to address both questions and discovered programming communities of cooperating users, which were both complex and of great practical value to end-users. Indeed, she found these communities to be the norm and she argues that end-user programming environments must provide facilities tailored to the differential roles of users in such communities. It is a complex and colourful picture. We do not add to it; rather, our aim was simply to address the second question and characterize what the informants believe to be important about their spreadsheets. To tap into these beliefs, the informants were asked to choose a spreadsheet and to explain it to the interviewer as if he were a colleague who needed to understand it. The assertions were organized under four headings which summarized the type of information conveyed by the assertion.

3.4.1. Descriptions of the internal and external origins of data

Informants explained where data in the spreadsheet came from and where it was going. Sometimes, it was necessary to refer to an external source, as in these assertions:

- “... the organisers of the game give you an administration parameters sheet...”
- “... and here is some comparative concentrations because what I want to do with this spreadsheet is actually simulate somebody else’s results.”

At other times, the source of data was internal to the spreadsheet, as in:

- “... page 14 is... an accumulation of data... and the reason for putting it altogether in one place was then when we’re using the spreadsheet in what-if mode...”
- “... on page 6 we’ve got a whole load more figures which are worked out from figures on the previous page...”
- “This estimate is used by our sales forecasting mechanism.”
When one looks at the spreadsheet's surface, which is neatly organized into columns and rows with headings, it is often rather difficult to infer the origins of the data. Of course, it is possible to put extended comments into neighbouring cells but doing this often hinders the presentation of the data. Sometimes, the headings give clues, but often the data beneath a heading comes from a variety of places. To learn where the data comes from and where it is going, users often have to reveal the underlying formulae and examine many parts of the spreadsheet.

3.4.2. Problem domain descriptions of data
Of course, understanding a spreadsheet, like any piece of code, is much easier if you understand how it relates to the problem domain; in fact, good programmers spend considerable time switching back-and-forth between computational and problem-domain representations (Pennington, 1987). Consequently, it is not surprising, just as Nardi and Miller (1991) observed, that the informants often point to a part of a spreadsheet and explain its relevance in the problem domain. For example:

- "This is a set of data that we need, as basically correction factors which are going to be used in each year."
- "... page three, which is basically a complete breakdown of all the component prices, the sale prices, and what the price per line works out to."

Sometimes, a section of a spreadsheet triggered informants to make rather detailed comments about the problem domain. For example:

- "Right, so page 12 gets a bit more complicated in that it's on page 12 that the figures used to forecast our sales for the next period are included, and the way we've done that is just do a linear regression..."
- "So page 13 is just those actual and predicted figures as they work out, not always very close, but generally better than if we just stuck a finger into the air."

3.4.3. Problem domain explorations of data roles
In addition to pointing to a part of the spreadsheet and saying what it is, informants often explained how a part of the spreadsheet should be interpreted or how it should be used. These were somewhat more than just descriptions, they were explanations of why something was done, guidance for how something should be interpreted, or instructions for how something should be used. Here are some examples:

- "So a lot of these pages are really information pages... they were things that we would look at infrequently to help us make some specific decision..."
- "... we could look at them quite quickly and see how much variation there was between the different ways of calculating contribution...[because] there wasn't so much difference we... arrived at the same decision whichever way you worked it out"
- "So basically what I do in trying to use it is I go back to the input block, change some of the parameters in order to try and get this output simulation looking reasonable here"
- "... I've password protected it... That guarantees that I give a description of how it works before they start hacking..."
The impression that informants gave is that the spreadsheet served as an excellent trigger for discussing how data should be analysed, presented, and interpreted. However, it should be pointed out that very little of this information was actually recorded on the sheet.

3.4.4. *Computational domain explanations of the roles of spreadsheet regions*

In contrast to explaining the purpose of a region of a spreadsheet in terms of the problem domain, informants sometimes described a region in terms of its computational role. For example:

- "the next diagonal bit is the first bit of the calculation..."
- "coming down to the final results... so that's basically the output of the spreadsheet"
- "There is then a page which basically does a lot of lookups on that which calculates what options are technically feasible."
- "In general when I do spreadsheets I try to make them work from backwards, forward"
- "The reason it does the calculations there is that the formulae at the top would get so absolutely immense that they would be repeated on every line."
- "And I've shown the workings for that reason so you can kind of validate the workings..."

These type of explanations would appear to be most relevant in the cases when the spreadsheets were to be modified or if a similar spreadsheet needed to be created. However, notice that the explanations are at a rather general level—specific details of exactly what steps are involved in a computation were generally not given.

3.5. EXPERTISE AND IMPASSES

Despite their experience, the informants gave the impression that they were little more than novices when it came to solving unfamiliar problems in some areas. One example was Sue's difficulty in solving the sum-blocks problem (see Figure 1), another was the tally-items problem presented in Section 3.3.2. These are typical impasses, where a end-user is faced with a problem, but sees no way to attack it. Often they conclude that the problem is insoluble, and this can be very frustrating when the problem is as trivial as summing blocks or tallying items.

Why could the informants not find the required techniques? Why are experienced users unable to resolve fairly simple impasses?

There are two usual routes to resolving impasses. One route is to consult a mental model or some other abstract, high-level representation. This is the method of problem solution which characterizes expert performance in solving physics problems, leading to forwards reasoning from problem statement to solution based on high-level abstractions about problem types (Larkin, 1981).

The second route is to look for some command which apparently has something to do with the problem. In the case of Larkin's physics study, this would be equivalent to the behaviour of novices, who reasoned backwards from the solution by finding a
rule which generated one of the desired quantities, then finding a rule which
generated a term in the first rule, and so on until the terms of the problem statement
were reached. Similar activities have been found for novice and expert programmers
(Rist, 1986; Davies, 1993).

Unfortunately, both these routes appear blocked for users of spreadsheets. The
problem with the first route is that it is extremely difficult to form a coherent mental
model of spreadsheet operation. There are many, nearly isolated, niches of function,
including such things as domain-oriented functions which apply to data stored in
ranges (e.g. NetPresentValue), cell referencing techniques, functions for inspecting
and manipulating the grid (e.g. MakeRange, FormulaText, etc.), functions for
computing iterative solutions, and database functions. Each niche requires different
skills. Obviously there is some overlap, but the differences are greater than the
similarities, mostly because for each niche the applicable formulae have a different
set of environmental options for their control. None of the informants advanced
beyond the beginning niches, those of domain-oriented functions and cell referenc-
ing techniques, which is consistent with existing field studies (see Section 2.3).
Furthermore, even if the individual niches are mastered, learning about the
connections among them and mapping between niche and problem type is not
straightforward.

The problem with the second route, reasoning backwards from surface cues, is
that spreadsheet command names often give no cues which suggest how they can be
usefully applied to the problem at hand. Typically they are very explicit about
particular computations: e.g. Excel 4.0 contains commands called PEARSON
(Pearson product-moment correlation) and GAMMAINV (inverse gamma func-
tion). However, at the other extreme, many functions and commands which operate
on the sheet itself have bland, featureless names, and use a "magic number"
parameter to select one particular function from many others. To select the last
column used in a spreadsheet, the Excel command is GET.DOCUMENT(12), while
to select all cells which depend on a given cell the command is
SELECT.SPECIAL(10, 23, 2). These general-purpose commands have many
different unrelated effects (GET.DOCUMENT will do 68 different operations,
depending on its parameters, such as reporting the value of the check box for
Summary Columns in the Outline dialogue). There is no possible route to the
required command via the keywords in the indices of user guides and references.

In short, the informants appeared to be blocked from either using a mental model
to construct a solution, or reasoning backwards from the documentation. The
nearest parallel to their plight appears to be the problems encountered by
experienced Pascal programmers tackling their first problems in Ada and Icon (two
other programming languages, with various large differences from Pascal), as
reported by Scholtz and Wiedenbeck (1992). These programmers quickly sorted out
problems of syntax and semantics, but spent a lot of their time in coping with what
the authors called "implementation" issues.

Scholtz and Wiedenbeck point to two kinds of implementation problem. First is
forming appropriate plans in the new language: "Many of the programmers' difficulties arose in planning a solution that would take advantage of the strengths and orientation of the new language. This was especially true of Icon, which ... was very different from familiar languages like Pascal and C."
Second is finding out how to implement plans:

[Ada] subjects quickly understood that they had to put a reference to the appropriate input/output package into their program in order to be able to use input and output statements, but they had great difficulty determining the appropriate package to reference. Ada subjects also had trouble implementing perfectly good plans for case conversion because they could not find the Ada equivalents of the Pascal functions that they were familiar with. These equivalents existed, but their names and placements in the documentation were so different from what the programmers expected that they were often missed. (p. 214)

Spreadsheet users encounter both kinds of problem. Faced with an impasse, they often do not have a route to forming the appropriate plan; and when they have a plan, they often cannot find the appropriate function. When such a function is called GET.DOCUMENT(10), who can blame them?

4. Ease of use: towards a balanced view

4.1. EXPERT OPINIONS, USER EXPERIENCE

We can now compare what the HCI experts said with what our informants said. Roughly speaking, the experts seem to be good at seeing the many positive points of spreadsheets, while the users have more to say about the problems: the experts praise, while the users grumble. This is not to say that users do not like, or even love, their spreadsheets, only that users are more aware of limitations because they stress and tease spreadsheets in moments of urgency. Experts, though, can be more reflective—their goal has been to extract those qualities of the spreadsheet that have resulted in its enormous success and rework them in new designs.

Accurate summative analyses of existing interactive computing systems are extremely important because they allow one to see virtues to replicate and pitfalls to avoid when designing new applications. Myers et al. (1992) summarized a variety of exemplary systems, including the spreadsheet, for this very purpose. As is common practice, though by no means universal (contrast Nardi, 1993), the claims seem to be based on an informal comparison of traditional programming languages and an unreported amount of personal experience with spreadsheets. The paper comes from a committee of respected names, at a leading conference on HCI, who put considerable effort into the project. They distilled the spreadsheet’s design features into 12 positive claims which account for why users “love” the spreadsheet. Here is a sample of five of the claims (Myers et al., 1992: pp. 348–349):

“Levels of complexity. Using cell formulas is easy, but somewhat limited in their capabilities. Spreadsheet macros are considerably richer but also more difficult to use...”

“Smart’ assistance. Copying rows and columns automatically adjusts the formulas in the copied cells appropriately. This almost always gives users what they want.”

“Interactive. Spreadsheets are continuously evaluated, so that the computation is always up-to-date. This makes it easy to spot mistakes.”
“Visibility. The entire state of the computation is visible and available for manipulation. There is no hidden state.”

“Locality. The user’s focus is normally on a small region: a cell. . . . This is a simpler task than dealing with the program as a whole.”

Reading these claims, one could conclude that spreadsheet users rarely experience trouble, and undoubtedly this is true for some users, especially if Basic or Pascal is the standard for comparison; but much of the material presented in Section 3 contradicts these claims.

Levels of complexity. There are some problems which initially appear “natural” for spreadsheets but are, in fact, very difficult to solve because primitive non-task oriented functions are required (see Section 3.2.1).

Smart’ assistance. Deciding on the proper cell referencing technique can be difficult because there are many possible choices. Careful attention to which techniques are employed is necessary if a copied formula is to function the way it is intended. It appears that users often resort to trial-and-error to make a copied formula work, and that reverse-engineering is often needed later (see Section 3.2.2).

Interactive. Mistakes are often difficult to spot. Users often employ design techniques so that errors are easier to spot and easier to repair (see Section 3.2.3).

Visibility. Usually, the spreadsheet surface consists of neatly arranged columns, rows and blocks; but this presentation view can hide much of the underlying complexity. Headings, graphic annotation, and policies for spatial arrangement are often employed in an attempt to reveal underlying structure, but these are often inadequate (see Section 3.3.1).

Locality. Comparing separate computations is often necessary, and understanding how a formula works often requires the user to recursively track down the meaning of cell references; hence, much of the spreadsheet may need to be visited. The goal management can be cognitively complex (see Section 3.3.3).

To conclude, the informants did not find spreadsheets as easy to use as the prevailing view of the experts in HCI, and they would almost certainly consider the claims offered by Myers et al. (1992) to contain a substantial element of myth.

4.2. PROFILING THE SPREADSHEET’S QUALITIES

How can we get a summative view of the strengths and weaknesses of the spreadsheet? Certainly, given the tremendous success of the spreadsheet, there can be no doubts on the importance of converging on an accurate and balanced view—one which prevails throughout the large and amorphous HCI community.

Green (1991) developed “cognitive dimensions”, a set of broad-brush descriptors of information artifacts, intended to cover more of the activities and tasks normally
taking place than just initial creation. An important feature is that the vocabulary is designed to be applicable to any information artifact and is relatively well defined. It is thus possible to use the dimensions to reveal how the design features of an artifact trade-off against each other and how they compare with other systems, unlike the analyses offered either by our informants or by the experts we have cited. Green and Petre (in preparation) have applied this form of analysis to several types of end-user programming language, including spreadsheets, using armchair analysis to form their judgements. We shall present a thumbnail picture of the dimensions, and relate them briefly to the assertions of our informants. The resultant profile conveys more information than existing analyses.

The dimensions, actually a subset, follow (see Green and Petre, in preparation, for a more detailed treatment).

_Closeness of mapping_
The closeness of mapping is the "ease of mapping between the problem domain and the program domain", or "number of concepts in the programming world which are entirely intrinsic and have no domain counterparts". Typical programming languages require learners to become familiar with many concepts, some of which have no domain counterparts whatever. Some of the spreadsheets that the informants explained could obtain a very close mapping because mathematical functions that fitted the problem domain were used and because the conceptual features of the problem domain could be represented spatially on the two-dimensional grid. Other cases revealed much need for specialized "sheet" functions with no problem-domain counterpart (see examples in Section 3).

_Progressive evaluation_
It is a standard finding in many domains that novices need to evaluate their own problem-solving progress frequently, whereas experts can plan and execute relatively long sequences of action without feedback. This has been termed "progressive evaluation". One of the most notable features of a spreadsheet is that until the sheet becomes very extensive or very complex, it is possible to work with immediate evaluation. There is absolutely no test-edit-recompile cycle.

The informants revealed that when writing formulae, especially formulae with mixed cell referencing, they knew what effects they wanted but were unsure which cell referencing technique to use. The standard tactic seems to be to try a referencing technique, then immediately copy it to check that it is generating correct data. Progressive evaluation was crucial to resolving difficulties with the formula language. This observation is consistent with Payne, Squibb and Howes (1990), who showed that even experienced word-processor users were unsure of what their standard system did when told, for instance, to move the cursor forward by one word.

_Viscosity_
Some types of structure are easier to change than others; this difference we call viscosity. On the whole, spreadsheets do not suffer unduly from viscosity; the layout can be moved around quite fluidly, and the formulae can be rebuilt without excessive work. By this we mean that the root formula from which the others in a
block of cells are derived can be changed in small ways, then the formula can be
recopied over the block. Since the results are immediately displayed it is easy to
repeatedly change the formula until it works.

Hidden dependencies
Dependencies are badly hidden in spreadsheets. When cell A refers to cell B, and B
in turn refers to C, then:

(i) Although the formula for A mentions B, it has no built-in indication that B
mentions C.
(ii) There is no indication in either B or C that they refer to A.

Thus the view of dependencies in spreadsheets is local (next cell only) and is
one-way only. Tools exist to remedy these deficiencies, but they are rarely used,
according to our informants and those of Nardi and Miller (1991).

The impact of these deficiencies was well illustrated in Section 3.3.1, where the
informants spoke with feeling about the uncertainties of deleting cells, for example.

Visibility
Spreadsheet visibility is poor. Several informants illustrated the difficulties of poor
visibility, one saying that a textual program is more visible than a spreadsheet (see
Section 3.3.1). Two features of spreadsheets conspire to make the visibility of code
poor:

(i) The material is thoroughly dispersed—any information might be anywhere.
(ii) There is never a convenient way to compare several formulae amongst
themselves, while viewing their results. Normally, either a single formula
only—the currently selected cell—is visible and all data is visible, or
conversely, all formulae are visible but no data. Some spreadsheets provide
provisions for creating multiple views; for instance, a data view and a formula
view. Creating multiple views, however, usually requires much extra work:
selecting view options, resizing cell widths, positioning windows, laying out
note windows that have been attached to cells, and so on. None of these extra
operations are required when studying a textual programming language like
Basic.

To determine whether one variable depends on another is very simple, because the
environment contains an appropriate tool; “Select All Precedents”. But it should be
noted that discovering just how is not so easy; the user has to search a branching
dependency graph, in which each step is non-mnemonic. Another of Nardi and
Miller’s informants, Laura, spoke feelingly of spending “literally days going forward
and backward” sorting out dependencies.

Imposed guess-ahead
Some information devices force their users to make guesses before they want to.
“How much room do I need to leave?”; or, in the case of tightly-defined
programming languages, “I wonder what identifiers I need to declare?” Spread-
sheets score fairly well here, at least for provisional or exploratory work. Our
informant Jude used a clever strategy to minimize guess-ahead, described in detail in Hendry and Green (1993). He blocked data and computations diagonally from top-left to bottom-right, so that each block was independent of others in both rows and columns. Although this created large areas of white space, leaving 55% of the sheet blank, it meant that columns and rows could be formatted for one data purpose independently of all others. Other strategies, such as working steadily downwards, create complications which Jude finessed.

Guess-ahead may become a problem when high quality paper presentation is required, as was the case for half the informants. Discovering a suitable format may need a lot of trial and error. Fortunately, spreadsheets are a fluid, forgiving medium, so that users can cope with guess-ahead by successive approximation.

Secondary notation

Many programming languages allow extra information to be carried by other means than their formal syntax: indenting, choice of naming conventions, choice of programming construct, and grouping of related statements. These techniques have no place in the formal semantics of the algorithm, but they all convey meaning to the human reader, just as it is possible for one to recognize the contents of a document merely from a typographer’s sketch of it.

Spreadsheets carry secondary notation primarily by choice of layout. This can be very effective as an aid to comprehension: Saariluoma and Sajaniemi (1989, 1991) showed that comprehension of “simple” trees was four times faster than comprehension of “embedded” trees (see Figure 2). The so-called simple trees used layout as a form of secondary notation.

Using secondary notation comes with a cost. Layout B is much more compact than layout A. More importantly, some kinds of comparisons are easier in Layout B than in Layout A. About half the informants created spreadsheets which summarized data across a dimension (e.g. university departments) and for them the use of secondary notation to reveal the formula structure was impracticable. One

**Figure 2.** Costs of using secondary notation. In Layout A secondary notation is used to improve the comprehension of a formula tree, consisting of 7 nodes. The formula tree is C4 = B2 + B6, B2 = A1 + A3, B6 = A5 + A7. In Layout B three formulae of the exact same structure, but embedded within columns, fit into the same display space. In Layout B, comprehension of the formula tree is more difficult, but comparative analysis of data across the horizontal dimension is much easier.
method of compensation was the use of typographical features, including row and
column headings and borders of different line thickness and shading—other forms of
secondary notation.

**Abstraction level**
Classical spreadsheets have virtually no abstraction facilities. Conventional
programming languages have far more. Even humble drawing programs have more:
they usually have a grouping operator to allow disparate objects to be treated as one
(and notice that the groups can be nested hierarchically), and often they have
separate layers which are individually viewable, shape libraries, etc. It is true that
spreadsheets have named ranges and some even have outlining controls for
selectively folding rows and columns. These abstractions are important in some
niches, but they have nothing to do with the outstanding success of the spreadsheet.
In fact, the success of the spreadsheet is, in part, because there are few abstractions.

**Summary**
We characterize spreadsheets as having very low requirements for looking ahead
(low "premature commitment" or "imposed guess-ahead"); virtually no user-
defined abstractions; ability to re-arrange material swiftly ("low viscosity"), but at
the price of many "hidden dependencies", and poor visibility (not easy to view
formulae and results at the same time, for instance), leading to low "role-
expressiveness" (i.e. not easy to understand the purpose of any particular com-
ponent with respect to the whole computation).

These conclusions, drawn from the assertions made by the informants, confirm the
conclusions reached by Green and Petre (in preparation) on the "cognitive
dimensions" of spreadsheets. This concordance is evidence for the robustness of the
cognitive dimensions analysis technique, and that the profile of spreadsheet qualities
is accurate.

5. **Situated information use: what spreadsheets tell us**

The view that emerges from the previous section could be summarized in brief as,
spreadsheets solve immediate problems very quickly but store up future trouble in
debugging and comprehending. If you consider the spreadsheet as one component of
an information system, or of an activity involving information, it would seem that
their place is in systems where a particular style of working is encouraged.

Three different types of information systems can be clearly distinguished, forming
extreme points in a space of possibilities (in which most real systems are not located
at an extreme, of course). First, there are incremental growth systems. A totally
unorganized collection of books or tapes, such as most of us own, is perhaps the
paradigm example: very easy to add to, but relying exclusively on our own memory
to tell us what we own and where it is. The humble (or sophisticated?) pencil and
paper, as used by typographical designers (Black, 1990), is another fine example.
These systems are suitable for early stages of design, in which the emphasis must be on capturing ideas fluently. The spreadsheet has much in common with this category. Second, there are transcription systems. These systems are ones where one information structure is turned into another. Music software to transcribe from keyboard direct to a data file is one kind of example. Some programming languages have been designed to meet very stringent requirements in the transcription of specifications into code with extreme safety and reliability, with built-in mechanisms to reduce the likelihood of careless mistakes going unobserved. Transcription systems do not always make it easy to add second thoughts (and for safety-critical systems, that might well be a virtue). Finally, there are presentation systems, such as library catalogues and engineering drawings; these systems are intended to produce clear and unambiguous representations, but they may well require training and patience from their users, since their primary goal is extreme accuracy—the speed of learning, ease of use, etc., must take second place. Frequently they provide multiple views (such as the different viewpoints used in mechanical and engineering drawing, and the different types of index and contents list in libraries), and version control may be very tight.

Obviously, these are extreme positions. The unlisted collection of books may be supplemented by a list of books owned, without going to the trouble of setting up a full library system. But even if all the owner does is to write each new title on a list as the book is acquired, this little step takes extra work. Effort is always required to turn an incremental system into a more presentational one.

Sometimes an individual “information object” will pass through all these three types of activity (incremental growth, transcription and presentation) during its lifetime. Early ideas are loosely explored, and a specification emerges; the specification is transcribed into various rigorous notations, including program code; and the resulting object is presented to clients for acceptance. (Obviously, there are also cases where one activity is paramount.) From our informants, and also those of Nardi and Miller, spreadsheets are frequently used for incremental growth and for presentation, and we can safely assume that in many cases the same information proceeds from one to the other.

_Spreadsheets as incremental devices_

To see how the characteristics of spreadsheets support incremental or exploratory working, we can suppose that the user does not have training in program design, nor any external aids on which to sketch ideas. First ideas will therefore need to be recorded in some external form quickly so that they can be considered and if necessary modified. The spreadsheet enables that instant gratification, because it does not require abstractions to be created (no type definitions or class definitions), nor any other kind of enabling task; a certain risk of slips and mistakes is accepted, in return for quick entry of ideas. The user, normally closely engaged in the task, relies on working memory to record the role of each formula, i.e. what it does in the overall computation. The hidden dependencies are not a problem, at least during the stage of close interaction with the sheet, because the user can recall what each cell does.

During early stages of design of both software and hardware, it is characteristic that growth is not purely additive; false starts have to be altered, and new ideas
have to be explored. Typographers do the same, and indeed they have even evolved conventions to indicate when ideas are still provisional and likely to be changed. There is no reason to suppose that "idea sketching" is any different with spreadsheets. It would not be going too far to say that low viscosity is an essential ingredient of incremental devices being used in this manner. The low viscosity of the spreadsheet makes this tinkering a simple process.

In short, spreadsheets support incremental work well, by allowing quick gratification of first ideas and also making second thoughts easy: "quick fixes with little look-ahead", as Green and Petre (in preparation) put it.

**Spreadsheets as transcription devices**

Spreadsheets are not good transcription devices for most purposes. There are no built-in redundancies to help make sure that each step is correct, and we have shown how the hidden dependencies create difficulties in debugging. Because transcription is by definition a situation where the target is relatively well-defined, there is no great call for the spreadsheet's virtues of low viscosity and instant gratification; instead, transcription is a situation where abstractions are likely to be useful and visibility is essential.

**Spreadsheets as presentation devices**

Spreadsheets have considerable use as presentation devices, but although Nardi and Miller (1991) report that their informants thought they were excellent, we do not know what the competition was like. Were spreadsheets really excellent, or were they just a lot better than pencil and paper? Or was it maybe that they were used for presentation because the information had already been entered during the exploratory phase, and leaving it in spreadsheet form seemed easiest?

Whatever the situation, our own assessment is that spreadsheets are fair but not outstanding presentational devices. Their lack of abstractions, their poor visibility and their hidden dependencies tell severely against them. It is not easy to understand the purpose of any single component with respect to the computation as a whole. On the other hand, it is quick and easy to alter the layout and to apply typographical sugar so that the "visual rhetoric" is much better than the output of most programs and analysis tools. This gives a limited but extremely useful form of secondary notation.

### 5.1. Improving an Information Device

The limitations of spreadsheets restrict their usage. Of our three types of information device, they are best fitted for use as incremental devices; yet they are widely used as presentation devices. The profile of cognitive dimensions suggests how improvements might be made. It is the lack of abstractions to be used as secondary notation that most powerfully limits spreadsheets as presentation devices; one way to overcome that difficulty is to force users to create abstractions, but if abstractions are imposed as a necessity, the extra work would in turn limit the spreadsheet's usefulness as an incremental device. So the problem is how to allow
Figure 3. Screen dump of CogMap (Hendry & Green, 1993). On the left is the CogMap window for creating "tags" by keying in a descriptor and selecting a display colour; for assigning explanations for what the tags denote; for adding tags to the sheet and for removing them; and for finding ranges that contain certain tags. On the right is an abridged version of Tess’s spreadsheet which has been annotated with CogMap; the tag assigned to column D and its explanation are revealed to the right. The formula she reverse-engineered is shown in the entry bar (see Section 3.3.2). This formula, which is copied down from cell D8 to D17, computes how much stock should be ordered for a given week. Cell D7 contains the initialization formula:

\[ \text{Forecasting} \]

\[
D8 = \text{HI}((\text{Sum}(\$B8:$B9) + \text{Sum}(D7:D7) + \text{Sum}(B8:B10) - \text{Sum}(D7:D7) + C10 + C3:3))
\]

Four tags have been applied to the sheet to denote the roles played by the regions. These are displayed in colour in the live version.
Tagged Regions:

- amountToOrder: The stock that should be ordered for a week.
- demand: The stock actually required by customers for a week.
- forecast: The stock that is expected to be required by customers for a week.
- safetyStock: Extra stock.

Formula:

\[
\text{amountToOrder}\{1\} = \text{sum(demand}\{1..2\}\} + \text{forecast}\{3\} + \text{safetyStock}
\]

\[
\text{for } k = 2 \text{ to } 20 \\
\quad \text{if } \text{sum(demand}\{1..k\}\} > \text{sum(amountToOrder}\{1..k-1\}\) then \\
\quad \quad \text{amountToOrder}\{k\} = \text{sum(demand}\{1..k+1\}\} - \text{sum(amountToOrder}\{1..k-1\}\}) + \text{forecast}\{k+2\} + \text{safetyStock} \\
\quad \text{else} \\
\quad \quad \text{amountToOrder}\{k\} = 0 \\
\text{end if} \\
\text{end for}
\]

Figure 4. Structured view of the formulae in the range that have been tagged as amountToOrder. First, there is a declaration section, then representations of the formulae. The first assignment is a translation of the initialization formula given in Figure 3. The formulae found in cells D8 to D17 have been parsed and cell ranges have been replaced by names of tagged regions; then the copying gesture has been replaced by a FOR-loop.

useful abstractions without disturbing the qualities of instant gratification. The major cognitive limitation is the loss of the relationship between spreadsheet locality and domain reference. We have devised a technique to provide a “cognitive mapping” between areas on the spreadsheet, and the user’s individual view of the problem domain. This tool, “CogMap”, makes use of a “description level” in which the user can categorize regions, according to any criteria desired, and can mark links between them (Hendry & Green, 1993). The approach was first devised for improving the comprehensibility of object-oriented systems (Green et al., 1992), but has also proved effective for spreadsheets (Figure 3).

The approach can be taken a step further, to make the formulae themselves more comprehensible. A possible extension is shown in Figure 4, illustrating how a formula could be translated into a structured, Pascal-like view. The techniques described would require only simple parsing—no deep understanding of the structure would be needed—and would give the user a complementary, procedural representation of the computations.

The CogMap approach preserves the strengths of spreadsheets—progressive evaluation, no guess ahead, low viscosity—while remedying two of its weaknesses—poor visibility and hidden dependencies. We claim that CogMap is effective because:

Simplicity. CogMap was designed to be very simple; hence, the operations for creating, applying and managing tags are rudimentary. Users will record what they know about their sheets only if it is extremely easy to do so. Despite its simplicity, most of what the informants said about their sheets could be represented in CogMap (Hendry & Green, 1993).

No imposed guess-ahead. Creating a spreadsheet entails few start-up costs and, if
one chooses, little planning. CogMap is the same: creating tags and applying
instances of them to the sheet requires no planning or complicated start-up
procedure. CogMap can be used to annotate spreadsheets while they are being
created or after they are finished and, like most other features of spreadsheets, it
can be ignored altogether.

Closer mapping. Tagged spreadsheet regions are shown in colour and can be
inspected, if necessary, to find out what the tag means if the meaning of a colour is
not known. Tags can be searched for and filtered views can be created by
changing the colours of tags. These three simple operations—inspecting, finding
and filtering—are sufficient to highlight the roles and the meanings of parts of the
spreadsheet.

Improved visibility. CogMap improves the comprehensibility of the formula
language by presenting structured views of formulae using terminology of the
problem domain. The structured view brings together disparate information,
thereby eliminating much of the cell probing, scrolling and cross-referring
between cell references and headings that is normally required when compre-
prehending formulae.

6. Conclusions

We offer the following conclusions.

1. Even for simple problems spreadsheet formulae are not always easy to create
   or understand.
2. It is not enough to listen only to users or to HCI experts. At least in the
   present case, their different backgrounds and experience with spreadsheets
   have made different aspects salient.
3. The cognitive dimensions framework offers a balanced view.
4. The profile from the cognitive dimensions framework can give some useful
   indication of what type of activity the device is best suited for—incremental,
   transcription or presentation.
5. CogMap was presented, not so much to show "how to build a better
   spreadsheet" but more as an example of how this analysis can give design
   guidance on making the device more suitable for one type of activity without
   sacrificing its effectiveness for another type.
6. Finally, lest there be any doubts: there is still much to learn about how people
   use spreadsheets, and spreadsheet ecology.

We wish to thank the informants who took the time to participate in this research, Blair
Nonnecke for comments which improved the quality of this paper and Wai On Lee and Kim
Graham for bringing their spreadsheet problems to our attention. Bonnie Nardi's comments
on an earlier draft substantially improved the clarity of the presentation and alerted us to
some oversights. Any remaining errors or omissions are, of course, our responsibility.

Trademark owners or licensees: "Claris" and "Resolve", Claris Corporation; "Wingz",
Informix Software Ltd; "Excel", Microsoft Corporation; "Lotus 1-2-3", Lotus Development
Corporation.
References


*Paper accepted for publication by Editor Professor B. R. Gaines*

### Appendix

Some degree of familiarity with absolute and relative referencing techniques is assumed. A typical way to place the same (or similar) formula into a range of cells is to key in the formula once and copy it over a block of cells. The operands in the formula are references to cells containing data and, when the formula is copied, these references change depending on which of the following three referencing techniques are used: *relative* (G5), *absolute* ($H$5), or *mixed* (e.g. J$15 or $J15).
With relative references, the default technique, both the column and row components change by the distance copied (e.g. "=G5 + 2" becomes "=J9 + 2" when copied 4 cells down and 3 to the right). With absolute references, a method for referring to "constant" data such as the tax rate or the value of gravity, neither the column nor the row components change (e.g. "=$G$5 + 2" is always "=$G$5 + 2" no matter where it is copied). With mixed references, a method for creating more complicated structures, either the row or column component is fixed and the other changes by the distance copied (e.g. "=$G5 + 2" becomes "=$G9 + 2" when copied 4 cells down and 3 to the right). For details on this topic consult any spreadsheet user's guide.