Towards Safer Spreadsheets*

Robin Abraham
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, Oregon 97331, USA.

1 Introduction

Professional programmers are well aware that debugging, testing, code inspection, etc. are part and parcel of software development. Requiring end users to carry out the same activities to reduce spreadsheet errors might be asking too much. For one thing, they lack the expertise and for another, they might not be willing to invest the time and effort required by these activities. For example, testing is a standard and effective technique for detecting faults in programs. The downside is that testing requires reasonable domain knowledge (to come up with effective test cases at the very least) and understanding of the program. End users might be deficient in one or both areas. Another problem arises from the lack of tool support for running test suites in currently available commercial spreadsheet systems. This forces users to run one test at a time, thereby taking up more time.

2 Program Generator for Spreadsheets

We have developed a system that allows the user to create *specifications* that describe the structure of the initial spreadsheet [2]. The system (named Gencel) translates the specification into the initial spreadsheet instance and also generates customized update operations (insert/delete operations for rows and columns) for the given specification. This approach guarantees that a spreadsheet instance generated by application of any sequence of the update operations to the initial spreadsheet instance conforms to the user-defined specification. Moreover, given that the initial specification is type-correct, any spreadsheet instance generated by the application of the customized update operations is guaranteed to be free from omission, reference, or type errors.

One concern that might arise about this approach is that it could detract from the flexibility offered by spreadsheet systems because of the constraints imposed on the update operations. We believe that the advantages will outweigh the intial investment (in training and creation of the initial specifications) because of the huge savings in debugging and testing effort—the user only needs to audit the initial specification and the data values entered in the generated instances.

To support the wide-spread adoption of the Gencel system, we need tools that allow the user to extract the specifications from arbitrary spreadsheets. We plan to use some of the spatial analyses techniques developed in [1] to help with this task.

3 Conclusion

Most of the current approaches are aimed at helping end users detect errors in spreadsheets they have already created. Programming language environments used in commercial software development employ simple (syntax highlighting, auto completion) to sophisticated (type checkers, program generators) techniques to prevent the incidence of errors in programs. This makes a strong case in favor of systems that help the users create correct spreadsheets. In this context, we believe that the Gencel approach is a big step towards the prevention of errors in spreadsheets.

References

[1] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets Through Spatial Analyses. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, 2004.

[2] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Gencel — A Program Generator for Correct Spreadsheets. Technical Report TR04-60-11, School of EECS, Oregon State University, 2004.

*This work is supported by the National Science Foundation under the grant ITR-0325273 and by the EUSES Consortium (`http://eecs.oregonstate.edu/EUSES/`).

The Spreadsheet as a User Interface

Alan Blackwell

The first spreadsheet was a success because it provided a new user experience, not because of any contemporary theories in human computer interaction (or in computing, despite a prior patent of which Bricklin and Frankston were unaware – Mattessich 1964). It is useful to consider how and why VisiCalc was different to the research agenda in HCI at the time, how its success was subsequently rationalised, and what advances have been made in the usability of spreadsheets since then.

Most HCI research in the late 1970s was still continuing directions originally motivated by military research through Licklider, Sutherland and Taylor's directorships of the ARPA Information Processing Techniques Office. New interaction paradigms were clearly derived either from SAGE command and control scenarios, or from CAD for aerospace modelling and machine tooling. Engelbart adopted the more humanistic vision of Bush's Memex, but developed it in the military research atmosphere of ARPANET. In contrast, Kay's radical pursuit of an educational agenda from Papert emphasised creative exploration rather than conventional literacy and numeracy. None of these strands of research gave close consideration to business computing needs, beyond the anodyne commercialisation of NLS WYSIWYG documentation facilities into word processors.

VisiCalc was developed in isolation from this research environment in which the concepts of "direct manipulation" and "metaphor" were gaining currency. Bricklin's original concept was prototyped in BASIC (restricted to five columns by 20 rows), with the vision of creating an "electronic blackboard". It was targeted at the platform that was closest to a "commodity" PC – the Apple II – on the advice of a founding editor from Byte magazine, and Franskston optimised it to run fast in only 20K of memory.

Later developers did little to change this basic paradigm, but adopted more features that had become familiar from work at PARC and Apple. Lotus 1-2-3 added presentation features such as charts and plots, as well as database capabilities, after Kapor had previously developed similar extensions for VisiCalc. Excel was written by Microsoft for the 512K Macintosh in 1984. It provided pull-down menus and mouse pointing (and was the flagship application for Windows 3.0).

HCI rhetoric of the 1980's claimed that spreadsheets were "natural" because of the adoption of a well-chosen metaphor, as with other "desktop" features. In fact, the metaphor was minimal. More mundane features were the fact that few other applications organised data properly into columns, and most calculators kept no record of previous calculations – both features that are essential in book-keeping. Subsequent spreadsheets were successful to the extent that they preserved these essential features. My talk will consider how innovations in the spreadsheet paradigm can be designed and assessed in the light of these critical attributes.

Levy, S. (1994). Insanely great: The life and times of Macintosh, the Computer that changed everything. London: Penguin

Mattessich, R. (1964). Simulation of the firm through a budget computer program. Irwin.

Power, D. J., "A Brief History of Spreadsheets", DSSResources.COM, World Wide Web, http://dssresources.com/history/sshistory.html, version 3.6, 08/30/2004.

Rheingold, H. (2000). Tools for thought: The history and future of mind-expanding technology (2nd ed). Cambridge, MA: MIT Press

Smith, D.K. and Alexander, R.C. (1988). Fumbling the future: How Xerox invented, then ignored, the first personal computer. New York: William Morrow

A Spreadsheet-Based View of the End-User Software Engineering Concept[1]

Margaret Burnett, Curtis Cook and Gregg Rothermel
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331 USA
{burnett, cook, grother}@eecs.orst.edu

End-user programming is arguably the most common form of programming in use today, but there has been little investigation into the dependability of the programs end users create. Instead, most environments for end-user programming support *only* programming. Giving end-user programmers ways to easily create their own programs is important, but it is not enough. Like their counterparts in the world of professional software development, end-user programmers need support for other aspects of the software lifecycle.

We have been investigating ways to address this problem by developing a software engineering paradigm viable for end-user programming, an approach we call *end-user software engineering*. Because end users are different from professional programmers in background, motivation and interest, the end user community cannot be served by simply repackaging techniques and tools developed for professional software engineers. For this reason, end-user software engineering does not mimic the traditional approaches of segregated support for each element of the software engineering life cycle, nor does it ask the user to think in those terms. Instead, it employs a feedback loop supported by highly incremental testing, fault localization heuristics, and deductive reasoning, which collaborate to help monitor dependability as the end user's program evolves. This approach helps guard against the introduction of faults in the user's program and, if faults have already been introduced, helps the user detect and locate them.

We have prototyped our approach in the spreadsheet paradigm. Our prototypes employ the following end-user software engineering devices:

- Interactive, incremental systematic testing facilities.
- Interactive, incremental fault localization facilities.
- Interactive, collaborative assertion generation and propagation facilities.
- Motivational devices that gently attempt to interest end users in appropriate software engineering behaviors at suitable moments.

We have conducted more than a dozen empirical studies related to this research, and the results have been very encouraging. (More details about the studies are at http://www.engr.oregonstate.edu/~burnett/ITR2000/empirical.html.) Directly supporting these users in software development activities beyond the programming stage—while at the same time taking their differences in background, motivation, and interests into account—is the essence of the end-user software engineering vision.

For further reference:

M. Burnett, C. Cook, and G. Rothermel, "End-User Software Engineering," *Communications of the ACM*, September 2004.

[1] This work has been supported in part by NSF under ITR-0082265 and in part by the EUSES Consortium via NSF's ITR-0325273.

-

IEEE FOS (Foundations of Spreadsheets) Workshop

Rome September 30 2004

Position Statement

Pat Cleary

Researchers need a shift in perspective. End User Development (EUD), and in particular the use of spreadsheets, is essentially an organisational issue, not a technical one. We must understand organisations and how they behave. Organisations consist of people interacting within some sort of structure. People are complex, far more complex than we as technologists understand, and people in organisations are even more complex. As such, the solutions to EUD problems are organisational not technical. We need to understand the context in which EUD takes place. Why do users choose to model a business process using a spreadsheet rather than some alternative vehicle? If a decision-maker is forced through organisational policy to adopt an alternative, is there likely to be a loss of motivation? The answers are likely to lie in the domain of psychology rather than computing. Ray Panko (Panko 2003) has been encouraging us to look outside our own disciplines to seek understanding and knowledge to help our research. At UWIC, we are embarking on a programme of research aimed at understanding spreadsheet use and then attempting to provide a framework for risk reduction:

- Categorise spreadsheet use within an organisation according to some agreed criteria e.g. an estimate of financial risk; complexity; number of potential users; motivation of the modeller;
- For each category, formulate a strategy for risk reduction; this may vary from do nothing (continue as before) to do not use spreadsheets for this category. Between these two extremes of the continuum, a variety of strategies may use the variety of tools and techniques already available or may demand new tools to be developed.
- Implement the strategies and monitor the effect.

A number of issues need to be understood and resolved at this initial stage, e.g. how do you measure spreadsheet use? In particular, how do you measure motivation/de-motivation? Clearly, without a suitable metric(s), it is not going to be possible to recognise success and failure.

Reference:

Panko, R. R., ‘Reducing Overconfidence in Spreadsheet Development’, *Proceedings of EuSpRIG Conference*, Dublin, 2003

Foundations of Spreadsheets

Rome 2004

Position Statement by Grenville Croll

Background

By way of introduction, I should mention that as a young software engineer, I was responsible for re-engineering Lotus 1-2-3 for the European marketplace, way back in 1984. I had the good fortune to meet and work with Mitch Kapor, Jonathan Sachs and the software engineers who took Lotus 1-2-3 through its early versions. Subsequently, for an unbroken period of nearly fifteen years I ran a couple of small UK companies (4-5-6 World and Eastern Software Publishing), developing and marketing Lotus and Excel add-ins and related training. The products provided anything from basic functionality – graphics, function libraries and printer drivers – through to more advanced technologies including Monte Carlo Simulation, Neural Networks and Linear Programming. My present employer, Frontline Systems, was founded and is managed by Dan H. Fylstra who previously founded Personal Computer Software, later renamed VisiCorp, publishers of VisiCalc, the first mass market spreadsheet. Frontline Systems presently supply a diverse range of optimisation software products for Microsoft Excel. For the last five years I have been closely involved with the European Spreadsheet Risks Interest Group (EuSpRIG). At the Amsterdam conference in 2001, I gave a presentation on the work of Mattesich, the originator of the first electronic spreadsheet.

Frame Questions

HCI perspective. Given the 25 year history of spreadsheets, we can look forward with considerable certainty to at least another 25 years of their business use in essentially their present form. With this in mind, what set of five and ten year objectives might it be reasonable to aspire to in order to positively influence the work and leisure lives of over 100 million spreadsheet users over a period of this length?

Business Perspectives. We know almost nothing about how spreadsheets are used in business, beyond our own experience – what are the uses of spreadsheets*?* We assume that we make better business decisions using spreadsheets, but to what extent is this actually true? Can we identify areas where spreadsheets should not be used and create or recommend a replacement? Are there new areas where spreadsheets could be effectively deployed?

Programming perspective. An enduring theme through the life of spreadsheets has been the desire to create and manipulate them programmatically, to compile them having been written manually, then to decompile them automatically to assist in debugging. Can we conceive of and implement a simple to use, integrated architecture that can achieve all this?

Quality Perspective. How can we continue to improve the educational process relating to spreadsheets - from their active use in primary education through their role in the teaching of quantum chemistry.

FOS'04 Workshop — Position Statement

Martin Erwig, Oregon State University

We believe the two most promising ways to improve reliability of spreadsheets are the development of:

- Automatic tools for error detection
- Tools for automatically generating correct spreadsheets from specifications

The focus should be on *automatic tools*, because anything that has to be done "manually" in addition to creating a spreadsheet takes time, which end users are reluctant to spend.

One promising approach to automatic error-detection tools is to define type systems that exploit the labels and spatial structure of spreadsheets [6, 4, 2, 3, 1].

However, an even greater potential lies in the development of new programming approaches for spreadsheets. Existing spreadsheet systems work with a simple programming model of a flat collection of cells that do not contain any structure other than their arrangement on a grid. In particular, cells are identified by global row and column numbers (letters) so that references have to be expressed using these global addresses. This lack of structure puts current spreadsheet systems into the category of assembly languages when compared to the state of the art in other programming languages. This situation is peculiar because spreadsheet systems are equipped with very sophisticated user interfaces offering many fancy features, which can distract from their intrinsic language limitations. The rigid, global addressing scheme makes computations vulnerable to changes in the structure of the spreadsheet—much like in the old days of assembly language programming where the introduction of a new item into the memory could cause some references to become invalid.

Instead of revealing this low-level memory structure to the user, we believe that spreadsheets should be built using higher-level abstractions, such as, *tables*, *headers*, and *repeating blocks*. Correspondingly, instead of creating spreadsheets through arbitrary, uncontrolled cell manipulations, spreadsheets should be allowed to evolve only according to specification that describes the principal structure of the initial spreadsheet and all of its future versions. Such a specification defines a schema or template for a spreadsheet that allows only those update operations that keep changed spreadsheets within the schema. A program generator can create from the specification an initial spreadsheet together with customized update operations for changing cells and inserting/deleting rows and columns for this particular specification [5]. These customized operations ensure that the spreadsheet can be changed only into new versions that always adhere to the table specification. A type system for specifications can guarantee that all spreadsheets that evolve through the customized update operations from a type-correct specification will never contain any reference, omission, or type errors.

References

[1] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets Through Spatial Analyses. *IEEE Symp. on Visual Languages and Human-Centric Computing*, 2004.

[2] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A Type System for Statically Detecting Spreadsheet Errors. *18th IEEE Int. Conf. on Automated Software Engineering*, pp. 174–183, 2003.

[3] T. Antoniu, P. A. Steckler, S. Krishnamurthi, Neuwirth, and M. Felleisen. Validating the Unit Correctness of Spreadsheet Programs. *Int. Conf. on Software Engineering*, 2004.

[4] M. M. Burnett and M. Erwig. Visually Customizing Inference Rules About Apples and Oranges. *2nd IEEE Int. Symp. on Human Centric Computing Languages and Environments*, pp. 140–148, 2002.

[5] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Gencel — A Program Generator for Correct Spreadsheets. Technical Report TR04-60-11, School of EECS, Oregon State University, 2004.

[6] M. Erwig and M. M. Burnett. Adding Apples and Oranges. *4th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pp. 173–191, 2002.

Hodnigg Karin, Roland Mittermeir,

Computational Models of Spreadsheet-Development

POSITION STATEMENT

Amongst multiple causes for high error rates in spreadsheets, lack of proper training and of deep understanding of the model behind spreadsheet computation and development is not the least among them. The fact that developing spreadsheets is programming and thus needs proper training somehow contradicts the intuitiveness of developing simple spreadsheets. Immediate feedback representation of values only, the possibility to shift complexity by splitting formulas over different cells, and the tabular layout hide intricacy. This is useful, when writing a spreadsheet program, disturbing, when trying to understand or maintain a spreadsheet.

Auditing tools help to reduce error rates. But, powerful as they are, they are expert tools. Spreadsheets, though, are quite often written by people with minimal formal spreadsheet training. To provide training to this community in acceptably small doses, it is important that it can rest on a simple but nevertheless solid conceptual model. The three layers of a spreadsheet program – the value, the formula and the data flow layer are a challenge. Thus, a spreadsheet programmer has (more or less) the notion of a data flow graph in mind which requires to memorize the coherence of a spreadsheet program with no explicit representation.

Considering spreadsheet system implementations, both, data flow and graph reduction models almost fit to the semantics of spreadsheets. But none of them fulfils all requirements of a correct conceptual spreadsheet model. E.g., neither loops nor circular references are part of the main spreadsheet paradigm. Moreover, the interactive evaluation process corresponds neither to graph reduction nor data flow programs. Thus, a conceptual model consistent with the spreadsheet paradigm is required. Differences in implementations aggravate the situation. Common operations (like copy-and-paste or drag-and-drop) are implemented differently on different systems. One main and profound problem is the approach of treating circular references into the spreadsheet paradigm. On one hand, circular references are likely to happen by accident (and thus are errors), on the other hand, the loop concept is used (in some implementations) to support scientific computations. Nevertheless, these approaches differ in their depth and implementation.

To establish a common and consistent conceptual model, these differences have to be taken into account. Users familiar with the tabular grid have to understand, that there is still some scoping among the different cells. This has been conceptualised in a projector-screen model. Corresponding to the spreadsheet peculiarities, it relies rather on visibility than on data flow, since the reference points to a cell – irrespective to its content. Moving operations influence the address of a cell, not the content. However, the concept of visibility is native to spreadsheets if one considers a cell seeing all the cells it is referencing. It does not "see", however, cells that are referencing this cell itself. Cells containing range formulas observe cells in a geometrical pattern. According to common approaches in spreadsheet programs, some typical patterns of references could be identified. Examples are: many-handed figures (one cell referencing a set of other cells just like a squid), the queue on a staircase (a sequence of cells referencing exactly its (geometrical) predecessor), flying carpets (range references over a geometrical pattern of cells) and recursive images (to explain circular references) may be useful patterns to provide an in-depth understanding of spreadsheet programming. This approach is discussed in-depth in "*Computational Models of Spreadsheet Development – Basis for educational approaches*", Hodnigg, Clermont, Mittermeir, EuSpRiG 2004.

Position Statement

David Wakeling [1]

Bioinformatics Group, University of Exeter, Exeter, United Kingdom

Cell biologists often create mathematical models of cellular processes in an attempt to understand them. Usually, the model is converted to a form suitable for computer simulation, evaluated by comparing the simulated and observed behaviour, and repeatedly revised until the two agree. Unfortunately, though, the design, implementation and documentation of many cell simulators can make this so wearing that all but the most determined cell biologists soon give up.

In this context, we argue that spreadsheets are useful:

- *from an HCI perspective*, because they provide a familiar setting in which to revise a model by asking "what if" questions;
- *from a programming language perspective*, because their natural *purely functional style* avoids the (often troublesome) use of macros;
- *from a quality perspective*, because a type system could be added to prevent the confusion of *types*, *dimensions* and *units* leading to nonsensical results.

[1] Email: `D.Wakeling@exeter.ac.uk`