

# Spreadsheet Programming

Robin Abraham, Microsoft Corporation  
Margaret Burnett, Oregon State University  
Martin Erwig, Oregon State University

Spreadsheets are among the most widely used programming systems in the world. Individuals and businesses use spreadsheets for a wide variety of applications, ranging from performing simple calculations to building complex financial models. In this article, we first discuss how spreadsheet programs are actually functional programs. We then describe concepts in spreadsheet programming, followed by a brief history of spreadsheet systems. Widespread use of spreadsheets, coupled with their high error-proneness and the impact of spreadsheet errors, has motivated research into techniques aimed at the prevention, detection, and correction of errors in spreadsheets. We present an overview of research effort that seeks to rectify this problem.

## 1 Introduction

For many people, the programming language of choice is a spreadsheet. In fact, spreadsheets are probably the most widely used *end-user* programming systems [1]. The people who use spreadsheets to program are often end-user programmers. End-user programmers are people who often have little or no training at programming but still do some amount of programming. In the U.S. alone, the number of end-user programmers is conservatively estimated at 11 million, compared to only 2.75 million other, professional programmers [2].

Even though spreadsheets have not been traditionally considered as such, spreadsheet programming is, in essence, first-order functional programming. It may seem odd to describe a spreadsheet as a programming language. Indeed, one of the great merits of spreadsheets is that users need not think of themselves as doing “programming”, let alone functional programming—rather, they simply “write formulas” or “build a model”. However, one can imagine printing the cells of a spreadsheet in textual form, like this:

```
A1 = 3  
A2 = A1 - 32  
A3 = A2 * 5/9
```

and then it plainly is a (functional) program. Considering spreadsheets as programs helps to identify clear opportunities for reasoning about spreadsheet errors and correctness.

Unlike most other programming languages, it is possible to begin using spreadsheets without training in programming. For example, a user can start to use a spreadsheet system by simply entering values in a few cells in a column. If the user later sums them up at the bottom of the column, for example with the “+” operator or the “SUM” built-in function, the spreadsheet has become a program. Spreadsheets are also forgiving in the sense that the user constantly gets feedback as to how they are making progress with the construction of their spreadsheet program, even when the program is not complete or contains errors.

The ease of use of spreadsheets has led to the development of spreadsheet-based approaches in other domains as well. In the domain of grid and parallel computing, this has led to the development of middleware which enables Excel users to develop parallel programs on a desktop grid computing environment [3]. Spreadsheet programming of web-based applications is the domain of Formulate [4] and FAR [5]. Information visualization spreadsheets, whose cells might contain abstract data sets, selection criteria, viewing specifications, and other information required to customize specific views, have been developed to allow end users access to rich visualizations of data [6]. Integration

of components of logic programming within the spreadsheet paradigm have also been proposed to reduce the incidence of errors within spreadsheets [7]. Somewhat related to spreadsheets are the rule-based languages AgentSheets [8] and KidSim [9], which are examples of visual programming environments that use spatial grids for the layout of program elements.

## 2 Concepts in Spreadsheet Programming

At a basic level, a *spreadsheet* is composed of cells. Cells can contain values, references to other cells, or formulas (which can also include references to other cells). A *spreadsheet system* is a programming environment that allows a user to create or modify a spreadsheet. In most of the widely used spreadsheet systems, the cells of a spreadsheet are part of rectangular grid. The underlying specification for which a spreadsheet is being developed is oftentimes referred to as a *spreadsheet model*. In many cases, especially with end-user programmers, the spreadsheet model is rather vague or even unknown. The programmers understand and refine it as they create their spreadsheet through what software engineers would recognize as evolutionary prototyping.

*Cell addresses* can be used to refer to the contents of a cell. Most spreadsheet systems support multiple forms of cell addresses. The following are three commonly used addressing schemes.

1. In *A1-style* addressing, a cell is referenced by its row and column position within the spreadsheet grid. For example, a cell at column D and row 7 would have the address D7. A worksheet in Excel 2007 can have up to 1048576 rows and 16384 columns.
2. In *R1C1-style* addressing, a cell address consists of the letter “R”, followed by the row number, and the letter “C”, followed by the column number. In this addressing scheme, a cell at column 4 and row 7 would be referenced as R7C4.
3. In *named* addressing, a cell can be named and the name can be used to reference it. The cell name must be unique within the spreadsheet, and once defined, the name can be used to reference the cell instead of the addressing schemes discussed above.

A cell reference can be *relative* or *absolute*. A relative reference is one that changes if the formula it is in is copied to a new cell. An absolute reference, on the other hand, points to the same cell even when the formula it is in is copied to another cell. In A1-style addressing, an absolute reference is specified by putting a “\$” in front of the part of the reference. For example, \$D7 is composed of an absolute reference to column 4 and a relative reference to row 7. The R1C1-style of referencing is absolute by default. In the relative addressing scheme, row or column number have to be enclosed in square brackets. For example, R[7]C4 is relative reference to row 7 and absolute reference to column 4. Named references are also absolute by default; they can be copy-pasted to any other cell and the reference remains unchanged.

Sometimes, functions operate over a group of cells within the spreadsheet. Such groups of cells can be specified using *cell ranges*. For example, the formula SUM(A1:A5) sums the values in cells A1, A2, A3, A4, and A5. Most spreadsheet systems have many predefined functions the user can use in building formulas. Microsoft Excel 2007 has over 300 predefined functions. Moreover, a user can also use support for *user-defined functions* to build custom functions which they can then use in their formulas.

From a programming point of view, the cells within a spreadsheet can be classified as follows:

1. Cells that contain constants that are not referenced by other cells are essentially labels. They might be used as a form of documentation, to provide additional information about the overall spreadsheet or some cells or region within the spreadsheet. For example, the string constant in cell A1 in Figure 1 is a label since it is not referenced by any formula. Based on the layout of the spreadsheet, a user might infer that A1 documents the fact that the integer 2000 in cell B1 is the “Total Budget”.
2. The cells that contain constants that are referenced by other cells within the spreadsheet constitute inputs to the spreadsheet program. For example, in Figure 1, the integer constant 20 in B4 is an input to the formula in D4.

	A	B	C	D
1	Total Budget	2000		
2				
3	Items	Units to order	Unit price	Item total
4	Pens	20	5	=B4*C4
5	Paper Clips	100	3	=B5*C5
6	Paper	25	15	=B6*C6
7	Total Cost			=SUM(D4:D6)
8				
9	Budget OK?	=IF(D7>B1,"Not ok","Budget ok")		
10				

Figure 1: Budget spreadsheet

3. The cells that contain non-trivial formulas<sup>1</sup> constitute the spreadsheet program. Those formula cells that are referenced by other cells with formulas can be considered intermediate computations. Cells with formulas that are not referenced by other cells can be considered output cells of the spreadsheet. Since the formulas in cells B4, B5, and B6 in Figure 1 are referenced by the formula in cell D7, they can be considered intermediate computations. Cells D7 and B9, on the other hand, can be considered outputs of the spreadsheet since they are not referenced by any other cells.

A spreadsheet represents a program, its input, and output, all at the same time. A user might carry out a *spreadsheet update operation* like changing the value in a cell that contains a constant, modifying a formula, or inserting or deleting a row or column of cells within the spreadsheet. Seemingly similar update operations have different impacts on the spreadsheet program. For example, a change of a constant amounts to changing program input, whereas a change in a formula means a change to the spreadsheet program.

Support for repetition in spreadsheets is not done using FOR-NEXT or WHILE loops. Instead, users can use rows (or columns) for this purpose. It is also possible to express cyclic formulas. Under normal circumstances, spreadsheet systems typically report an error in a formula if it refers, directly or indirectly (through a chain of references), to the cell that contains the formula. The formula is said to have a *circular reference*, and it is marked as an error since circular references can iterate indefinitely. However, in such situations, the user can control the iteration by specifying the maximum number of iterations performed, or the amount of acceptable change between iterations. In case the user specifies the maximum number of iterations to be performed, recomputation stops if the specified number of iterations is exceeded. Otherwise, recomputation stops when the change between the computed output in the current and previous steps falls below the specified acceptable change value.

Most commercial spreadsheet systems also provide programming environments that support the development of macros. For example, macros in Excel can be programmed in Visual Basic for Applications (VBA). A user can use VBA to automate spreadsheet tasks and also to develop user-defined functions. However, VBA is different from the Excel formula language and the programming environment is different from that of the spreadsheet system. To work around these issues, extensions to Excel that integrate user-defined functions within the spreadsheet grid have been proposed [10].

### 3 The Landscape of Spreadsheet Systems

VisiCalc [11] was the first spreadsheet system that supported essential interaction features found in the spreadsheet paradigm like the “What you see is what you get” (WYSIWYG) user interface, ranges with relative and absolute addresses, immediate visual feedback through automatic recalculation, and formula edit through selection of cells and ranges using a mouse. The popularity of VisiCalc was in part responsible for the success of the Apple II computer. Lotus 1-2-3 [12] and Borland Quattro were spreadsheet systems developed for the IBM PC platform. Lotus 1-2-3 had

<sup>1</sup>Formulas like 2+3 are considered trivial since they can be replaced by a constant (5 in this case) without altering the logic of the spreadsheet.

the same effect on the sales and adoption of the IBM PC as VisiCalc had the sales of Apple II, and was the dominant commercial spreadsheet system until the early 1990s when Microsoft Excel took over.

Lotus Improv [13] was the first attempt at separating the underlying model from the concrete spreadsheet. The basic idea was to separate the data, views of the data, and computations done on the data. This approach allowed users to define higher-level abstractions of their data in terms of items, categories, and groups. It introduced many novel ideas including named addressing. However, the system did not gain widespread acceptance, perhaps because its user interaction model was very different from that of the other spreadsheet systems available at that time.

There are a few commercial spreadsheet systems available today. Among them, Microsoft Excel [14] is the most widely used one. Excel is a part of the Microsoft Office suite of products. OpenOffice Calc [15] is a free, open-source spreadsheet system. Google's document suite of applications also includes a spreadsheet system [16]. Both, Calc and Google's spreadsheet system, can read and export files in Excel format.

The Forms/3 spreadsheet system has been developed as a research vehicle for the spreadsheet programming paradigm [5]. Unlike more traditional and commonly available spreadsheet systems, spreadsheets in Forms/3 do not have a grid structure or the global addressing scheme based on it. Instead, cells in Forms/3 can be arranged any way the user want to. The cells can be named by the user (or default to a system-generated name) and this name can then be used to reference the cell. An example of a Forms/3 spreadsheet is shown in Figure 3. Forms/3's purpose is to explore the boundaries of the spreadsheet paradigm in ways not traditionally supported by spreadsheets, such as procedural abstraction, data abstraction, and graphics output. It also has been used to explore aspects of end-user programming, such as helping end users to systematically test and debug their spreadsheets [17].

CoreCalc [18] is a spreadsheet implementation in C#. Like Forms/3, it is a platform for spreadsheet research. The source code for the implementation can be downloaded from the web and the system can be used to prototype new tools and functionality.

The popularity of spreadsheets and the supported interaction model has led to the development of many spreadsheet-like systems for various purposes. Haxcel is a spreadsheet-like interface to the functional programming language Haskell [19]. Model Master is a spreadsheet implementation in Prolog that supports querying using Prolog predicates [20]. Intellisheet is a spreadsheet system that allows the entry of expressions that represent linear and finite-domain constraints, along with arithmetic expressions, in individual cells. The system has a built-in constraint solver that solves the constraints and displays results to the user [21].

## 4 Spreadsheet Research

### 4.1 Errors

Surveys have shown that a majority of commercial spreadsheets have non-trivial errors in them. Because of the wide variety of uses spreadsheets are put to, spreadsheet errors can have serious impact in many different ways. Instances of spreadsheet errors that have received publicity in the press as a result of companies losing millions of dollars have been documented on the European Spreadsheet Risks Interest Group (EuSpRIG) website [22].

The flexibility afforded by spreadsheet systems makes it very easy to create erroneous spreadsheets or introduce errors in existing ones. Such risks are unlikely to decrease since studies have shown that end users are creating more and more complex spreadsheets [23], with typical corporate spreadsheets doubling in size and formula content every three years [24]. For example, studies have shown that when end users are faced with failures within spreadsheets they are maintaining, and are not sure of the underlying model, they overwrite the cell formulas with a constant value (the expected output for the cell) [25]. This approach might solve the problem they are facing at that point (an error with an unknown cause in a formula cell), but in the long run, it degrades the maintainability of the spreadsheet and generally introduces errors.

The distinction between different classes of spreadsheet errors is important since they usually require different types of error prevention and detection techniques [26].

1. *Mechanical errors* are simple slips that may arise due to carelessness, mental overload, or distractions. Examples include mistyping a number or a reference, pointing at a wrong cell address, or selecting an incorrect range of

cells or values.

2. *Omission errors* arise from the programmer leaving something out of the spreadsheet model by accident or due to ignorance.
3. *Logic errors* are the result of the programmer choosing an incorrect algorithm for solving a problem. These errors typically manifest themselves as incorrect formulas, and oftentimes require domain-specific knowledge to detect and correct.

Researchers have also classified errors as *quantitative* and *qualitative* errors [27]. Quantitative errors usually manifest themselves as incorrect results in the spreadsheet. Qualitative errors, on the other hand, take the form of poor spreadsheet design and format. Qualitative errors in a spreadsheet might not be visible or cause incorrect results in cells right away, but they lead to quantitative errors after spreadsheet updates since poor design makes spreadsheets hard to maintain. Systematic spreadsheet design reviews might help identify qualitative errors. On the other hand, techniques like consistency checking, testing, and spreadsheet code inspection are more useful in detecting quantitative errors.

Spreadsheet research has focused on tools and approaches that seek to reduce the number of spreadsheet errors, either by detection and subsequent correction of errors, or by prevention of errors. A commonly accepted constraint of this research is to provide support without being intrusive. Tools or approaches aimed at helping spreadsheet users therefore tend to provide incremental and immediate feedback, in keeping with the operation of the spreadsheet systems themselves. Research also indicates that the systems should be tolerant to errors in feedback provided by the user. For example, end users are not very accurate oracles during testing [28]. Moreover, in order to gain acceptance, the benefits of using the system should be obvious and outweigh the investment (in terms of time and effort required of the user) [17].

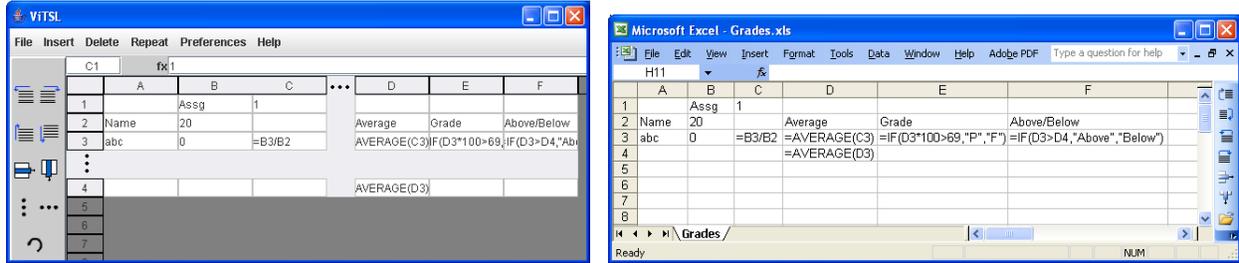
## 4.2 Design and Maintenance

Researchers have studied the occurrence of qualitative and quantitative errors and have made recommendations on how to avoid them. Mason and Keane have suggested that organizations should have a “Model Administrator” (along the lines of Database Administrators for enterprise-level databases) to regulate spreadsheet models within the organization [29]. Williams has recommended the adoption of organization standards [30]. These standards would consist of recommendations for best practices that cover spreadsheet specification, documentation, maintenance, and security that employees should follow while creating spreadsheets. The use of automated spreadsheet audit software has been recommended by Simkin [31]. Teo and Tan have shown that spreadsheet errors are hard to detect during “what-if” analyses if the spreadsheets are not well designed [32]. Based on their studies, they have recommended that courses designed for teaching spreadsheets should focus more on spreadsheet design, targeting ease of maintenance and debugging, rather than demonstrating the features available in spreadsheet systems [33].

Ethnographic interviews with spreadsheet users have shown that spreadsheets are often developed through collaborative effort in the work environment. The collaborators often had different levels of programming and domain expertise and shared responsibility for error detection [34]. Studies have also shown that spreadsheet auditing, as an error-detection technique, takes a lot of time and effort, and also requires a lot of expertise. It has also been shown in empirical studies that code inspections of spreadsheet formulas done by spreadsheet programmers working alone detects 63% of the seeded errors. Group code inspections are only slightly more effective and detect 83% of the errors [35]. The costs involved are already rather high for medium-size spreadsheets and get worse for large spreadsheets that are used in business [25]. Moreover, traditional spreadsheet auditing techniques (without tool support) do not scale very well since the visible area is limited by the size of the screen and the inspections are done at the cell level.

Isakowitz et al. recommend the treatment of spreadsheets from two different perspectives: logical and physical. The logical level deals with the model-related and formal aspects, and the physical level concerns details such as storage, formatting, user interface, and other aspects related to the implementation of the model. Additional information about the underlying model would be helpful to the programmer in understanding the spreadsheet, and can oftentimes be understood by:

1. detecting tables or areas with similar formulas in the spreadsheet,



(a) ViTSL template

(b) Generated spreadsheet in Gencel

Figure 2: Generating spreadsheets from specifications

2. analyzing the data flow in order to find regularities between different spreadsheet regions, or
3. finding logical relationships between cells that are not physically adjacent.

Field audits have shown that spreadsheet programs in business applications can become very large over a period of time. These spreadsheets usually consist of a set of unique formulas that are copied throughout the spreadsheet and then often only slightly modified, usually to change constants or relative references [25]. Some of the spreadsheets developed in the scientific computing domain, for example, use logically equivalent formulas to simulate iteration [36]. Identification of such logically related blocks would also help with comprehension and testing of spreadsheet programs. Visualization techniques have been developed to help end users identify cells that are outliers [37, 38].

The lack of modularity and abstractions make it very difficult to program and maintain spreadsheets that implement complex models. This problem has led to the development of specification languages like ViTSL (Visual Template Specification Language) [39] and ClassSheets [40] that can be used to capture the underlying specification of a spreadsheet. ViTSL specifications can be imported into a related system, called Gencel [41], which can then be used to generate spreadsheets that conform to the specification. Figure 2(b) shows the first instance of a spreadsheet generated by importing the ViTSL specification shown in Figure 2(a) into the Gencel system. In an organizational setting, the templates can be created by domain experts and programmers, tested, and then handed off to others for use. In this approach, the user only needs to update data and labels within the spreadsheet; the formulas are automatically generated and updated by Gencel to enforce the underlying specification. Parcel is a tool that has been developed to help with the related problem of inferring the underlying model from given spreadsheets, thereby allowing users to import legacy spreadsheets into the ViTSL/Gencel framework [42].

### 4.3 Checking, Testing, and Debugging

Automatic consistency checking is one of the relatively cheaper ways to identify potential problems in spreadsheets. For example, the UCheck system automatically infers the labels associated with cells and uses this information to carry out consistency checking of the formulas [43]. Some consistency checkers require the user to provide additional cell annotations which are then used to carry out consistency checking [44, 45].

Even professional programmers spend a major portion of their time identifying and correcting faults within their programs. A recent study conducted in the U.S. by the National Institute of Standards and Technology (NIST) has found that professional software engineers typically spend 70 to 80% of their time testing and debugging, and on average, errors take 17.4 hours to find and fix [46].

Testing a piece of code helps develop confidence that it works as expected. That is why considerable effort is spent on testing during the course of software development. Commercially available spreadsheet systems do not support systematic testing of spreadsheets. Moreover, end users might not be aware of how to go about effectively testing their spreadsheets. As a result, they might do simple tests using historical data and come away with an unjustified high level of confidence in the correctness of their spreadsheets. The “What You See Is What You Test” (WYSIWYT) testing methodology [47], implemented in the Forms/3 spreadsheet system, allows users to systematically test their

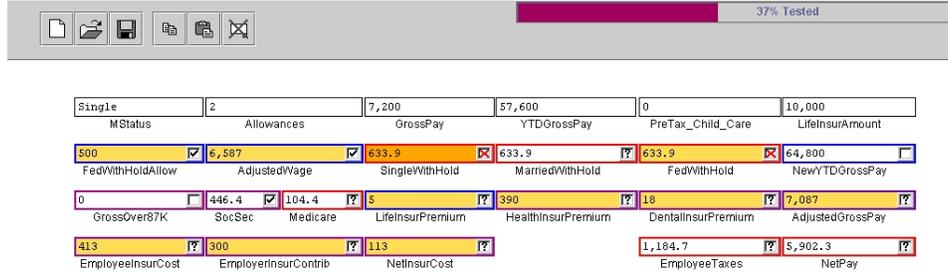
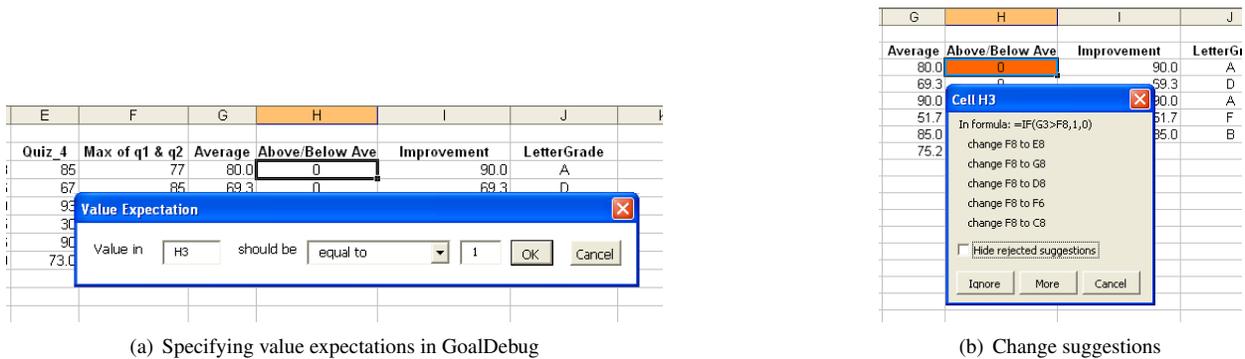


Figure 3: A Forms/3 spreadsheet with WYSIWYT (“What You See Is What You Test”)



(a) Specifying value expectations in GoalDebug

(b) Change suggestions

Figure 4: The spreadsheet debugger GoalDebug

spreadsheets. For a set of input values, the user can inspect the output in cells with formulas to verify if they are correct or not. If the output is correct, the user can place a ✓ mark in the cell. On the other hand, if the computed output is incorrect, the user can place a ✗ mark in the cell. Both of these actions are used to reason about how well tested the formulas and their interrelationships are, as well as which cells’ formulas contain the faults that led to values marked as being incorrect. WYSIWYT testing feedback colorizes the cell borders to keep the user apprised of how thoroughly tested they are. The fault-localization feedback of the system shades the cells—darker shades for cells that have a higher likelihood of faults, compared to lighter shades for cells that are less likely to contain faults (as shown in Figure 3).

Sometimes users can run into difficulty in creating effective test cases to test their spreadsheets. This problem has led to the development of two automatic test-case generation systems for spreadsheets: “Help Me Test” [48], which has been integrated into Forms/3 and WYSIWYT, and AutoTest [49], which has been prototyped in Microsoft Excel. Both these systems support incremental testing targeting the *definition-use* adequacy criterion. In a spreadsheet, every cell defines at least one value. Cells that contain formulas with conditionals generally give rise to two or more definitions, contained in the different branches. Likewise, cell formulas might contain references to other cells. Such references constitute uses of the definitions of the cells that are being referenced. The idea behind definition-use adequacy criterion is to test for each definition of a cell all of its uses.

Detecting errors in spreadsheets is only one part of the problem. Even in cases where failures within their spreadsheets are identified by end users, the faults that led to the failures might not be clear. Moreover, the formula changes that need to be carried out to correct the faults might not be obvious. These observations led to the development of GoalDebug, a spreadsheet debugger that allows the user to specify the expected output for any cells whose output is incorrect [50]. The interface for providing user expectation on cell outputs is shown in Figure 4(a). The system then

uses static and dynamic slicing techniques to generate change suggestions, any one of which when applied, would result in the expected output being computed in the cell. The generated change suggestions are ranked using a set of heuristics, and the cell with the highest ranked change suggestion is shaded orange. The user can then view the ranked change suggestions (see Figure 4(b) for an example of the automatically generated change suggestions). The change suggestions that get generated can be directly applied to the spreadsheet formula, thereby minimizing the need for any formula edits by the end user.

It bears mention that none of the approaches discussed above can detect or prevent all possible errors within a spreadsheet. Each of the approaches has its own strengths and weaknesses. Therefore, it is essential to continue research into tools and techniques that minimize errors in end-user spreadsheets.

## 5 Summary

The spreadsheet programming model is widely used and well accepted. Even though spreadsheet systems continue to offer new features, programming spreadsheets remains an error-prone endeavor. Therefore, efforts that help to improve the quality of spreadsheets are probably more important than adding functionality. The challenge lies in developing tools and methodologies that can help end users improve the dependability of their spreadsheets without being intrusive or interfering with their goals. It therefore seems most promising to employ successful methods that have been developed in the area of software engineering to help spreadsheet users with their tasks.

## References

- [1] R. R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. In *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.
- [2] C. Scaffidi, M. Shaw, and B. Myers. Estimating the Numbers of End Users and End User Programmers. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.
- [3] K. Konishi and T. Nakazawa. Grid Computing Middleware for Microsoft Excel. *Systems Modeling and Simulation*, pages 309–313, 2007.
- [4] A. Ambler and J. Leopold. Public Programming in a Web World. In *Visual Languages and Human-Centric Computing*, pages 100–107, 1998.
- [5] M. M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming*, 11(2):155–206, March 2001.
- [6] E. H. Chi, J. Riedl, P. Barry, and J. Konstan. Principles for Information Visualization Spreadsheets. *IEEE Computer Graphics and Applications*, 18(4):30–38, 1998.
- [7] P. T. Cox. Enhancing the Programmability of Spreadsheets with Logic Programming. In *Visual Languages and Human-Centric Computing*, pages 87–94, 2007.
- [8] A. Repenning and T. Sumner. Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *IEEE Computer*, 28(3):17–25, 1995.
- [9] A. Cypher and D. C. Smith. KidSim: End User Programming of Simulations. In *Conference Companion on Human Factors in Computing Systems*, pages 35–36, 1995.
- [10] S. L. Peyton Jones, A. Blackwell, and M. M. Burnett. A User-Centered Approach to Functions in Excel. In *ACM Int. Conf. on Functional Programming*, pages 165–176, 2003.

- [11] VisiCalc. <http://www.bricklin.com/visicalc.htm>.
- [12] IBM. Lotus 1-2-3. <http://www-306.ibm.com/software/lotus/products/123/>.
- [13] Lotus. Lotus Improv. <http://en.wikipedia.org/wiki/Lotus.Improv>.
- [14] Microsoft. Microsoft Excel. <http://office.microsoft.com/excel/>.
- [15] OpenOffice. OpenOffice Calc. <http://www.openoffice.org/product/calc.html>.
- [16] Google. Google Spreadsheets. <http://docs.google.com/>.
- [17] M. M. Burnett, C. Cook, and G. Rothermel. End-user Software Engineering. *Communications of the ACM*, 49(9):53–58, September 2004.
- [18] P. Sestoft. CoreCalc: A Spreadsheet Core Implementation in C#. <http://www.itu.dk/people/sestoft/corecalc/>.
- [19] B. Lisper and J. Malmstrom. Haxcel: A Spreadsheet Interface to Haskell. In *Proc. Int. Workshop on the Implementation of Functional Languages*, pages 206–222, September 2002.
- [20] J. Paine. ModelMaster: A Tool For Spreadsheet Intelligibility, Spreadsheet Integrity, and Spreadsheets on the Web. [http://www.j-paine.org/mm\\_publications.html/](http://www.j-paine.org/mm_publications.html/).
- [21] Y. Adachi. Intellisheet: A Spreadsheet System Expanded by Including Constraint Solvers. *IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, page 173, 2001.
- [22] EuSpRIG. European Spreadsheet Risks Interest Group. <http://www.eusprig.org/>.
- [23] M. Shaw. Avoiding Costly Errors in Your Spreadsheets. *Contractor's Management Report*, 11:2–4, 2004.
- [24] D. Whittaker. Spreadsheet Errors And Techniques For Finding Them. *Management Accounting*, 77(9):50–51, 1999.
- [25] M. Clermont, C. Hanin, and R. Mittermeir. A Spreadsheet Auditing Tool Evaluated in an Industrial Context. *Spreadsheet Risks, Audit and Development Methods*, 3:35–46, 2002.
- [26] R. R. Panko and R. P. Halverson, Jr. Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks. In *29th Hawaii Int. Conf. on System Sciences*, 1996.
- [27] T. Teo and M. Tan. Quantitative and Qualitative Errors in Spreadsheet Development. In *30th Hawaii Int. Conf. on System Sciences*, pages 25–38, 1997.
- [28] R. R. Panko. What We Know About Spreadsheet Errors. *Journal of End User Computing (Special issue on Scaling Up End User Development)*, 10(2):15–21, 1998.
- [29] D. Mason and D. Keane. Spreadsheet Modeling in Practice: Solution or Problem. In *Interface*, pages 82–84, 1989.
- [30] T. Williams. Spreadsheet Standards. *Tech. Report*, 1987.
- [31] M. G. Simkin. How to Validate Spreadsheets. *Journal of Accountancy*, 180(11):130–138, 1987.
- [32] T. Teo and M. Tan. Spreadsheet Development and 'What-if' Analysis: Quantitative Versus Qualitative Errors. *Accounting Management and Information Technologies*, 9:141–160, 2000.
- [33] S. G. Powell and K. R. Baker. *The Art of Modeling with Spreadsheets: Management Science, Spreadsheet Engineering, and Modeling Craft*. Wiley, 2004.

- [34] B. A. Nardi and J. R. Miller. Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development. *Int. J. Man-Mach. Stud.*, 34(2):161–184, 1991.
- [35] R. R. Panko. Applying Code Inspection to Spreadsheet Testing. *Journal of Management Information Systems*, 16(2):159–176, 1999.
- [36] G. Filby. *Spreadsheets in Science and Engineering*. Springer, 1995.
- [37] J. Sajaniemi. Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *Journal of Visual Languages and Computing*, 11(1):49–82, 2000.
- [38] T. Igarashi, J. Mackinlay, B.-W. Chang, and P. T. Zelleweger. Fluid Visualization of Spreadsheet Structures. In *IEEE Symp. on Visual Languages*, 1998.
- [39] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual Specifications of Correct Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 189–196, 2005.
- [40] G. Engels and M. Erwig. ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications. In *20th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 124–133, 2005.
- [41] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein. Gencil — A Program Generator for Correct Spreadsheets. *Journal of Functional Programming*, 16(3):293–325, 2006.
- [42] R. Abraham and M. Erwig. Inferring Templates from Spreadsheets. In *28th IEEE Int. Conf. on Software Engineering*, pages 182–191, 2006.
- [43] R. Abraham and M. Erwig. UCheck: A Spreadsheet Unit Checker for End Users. *Journal of Visual Languages and Computing*, 18(1):71–95, 2007.
- [44] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A Type System for Statically Detecting Spreadsheet Errors. In *18th IEEE Int. Conf. on Automated Software Engineering*, pages 174–183, 2003.
- [45] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the Unit Correctness of Spreadsheet Programs. In *26th IEEE Int. Conf. on Software Engineering*, pages 439–448, 2004.
- [46] G. Tassej. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology*, RTI Project Number 7007.011, 2002.
- [47] G. Rothermel, M. M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A Methodology for Testing Spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10(2):110–147, 2001.
- [48] M. Fisher II, G. Rothermel, D. Brown, M. Cao, C. Cook, and B. Burnett. Integrating Automated Test Generation into the WYSIWYT Spreadsheet Testing Methodology. *ACM Trans. on Software Engineering and Methodology*, 15(2):150–194, 2006.
- [49] R. Abraham and M. Erwig. AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 43–50, 2006.
- [50] R. Abraham and M. Erwig. GoalDebug: A Spreadsheet Debugger for End Users. In *29th IEEE Int. Conf. on Software Engineering*, pages 251–260, 2007.