Int. Workshop on the Foundations of Spreadsheets (FOS'04)

Rome, Italy
September 30th, 2004

http://eecs.oregonstate.edu/~erwig/FOS/

Int. Workshop on the Foundations of Spreadsheets (FOS'04)

Program Committee

Alan Blackwell	University of Cambridge, UK
Margaret Burnett	Oregon State University, USA
Martin Erwig	Oregon State University, USA (chair)
Shriram Krishnamurthi	Brown University, USA
Clayton Lewis	University of Colorado, USA
Simon Peyton Jones	Microsoft Research, Cambridge, UK
Jorma Sajaniemi	University of Joensuu, Finland
Susan Wiedenbeck	Drexel University, USA

I. Position Statements

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.

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

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.

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

II. Papers

International Workshop on the Foundations of Spreadsheets
Rome, Italy
September 30, 2004

Spreadsheet Modeling for Insight

Stephen G. Powell [1,2]

Tuck School of Business
Dartmouth College
Hanover, NH 03755 USA

Abstract

It is widely recognized that spreadsheets are error-filled, their creators are overconfident, and the process by which they are developed is chaotic. It is less well-understood that spreadsheet users generally lack the skills needed to derive *practical insights* from their models. Modeling for insight requires skills in establishing a base case, performing sensitivity analysis, using back-solving, and (when necessary) carrying out optimization and simulation. Some of these tasks are made possible only with specialized add-ins to Excel. In this paper we present an overview of the skills and software tools needed to model for insight.

Key words: sensitivity analysis, software engineering, spreadsheet engineering

1 Introduction

There is ample evidence that spreadsheets as actually used in industry are highly problematic [1]. Many, if not most, spreadsheets harbor serious bugs. The end-users who typically design and use spreadsheets are under-trained and overconfident in the accuracy of their models. The process that most spreadsheet developers use is chaotic, leading to time wasted in rework and in high error rates. Few spreadsheets are tested in any formal manner. Finally, many organizations fail to follow standard procedures for documentation or version control, which leads to errors in use even if the spreadsheets themselves are correct. While these problems are well known to a few researchers, and widely suspected by many managers, few companies recognize the risks that spreadsheet errors pose.

This paper is concerned with a much less well understood problem, involving missed opportunities to extract useful business insights from spreadsheet

[1] Thanks are due to Ken Baker at Tuck School of Business, Dartmouth College, who co-developed most of the ideas in this paper.
[2] Email: Stephen.G.Powell@Dartmouth.EDU

models. Many spreadsheet developers have extensive skills in Excel itself but far fewer have a disciplined approach to using a model to inform a decision or shed light on a business problem. We advocate an engineering approach to the process of designing and building a spreadsheet. In the same spirit, the analysis process itself can be improved by providing structure and specific software tools. We will discuss in particular four analytic tools that are contained in the Sensitivity Toolkit, a publicly-available Excel add-in we built.

2 Elements of Spreasheet Engineering

Spreadsheet modeling is a form of computer programming, although it is usually carried out by people who do not think of themselves as programmers. Moreover, few spreadsheet developers are trained in software engineering. In order to improve this situation we have undertaken the task of translating the principles of software engineering into a form that end-users in business can actually use. We call the resulting discipline *spreadsheet engineering*. Our motivation is to improve both the efficiency and the effectiveness with which spreadsheets are created. An *efficient* design process uses the minimum time and effort to achieve results. An *effective* process achieves results that meet the users' requirements. Although spreadsheet modeling is a creative process, and thus cannot be reduced to a simple recipe, every spreadsheet passes through a predictable series of four stages: designing, building, testing, and analysis. Some of the principles in each of the first three phases are given below:

- designing
  - sketch the spreadsheet
  - organize the spreadsheet into modules
  - start small
  - isolate input parameters
  - design for use
  - keep it simple
  - design for understanding
  - document important data and formulas
- building
  - follow a plan
  - build one module at a time
  - predict the outcome of each formula
  - Copy and Paste formulas carefully
  - use relative and absolute addresses to simplify copying
  - use the Function Wizard to ensure correct syntax
  - use range names to make formulas easy to read
- testing
  - check that numerical results look plausible

· check that formulas are correct
· test that model performance is plausible

Since the focus of this paper is on improving the analysis phase, we will not discuss the first three phases of spreadsheet engineering further in this paper. These are described in more detail in [2].

3 Insight: The Goal of Spreadsheet Modeling

In many business applications, the ultimate goal of a spreadsheet modeling effort is not numerical at all; rather, it is an *insight* into a problem or situation, often a decision facing the organization. In our minds, an insight is never a number but can be expressed in natural language that managers understand, often in the form of a graph. Insights often arise from surprises. For example, Option A looks better than Option B on first glance, but our analysis shows why B actually is a better choice. Many insights involve trade-offs. For example, as we add capacity we find at first that service improves faster than cost increases, but eventually increasing costs swamp improvements in service.

If we accept the notion that the purpose of many spreadsheet models is to identify insights, it follows that the spreadsheet itself is not a particularly good vehicle for the purpose. As convenient as the spreadsheet format is, it does not display the *relationships* involved in a model, but hides them behind a mass of numbers. Nor does it show how changes in inputs affect outputs, which is where insight begins. Users of spreadsheets need to be taught how to make the row-and-column format work for them to generate insights. There are several powerful but obscure features built into Excel (like Goal Seek and Data Table) that can assist in this process. To augment these tools we have built a Visual Basic add-in called the Sensitivity Toolkit that automates some of the most powerful sensitivity analysis tools. (This add-in is publicly available at `http://mba.tuck.dartmouth.edu/toolkit/`)

Although Excel itself has thousands of features, most of the analysis done with spreadsheets falls into one of the following five categories:

- base-case analysis
- what-if analysis
- breakeven analysis
- optimization analysis
- risk analysis

Within each of these categories, there are specific Excel tools, such as the Goal Seek tool, and add-ins, such as Solver [3] and Crystal Ball [4], which can be used either to automate tedious calculations or to find powerful business insights that cannot be found any other way. Some of these tools, such as Solver, are quite complex and can be given only a cursory treatment here. By contrast, some of the other tools we describe are extremely simple, yet are

underutilized by the majority of spreadsheet users.

We will use the spreadsheet model Advertising Budget (see Figure 1) to illustrate each of these five categories of analysis. This model takes various inputs, including the price and unit costs of a product, and calculates revenues, total costs, and profit over the coming year by quarters. The essential relationship in the model is a sales response to advertising function characterized by diminishing returns. The fundamental question the model will be used to answer is how we should allocate a fixed advertising budget across quarters.

3.1 Base-case analysis

Almost every spreadsheet analysis involves measuring outcomes relative to some common point of comparison, or **base case**. Therefore, it is worth giving some thought to how the base case is chosen. A base case is often drawn from current policy or common practice, but there are many other alternatives. Where there is considerable uncertainty in the decision problem, it may be appropriate for the base case to depict the most likely scenario; in other circumstances, the worst case or the best case might be a good choice.

Sometimes, several base cases are used. For example, we might start the analysis with a version of the model that takes last year's results as the base case. Later in the analysis, we might develop another base case using a proposed plan for the coming year. At either stage, the base case is the starting point from which an analyst can explore the model using the tools described in this paper, and thereby gain insights into the corresponding business situation.

In the Advertising Budget example, most of the input parameters such as price and cost are forecasts for the coming year. These inputs would typically be based on previous experience, modified by our hunches as to what will be different in the coming year. But what values should we assume for the decision variables, the four quarterly advertising allocations, in the base case? Our ultimate goal is to find the best values for these decisions, but that is premature at this point. A natural alternative is to take last year's advertising expenditures ($\$10,000$ in each quarter) as the base-case decisions, both because this is a simple plan and because initial indications point to a repeat for this year's decisions.

3.2 What-if analysis

Once a base case has been specified, the next step in analysis often involves nothing more sophisticated than varying one of the inputs to determine how the key outputs change. Assessing the change in outputs associated with a given change in inputs is called **what-if analysis**. The inputs may be *parameters*, in which case we are asking how sensitive our base-case results are to forecasting errors or other changes in those values. Alternatively, the inputs we vary may be *decision variables*, in which case we are exploring whether changes in our decisions might improve our results, for a given set

of input parameters. Finally, there is another type of what-if analysis, in which we test the effect on the results of changing some aspect of our model's *structure*. For example, we might replace a linear relationship between price and sales with a nonlinear one. In all three of these forms of analysis, the general idea is to alter an assumption and then trace the effect on the model's outputs.

We use the term **sensitivity analysis** interchangeably with the term what-if analysis. However, we are aware that sensitivity analysis sometimes conveys a distinct meaning. In optimization models, where optimal decision variables themselves depend on parameters, we use the term sensitivity analysis specifically to mean the effect of changing a parameter on the *optimal* outcome. (In optimization models, the term what-if analysis is seldom used.)

When we vary a *parameter*, we are implicitly asking what would happen if the given information were different. That is, what if we had made a different numerical assumption at the outset, but everything else remained unchanged? This kind of questioning is important because the parameters of our model represent assumptions or forecasts about the environment for decision making. If the environment turns out to be different than we had assumed, then it stands to reason that the results will also be different. What-if analysis measures that difference and helps us appreciate the potential importance of each numerical assumption.

In the Advertising Budget example, if unit cost rises to \$26 from \$25, then annual profit drops to \$53,700. In other words, an increase of 4 percent in the unit cost will reduce profit by nearly 23 percent. Thus, it would appear that profits are quite sensitive to unit cost, and, in light of this insight, we may decide we should monitor the market conditions that influence the material and labor components of cost.

When we vary a *decision variable*, we are exploring outcomes that we can influence. First, we'd like to know whether changing the value of a decision variable would lead to an improvement in the results. If we locate an improvement, we can then try to determine what value of the decision variable would result in the best improvement. This kind of questioning is a little different from asking about a parameter, because we can act directly on what we learn. What-if analysis can thus lead us to better decisions.

In the Advertising Budget example, if we spend an additional \$1,000 on advertising in the first quarter, then annual profit rises to \$69,882. In other words, an increase of 10 percent in the advertising expenditure during $Q1$ will translate into an increase of roughly 0.3 percent in annual profit. Thus, profits do not seem very sensitive to small changes in advertising expenditures in $Q1$, all else being equal. Nevertheless, we have identified a way to increase profits. We might guess that the small percentage change in profit reflects the fact that expenditures in the neighborhood of \$10,000 are close to optimal, but we will have to gather more information before we are ready to draw conclusions about optimality.

In addition to testing the sensitivity of results to parameters and decision variables, there are situations in which we want to test the impact of some element of model structure. For example, we may have assumed that there is a linear relationship between price and sales. As part of what-if analysis, we might then ask whether a nonlinear demand-price relationship would materially alter our conclusions. As another example, we may have assumed that our competitors will not change their prices in the coming year. If we then determine that our own prices should increase substantially over that time, we might ask how our results would change if our competitors were to react to our pricing decisions by matching our price increases. These what-if questions are more complex than simple changes to a parameter or a decision variable because they involve alterations in the underlying structure of the model. Nonetheless, an important aspect of successful modeling is testing the sensitivity of results to key assumptions in the structure of the model.

In the Advertising Budget example, the relationship between advertising and sales is given by the nonlinear function:

$$\text{Sales} = 35 \times \text{Seasonal Factor} \times \sqrt{\text{Advertising} + 3000}. \tag{1}$$

In the spirit of structural sensitivity analysis, we can ask how different our results would be if we were to replace this relationship with a linear one. For example, the linear relationship

$$\text{Sales} = 3,000 + 0.1(\text{Advertising} \times \text{Seasonal Factor}) \tag{2}$$

lies close to the nonlinear curve for advertising levels around $\$10,000$. When we substitute this relationship into the base-case model, holding advertising constant at $\$10,000$ each quarter, we find that profit changes only slightly, to $\$70,000$. But in this model, if we then increase $Q1$ advertising by $\$1,000$, we find that profit *decreases*, while in the base-case model it increases. Evidently, this structural assumption does have a significant impact on the desired levels of advertising.

We have illustrated what we might call a "one-at-a-time" form of what-if analysis, where we vary one input at a time, keeping other inputs unchanged. We could, of course, vary two or more inputs simultaneously, but these more complex experiments become increasingly difficult to interpret. In many cases, we can gain the necessary insights by varying the inputs one at a time.

It is important not to underestimate the power of this first step in analysis. Simple what-if exploration is one of the most effective ways to develop a deeper understanding of the model and the system it represents. It is also part of the debugging process. When what-if analysis reveals something unexpected, we have either found a useful insight or perhaps discovered a bug.

Predicting the outcome of a what-if test is an important part of the learning process. For example, in the Advertising Budget example, what would be the result of doubling the selling price? Would profits double as well? In the base

case, with a price of $40, profits total $69,662. If we double the price, we find that profits increase to $612,386. Profits increase by much more than a factor of two when prices double. After a little thought, we should see the reasons. For one, costs do not increase in proportion to volume; for another, demand is not influenced by price in this model. Thus, the sensitivity test helps us to understand the nature of the cost structure — that it's not proportional — as well as one limitation of the model — that no link exists between demand and price.

3.2.1 Data Sensitivity

The **Data Sensitivity** tool automates certain kinds of what-if analysis. It simply recalculates the spreadsheet for a series of values of an input cell and tabulates the resulting values of an output cell. This allows the analyst to perform several related what-if tests in one pass rather than entering each input value and recording each corresponding output.

The Data Sensitivity tool is one module in the Sensitivity Toolkit, which is an add-in to Excel (available at `http://mba.tuck.dartmouth.edu/toolkit`). Once the Toolkit is installed, the Sensitivity Toolkit option will appear on the far right of the menu bar (see Figure 1). Data Sensitivity and the other modules can be accessed from this menu. (An equivalent tool called Data Table is built into Excel.)

We illustrate the use of the Data Sensitivity tool in the Advertising Budget model by showing how variations in unit cost from a low of $20 to a high of $30 affect profit. (Note: we will not describe the specific steps required to run any of the tools in the Toolkit in this paper: details can be found in [2] or in the Help Facility in the Toolkit itself).

Figure 2 shows the output generated by the Data Sensitivity tool. A worksheet has been added to the workbook, and the first two columns on the sheet contain the table of what-if values. In effect, the what-if test has been repeated for each unit-cost value from $20 to $30 in steps of $1, and the results have been recorded in the table. In addition, the table is automatically converted to a graph. As the table and graph both show, annual profits drop as the unit cost increases, and the cost-profit relationship is linear. We can also see that the breakeven value of the unit cost falls between $29 and $30, since profits cross from positive values to negative values somewhere in this interval.

Note that the Data Sensitivity tool requires that we provide a single cell address to reference the input being varied in a one-way table. The tool will work correctly only if the input has been placed in a single location. By contrast, if an input parameter had been embedded in several cells, the tool would have given incorrect answers when we tried to vary the input. Thus, the use of single and separate locations for parameters (or for decisions), which is a fundamental principle of spreadsheet engineering, makes it possible to take advantage of the tool's capability.

We can also use the Data Sensitivity tool to analyze the sensitivity of an output to *two* inputs. This option gives rise to a two-way table, in contrast to the one-way sensitivity table illustrated above. To demonstrate this feature, we can build a table showing how profits are affected by both $Q1$ advertising and $Q2$ advertising. By studying the results in Figure 3, we can make a quick comparison between the effect of additional spending in $Q1$ and the effect of the same spending in $Q2$. As we can observe in the table, moving across a row generates more profit than moving the same distance down a column. This pattern tells us that we can gain more from spending additional dollars in $Q2$ than from the same additional dollars in $Q1$. This observation suggests that, starting with the base case, we could improve profits by shifting dollars from $Q1$ to $Q2$. We can also note from the table, or from the three-dimensional chart that automatically accompanies it, that the relationship between profits and advertising expenditures is not linear. Instead, profits show diminishing returns to advertising.

3.2.2 Tornado charts

Another useful tool for sensitivity analysis is the **tornado chart**. In contrast to the information produced by the Data Sensitivity tool, which shows how sensitive an output is to one or perhaps two inputs, a tornado chart shows how sensitive the output is to several different inputs. Consequently, it shows us which parameters have a major impact on the results and which have minor impact.

Tornado charts are created by changing input values one at a time and recording the variations in the output. The simplest approach is to vary each input by a fixed percentage, such as ±10 percent, of its base-case value. For each parameter in turn, we increase the base-case value by 10 percent and record the output, then decrease the base-case value by 10 percent and record the output. Next, we calculate the absolute difference between these two outcomes and depict the results in the order of these differences.

The Sensitivity Toolkit contains a tool for generating tornado charts. The Tornado Chart tool provides a choice of three options:

- Constant Percentage
- Variable Percentage
- Percentiles

We will illustrate the Constant Percentage case first. The tornado chart appears on a newly inserted worksheet, as shown in Figure 4. The horizontal axis at the top of the chart shows profits; the bars in the chart show the changes in profit resulting from ±10 percent changes in each input. After calculating the values (which are recorded in the accompanying table on the same worksheet), the bars are sorted from largest to smallest for display in the diagram. Thus, the most sensitive inputs appear at the top, with the largest horizontal spans. The least sensitive inputs appear toward the bottom, with

the smallest horizontal spans. Drawing the chart using horizontal bars, with the largest span at the top and the smallest at the bottom, suggests the shape of a tornado, hence the name. If some of the information in the chart seems unclear, details can usually be found in the accompanying table, which is constructed on the same worksheet by the Tornado Chart tool. In our example, we can see in the table that price has the biggest impact (a range of more than $\$108,000$), with unit cost next (a range of nearly $\$80,000$), and the other inputs far behind in impact on profit.

The standardization achieved by using a common percentage for the change in inputs (10 percent in our example) makes it easy to compare the results from one input to another, but it may also be misleading. A 10 percent range may be realistic for one parameter, while 20 percent is realistic for another, and 5 percent for a third. The critical factor is the size of the forecast error for each parameter. If these ranges are significantly different, we should assign different percentages to different inputs. This can be accomplished using the Variable Percentage option in the Tornado Chart tool.

To illustrate the Variable Percentage option in the Advertising Budget example, suppose we limit ourselves to seven parameters: price, cost, four seasonal factors, and overhead rate. Suppose that, based on a detailed assessment of the uncertainty in these parameters, we choose to vary price by 5 percent, cost by 12 percent, seasonal factors by 8 percent, and overhead rate by 3 percent. The resulting tornado chart is shown in Figure 5. As the results show, cost now has the biggest impact on profits, partly because it has a larger range of uncertainty than price.

3.3 Breakeven analysis

Many managers and analysts throw up their hands in the face of uncertainty about critical parameters. If we ask a manager to directly estimate market share for a new product, the reply may be: "I have *no idea* what market share we'll capture". A powerful strategy in this situation is to reverse the sense of the question and ask not, "What will our market share be?" but rather, "How high does our market share have to get before we turn a profit?" The trick here is to look for a **breakeven**, or cutoff, level for a parameter — that is, a target value of the parameter at which some particularly interesting event occurs, such as reaching zero profits or making a 15 percent return on invested assets. Managers who cannot predict market share can often determine whether a particular breakeven share is likely to occur. This is why breakeven analysis is so powerful.

Even if we have no idea of the market share for the new product, we should be able to build a model that calculates profit given some *assumption* about market share. Once market share takes the role of a parameter in our model, we can use the Data Sensitivity tool to construct a graph of profit as a function of market share. Then, from the graph, we can find the breakeven

market share quite accurately.

New capital investments are usually evaluated in terms of their net present value, but the appropriate discount rate to use is not always obvious. Rather than attempting to determine the appropriate discount rate precisely, we can take the breakeven approach and ask how high would the discount rate have to be in order for this project to have an NPV of zero? (The answer to this question is generally known as the internal rate of return.) If the answer is 28 percent, we can be confident that the project is a good investment. On the other hand, if breakeven occurs at 9 percent, we may want to do further research to establish whether the appropriate discount rate is clearly below this level.

Breakeven values for parameters can be determined manually, by repeatedly changing input values until the output reaches the desired target. This can often be done fairly quickly by an intelligent trial-and-error search in Excel. In the Advertising Budget example, suppose we want to find the breakeven cost to the nearest penny. Recall our example earlier, where we noted that profit goes to zero between a unit cost of $29 and a unit cost of $30. By repeating the search between these two costs in steps of $0.10, we can find the breakeven cost to the nearest dime. If we repeat the search once more, in steps of $0.01, we will obtain the value at the precision we seek.

However, Excel also provides a specialized tool called **Goal Seek** (in the Tools menu) for performing this type of search. The tool locates the desired unit cost as $29.36, and the corresponding calculations will be displayed on the spreadsheet.

Note that the Goal Seek tool searches for a prescribed level in the relation between a single output and a single input. Thus, it requires the parameter or decision being varied to reside in a single location, reinforcing one of our design principles.

3.4 Optimization analysis

Another fundamental type of managerial question takes the form of finding a set of decision variables that achieves the best possible value of an output. In fact, we might claim that the fundamental management task is to make choices that result in optimal outputs. **Solver** is an important tool for this purpose. Solver is an add-in for Excel that makes it possible to optimize models with multiple decision variables and possibly constraints on the choice of decision variables. Optimization is a complex subject, and we can only provide a glimpse of its power here by demonstrating a simple application in the Advertising Budget example.

Suppose we wish to maximize total profits with an advertising budget of $\$40,000$. We already know that, with equal expenditures in every quarter, annual profits come to $\$69,662$. The question now is whether we can achieve a higher level of annual profits. The answer is that a higher level is, in fact,

attainable. An optimal reallocation of the budget produces annual profits of $71,447. The chart in Figure 6 compares the allocation of the budget in the base case with the optimal allocation. As we can see, the optimal allocation calls for greater expenditures in quarters $Q2$ and $Q4$ and for smaller expenditures in $Q1$ and $Q3$.

This is just one illustration of Solver's power. Among the many questions we could answer with Solver in the Advertising Budget example are these:

- What would be the impact of a requirement to spend at least $8,000 each quarter?
- What would be the marginal impact of increasing the budget?
- What is the optimal budget size?

One way to answer this last question would be to run Solver with increasing budgets and trace out the impact on profit. We could do this manually, one run at a time, but it would be more convenient to be able to accomplish this task in one step. The Sensitivity Toolkit contains a tool called **Solver Sensitivity** that does just this: it runs Solver in a loop while varying one (or two) input parameters. Figure 7 shows the results of running Solver for advertising budgets from $40,000 to $100,000 in increments of $5,000. The table shows that profit increases at a decreasing rate as the budget increases, and beyond about $90,000 there is no discernible impact from additional budget. It also shows how the four decision variables ($Q1$–$Q4$ advertising) change as the budget changes.

3.5 Simulation and risk analysis

Uncertainty often plays an important role in analyzing a decision, because with uncertainty comes risk. Until now, we have been exploring the relationship between the inputs and outputs of a spreadsheet model as if uncertainty were not an issue. However, risk is an inherent feature of all managerial decisions, so it is frequently an important aspect of spreadsheet models. In particular, we might want to recognize that some of the inputs are subject to uncertainty. In other words, we might want to associate probability models with some of the parameters. When we take that step, it makes sense to look at outputs the same way — with probability models. The use of probability models in this context is known as **risk analysis**.

One tool for risk analysis in spreadsheets is Crystal Ball, an add-in for Monte Carlo simulation (another is @Risk [5]). This tool allows us to generate a probability distribution for any output cell in a spreadsheet, given probability assumptions about some of the input cells. Simulation and risk analysis are complex subjects. Here, we simply illustrate how Crystal Ball can help us answer an important question about risk.

In the Advertising Budget example, we return to the base case, with equal expenditures of $10,000 on advertising each quarter. Our base-case analysis,

which assumed that all parameters are known exactly, showed an annual profit of $69,662. However, we might wonder about the distribution of profits if there is uncertainty about the unit price and the unit cost. Future prices depend on the number of competitors in our market, and future costs depend on the availability of raw materials. Since both level of competition and raw material supply are uncertain, so, too, are the parameters for our price and cost. Suppose we assume that price is normally distributed with a mean of $40 and a standard deviation of $10, and that unit cost is equally likely to fall anywhere between $20 and $30. Given these assumptions, what is the probability distribution of annual profits? And how likely is it that profits will be *negative*?

Figure 8 shows the probability distribution for profits in the form of a histogram, derived from the assumptions we made about price and cost. The graph shows us that the estimated average profit is $69,721 under our assumptions. It also shows that the probability is about 30 percent that we will lose money. This exposure may cause us to reevaluate the desirability of the base-case plan.

Once again, we often wish to know how sensitive our simulation results are to one or more input parameters. This suggests running Crystal Ball in a loop while we vary the inputs, and to do so we have included the appropriate tool, called **CB Sensitivity**, in the Sensitivity Toolkit. Figure 9 shows the results of running Crystal Ball while we vary the budget from $40,000 to $100,000 in increments of $5,000, keepting advertising spending equal across the quarters. We plot here not only the mean profit, but the maximum and minimum values from each of the simulations, to give an idea of the range of outcomes likely at each step.

4 Research Issues

In contrast to software engineering, which has seen decades of development, spreadsheet engineering is in its infancy. Most of the ideas in this paper have been adapted from software engineering and tested informally in various instructional settings. However, there is little laboratory or field research to support claims that one or another spreadsheet engineering principle is effective in actual practice.

Spreadsheet engineers are fundamentally different from software engineers. Most of them would not describe themselves as programmers and most are not aware that they are under-trained for the spreadsheet design and analysis tasks they perform. Few recognize the risks they and their companies run when they use chaotic development processes or fail to use the powerful analytic tools described here.

The research needs are clear, although how best to carry out this kind of research is not. We need to know much more than we do about how spreadsheets are designed and used in industry. We also need to test various interventions,

including training programs and software add-ins, to see which really improve practice and which do not. We also need to study how corporate standards for training and use of spreadsheets influence the culture and performance of end-users. While spreadsheet programming has little cache in the computer science profession, it is likely that more computer programs are written by the millions of spreadsheet end-users than all professional programmers combined. The positive impacts of improving this aspect of programming practice are correspondingly high.

References

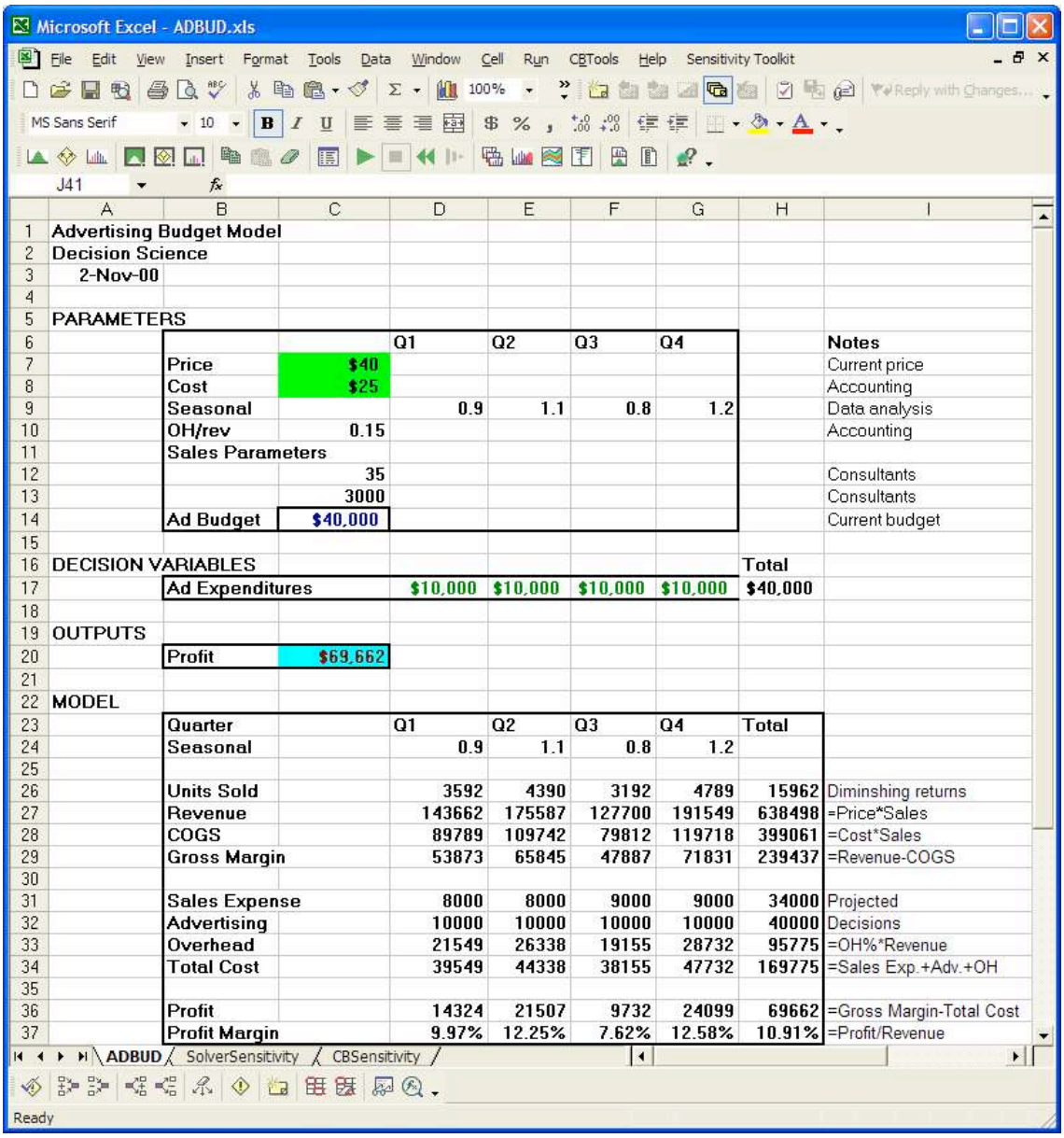
[1] Panko, Ray, What We Know About Spreadsheet Errors, *Journal of End User Computing*, Special issue on Scaling Up End User Development, Volume 10, No 2. Spring 1998, pp. 15-21.

[2] Powell, Stephen G. and Kenneth Baker, *The Art of Modeling with Spreadsheets*, New York: John Wiley, 2004.

[3] Solver: http://www.solver.com/

[4] Crystal Ball: http://www.decisioneering.com/crystal_ball/index.html/

[5] @Risk: http://www.palisade.com/html/risk.asp



	A	B	C	D	E	F	G	H	I
1	Advertising Budget Model								
2	Decision Science								
3	2-Nov-00								
4									
5	PARAMETERS								
6				Q1	Q2	Q3	Q4		Notes
7		Price	$40						Current price
8		Cost	$25						Accounting
9		Seasonal		0.9	1.1	0.8	1.2		Data analysis
10		OH/rev	0.15						Accounting
11		Sales Parameters							
12			35						Consultants
13			3000						Consultants
14		Ad Budget	$40,000						Current budget
15									
16	DECISION VARIABLES							Total	
17		Ad Expenditures		$10,000	$10,000	$10,000	$10,000	$40,000	
18									
19	OUTPUTS								
20		Profit	$69,662						
21									
22	MODEL								
23		Quarter		Q1	Q2	Q3	Q4	Total	
24		Seasonal		0.9	1.1	0.8	1.2		
25									
26		Units Sold		3592	4390	3192	4789	15962	Diminshing returns
27		Revenue		143662	175587	127700	191549	638498	=Price*Sales
28		COGS		89789	109742	79812	119718	399061	=Cost*Sales
29		Gross Margin		53873	65845	47887	71831	239437	=Revenue-COGS
30									
31		Sales Expense		8000	8000	9000	9000	34000	Projected
32		Advertising		10000	10000	10000	10000	40000	Decisions
33		Overhead		21549	26338	19155	28732	95775	=OH%*Revenue
34		Total Cost		39549	44338	38155	47732	169775	=Sales Exp.+Adv.+OH
35									
36		Profit		14324	21507	9732	24099	69662	=Gross Margin-Total Cost
37		Profit Margin		9.97%	12.25%	7.62%	12.58%	10.91%	=Profit/Revenue

Fig. 1. The advertising budget spreadsheet.

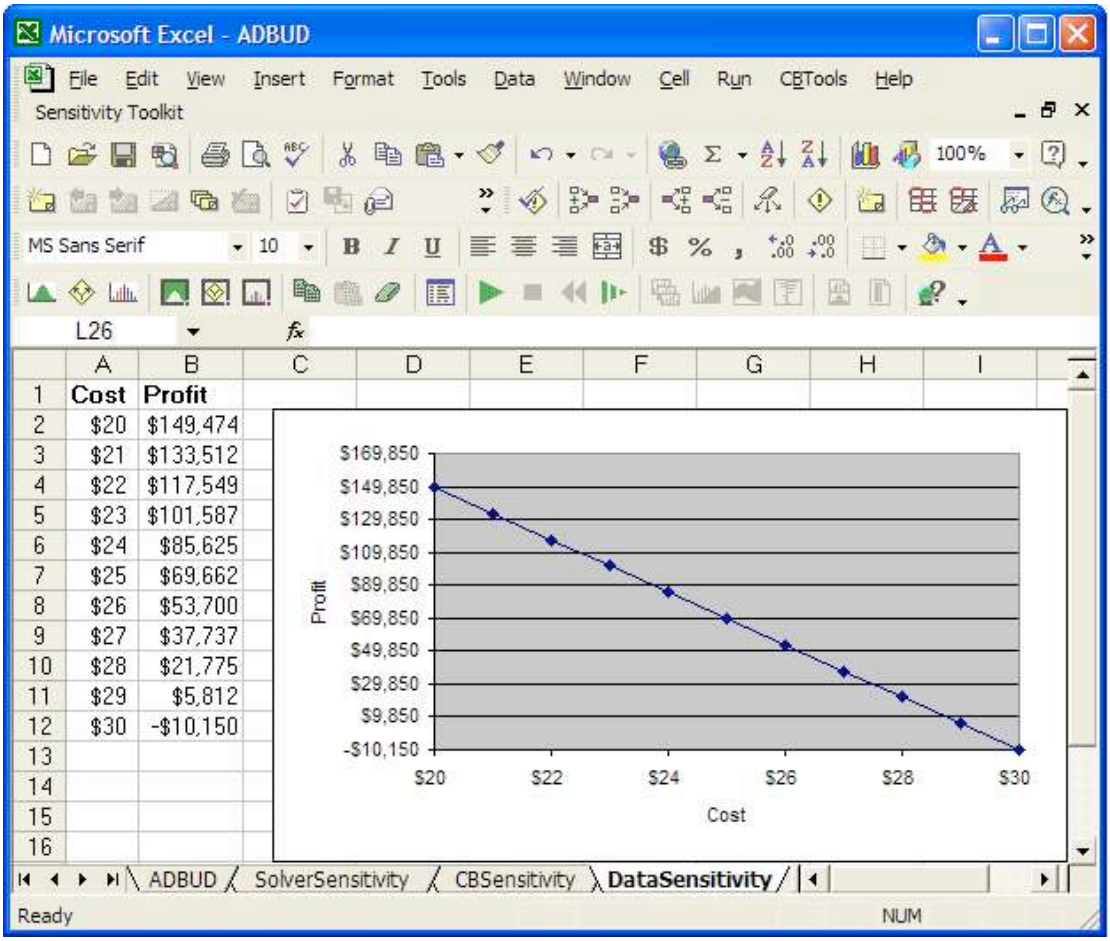


Fig. 2. A graph based on the Data Sensitivity.

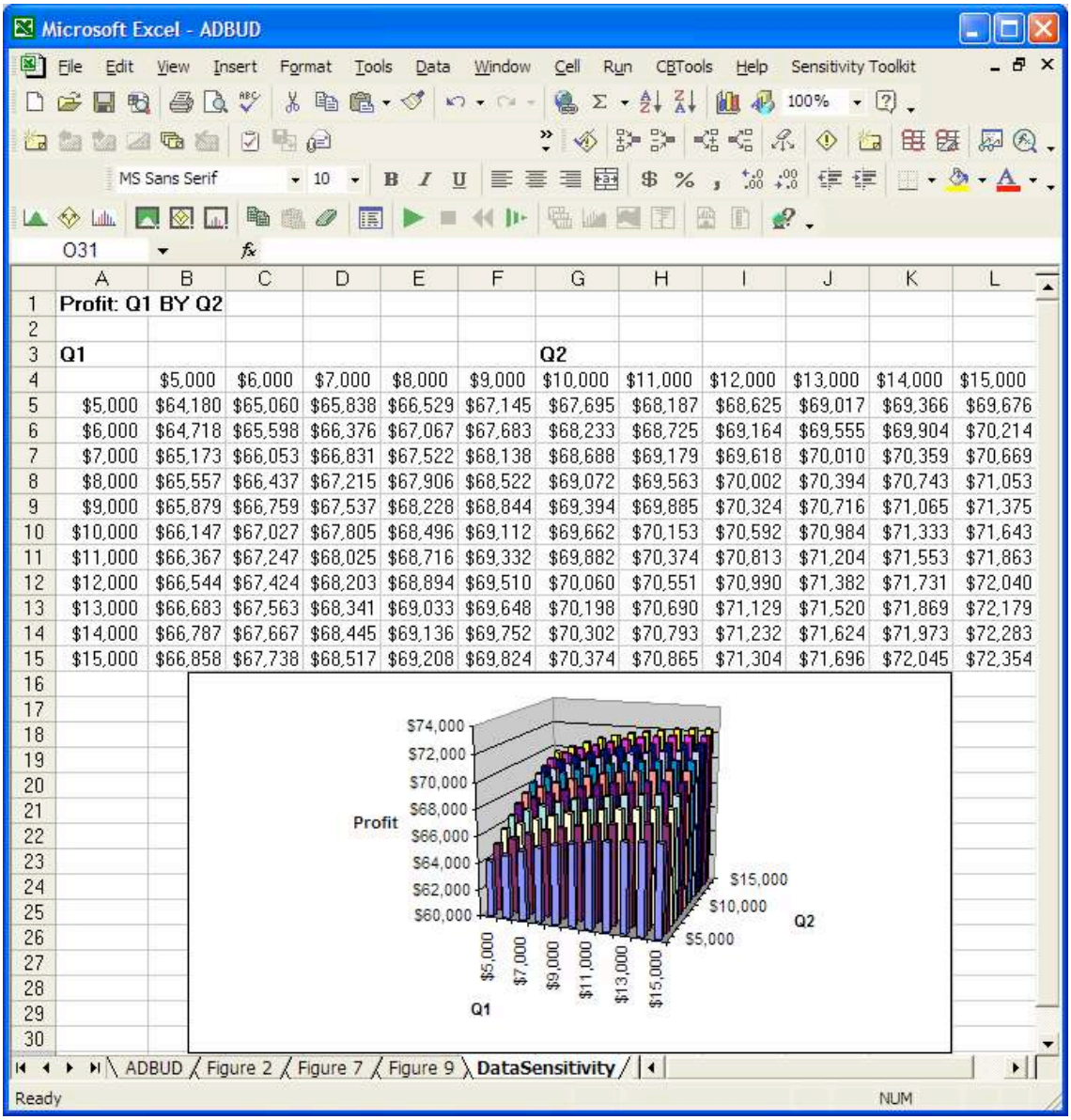
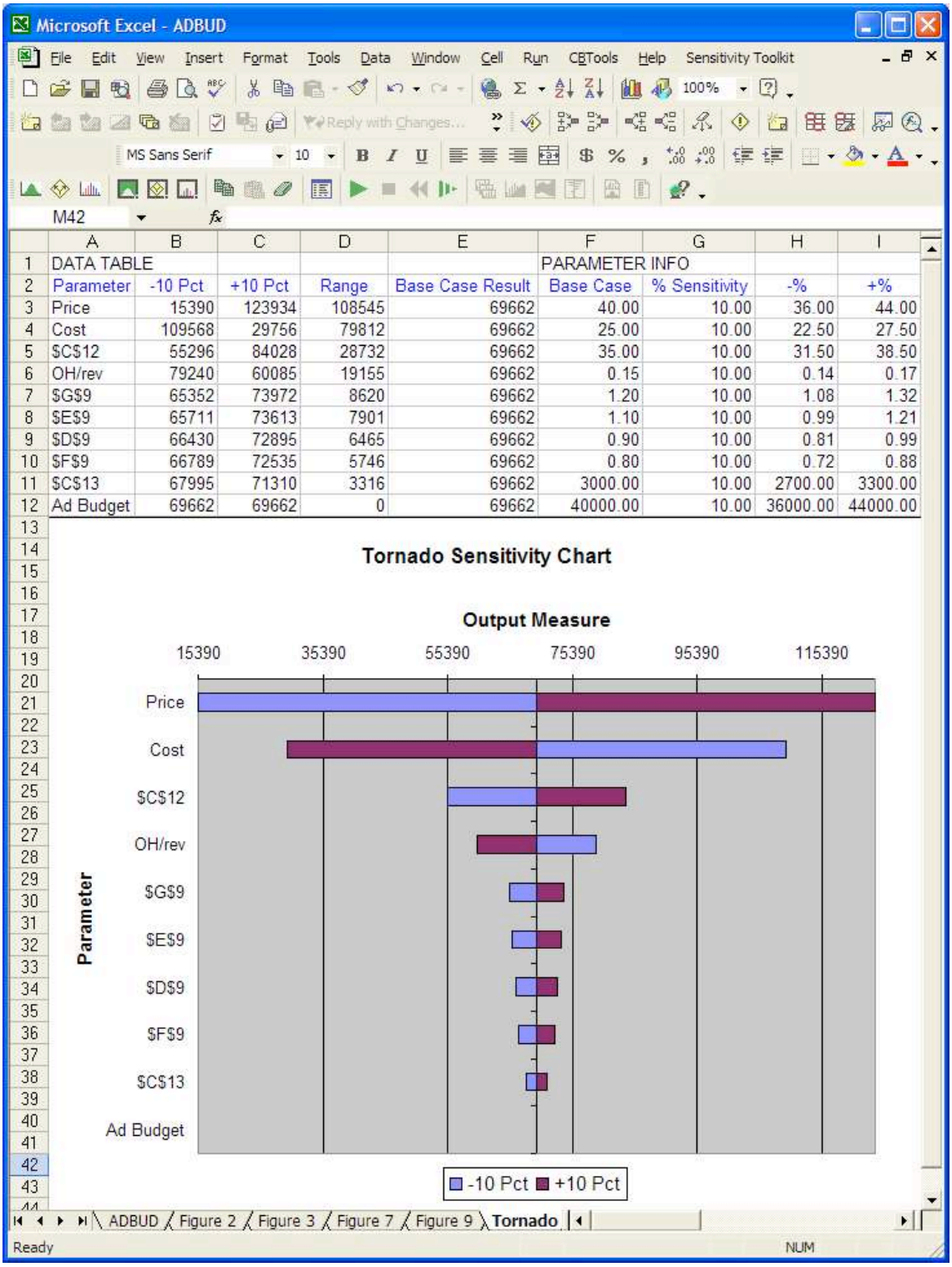
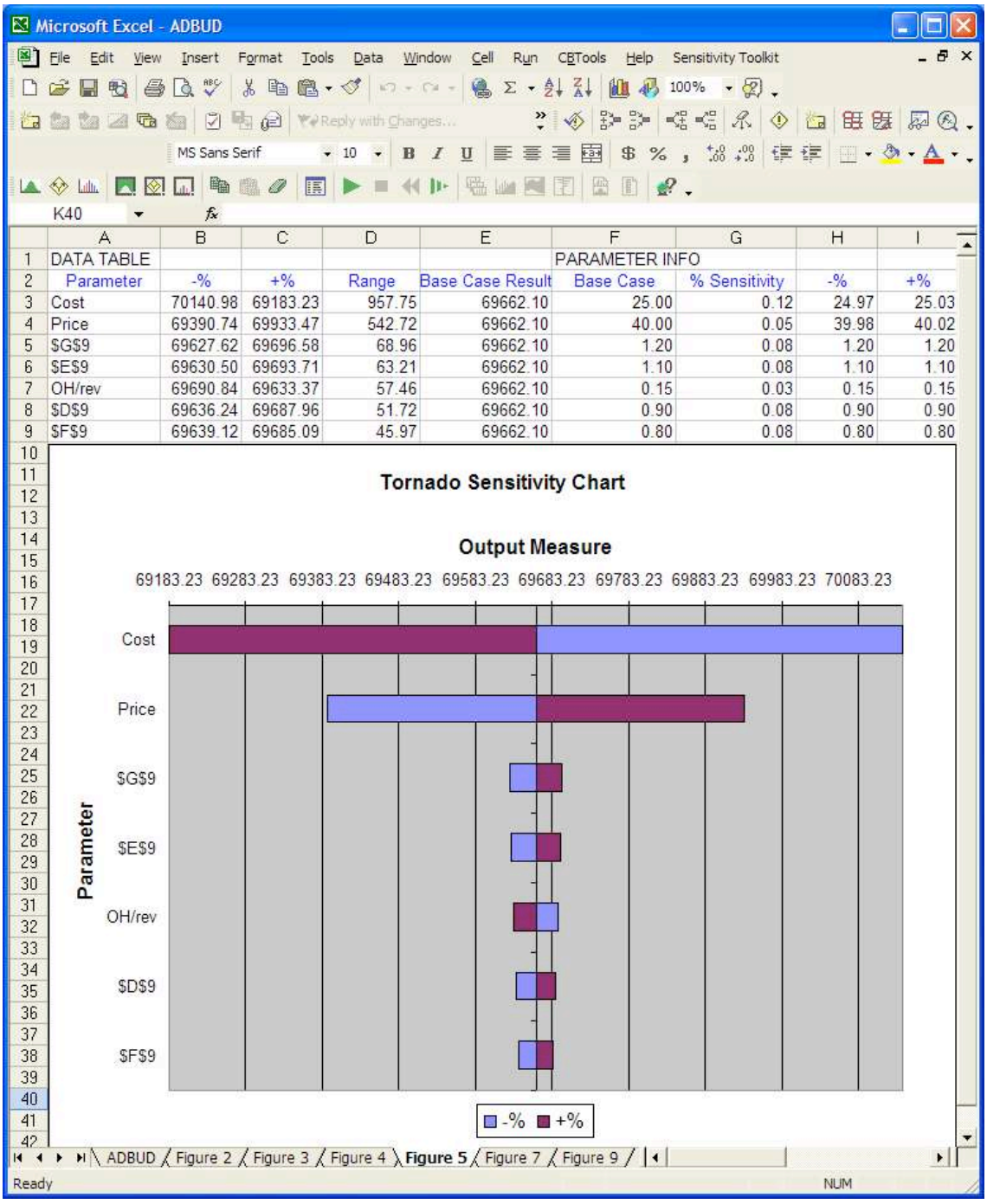


Fig. 3. Two-way Data Sensitivity: profit as a function of $Q1$ and $Q2$ advertising.



	A	B	C	D	E	F	G	H	I
1	DATA TABLE					PARAMETER INFO			
2	Parameter	-10 Pct	+10 Pct	Range	Base Case Result	Base Case	% Sensitivity	-%	+%
3	Price	15390	123934	108545	69662	40.00	10.00	36.00	44.00
4	Cost	109568	29756	79812	69662	25.00	10.00	22.50	27.50
5	C12	55296	84028	28732	69662	35.00	10.00	31.50	38.50
6	OH/rev	79240	60085	19155	69662	0.15	10.00	0.14	0.17
7	G9	65352	73972	8620	69662	1.20	10.00	1.08	1.32
8	E9	65711	73613	7901	69662	1.10	10.00	0.99	1.21
9	D9	66430	72895	6465	69662	0.90	10.00	0.81	0.99
10	F9	66789	72535	5746	69662	0.80	10.00	0.72	0.88
11	C13	67995	71310	3316	69662	3000.00	10.00	2700.00	3300.00
12	Ad Budget	69662	69662	0	69662	40000.00	10.00	36000.00	44000.00

Fig. 4. Tornado Chart using the Constant Percentage option.



	A	B	C	D	E	F	G	H	I
1	DATA TABLE					PARAMETER INFO			
2	Parameter	-%	+%	Range	Base Case Result	Base Case	% Sensitivity	-%	+%
3	Cost	70140.98	69183.23	957.75	69662.10	25.00	0.12	24.97	25.03
4	Price	69390.74	69933.47	542.72	69662.10	40.00	0.05	39.98	40.02
5	G9	69627.62	69696.58	68.96	69662.10	1.20	0.08	1.20	1.20
6	E9	69630.50	69693.71	63.21	69662.10	1.10	0.08	1.10	1.10
7	OH/rev	69690.84	69633.37	57.46	69662.10	0.15	0.03	0.15	0.15
8	D9	69636.24	69687.96	51.72	69662.10	0.90	0.08	0.90	0.90
9	F9	69639.12	69685.09	45.97	69662.10	0.80	0.08	0.80	0.80

Fig. 5. Tornado Chart using the Variable Percentage option.

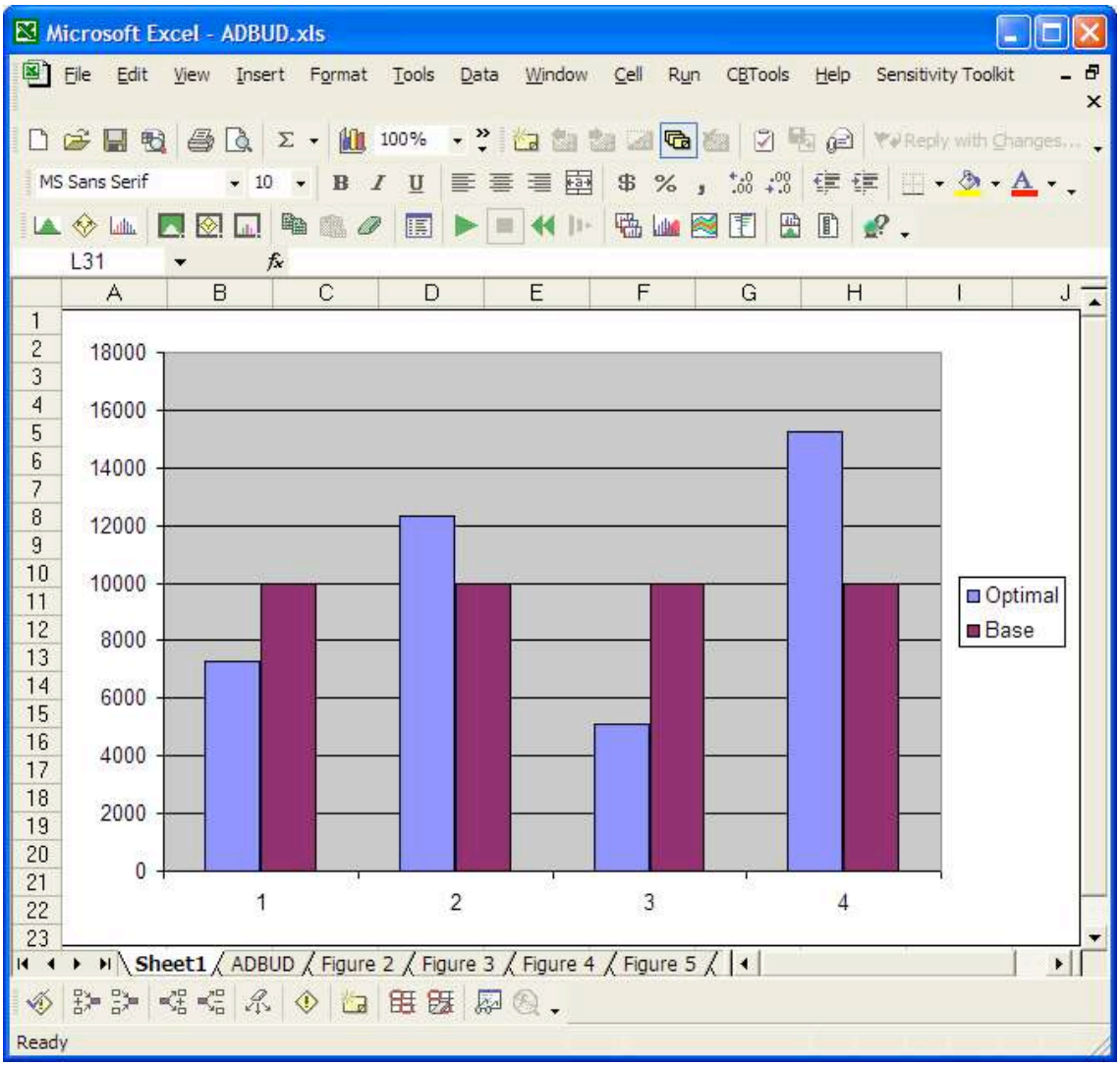
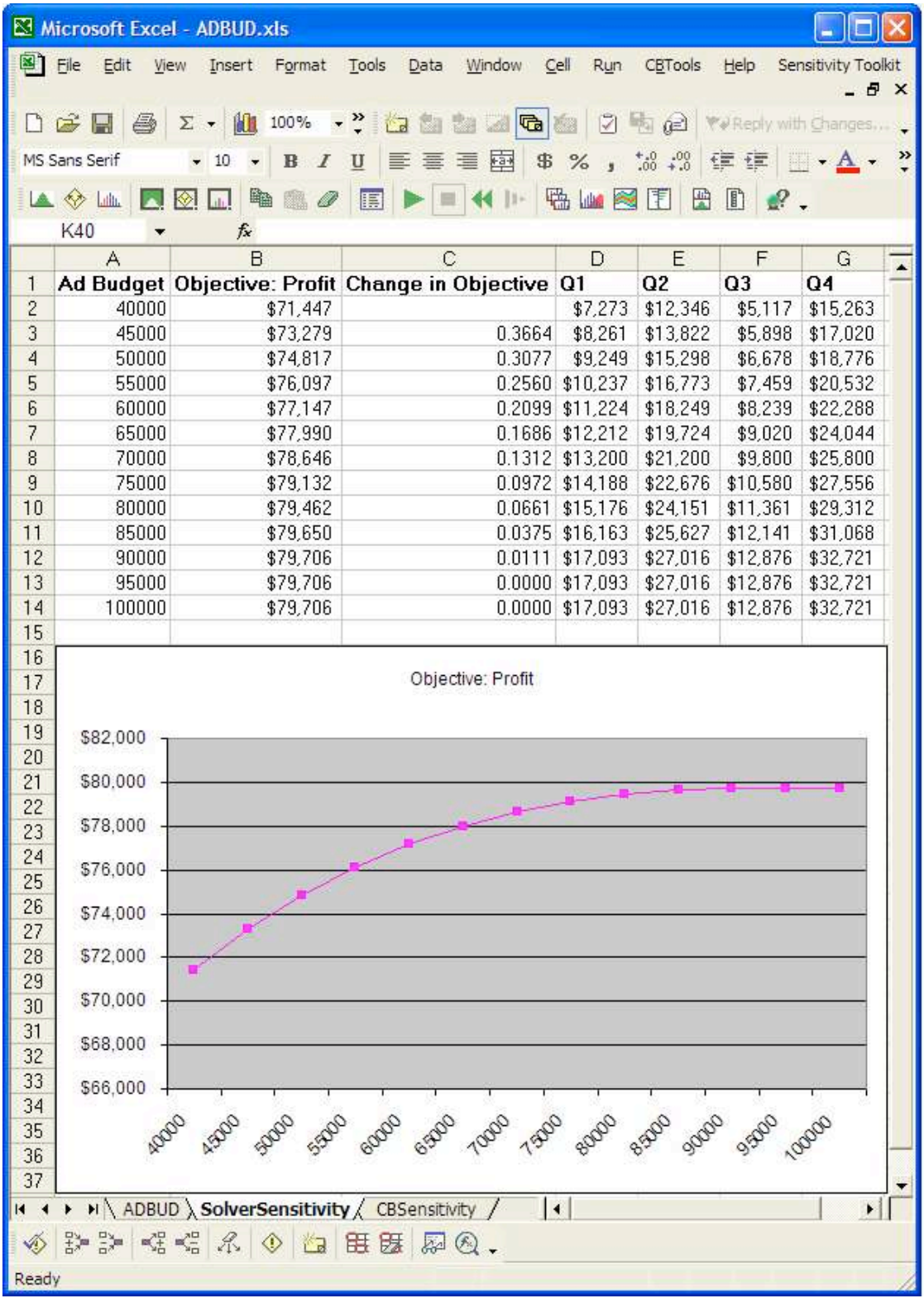


Fig. 6. Comparison of base-case and optimal allocations.



	A	B	C	D	E	F	G
1	**Ad Budget**	**Objective: Profit**	**Change in Objective**	**Q1**	**Q2**	**Q3**	**Q4**
2	40000	$71,447		$7,273	$12,346	$5,117	$15,263
3	45000	$73,279	0.3664	$8,261	$13,822	$5,898	$17,020
4	50000	$74,817	0.3077	$9,249	$15,298	$6,678	$18,776
5	55000	$76,097	0.2560	$10,237	$16,773	$7,459	$20,532
6	60000	$77,147	0.2099	$11,224	$18,249	$8,239	$22,288
7	65000	$77,990	0.1686	$12,212	$19,724	$9,020	$24,044
8	70000	$78,646	0.1312	$13,200	$21,200	$9,800	$25,800
9	75000	$79,132	0.0972	$14,188	$22,676	$10,580	$27,556
10	80000	$79,462	0.0661	$15,176	$24,151	$11,361	$29,312
11	85000	$79,650	0.0375	$16,163	$25,627	$12,141	$31,068
12	90000	$79,706	0.0111	$17,093	$27,016	$12,876	$32,721
13	95000	$79,706	0.0000	$17,093	$27,016	$12,876	$32,721
14	100000	$79,706	0.0000	$17,093	$27,016	$12,876	$32,721

Fig. 7. Results of Solver Sensitivity.

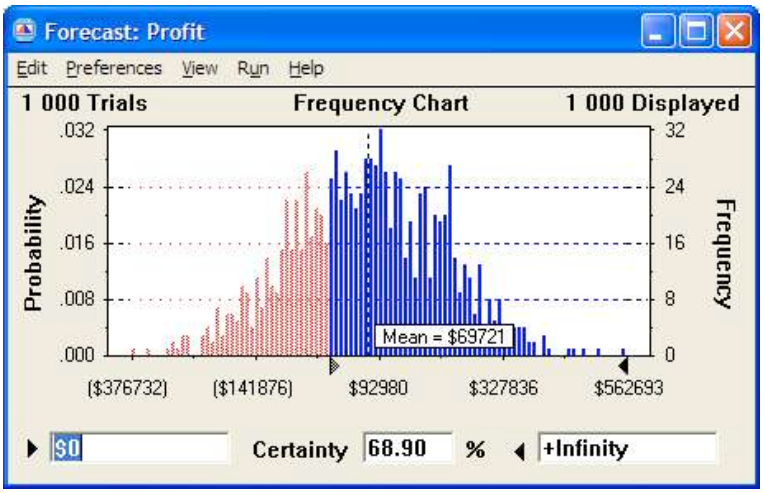
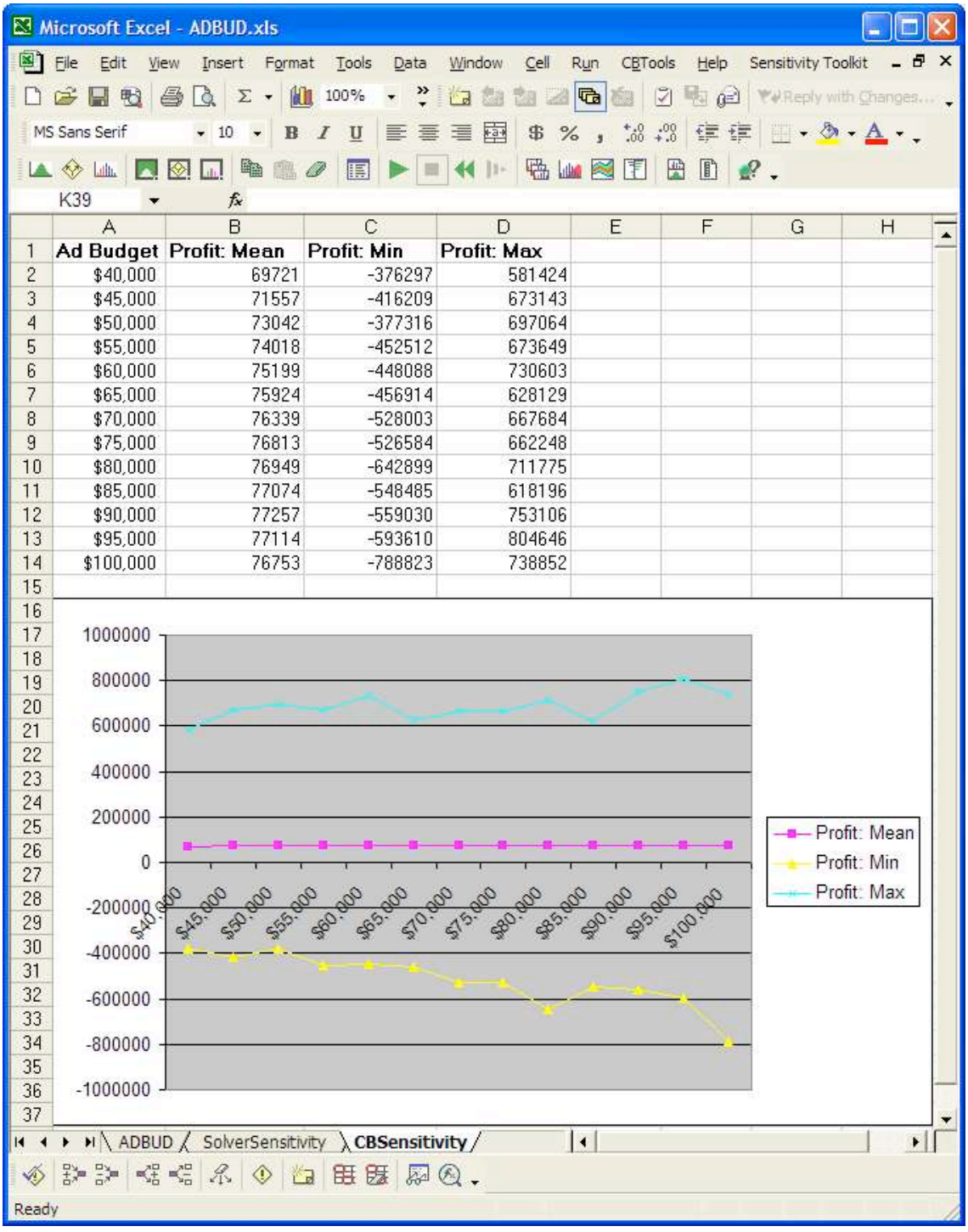


Fig. 8. Distribution of profits from the Advertising Budget example.



	A	B	C	D
1	Ad Budget	Profit: Mean	Profit: Min	Profit: Max
2	$40,000	69721	-376297	581424
3	$45,000	71557	-416209	673143
4	$50,000	73042	-377316	697064
5	$55,000	74018	-452512	673649
6	$60,000	75199	-448088	730603
7	$65,000	75924	-456914	628129
8	$70,000	76339	-528003	667684
9	$75,000	76813	-526584	662248
10	$80,000	76949	-642899	711775
11	$85,000	77074	-548485	618196
12	$90,000	77257	-559030	753106
13	$95,000	77114	-593610	804646
14	$100,000	76753	-788823	738852

Fig. 9. Results of CB Sensitivity.

Design, Implementation and Animation of Spreadsheets in the Lrc System

João Saraiva

Department of Computer Science,
University of Minho, Portugal.
e-mail: jas@di.uminho.pt

Abstract

This paper presents techniques for the design, implementation and animation of spreadsheet-like tools in the attribute grammar formalism. A real spreadsheet is formally specified and attribute grammar components that define user interfaces, querying languages and animations are plugged into the specification through higher-order attributes. From such a specification an incremental implementation is automatically derived and experimental results are presented.

Key words: Spreadsheets, Attribute Grammars, Incremental Evaluation, Functional Programming

1 Introduction

Attribute grammars (AG) [17] have proven to be a suitable formalism to the design and implementation of both domain specific and general purpose languages. In fact, powerful systems based on the attribute grammar formalism [23,13,2,9,20,18] have been constructed. These systems automatically produce very efficient/optimised implementations for languages specified via attribute grammars. While, in the beginning, AG-based systems were used mainly to specify and derive efficient (batch) compilers for formal languages, nowadays, AG-based systems are powerful tools that not only specify compilers, but also structured-based editors [23], programming environments [18], visual languages [15], complex pretty printing algorithms [28], program animations [20,25], etc. Furthermore, attribute grammars are also a suitable setting to express (circular) lazy programs [12,19,24,5], aspect oriented compilers [6], incremental algorithms [27], the XML technology [3], etc.

The purpose of this paper is three fold: first, to show that spreadsheet-like tools can be formally and concisely specified in the higher-order attribute grammar (HAG) formalism [30], and that well-known attribute grammar techniques can be used to reason about such formal specifications. For example, the AG circu-

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

larity test can be used to statically detect circularities in the specification which may induce the non-termination of the spreadsheet. In this paper we present the specification in the attribute grammar formalism of a student spreadsheet, its interactive interface, and the specification of querying language to query the student database. Second, to show that efficient incremental implementations can be automatically derived from an attribute grammar. Spreadsheets heavily rely on a incremental computation model since they have to provide immediate feedback after user interaction. To derive incremental implementations from AG (usually called attribute evaluators) we use the *Lrc* system: a purely functional attribute grammar-based system [18]. In *Lrc* efficient incremental evaluation is obtained via function memoisation. And, finally, to show that the visualisation and animation of the (incremental) execution of the spreadsheet can be obtained from the AG specification. Such animations provide a visual debugger which allows the user, for example, to easily understand how the incremental evaluation is performed.

This paper is organised as follows: Section 2 briefly describes higher-order attribute grammars and the *Lrc* system. Section 3 models a spreadsheet within the AG formalism. AG components for defining an interactive interface, a query language and a visualisation and animation are presented. Section 4 discusses the incremental implementations derived by the *Lrc* system from attribute grammar sprecifications and presents the results of the incremental behaviour of the spreadsheet. Section 5 presents the conclusions.

2 The *Lrc* Attribute Grammar based System

The techniques presented in this paper are based on the *higher-order attribute grammar* formalism [29]. Higher-order attribute grammars are an important extension to the attribute grammar formalism. Conventional attribute grammars are augmented with *higher-order attributes*, the so-called *attributable attributes*. Higher-order attributes are attributes whose value is a tree. We may associate, once again, attributes with such a tree. Attributes of these so-called *higher-order trees*, may be higher-order attributes again.

The *Lrc* system accepts as input a higher-order attribute grammar and generates purely functional implementations, the so-called attribute evaluators. *Lrc* generates both strict, multiple traversal attribute evaluators (C, HASKELL, and OCaml based attribute evaluators) and lazy attribute evaluators (expressed as circular lazy programs in HASKELL). Efficient incremental attribute evaluation is obtained via function memoization. The *Lrc* system not only produces batch tools (*e.g.*, compilers), but also programming environments. Such environments have a modern graphical user interface.

The higher-oder attribute grammar formalism and the *Lrc* system will be explained in detail in the next section where we present the HAG to specify a spreadsheet-like tool.

3 The Students Spreadsheet Attribute Grammar

Suppose that we have a (textual) database of students registered in one course. Each student (*i.e.*, register) has several attributes such as: identification number, name, and a list of marks (pairs containing the mark identification, and the value the student got). A possible (concrete) instance of the database is presented below.

```
1,"Ana",tm=16,p1=15,p2=17
2,"Eduardo",tm=12,p1=13,p2=15
3,"David",tm=16,p1=15,p2=17
```

The first register expresses that the student with number *1*, named *Ana* got the mark 16 as theoretical mark (tm), 15 in the first project (p1) and 17 in the second one (p2). This database/language is defined by the following context-free grammar. A production p is denoted as $X_0 = \mathrm{P}\ X_1\ \ldots\ X_n$, where the name of the production, *i.e.*, p, also indicates the term constructor function P. The type of the constructor function is $\mathrm{P} :: X_1 \to \cdots \to X_n \to X_0$ and we say that function P takes as arguments values of type $X_1 \cdots X_n$ and returns a value of type X_0. Roughly speaking, non-terminal symbols correspond to tree type constructors, and productions correspond to value constructors. We focus on the abstract structure of the language and we ignore its syntactic sugar, *e.g.*, punctuation symbols, etc.

$$
\begin{array}{lll}
\mathit{Students} & = & \textsc{ConsStuds}\ \mathit{Stud}\ \mathit{Students} \\
 & | & \textsc{NoStuds} \\
\mathit{Stud} & = & \textsc{OneStud}\ \ \mathit{Int}\ \mathit{String}\ [\mathit{Mark}] \\
\mathit{Mark} & = & \textsc{OneMark}\ \ \mathit{String}\ \mathit{Real}
\end{array}
$$

Fragment 1: The abstract grammar for the students database.

We represent lists of non-terminals by using both the usual functional notation and explicit constructors.

Having a database with the information about the students marks, the natural operations we would like to perform on that database are the mapping of a given formula through all the students in order to calculate, for example, their final classification. That is to say that we wish to construct a spreadsheet-like tool. To express a formula we consider a domain specific language (DSL) very much like the desk calculator language presented in [22]. A concrete example of a formula is as follows:

```
FinalMark = (tm + pm)/2
   where pm = (p1 + p2)/2
```

To define the formula that is applied to each student we have two possibilities:

- We may use a straightforward AG approach where the formula is defined as a semantic function, written in the declarative language used to express semantic functions in the AG formalism. As a result, this semantic function will not be analysed (to infer termination properties) nor optimised by AG techniques. In

this approach, the semantic function is part of the AG specification of the tool. As a result, the formula is processed statically and not dynamically. Thus, if we wish to use a different formula, the AG has to be modified, analysed and compiled in order to produced the desired tool.

- Or, we may use a key characteristic of higher-order attribute grammars: within higher-order attribute grammars (HAG) every inductive computation can be modeled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. Thus, we can model the formula, or, more precisely, that language of formulas as a higher-order attribute of our spreadsheet. That is, we extend our context-free grammar with new symbols and productions to define the language of formulas. Then, we associate inerited and synthesised attributes to its symbols to move context information to the formula sub-expressions and to synthesise the result of the formula.

  In the context of the specification of the student spreadsheet, this is done as follows: first we define a grammar describing the (abstract) structure of the language of formulas and we extend it with attributes and equations. We introduce an inherited attribute to pass the list of marks as the "argument" of the formula, and a synthesised attribute to deliver the result of "applying" the formula to its inherited/argument. After that we have to "apply" such "function" in the context of every student. To do this, we just introduce a higher-order attribute to represent the abstract formula, we instantiate its inherited attribute (with the marks of a particular student) and we use its synthesised result. So, we define a DSL for formulas as a sub-language of our spreadsheet. Note that in this case the formula is processed dynamically since it is part of the input sentence. As a result, if the spreadsheet user wishes to change the formula, he just changes the part of the input sentence where the formula is defined.

Next we discuss in detail how this latter approach can be implemented in the *Lrc* system. The structure of the spreadsheet is defined through the following grammar, where non-terminal *Students* is defined in Fragment 1, and non-terminal *Formula* represents the (abstract structure) of the formula under consideration.

SpreadSheet = ROOTPROD *Formula Students*

Formula = ONEFORM *Exp Decls*

Fragment 2: The abstract grammar for the Spreadsheet-like language.

Let us assume that we have an off-the-shelf AG component whose root non-terminal is *Formula*. This non-terminal has one inherited attribute (representing a finite function mapping variable names to values) and it synthesises one attribute with the value expressed by the formula. Synthesised (inherited) attributes are prefixed with the up (down) arrow $\uparrow$ ($\downarrow$).

$$Formula <\downarrow env : Env, \uparrow res : Real >$$

We omit here the attribute declaration and equations of the (trivial) definition

of this component. We shall consider that this AG component is included in our specification in order to create a monolithic AG, which is then analysed and the respective implementation derived.

In order to map the formula through the database of students, we have to move the (abstract) formula to the context of every student. Thus, we use one characteristic of the HAG, the so-called *syntactic references*, meaning that the abstract tree can be used directly as a value within a semantic equation. In our example the "syntactic" symbol *Formula* is used within an equation as follows:

SpreadSheet = ROOTPROD *Formula Students*

Students.form = *Formula*

Fragment 3: Passing the formula to the students.

Instead of defining attributes and equations to move the (abstract) formula downwards in the tree (*i.e.*, the student list) via trivial *copy rules*, we use a special notation to access a remote attribute (up in the tree). The expression {*Students.form*} refers to the local attribute *form* at the non-terminal *Students* [23,10] [1] .

Now that we are able to access the formula in the desired context, *i.e.*, in production ONESTUD, we use a higher-order attribute to model the semantic function that "applies" the formula to the list of marks of a student. Note that the inherited attribute *form* is a higher-order attribute: it is a (higher-order) tree that has attributes as well. In order to access those attributes we have to use the higher-order extension to the AG formalism. This is done as follows: first, we declare a higher-order attribute, *i.e.*, attributable attribute (*ata*) named *form* of type *Formula*, to represent the formula. Then, we instantiate this attribute with the inherited global formula. After that, we instantiate the inherited attribute *env* of that *ata* with the list of marks of the student (and we use a syntactic reference once again). And finally, we access the synthesised value of the formula. We use a local attribute (*finalMark*) to store the computed value. In the higher-order attribute grammar notation this is expressed as follows:

Stud = ONESTUD *Int Name* [*Mark*]

ata *form* : *Formula* -- Declaration of the ata

local *finalMark* : *Real* -- Declaration of a local attr.

form = {*Students.form*} -- Instantiation of the ata

form.env = [*Mark*] -- Instantiation of the inherited attr.

finalMark = *form.res* -- Use of the synthesised attr.

Fragment 4: The formula as a higher-order attribute.

Before we proceed, let us compare this higher-order style of defining such computations with the classical attribute grammar one. In the classical style we could

[1] See [16,28] for a survey of special notation for common attribute propagation patterns.

express this inductive computation by defining a semantic function that accepts as arguments the abstract representation of the formula and the list of marks of the students. It delivers the result of applying the formula to the list of marks. We can write it as follows:

```
Stud = OneStud Int Name [Mark]
    finalMark = evalFormula({Students.form}, [Mark])
```

Fragment 5: The formula as a semantic function.

where *evalFormula* is the inductive function. This function has to be defined and included in the AG specification. If this semantic function is semantically equivalent to the formula AG component, then, the previous two AG fragment are semantically equivalent as well. Furthermore, this approach also supports the dynamic update of the formula, since its representation is an argument of the semantic function. But, there are two important differences between these two approaches: while in the higher-order one, the AG techniques analyse the attribute dependencies, check for termination properties and, if no circularities are induced, finally, produce an optimised implementation. In the classical attribute grammar approach, the function is simply translated to the output without any analysis nor optimisation. Thus, it can cause the non-termination of the attribute evaluator, and consequently the non-termination of the spreadsheet!

Spreadsheets are usually displayed in a table-like representation. In [28] we have presented a generic, off-the-shelf pretty-printing AG component that can be plugged into our students spreadsheet so as to obtain the desired representation [2]. This AG component is based on a processor for HTML style tables. It computes a pretty-printed textual table from a HTML (table) text. More recently we have extended our original AG in order to synthesise a LaTeX, a XML, a VRML, and HTML table representation. Thus, we have a representation for our abstract tables in all these concrete languages. Figure 1 displays the ascii representation of a pretty printed table.

3.1 The Specification of the Spreadsheet Programming Environment

As it was previously stated, types can be defined within the attribute grammar formalism via non-terminal symbols. So, we may use this approach to introduce a type that defines an abstract representation of the interface of spreadsheet-like tools (or more generally, language-based tools). In other words, we use an abstract context-free grammar to define an abstract interface. The productions (or constructors) of such a grammar represent "standard" graphical user interface objects, like menus, buttons, list boxes, pull donw menus, etc. Next, we present the so-called *Lrc abstract interface grammar*.

[2] Actually, attribute grammar systems provide a special domain specific language (or, in other words, a fixed number of combinators) to pretty-print the syntax tree (usually called *unparsing rules*).

Visuals = CVISUALS [*Toplevel*]

Toplevel = TOPLEVEL *Frame String String*

Frame = LABEL *String*
| LISTBOX *Entrylist*
| PULLDOWNMENU *String MenuList*
| PUSHBUTTON *String*
| UNPARSE *Ptr*
| HLIST [*Frame*]
| VLIST [*Frame*]

Fragment 6: The *Lrc* abstract interface grammar.

The non-terminal *Visual* defines the type of the abstract interface of the tool: it is a list of TOPLEVEL objects, that may be displayed in different windows. A TOPLEVEL construct displays a frame in a window. It has three arguments: the frame, a name (for future references) and the window title. The productions applied to non-terminal *Frame* define concrete visual objects. For example, production PUSHBUTTON represents a *push-button*, LISTBOX represents a *list box*, etc.

The production UNPARSE represents a visual object that provides *structured text editing* [23]. It displays a pretty-printed version of its (tree) argument and allows the user to interact with it. Such beautified textual representation of the abstract syntax tree is produced according to the unparse rules specified in the grammar. It also allows the user to point to the textual representation to edit it (via the keyboard), or to transform it using user defined transformations. The productions VLIST and HLIST define combinators: they vertically and horizontally (respectively) combine visual objects into more complicated ones. These non-terminals and productions can be directly used in the attribute grammar to define the interface of the environments. Thus, the interface of the spreadsheets is specified through attribution, *i.e.*, within the AG formalism.

To define a concrete interface, we need, as we have said above, to define the mapping from the abstract interface representation into a concrete one. Instead of defining a concrete interface from scratch, we synthesise a concrete interface for a existing GUI toolkit, *e.g.*, the TCL/TK GUI toolkit [21]. Indeed, this GUI component synthesises TCL/TK code defining the interface in the attribute named *tk*.

Next, we present an attribute grammar fragment that glues the spreadsheet HAG with this graphical user interface attribute grammar component. It defines an interactive interface consisting of two visual objects that are vertically combined, namely: a push-button and the unparsing of the input under consideration. The root symbol *Spreadsheet* synthesises the TCL/TK concrete code in the attribute occurrence *concreteInterface*.

```
SpreadSheet    <↑ concreteInterface : Tk >
SpreadSheet = RootProd Formula Students
              ata absInterface : Visuals
              absInterface = let { button = PushButton "XML"
                                   editor = Unparse & SpreadSheet
                                   comb   = VList [ button , editor ]
                                 } in [ Toplevel comb "edit" "Students Editor" ]
              SpreadSheet.concreteInterface = absInterface.tk
```

Fragment 7: The Spreadsheet graphical user interface.

Figure 1 displays a snapshot of the students spreadsheet produced by *Lrc* from a AG specified using the techniques presented in this paper. In the background, a frame contains a syntax-editor to edit the pretty-printed formula (actually, we use a list of formulas) and the student database. In this syntax-editor, the user can point to a formula and dynamically change it. The user can also point to a particular student and select it through a mouse button. Then, the information of the student is displayed in a new window, where it can be easily updated. All of these actions are modelled as (abstract syntax) tree transformations, since the *Lrc* system maintains an abstract syntax tree to represent the input under consideration. Indeed, the (pretty printed) text displayed in the environments, corresponds to the textual view of such a tree. The *Lrc* system uses incremental attribute evaluation, in order to provide immediate real-time feedback, after a user action. This will be discussed in Section 4.

3.2 Querying the Spreadsheet Through Attribute Grammars

Having specified the students database, the formula to compute their final marks and how such formula is applied to each of the students, we may wish to compute which students got good or bad results. Or, we may wish to compute the students that have a final mark greater than a given number. In other words, we would like to have some mechanism to be able to query our database.

Rather than defining a particular query language for our student database, we want to define a generic query language that can not only be used for querying this particular example, but also to query any other textual database defined within the AG formalism. That is to say that we want to define a domain specific language that can be easily embedded in any AG specification. In order to not introduce yet-another querying language, we will consider the *XQuery* language: a typed, functional language for querying *Xml*, currently being designed by the *Xml* Query Working Group of the World-Wide Web Consortium [7,31]. We choose *XQuery* for two reasons: first, because attribute grammars and *Xml* technology are closely related [4] (both extend the context-free grammar formalism), thus *XQuery* is indeed

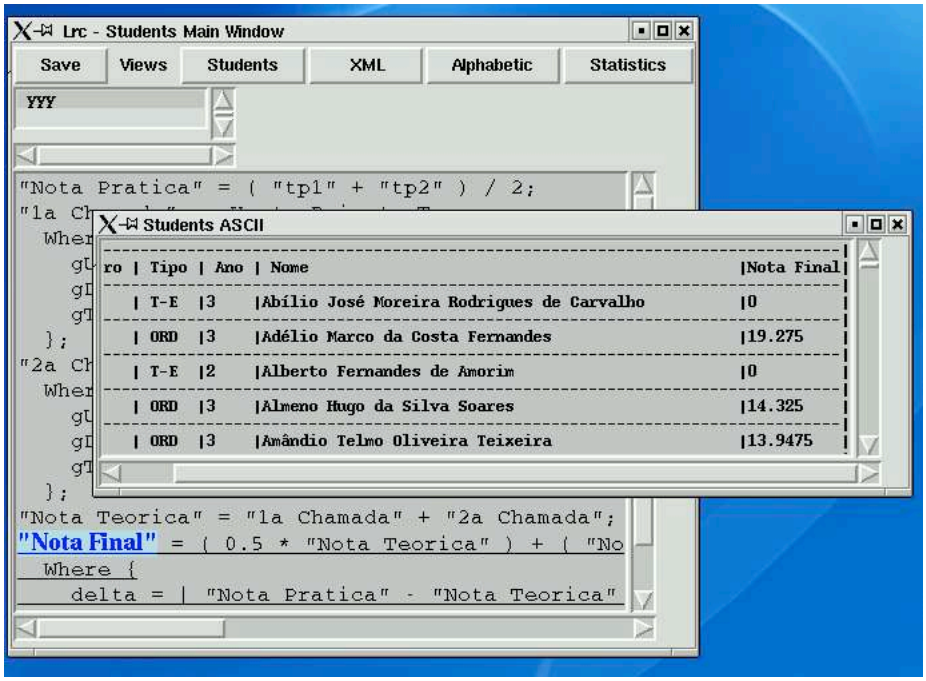


Fig. 1. The spreadsheet programming environment produced by the *Lrc* system from the students spreadsheet higher-order attribute grammar.

a suitable declarative language to express queries on AG-based language specifications. Second, an *Xml* combinator library to map abstract grammars (or trees) into *Xml* documents (or *Xml* trees) is already defined (via attribute grammars) in *Lrc*. Thus, we may re-use such a library, firstly to map the abstract grammar of the language under consideration to an *Xml* document, and then to query that *Xml* document.

Before we briefly explain the *XQuery* language, let us present a fragment of the abstract grammar defining the structure of an *Xml* document.

$$
\begin{array}{lcll}
\mathit{Document} & = & \textsc{CDocument} & \mathit{Prolog}\ \ \mathit{Miscs}\ \ \mathit{Element} \\
\mathit{Element} & = & \textsc{CElem} & \mathit{Name}\ \ [\,\mathit{Attribute}\,]\ \ [\,\mathit{Content}\,] \\
\mathit{Content} & = & \textsc{CElement} & \mathit{Element} \\
 & | & \textsc{CString} & \mathit{CharData}
\end{array}
$$

Fragment 8: The abstract grammar defining *Xml* documents.

XQuery uses *path expressions* that is a mechanism very much like the Unix notation to define paths on its file system. Instead of using the directory names, however, it uses the tags contained in an *Xml* document to indicate the path. Such tags correspond to the production names (or constructors) of the attribute grammar.

To introduce *XQuery*, let us consider some example queries on our student database. To list the students registered in the course we have to write the following

simple *XQuery* sentence (see non-terminals and productions on fragments 1 and 2):

```
RootProd/Students
```

To list the students that got a final mark greater than `13` we can write the following query:

```
RootProd/Students/OneStud[//OneMark/@FinalMark.>.13]
```

This query selects the element `OneStud` (or the subtree constructed with constructor OneStud), that contains as descendant (the double slash `//` means that the tagged element can be a direct or an indirected descendant) an element `OneMark` where the attribute `FinalMark` is defined and it is greater than `13`.

To model *XQuery* in the HAG formalism we start by defining the (abstract) structure of this language via the following abstract grammar:

Query	=	ProdQuery	*AQuery*
AQuery	=	CurrentContext	*TQuery*
	|	RootContext	*TQuery*
	|	DeepContextFromRoot	*TQuery*
TQuery	=	ProdBracketTQ	*TQuery XQuery*
	|	ProdTag	*XQuery*

Fragment 9: The abstract grammar defining the *XQuery* language.

We extend this grammar with attributes and equations in order to synthesise the desired information, that is to say, the answer to the query under consideration. The result of a query is another *Xml* document that contains the elements of the original document that answers the query. Thus, to perform a query is to evaluate a function, say *query*, that takes the query and the *Xml* document as arguments and returns another *Xml* document. In our setting, such a function has type:

$$query :: AQuery \rightarrow Document \rightarrow Document$$

This is, once again, an inductive function that can be efficiently defined within the style of higher-order attribute grammar programming. We omit here the definition of the attributes and respective equations since they are not relevant to understand our technique nor to re-use such a query language AG component.

Now that we have introduced this generic query component, we can embed it in any attribute grammar specification. For example, we can embed it in a bibliographic database processor (*e.g.*, *BibTeX* [24]) to list the books written by a given author. Or, we can embed it in our spreadsheet specification in order to extend the spreadsheet environment shown in Figure 1 with a powerful querying language. Figure 2 displays a new window (automatically) included in the spreadsheet that

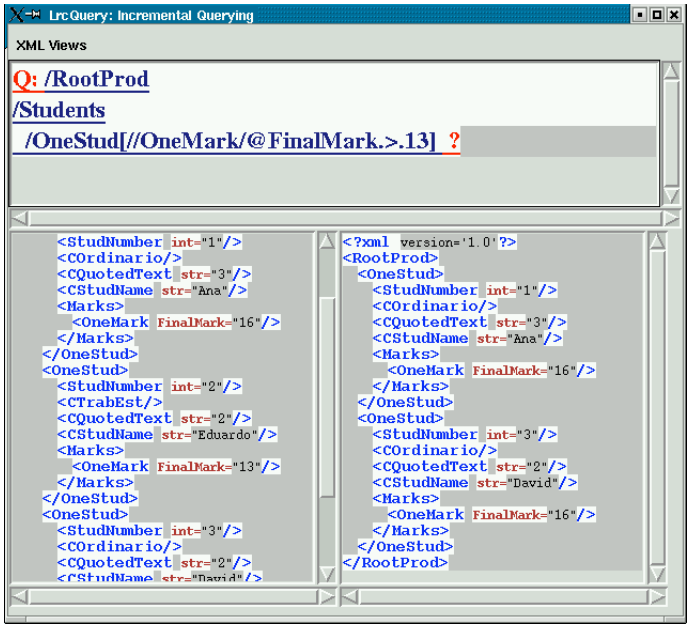


Fig. 2. Querying the students spreadsheet: the top frame displays the beautified query that is performed on the *Xml* representation of the database (frame on the left). The answer to the query is displayed as a *Xml* document on the right frame..

provides the user with a syntax editor to interactively query the student database. The query being displayed is the example query discussed above. Obviously, the user can dynamically modify this query and, in this case, the spreadsheet will incrementally compute the answer.

3.3 Visualisation and Animation of Spreadsheets

Attribute grammar-base systems statically schedule the computations, and, automatically generate implementations for the AGs - the so-called attribute evaluators - based on the dependencies induced by the attribution rules. Such evaluators are usually implemented as *tree-walk evaluators*: a function that walks over the abstract tree (representing the input under consideration) while computing attribute values (this task is usually called tree decoration). Thus, attribute grammars can be visualised by displaying the abstract tree in a graphical representation, and animated by displaying the tree decoration process in that tree (for example, by marking in the graphical representation the tree node being visited and by displaying the attribute values being computed).

Thus, we introduce a generic component for the visualisation and animation of AGs. We wish to use this AG as a *generic visual and animation AG component*. We start by defining an abstract grammar that is sufficiently generic to define all possible abstract tree structures we may want to visualise and animate. The grammar is as follows:

```
TreeViz   = CTreeViz     TreeId [TreeStmt]
TreeStmt  = CStmtNode    NodeStmt
          | CStmtEdge    EdgeStmt
          | CStmtAttr    AttrStmt
NodeStmt  = CNodeStmt    NodeId [Attr]
EdgeStmt  = CEdgeStmt    NodeId [EdgeRHS] Attrs
EdgeRHS   = CRHSExpNode  EdgeOp NodeId
Attr      = CAttr        AttrId AttrVal
```

The non-terminals *TreeId*, *NodeId*, *EdgeOp*, *AttrId*, *AttrVal* define sequences of characters (strings). In order to make it easier to use this component, we define a set of functions/macros that, using the productions of this AG component, define usual occurring node formats in our trees. Next, we present four functions that define the shape of a node as a record (*attrShapeRecord*), as a circle (*attrShapeCircle*), as the value of a node label (*attrLabel*), and, finally, as a node that contains a value and an arrow to a child node. These functions are presented next.

```
attrShapeRecord                = CAttr "shape" "record"
attrShapeCircle                = CAttr "shape" "circle"
attrLabel label                = CAttr "label" label
nodeRecord1 val father child   =
  [CStmtNode (CNodeStmt father) [attrShapeRecord , attrLabel (val ++ "|<c>")]
  ,CStmtEdge (CEdgeStmt "c") [CRHSExpNode "->" child] ]
```

The label is a string that defines the format of the node record. The non-terminal *EdgeOp*is a string defining the direction of the arrow.

The above grammar defines the abstract structure of abstract trees only. To have a concrete graphical representation of the trees, however, we need to map such abstract tree representation into a concrete one. Rather than defining a concrete interface from scratch and implementing a tree/graph visualization system (and reinventing the wheel!), we can synthesise a concrete interface for existing high quality graph visualization systems, *e.g.*, the GraphViz system [8]. We omit here again attributes and attribution rules that we have associated to the visualization grammar since they are neither relevant to reuse this component nor to understand our techniques.

This grammar component is context-free (it does not have any inherited attributes) and synthesises two attributes *graphviz* and *xml*, both of type string. These two attributes synthesise a textual representation of trees in the GraphViz input language. The first attribute displays trees in the usual graphic tree representation, while the second one uses a Xml tree-like representation (where the production names are the element tags).

```
TreeViz  <↑ graphviz : String, ↑ xml : String >
```

We are now in position to "glue" this component to the spreadsheet AG. Let us start by defining the attribute and the equations that specify the construction of the GraphViz representation.

```
Students    <↑ viztree : [TreeStmt] >
Students = NoStuds
           Students.viztree = nodeEmptyCircle treeRef(Students)
         | ConsStuds Stud Students
           Students1.viztree = (nodeRecord2 "" (treeRef Students1) (treeRef Stud)
                               (treeRef Students2)) ++ Stud.viztree ++ Students2.viztree
Stud        <↑ viztree : [TreeStmt] >
Stud     = OneStud  Int String [Mark]
           Stud.viztree = nodeRecord1 ("Number : " ++ (int2str Int)) (treeRef [Marks])
```

Fragment 10: Constructing the Visual Tree.

Where the function *treeRef* returns a unique identifier of its tree-value argument (the tree pointer). Next, we declare a higher-order attribute, *i.e.*, attributable attribute (*ata*) named *visualTree*, in the context of the single production applied to the root non-terminal of the spreadsheet AG. The HAG fragment looks has follows:

```
Spreadsheet    <↑ String : visualTree >
Spreadsheet = RootProd Formula Students
              ata visualTree : TreeViz
              visualTree              = CTreeViz "Tree" (Formula.viztree
                                          ++ Students.viztree)
              Spreadshhet.visualTree = visualTree.graphviz
```

Figure 3 shows two different snapshots (displayed by GraphViz) of the tree that is obtained as the result of running the spreadsheet with the example sentence. As we can see the tree is displayed as a *Direct Acyclic Graphs*. This happens because we are using the incremental model of attribute evaluation of *Lrc*. We will return to this subject in the next section.

Besides computing the graphical representation of the tree, the processor generated by *Lrc* also produces a sequence of node transitions. This is exactly the sequence of visits the evaluator performs to decorate the tree under consideration. Such sequence can be loaded in and animated in GraphViz, either in single step or in continuous mode, forwards and backwards. Different colors (or different shadow intensities in a black and white printing) are used to identify the number of times

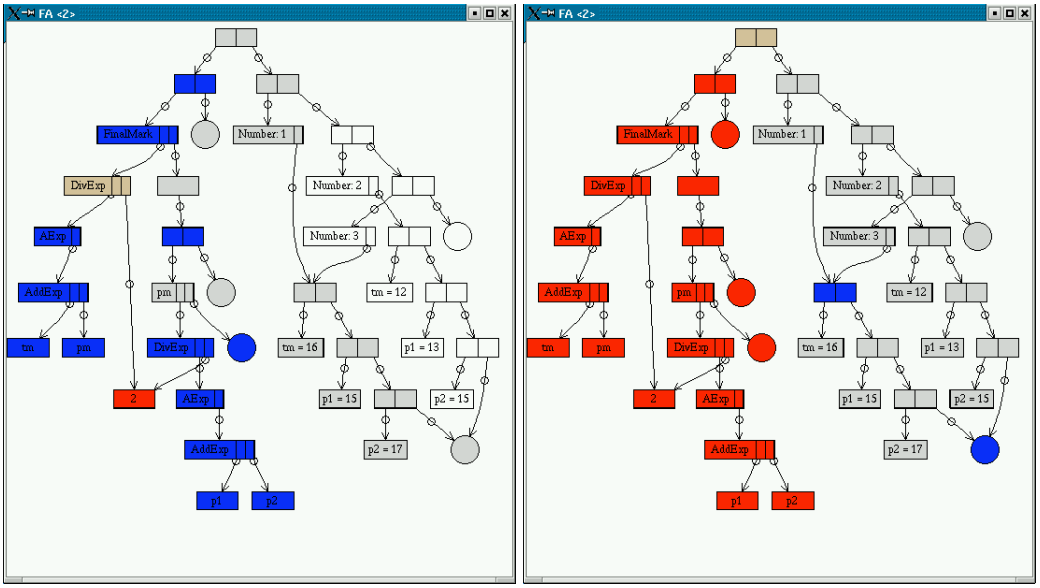


Fig. 3. The DAG representing the example sentence after processing the first student (left) and after attribute evaluation (right).

the nodes have been visited. Thus, light gray means a single visit, gray (or color blue) meand two visits and drak gray (or color red) means 3 or more visist.

The snapshot on the left shows the attribute evaluator when processing the first student. The shadowed nodes are the nodes that have already been visited. The snapshot on the right shows the tree after attribute evaluation. As we can see, the (shared) list of marks of students number 1 and 3 has been visited only once (light gray nodes). Its root, however, has been visited twice (gray node). On its second visit a cache hit occurred: the formula under consideration has already been "applied" to that argument.

4 Implementation of Attribute Grammar-based Spreadsheets

A spreadsheet-like system has to react and to provide answers in real-time. Consequently, the delay between the user interaction and the system response is an extremely important aspect in such interactive systems. Thus, one of the key features to handle such interactive environments is the ability to perform efficient recomputations. That is to say that spreadsheets use an efficient incremental computational model. Implementing from scratch an efficient incremental engine is a complex task.

We use an efficient incremental computational model that efficiently (and elegantly) handles HAGs: the memoization (and posterior reuse) of calls to the functions of the attribute evaluator. Such functions traverse/visit the tree in order to assign a meaning to input under consideration. To achieve efficient function mem-

oization we use the following combination of techniques:

- *Purely functional attribute evaluators:* Syntax trees are visited and decorated by strict, purely functional attribute evaluators. The attribute evaluators are based on the *visit-sequence* paradigm [14]: The attribute evaluator consists of a set of *visit-functions*, each of which perform the computations scheduled (by the attribute grammars scheduling algorithm) for a particular traversal of the evaluator. The attribute instances are not stored in the tree nodes, but, instead, they are the arguments and the results of pure (side-effect free) functions: the *visit-functions*. The different traversal functions are "glued" by intermediate data structures: the *visit-trees*. Such redundant intermediate structures can be eliminated by using our deforestation techniques for AGs [26].
- *Data constructor memoization:* Since attribute instances are not stored in the syntax tree, multiple instances of the syntax tree can be shared. That is, trees are collapsed into minimal Direct Acyclic Graphs (DAG) (Figure 3 displays the DAG constructed for the example input). DAGs are obtained by constructing trees bottom-up and by using constructor function memoization to eliminate replication of common sub-expressions. This technique, also called *hash-consing* [11,1], guarantees that two identical objects share the same records on the heap, and thus are represented by the same pointer.

  Data constructor memoization considerably reduces the memory usage, allows for efficient equality tests between all terms because a pointer comparison suffices and, as we will explain next, makes efficient visit-function memoization possible.
- *Visit-function memoization:* Due to the pure nature of the visit-functions, incremental evaluation is obtained by memoizing calls to the evaluator's strict visit-functions. Memoization is obtained by storing in a *function cache* calls to visit-functions. Every call corresponds to a *entry*, in the *function cache*, that records both the arguments and the results of one call to a visit-function.

  The essence of the visit-function memoization is as follows: each time a memoized visit-function is applied to a subtree and to a set of remaining arguments (*i.e.*, values of attribute instances), we search a *cache* to check whether that function was previously applied to those arguments, or not. If the cache contains an entry corresponding to the call, the result in that entry is returned. If no such entry exists, the visit-function is applied to the arguments and the call is memoized.

  In the animations produced by *Lrc* the reuse of a function call can be easily identified since when visiting a node the animation skip the visits to the children of that node. In our running example this occured in the second visit to the root of the shared list of marks.

4.1 Benchmarking the Spreadsheet

Next, we present results obtained when executing the spreadsheet with a real student database: the database contains the students, and all their marks, who follow the compiler construction course where this tool was proposed as the course project.

The number of students attending this course was 144 and to each of them corresponds 15 evaluation elements (*i.e.*, partial evaluation marks) which are used by a list of 7 formulas to compute different aspects of their evaluation (for example, a particular exam, the projects mark, the final mark, etc). This database was constructed and is maintained by the spreadsheet tool. Actually, we have used this tool to manage the course, as opposed to the use of a commercial tool. The table below presents results obtained both with non-incremental evaluation, *i.e.*, without memoization of the calls to the evaluator functions, and with incremental evaluation, *i.e.*, with memoization of the function calls. It shows the number of functions evaluated (*cache misses*), functions reused (*cache hits*), and time (in seconds on an 2.4 GHz Intel Pentium processor, running Linux Red-Hat 9). We consider six different situations: the processing of the database from scratch (*i.e.*, starting with an empty function cache), the editing of the database (*i.e.*, the reaction after adding a new formula, editing one, and editing a student mark), and the querying of the database (*i.e.*, performing a query to select a student and performing the example query).

	Non-Incremental			Incremental		
	cache misses	cache hits	time secs	cache misses	cache hits	time secs
Scratch	408568	-	7.8	41730	37052	1.65
Editing:						
Add a formula	435212	-	11.3	18356	34317	1.30
Edit a formula	408568	-	10.4	8148	11409	0.45
Edit a stud. mark	408568	-	10.4	3393	3016	0.23
Querying:						
Select a student	501208	-	12.5	22158	47412	1.63
Example query	623000	-	13.3	27871	59307	1.73

As the above table shows, our incremental model of attribute evaluation produces efficient implementations. Even when processing an input from scratch, the incremental evaluator computes 10% of the functions as compared to when no incrementally is used (37052 functions evaluated against 408568, respectively) and is almost 5 times faster. The reused functions are, in this case, due to the decoration of the same tree (representing the formula) with the same inherited attributes (the same marks). Or, in other words, the reuse of previous evaluations of the formula with the same "arguments". As expected, the tool handles very well updates of the input: adding a new (global) formula requires the re-evaluation of 0.04% of the functions computed with non-incremental evaluation and 44% of the functions if we consider incremental evaluation. Better results are obtained with local changes (*e.g.*, editing

a student mark). A query in this spreadsheet performs like a global change since the results of applying formulas are being considered by the query. Nevertheless, using the incremental model is 8 times faster than the non-incremental one.

5 Conclusions

A spreadsheet was efficiently and elegantly specified within the style of attribute grammar programming. The *Lrc* system processed such a specification and derived a correct and efficient implementation. The results of incremental evaluation show that spreadsheets are a natural context for incremental evaluation. Actually, these results are much better than previous results of incremental evaluation (mainly produced in the context of syntax-based editing).

References

[1] A. W. Appel and M. J. R. Gonçalves. Hash-consing Garbage Collection. Technical Report CS-TR-412-93, Princeton University, Dept. of Computer Science, Feb. 1993.

[2] L. Augusteijn. The elegant compiler generation system. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990. Paris.

[3] H. Boley. Attribute grammars and xml. In *Workshop on Attribute Grammars, XML, and RDF*. University of St. Gallen, Sept. 2000.

[4] N. Bradley. *The XML Companion*. Addison Wesley, 1998.

[5] O. de Moor, K. Backhouse, and D. Swierstra. First-Class Attribute Grammars. In D. Parigot and M. Mernik, editors, *Third Workshop on Attribute Grammars and their Applications, WAGA'00*, pages 1–20, Ponte de Lima, Portugal, July 2000. INRIA Rocquencourt.

[6] O. de Moor, S. Peyton-Jones, and E. van Wyk. Aspect-Oriented Compilers. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, volume 1799 of *LNCS*. Springer-Verlag, Sept. 1999.

[7] W. W. Draft. *XQuery 1.0: An XML Query Language*, April 2002.

[8] E. R. Gransner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 00(S1):1–29, 1999.

[9] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM,*, 35(2):121–131, February 1992.

[10] G. Hedin. Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 153–172, Amsterdam, The Netherlands, Mar. 1999. INRIA rocquencourt.

[11] J. Hughes. Lazy memo-functions. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 129–146. Springer-Verlag, September 1985.

[12] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173. Springer-Verlag, September 1987.

[13] M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, volume 25, pages 209–222. ACM, June 1990.

[14] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[15] U. Kastens and C. Schmidt. Vl-eli: A generator for visual languages. In M. van den Brand and R. Laemmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[16] U. Kastens and W. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, June 1994.

[17] D. E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[18] M. Kuiper and J. Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In K. Koskimies, editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

[19] M. Kuiper and D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*, November 1987.

[20] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Lisa: An interactive environment for programming language development. In N. Horspool, editor, *International Conference on Compiler Construction, CC/ETAPS'02*, volume 2304 of *LNCS*, pages 1–4. Springer-Verlag, April 2002.

[21] J. Ousterhout. *Tcl and the Tk toolkit*. Addison Wesley, 1994.

[22] J. Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[23] T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer, 1989.

[24] J. Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999. `ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Saraiva/`.

[25] J. Saraiva. Component-based Programming for Higher-Order Attribute Grammars. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2002, Held as Part of the Confederation of Conferences on Principles, Logics, and Implementations of High-Level Programming Languages, PLI 2002, Pittsburgh, PA, USA, October 3-8, 2002*, volume 2487 of *LNCS*, pages 268–282. Springer-Verlag, October 2002.

[26] J. Saraiva and D. Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction, CC/ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, Mar. 1999.

[27] J. Saraiva, D. Swierstra, and M. Kuiper. Functional Incremental Attribute Evaluation. In David Watt, editor, *9th International Conference on Compiler Construction, CC/ETAPS2000*, volume 1781 of *LNCS*, pages 279–294. Springer-Verlag, Mar. 2000.

[28] D. Swierstra, P. Azero, and J. Saraiva. Designing and Implementing Combinator Languages. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Third Summer*

School on Advanced Functional Programming, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, Sept. 1999.

[29] H. Vogt, D. Swierstra, and M. Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 131–145. ACM, July 1989.

[30] H. Vogt, D. Swierstra, and M. Kuiper. Efficient incremental evaluation of higher order attribute grammars. In J. Maluszynki and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *LNCS*, pages 231–242. Springer-Verlag, 1991.

[31] P. Wadler. Xquery: a typed functional language for querying XML. In *Fourth Summer School on Advanced Functional Programming, Oxford*, August 2002.

Dealing with life in the cells: An experimental study

David Wakeling [1]

Bioinformatics Group, University of Exeter, Exeter, United Kingdom

Abstract

Cell biologists have access to the "parts list" data for the cells of many organisms, but limited understanding of how these parts go together to make a cell "come alive". A cell simulator could improve this understanding considerably. Our goal is to make cell simulation as convenient as spreadsheet calculation by making the simulator resemble a spreadsheet. In this paper, we describe our prototype simulator and report on our early experience with it.

Key words: cell simulation, spreadsheet

1 Introduction

Cell biologists have access to "parts list" data for the cells of many organisms, but limited understanding of how these parts go together to make a cell "come alive". A cell simulator could improve this understanding considerably, as Tomita remarks

> To say that we understand the overall behaviour of the cell, we must be able to answer questions such as: 'How would the cell behave if we change the environment, for example, by adding or decreasing a certain substance?' and 'What is the result if a certain gene gets knocked out or over-expressed?' Slightly more sophisticated questions include 'What gene needs to be inserted for the cell to behave in such a way?' and 'What is the ideal culture medium in which to maximize the cell's ability to do such a thing?' [10].

Our goal is to make cell simulation as convenient as spreadsheet calculation by making the simulator resemble a spreadsheet. In this paper, we describe our prototype simulator and report on our early experience with it. The paper is organised as follows. Section 2 provides a brief reminder of how ordinary differential equations are solved numerically. Section 3 describes our simulator. Section 4 gives its evaluation and typing rules. Section 5 presents an example

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

model. Section 6 shows how this model can be simulated with our simulator. Section 7 shows how it can be simulated with the E-Cell simulator. Section 8 reports the results of a small an experimental study. Section 9 considers some closely related and possible future work. Section 10 concludes.

2 Solving ordinary differential equations numerically

An *ordinary differential equation* takes the form $dx/dt = f(t, x)$, and describes how the *dependent variable*, x, representing some value of interest changes with the *independent variable*, t, usually representing time. An *initial value problem* is one where the value of $x = x_0$ is specified at time $t = t_0$. A simulation must solve systems of such equations numerically, and one common way to do so is to use the fourth-order Runge-Kutta method, summarised by the equations

$$
\begin{aligned}
k_1 &= hf(t_n, x_n) \\
k_2 &= hf(t_n + \frac{h}{2}, x_n + \frac{k_1}{2}) \\
k_3 &= hf(t_n + \frac{h}{2}, x_n + \frac{k_2}{2}) \\
k_4 &= hf(t_n + h, x_n + k_3)
\end{aligned}
$$

$$
\begin{aligned}
x_{n+1} &= x_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k4}{6} \\
t_{n+1} &= t + h
\end{aligned}
$$

where the constant h is the time step size.

3 Our prototype simulator

Our prototype simulator incorporates several ideas.

3.1 A new approach to functions

Recently, Peyton Jones *et. al.* proposed a new approach to functions in spreadsheets [6]. Functions are entered into a worksheet [2], with some cells allocated for the arguments, and another for the result. See figure 1. Here, the workbook has a worksheet for the function `f2c` that converts from Fahrenheit to Celsius. By (our) convention, the first few cells in column A of a worksheet that defines a function are allocated for the argument values, and the one below those is allocated for the result formula.

[2] Throughout this paper, we use the terminology and features of Microsoft's popular Excel spreadsheet for our discussion and examples. Other spreadsheets are similar.

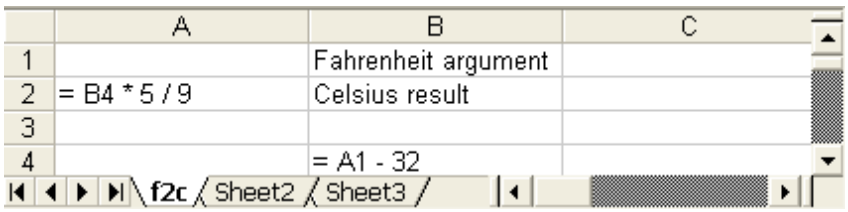


Fig. 1. A worksheet defining a function.

3.2 A history mechanism

Wray and Fairbairn introduced a *history mechanism* in their *Nas* spreadsheet [13]. Here, the value of a cell may depend on its own "last" value, and calculation takes place when a `Tick` button is pressed. Thus, a cell might contain the value and formula `42 = last + 1` which, after pressing `Tick`, would change to `43 = last + 1`. There must always be a value in a cell with a formula involving `last`.

3.3 A built-in function for solving ordinary differential equations

Spreadsheets usually provide a number of *built-in functions* for common mathematical, statistical and scientific operations. In the same way, a built-in function can be provided for solving ordinary differential equations. To solve an equation of the form $dx/dt = f(t, x)$ with an initial value $x = x_0$ at $t = t_0$, a cell might contain the value and formula x_0 `= ODE(`f`(`t`, last),` h`)`, where t is a reference to a cell containing the value and formula t_0 `= last +` h, and h is a reference to a cell containing the value of the step size. Each `ODE` application takes one step towards solving the equation. Again, there must always be a value in a cell with a formula involving an `ODE` application, which must always be outermost.

Most often, a *system* of ordinary differential equations must be solved, in which several equations are defined in terms of one another. In a functional setting, each of these equations must be passed the current values of the others as additional arguments, $a_1, \ldots, a_n$. A definition of `ODE` using the Runge-Kutta method is as follows

$$
\begin{aligned}
&\texttt{ODE(}f(t, a_1, \ldots, a_n, x)\texttt{,}\ h\texttt{)}\\
&\quad = x + k_1/6 + k_2/3 + k_3/3 + k_4/6\\
&\qquad \texttt{where}\\
&\qquad k_1 = hf(t, a_1, \ldots, a_n, x)\\
&\qquad k_2 = hf(t + h/2, a_1, \ldots, a_n, x + k_1/2)\\
&\qquad k_3 = hf(t + h/2, a_1, \ldots, a_n, x + k_2/2)\\
&\qquad k_4 = hf(t + h, a_1, \ldots, a_n, x + k_3)
\end{aligned}
$$

Of course, other definitions are possible using other methods.

4 Evaluation and typing rules

In programming language research, valuable clarity comes from formalising rules for evaluation and typing, and we do something similar for our prototype simulator.

4.1 Terms and types

A *workbook* maps from names to worksheets, and a *worksheet* from cell references to cells. A *cell* is a pair of expressions representing a value and a formula, and an *expression* is either a label, a number, a cell reference, the history variable, or the application of a name to an argument. See figure 2. In reality, of course, it is essential to allow the basic mathematical operators,

$$\begin{array}{lcll} w & ::= & \{g_i \Rightarrow s_i\}_{i=1}^{n} & \text{— workbooks} \\ s & ::= & \langle r_i \Rightarrow c_i \rangle_{i=1}^{m} & \text{— worksheets} \\ c & ::= & (e_1, e_2) & \text{— cells} \\ e & ::= & l \mid n \mid r \mid \texttt{last} \mid g\ \texttt{(}e\texttt{)} & \text{— expressions} \end{array}$$

Fig. 2. The syntax of terms.

parentheses for grouping, built-in functions, and application of a worksheet to more than one argument. These extensions present no difficulty, and are omitted here only to save space.

A *type* is either a label, number or function type. See figure 3.

$$T \quad ::= \quad \text{Label} \mid \text{Number} \mid (\text{Number}) \to \text{Number}$$

Fig. 3. The syntax of types.

4.2 Evaluation rules

An *evaluation judgement* takes the form $s : t \Downarrow s' : t'$, and says that evaluating a term t in a state s yields a term t' in a state s'. A state may be made up of a mapping Γ from names to worksheets, a mapping Δ from cell references to cells and a value ϕ to be used for the history variable. To the left of "$\Downarrow$", $M[a \mapsto b]$ indicates extension of a mapping, and $M[\ldots, a \mapsto b, \ldots]$ reveals some part of it. To the right of "$\Downarrow$", such details are omitted, since any changes can easily be inferred from elsewhere.

Evaluation judgements can be used to write *evaluation rules* for terms. The rule E-Book says that evaluating a workbook involves evaluating *only* the visible worksheet (this is the one being "ticked"). See figure 4. The rule E-Sheet says that evaluating a worksheet involves evaluating *all* of its cells (this must be done in *natural order*). See figure 5. There are two evaluation

$$\text{E-Book}\ \frac{\Gamma[g_i \mapsto s_i]_{i=1}^n : s_{\surd} \Downarrow \Gamma' : s'_{\surd}}{\Gamma : \{\ldots, g_{\surd} \Rightarrow s_{\surd}, \ldots\}_{i=1}^n \Downarrow \Gamma' : \{\ldots, g_{\surd} \Rightarrow s'_{\surd}, \ldots\}_{i=1}^n}$$

Fig. 4. The evaluation rule for workbooks.

$$\text{E-Sheet}\ \frac{\Gamma;\ \Delta[r_i \mapsto c_i]_{i=1}^m : c_i \Downarrow \Gamma';\ \Delta' : c'_i}{\Gamma : \langle r_i \Rightarrow c_i \rangle_{i=1}^m \Downarrow \Gamma' : \langle r_i \Rightarrow c'_i \rangle_{i=1}^m}, r_i \text{ in natural order}$$

Fig. 5. The evaluation rule for worksheets.

rules for cells. See figure 6. The rules E-Cell1 and E-Cell2 say that evaluating

$$\text{E-Cell1}\ \frac{}{\Gamma;\ \Delta : (v, ?) \Downarrow \Gamma;\ \Delta : (v, ?)}$$

$$\text{E-Cell2}\ \frac{\Gamma;\ \Delta;\ v : f \Downarrow \Gamma';\ \Delta' : e}{\Gamma;\ \Delta : (v, f) \Downarrow \Gamma';\ \Delta' : (e, f)}$$

Fig. 6. The evaluation rules for cells.

a cell is trivial if it has a value, but no formula; otherwise, the formula must be evaluated to give the value. There are five evaluation rules for expressions. See figure 7. The rules E-Lab and E-Num say that evaluating constants is

$$\text{E-Lab}\ \frac{}{\Gamma;\ \Delta;\ \phi : l \Downarrow \Gamma;\ \Delta : l}$$

$$\text{E-Num}\ \frac{}{\Gamma;\ \Delta;\ \phi : n \Downarrow \Gamma;\ \Delta : n}$$

$$\text{E-Ref}\ \frac{}{\Gamma;\ \Delta[\ldots, r \mapsto (v, f), \ldots];\ \phi : r \Downarrow \Gamma;\ \Delta : v}$$

$$\text{E-Hist}\ \frac{}{\Gamma;\ \Delta;\ \phi : \texttt{last} \Downarrow \Gamma;\ \Delta : \phi}$$

$$\text{E-App}\ \frac{\begin{array}{c}\Gamma;\ \Delta;\ \phi : x \Downarrow \Gamma';\ \Delta' : y \\ \Gamma'[\ldots, g \mapsto s, \ldots];\ s[\texttt{A1} \mapsto (y, ?)];\ \phi : \texttt{A2} \Downarrow \Gamma'';\ s'' : e\end{array}}{\Gamma;\ \Delta;\ \phi : g\texttt{(}x\texttt{)} \Downarrow \Gamma'';\ \Delta'' : e}$$

Fig. 7. The evaluation rules for expressions.

trivial. The rule E-Ref says that a cell reference evaluates to the value of a cell that must itself already have been evaluated because evaluation is done in natural order. The rule E-Hist says that `last` evaluates to the value of the cell whose formula caused it to be evaluated. The rule E-App says that evaluating an application amounts to evaluating the result cell of the function worksheet when its argument cell contains the argument value.

4.3 Typing rules

A *typing judgement* takes the form $A \vdash t :: T$, and says that under the assumptions A, a term t has type T. The assumptions are made up of a mapping Λ from names to types, and a mapping Ξ from cell references to types.

Typing judgements can be used to write *typing rules* for terms. The rule T-Book says that a workbook has (arbitrarily) the type of the visible worksheet. See figure 8. The rule T-Sheet says that a worksheet has a type formed from

$$\text{T-Book}\ \frac{\Lambda[\ldots, g_i :: T_i, \ldots] \vdash s_\surd :: T_\surd}{\Lambda \vdash \{\ldots, g_\surd \Rightarrow s_\surd, \ldots\}_{i=1}^{n} :: T_\surd}$$

Fig. 8. The typing rule for workbooks.

those of the argument and result cells, which must be Numbers. See figure 9.

$$\text{T-Sheet}\ \frac{\begin{array}{c}\Lambda;\ \Xi[\texttt{A1} :: \text{Number}] \vdash c_i :: T_i \\ \Lambda;\ \Xi[\texttt{A1} :: \text{Number}, r_i :: T_i] \vdash c_2 :: \text{Number}\end{array}}{\Lambda \vdash \langle \texttt{A1} \Rightarrow c_1, \texttt{A2} \Rightarrow c_2, r_i \Rightarrow c_i \rangle_{i=3}^{m} :: (\text{Number}) \to \text{Number}}$$

Fig. 9. The typing rule for worksheets.

There are two typing rules for cells. See figure 10. The rules T-Cell1 and T-

$$\text{T-Cell1}\ \frac{\Lambda;\ \Xi \vdash v :: T}{\Lambda;\ \Xi \vdash (v, ?) :: T}$$

$$\text{T-Cell2}\ \frac{\Lambda;\ \Xi \vdash f :: \text{Number}}{\Lambda;\ \Xi \vdash (v, f) :: \text{Number}}$$

Fig. 10. The typing rules for cells.

Cell2 say that a if a cell has value, but no formula, then its type is that of the value; otherwise, it is that of the formula, which must be Number. There are five typing rules for expressions. See figure 11. The rules T-Lab and T-Num say that constants have the obvious basic types. The rule T-Ref says that a cell reference has the type given in the assumptions, and the rule T-Hist says that `last` must have type Number. The rule T-App says that the application of a worksheet to an argument of type Number has type Number.

5 An example model

Hodgkin and Huxley modelled the current flow through the surface membrane of the nerve fibre of a squid as the sum of the *capacitative*, *sodium*, *potassium* and *leakage* currents [4].

$$\text{T-Lab}\ \frac{}{\Lambda;\ \Xi \vdash l :: \text{Label}}$$

$$\text{T-Num}\ \frac{}{\Lambda;\ \Xi \vdash n :: \text{Number}}$$

$$\text{T-Ref}\ \frac{}{\Lambda;\ \Xi[\ldots, r :: T, \ldots] \vdash r :: T}$$

$$\text{T-Hist}\ \frac{}{\Lambda;\ \Xi \vdash \texttt{last} :: \text{Number}}$$

$$\text{T-App}\ \frac{\Lambda;\ \Xi \vdash x :: \text{Number} \qquad \Lambda;\ \Xi \vdash g :: (\text{Number}) \to \text{Number}}{\Lambda;\ \Xi \vdash g(x) :: \text{Number}}$$

Fig. 11. The typing rules for expressions.

5.1 The capacitative current

The capacitative current, I_{cap}, is given by

$$I_{\text{cap}} = C_m \times \frac{dV_m}{dt}$$

where C_m is the membrane capacitance, and V_m the transmembrane potential.

5.2 The sodium current

The sodium current, I_{Na}, is given by

$$I_{\text{Na}} = g_{\text{Na,max}} \times m^3 \times h \times (V_m - E_{\text{Na}})$$

where $g_{\text{Na,max}}$ is the maximum conductance of sodium, and m and h satisfy the equations

$$\frac{dm}{dt} = \alpha_m \times (1 - m) - \beta_m \times m$$

$$\frac{dh}{dt} = \alpha_h \times (1 - h) - \beta_h \times h$$

The rate coefficients were found by curve-fitting to be

$$\alpha_m = \frac{0.1 \times (V_m + 25.0)}{\exp(0.1 \times (V_m + 25.0)) - 1.0}$$

$$\beta_m = 4 \times \exp(V_m/18.0)$$

$$\alpha_h = 0.07 \times \exp(V_m/20.0)$$

$$\beta_h = \frac{0.1}{\exp(0.1 \times (V_m + 30.0)) + 1.0}$$

5.3 The potassium current

The potassium current, I_{K}, is given by

$$I_{\mathrm{K}} = g_{\mathrm{K,max}} \times n^4 \times (V_m - E_{\mathrm{K}})$$

where $g_{\mathrm{K,max}}$ is the maximum conductance of potassium, and n satisfies the equation

$$\frac{dn}{dt} = \alpha_n \times (1 - n) - \beta_n \times n$$

The rate coefficients were again found by curve-fitting to be

$$\alpha_n = \frac{0.01 \times (V_m + 10.0)}{\exp(0.1 \times (V_m + 10.0)) - 1.0}$$

$$\beta_n = 0.125 \times \exp(V_m/80.0)$$

5.4 The leakage current

The leakage current, I_{L}, is given by

$$I_{\mathrm{L}} = g_{\mathrm{L}} \times (V_m - E_{\mathrm{L}})$$

where E_{L} is the reversal potential for the leakage current.

5.5 The total current

Combining the equations, the total current across a cell membrane, I, is given by

$$I = C_m \times \frac{dV_m}{dt} + I_{\mathrm{Na}} + I_{\mathrm{K}} + I_{\mathrm{L}}$$

This can be rearranged to

$$\frac{dV_m}{dt} = \frac{I - I_{\mathrm{Na}} - I_{\mathrm{K}} - I_{\mathrm{L}}}{C_m}$$

6 Simulation with our prototype simulator

In our simulator, the ordinary differential equations of the example model are entered into a workbook. See figure 12. The `run` worksheet drives the simulation. Cells A4 — A7 deal with calculation of the dependent variables `Vm`, `m`, `h` and `n`, and cells A2 and C1 with calculation of the dependent variable, `t`. The remaining cells are used for intermediate calculations, constant values and labels. Other worksheets are used in the calculation of the dependent variables. To save space, we have shown only the `calc_m` worksheet, as `calc_h`, `calc_n` and `calc_vm` are very similar. Simulation is performed by pressing a "recalculate" button and specifying the number of steps (or "ticks"). Our simulator is written from scratch in Haskell using the WxHaskell library.

File Help

	A	B	C	D	E
1			5.0e-3		step
2	0.0 = last + C1				t
3					
4	-75.0 = ODE(calc_vm(A2, B9, B10, B11, last), C1)				Vm
5	5.0e-2 = ODE(calc_m(A2, A4, last), C1)				m
6	0.6 = ODE(calc_h(A2, A4, last), C1)				h
7	0.325 = ODE(calc_n(A2, A4, last), C1)				n
8					
9		= C17 * A5 ^	3.0 * A6 * (A4 - C14)		INa
10		= C18 * A7 ^	4.0 * (A4 - C15)		IK
11		= C19 * (A4 -	C16)		IL
12					
13			-75.0		ER
14			= C13 + 115.0		ENa
15			= C13 - 12.0		EK
16			= C13 + 10.613		EL
17			120.0		gNaMax
18			36.0		gKMax
19			0.3		gL

calc_m calc_h calc_n calc_vm run

The `run` worksheet.

File Help

	A	B	C	D	E
1		t			
2		Vm			
3		m			
4	= B6 * (1.0 -	A3) - B7 * A3			
5					
6		= 0.1 * (A2 +	25.0) / (EXP(0.1 * (A2 + 25.0)) - 1.0)		
7		= 4.0 * EXP(A2 / 18.0)			

calc_m calc_h calc_n calc_vm run

The `calc_m` worksheet.

Fig. 12. Model entry with our prototype simulator.

7 Simulation with the E-Cell simulator

One of the most advanced cell simulators available is E-Cell, developed by Tomita and his team since 1996 [9]. In the E-Cell simulator, the ordinary differential equations of the example model are entered in a text file. See figure 13. Here, the single stepper object, `ODE`, is an `ODE45Stepper` — the

```
Stepper ODE45Stepper( ODE ) { }

System System( / ) {
 StepperID ODE;
 Variable Variable( m  ) { Value 5.0E-2; }
 Variable Variable( h  ) { Value 0.6;    }
 Variable Variable( n  ) { Value 0.325;  }
 Variable Variable( Vm ) { Value -75.0;  }

 Process ExpressionFluxProcess( proc_m ) {
  Expression "( (0.1 * (Vm.Value + 25.0)) /
                (exp(0.1 * (Vm.Value + 25.0)) - 1.0) ) *
                  (1 - m.Value)
                - (4 * exp(Vm.Value / 18.0) * m.Value)";

  VariableReferenceList [ m  Variable:.:m  1 ]
                        [ Vm Variable:.:Vm 0 ];
 }

 # ... similar processes to calculate h, n and Vm
}
```

Fig. 13. Model entry with the E-Cell simulator.

one recommended for solving systems of ordinary differential equations. It calls process objects and determines the next time step interval. The system object, `System`, contains variable and process objects that describe the system of equations. For each variable object, the `Value` property is set to the initial value. For each process object, the `Expression` property is set to an expression that changes the value of a variable, and the `VariableReferenceList` property is set to indicate those variables that it reads ('`0`'), and those that it reads or writes ('`1`'). Notice that we have had to break the `Expression` string to fit it on the page. Again to save space, we have shown only the process `proc_m`, as `proc_h`, `proc_n` and `proc_Vm` are similar. Simulation is performed by loading a text file into either a batch or an interactive session monitor. E-Cell is written in C++ and Python.

8 An experimental study

It is essential that cell biologists be able to enter their models into a simulator quickly and accurately. Otherwise, they will either give up on simulation and continue to use pencil-and-paper, or they will generate meaningless simulation results whose cause might be difficult to determine. A small experimental study was therefore conducted into the effectiveness of both our prototype simulator interface and the E-Cell simulator interface.

8.1 Subjects

Eight volunteer subjects took part in the study, ranging in age from 20 to 37 years old. Four were BSc Computer Science students, two were MSc Bioinformatics students, one was a Research Assistant in Bioinformatics, and one was a Research Assistant in Neural Networks. All had some experience of spreadsheets and text editors, but none of cell simulators. They were thus considered to have a representative skill-set.

8.2 Apparatus

A week in advance of the experiment, the subjects were issued with a four-page manual covering both simulators, and containing an example of how to use them to simulate a model with two ordinary differential equations. This manual was deliberately terse and example-based, reflecting the typical provision of documentation and the availability of examples. The test problem was to enter the four ordinary differential equations for the model given in [12].

8.3 Procedure

First, each subject was asked to fill out a short questionnaire, answering questions about themselves and their proficiency with spreadsheets, text editors and cell simulators. Next, they were given a five minute demonstration of the simulators. Finally, the procedure for the study was explained and the test problem was revealed. Afterwards, each subject was invited to comment on their experience.

8.4 Data collection

The subjects could choose to use either simulator first. Their choice was recorded, together with their claimed proficiency with spreadsheets and text editors. For our prototype, "begin" time was recorded when the subject started the simulator, and "end" time when they had successfully recalculated their workbook. The number of error dialogue boxes encountered, and the nature of the errors was also recorded. For E-Cell, "begin" time was recorded when the subject started the text editor, and "end" time when they had successfully loaded their file into the session monitor. The number of load

commands issued, and the nature of any errors was also recorded. Afterwards, the workbooks and files were examined and the number of mistakes that they contained was recorded.

8.5 Results

Table 14 and Table 15 show the results for our prototype and for E-Cell.

subject identifier	spreadsheet proficiency	time taken	static errors	dynamic errors
A	5	27	1	0
B	3	31	1	0
C	3	55+	13	1
D	3	31	3	0
E	4	35+	6	0
F	3	36	14	1
G	3	30	3	0
H	1	34	3	0

Fig. 14. Results for our prototype simulator.

subject identifier	text editor proficiency	time taken	static errors	dynamic errors
A	4	36	7	0
B	5	28	8	3
C	5	66+	17	4
D	3	69+	12	4
E	1	30	4	1
F	4	49	10	0
G	4	27	1	0
H	4	35	12	0

Fig. 15. Results for the E-Cell simulator.

In these tables, proficiency is measured from 0 (low) to 5 (high), and the time taken is measured in minutes, with a trailing "+" indicating that the

subject gave up. Static errors are counted as the number of error dialogue boxes or load commands, and dynamic errors as the number of mistakes.

8.6 Discussion

Regardless of their claimed proficiency with spreadsheets and text editors, all but one of the subjects chose to use E-Cell first, and their times taken to enter the test model with both simulators were broadly similar. No clear preference was expressed for either.

In the case of our simulator, two of the static errors were caused by accidental cycles between cells, and four by dangling cell references. These were eventually fixed, although our curt error messages of the form "`A1`: cycle exists" and "`A1`: dangling reference" proved difficult to comprehend. The remaining static errors were type (in fact, arity) errors. After some thought, most subjects fixed these, but the two who gave up did so because they had entirely misunderstood the worksheet argument-passing convention, and found error messages of the form "worksheet does not have type `(Number,Number,Number,Number) -> Number`" incomprehensible. Both of the dynamic errors were mistyped constants.

In the case of E-Cell, the majority of static errors were caused by missing parentheses, semicolons and the like. These were fixed after a few repeats of the the edit-load cycle. Of the two subjects who gave up, one did so because they declared their constants outside of a process rather that inside it, leading to an incomprehensible error message about unknown property slots, and the other did so because they used identifiers instead of `ExpressionFluxProcess`, leading to an incomprehensible message about loading shared object files. All of the dynamic errors were caused by misunderstandings about the role of `VariableReferenceList` integers.

9 Related and future work

A spreadsheet interface was used in an early version of the E-Cell simulator [9] and is still being used in the JigCell simulator [11]. In both cases, the spreadsheet describes chemical reactions, with rows used for objects (such as reactants and reactions) and columns for properties (such as names and types). The advantage of a constrained spreadsheet like this is that clear rules can be given about what should go in which cells, and these rules can be checked. The disadvantage is that these additional rules must be learned, and so much of the attractive simplicity of a free-form spreadsheet is lost. Actual experience with the spreadsheet interface has been mixed. On the one hand, it was abandoned in E-Cell, and so presumably was not a success. On the other hand, it continues to be used in JigCell, with encouraging results.

During our study, several subjects complained that our prototype was missing copy-and-paste, which they especially wanted to copy a table of all con-

stants onto all worksheets. They could also often be seen pointing at the screen, or heard muttering as they worked out which cell references to use because, although able to comment on the role of cells by placing labels nearby, they were unable to refer to them by those labels. Some subjects even engaged in rapid switching between worksheets to check function arities because only one worksheet is visible at a time. None of these problems would have arisen with Excel, which in future we plan to use as the interface to our simulator.

Our typing rules are in the tradition of functional programming, largely because the author is a former functional programmer. As our study has shown, though, they are not well suited to spreadsheets. Recently, there has been a growing interest in checking the consistency of spreadsheets by reasoning about their *units* [1,2,3]. In principle, any value declares a unit (for example, a label "Month" declares a unit Month). In practice, it is those values serving as column or row headers which provide the declarations that can serve as the basis for *unit inference* (for example, a label "October" in a column with a header label "Month" has the unit Month[October]). As Erwig and Burnett remark, unit inference provides a good way to make use of the "implicitly explicit" information already present for the purposes of documentation [3]. In future, we plan to build on this work to develop evaluation and typing rules to ensure that *types* (such as numbers and labels), *dimensions* (such as litres and volts) and *units* (such as reactants and catalysts) are used consistently.

Of course, other cell simulators exist with textual (for example, Jarnac [7]), diagrammatic (for example, GEPASI [5]) and graphical (for example, Virtual Cell [8]) user interfaces. Once we have improved our simulator as outlined, we plan to repeat our study, including simulators like these too.

10 Conclusions

In this paper, we have described our prototype cell simulator and reported on our early experience with it. The simulator interface is a spreadsheet that incorporates a new approach to functions, a history mechanism and a built-in function for solving ordinary differential equations. A small experimental study showed that it was not as effective as it might be. In future, we plan to redesign our prototype, to develop more suitable typing rules, and to undertake another study.

Acknowledgements

We are grateful to the anonymous referees for some useful comments and suggestions.

References

[1] Ahmad, Y., T. Antoniu, S. Goldwater and S. Krishnamurthi, *A type sysetm for statically detecting spreadsheet errors*, in: *IEEE International Conference on Automated Software Engineering* (2003), pp. 174–183.

[2] Antoniu, T., P. A. Steckler, S. Krishnamurthi, E. Neuwirth and M. Felleisen, *Validating the unit correctness of spreadsheet programs*, in: *Proceedings of the International Conference on Software Engineering* (2004), pp. 439–448.

[3] Erwig, M. and M. Burnett, *Adding apples and oranges*, in: *Proceedings of the International Symposium on Practical Aspects of Declarative Languages* (2002), pp. 173–191, lNCS 2257.

[4] Hodgkin, A. L. and A. F. Huxley, *A quantitative description of membrane current and its application to conduction and excitation in nerve*, Bulletin of Mathematical Biology **52** (1990), pp. 25–71.

[5] Mendes, P., *Biochemistry by numbers*, Trends in Biochemical Sciences **22** (1997), pp. 361–363.

[6] Peyton Jones, S., A. Blackwell and M. Burnett, *A user-centred approach to functions in Excel*, in: *Proceedings of the International Conference on Functional Programming* (2003), pp. 165–176.

[7] Sauro, H. M., *Jarnac: A system for interactive metabolic analysis*, in: *Proceedings of the International Meeting on BioThermoKinetics* (2000), pp. 221–228.

[8] Schaff, J. and L. M. Loew, *The virtual cell*, in: *Pacific Symposium on Biocomputing*, 1999, pp. 228–239.

[9] Takahashi, K., N. Ishikawa, Y.Sadamoto, H. Sasamoto, S. Ohta, A. Shiozawa, F. Miyoshi, Y. Naito, Y. Nakayama and M. Tomita, *E-Cell 2: Multi-platform E-Cell simulation system*, Bioinformatics **19** (2003), pp. 1727–1729.

[10] Tomita, M., *Whole-cell simulation: A grand challenge of the 21st century*, Trends in Biotechnology **19** (2001), pp. 205–210.

[11] Vass, M., C. A. Shaffer, J. J. Tyson, N. Ramakrishnan and L. T. Watson, *The JigCell model builder: A tool for modeling intra-cellular regulatory networks* (2002).

[12] Wodarz, D. and V. A. A. Jansen, *A dynamical perspective of CTL cross-priming and regulation implications for cancer immunology*, Immunology Letters **86** (2003), pp. 213–227.

[13] Wray, S. C. and J. Fairbairn, *Non-strict languages — programming and implementation*, The Computer Journal **32** (1989), pp. 142–151.