Markus Clermont

A Scalable Approach to Spreadsheet Visualization

DISSERTATION

zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften

Studium der Angewandten Informatik

Universität Klagenfurt

Fakultät für Wirtschaftswissenschaften und Informatik

1. Begutachter: O. Univ-Prof. Mag. Dipl.-Ing. Dr. Roland Mittermeir
Institut für Informatik-Systeme

2. Begutachter: Univ-Prof. Dipl.-Ing. Dr. Martin Hitz
Institut für Informatik-Systeme

März 2003

Ehrenwörtliche Erklärung

To Annabelle and Magdalena.

Acknowledgements

I would like to thank my advisor, Prof. Roland Mittermeir, for his hints, valuable tips and the amount of time he invested to supervise my research work. I am also very grateful to my colleagues in the research group for the numerous discussions we had.

I would like also to thank Prof. Martin Hitz, my second advisor, for his important comments while reading the thesis.

I am much obliged to Simone Pansi for proof reading the thesis and enhancing the readability. I am also grateful to all the other persons who read the thesis and helped me to fix potential errors.

Kurzfassung

Die möglichen Anwendungen von Spreadsheets reichen von Textverarbeitung, die auf numerische Dokumente ausgerichtet ist, bis zur Lösung und Visualisierung komplexer mathematischer Gleichungen. Trotz des zunehmenden Problembewusstseins haben sich systematische Tests oder strukturierte Entwicklungsmethoden als ungeeignet erwiesen. Tests sind in der Regel zu teuer, während strukturierte Methoden von Benutzern ein zu hohes Maß an IT-Kenntnissen erfordern.

Bedingt durch die vielfältigen Anwendungen, lassen sich verschiedene Arten von Spreadsheets identifizieren. Neben kleineren, aber berechnungsintensiven gibt es auch große Spreadsheets mit regulärer Struktur. In dieser Arbeit werden drei verschiedene Ansätze dargestellt, wie anhand der Visualisierung von bestimmten Strukturen im Spreadsheet Fehler leichter aufgefunden werden. Die drei Ansätze, *Logical Areas*, *Semantic Classes* und *Data Modules* unterstützen die Analyse großer, regulärer Spreadsheets. Logical Areas und Semantic Classes gruppieren Zellen zu abstrakten Einheiten, indem die Formeln in den individuellen Zellen verglichen werden, während Data Modules die Zusammenhänge von Zellen anhand des Datenflusses im Spreadsheet untersuchen.

Im Gegensatz zu anderen Visualisierungsansätzen werden so abstrakte Einheiten gebildet, die nicht auf benachbarten Zellen mit gleichen Formeln beschränkt sind. Die Einteilung der Formeln in abstrakte Einheiten berücksichtigt auch bestimmte Abweichungen zwischen den Formeln. *Logical Areas* werden auf der Basis einzelner Zellen gebildet, wobei deren räumliche Verteilung keine Rolle spielt, und lediglich eine ähnliche Formel gefordert wird. *Semantic Classes*, die auf den *Logical Areas* aufbauen, enthalten wiederkehrende Bereiche, also ähnliche Zellen mit ähnlichen Nachbarn. Eine *Semantic Class* kann als Menge von Bereichen aufgefaßt werden, die jeweils dieselbe Aufgabe erfüllen. Ein *Data Module* ist eine Menge von Zellen, die zum selben Resultat beitragen. Analog zur *Semantic Class* kann man ein *Data Module* daher als Menge von Zellen betrachten, die an der Lösung von ein und derselben Aufgabe arbeiten.

In der vorliegenden Arbeit wird anhand der Entwicklung von Spreadsheets in den letzten 40 Jahren sowie einer kurzen Beschreibung der vielfältigen Anwendungsmöglichkeiten die Bedeutung von Spreadsheets untermauert. Da Spreadsheets bis zum heutigen Zeitpunkt weitgehend als Sache der Endbenutzer angesehen wurden, gibt es auch noch kein eindeutiges Fachvokabular. Ein solches wird in dieser Arbeit eingeführt. Nach der Vorstellung von verschiedenen Studien, die Spreadsheets auf Fehler untersuchen, wird auch auf eine im Rahmen dieser Arbeit durchgeführte Untersuchung eingegangen, anhand der die Effizienz von *Logical Areas* beim Überprüfen von Spreadsheets gezeigt wurde. Weiters werden verschiedene Ansätze zur Formalisierung des Spreadsheet-Entwicklungsprozesses diskutiert und die theoretischen Grundlagen, Algorithmen zur Berechnung und Anwendungsmöglichkeiten von Logical Areas, Semantic Classes und Data Modules und eine Implementierung der Algorithmen als Prototyp vorgestellt.

Abstract

Applications of spreadsheets reach from word processing that is specialized on numeric documents to the solution and visualization of complex mathematical equations. Despite of an increasing problem awareness among the users, systematic testing and structured development methodologies are not widely applied. Testing is too expensive, and structured development methodologies do not take into account that spreadsheet users are end-users, and thus have only little IT-training.

Because of the diverse areas of application, many different kinds of spreadsheets can be identified. Aside from small, but computationally intensive spreadsheets, there are also huge spreadsheets with regular structure. Three different approaches for the visualization of regular structures in spreadsheet programs are introduced in this thesis. *Logical Areas*, *Semantic Classes*, and *Data Modules*, identify different structural aspects of spreadsheets and effectively reduce the complexity induced by the size of huge spreadsheets. The generated visualizations aim to support the spreadsheet writers to find errors.

To identify *Logical Areas* and *Semantic Classes* the cells' contents, i.e. the formulas, are examined in order to find regular structures. The *Data Modules* approach examines the data flow between cells. In contrast to other visualization techniques, the identified abstract units are not limited to adjacent cells with equal formulas, but will also contain similar formula. It is up to the auditor to state the desired degree of similarity. *Logical Areas* are made up from single cells with similar formulas, but without considering their spatial dispersion. *Semantic Classes* evolved from *Logical Areas*. They are made up from regularly recurring regions of cells, i.e. similar cells with similar neighbors. A *Semantic Class* can be considered to consist of sets of cells that fulfill the same kind of task. A *Data Module* is a set of cells that contribute to the same result of the spreadsheet. Thus, a data module can be considered a set of cells that cooperate to fulfill a given task.

To frame this work the importance of spreadsheets is pointed out by outlining the development of spreadsheet systems throughout the last 40 years and by a short description of different possible applications areas. Due to the fact, that spreadsheets are still considered to be in the responsibility of end-users, the vocabulary is still somewhat ambiguous. Hence, this thesis introduces a well defined terminology for dealing with spreadsheet-related issues. After the presentation of various spreadsheet error studies, including one that was carried out as part of this thesis, in order to show the auditing capabilities of *Logical Areas*, different approaches for the definition of the spreadsheet development process are presented. The theoretical foundations and algorithms for identifications of *Logical Areas*, *Semantic Classes* and *Data Modules* are presented, as well as different auditing strategies and a prototype using these visualization approaches.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis deals with spreadsheets and software quality. For an IT-professional it is obvious that spreadsheets are software. However, typical spreadsheet users are domain specialists and have little or no IT-training. For them spreadsheets are a computer supported realization of three tools that are fundamental for our modern civilization: paper, pencil, and calculator. As they are familiar with the use of these tools, they do not expect any problems even when creating complex spreadsheets. Therefore, they are not willing to make the extra effort to learn and apply software engineering methodology to increase the quality of their spreadsheets. Additionally, even if they want to decrease the error-rate in their spreadsheets, some of the techniques offered so far are either too expensive, too sophisticated or too restricting to be widely applied. To overcome these problems this thesis will introduce a visualization technique that supports the user in re-constructing the spreadsheet's conceptual model.

Based on the visualization, less complex abstractions of the spreadsheet can be generated. Consequently, the understanding of the internal logic and the calculations of the spreadsheet will be presented in a more comprehensive way. Critical regions and repeatedly used patterns of calculations are identified. Therefore, testing will become cheaper and faster. Maintenance and error-corrections can be performed without committing model-errors which are due to misunderstandings.

The visualization approaches that are introduced, focus on different aspects of the spreadsheet program, i.e. the formulas and constants in the spreadsheet. Thus, this work does not deal with erroneous input data at all.

In this chapter it will be shown why it is necessary to deal with spreadsheet quality problems. For this purpose, the motivation of the work introduced here will be pointed out. The development of spreadsheets from the introduction of computers to modern spreadsheet systems is briefly summarized in Section 1.2. Finally, an outlook on the thesis is given in Section 1.3.

1.1 Motivation

spreadsheet systems belong to the so-called *killer-applications* that are installed on nearly every desktop computer. Therefore, it is not surprising that spreadsheets are the basis for important decisions in daily business-life (see O'Brien and Wilde [OW96], Mittermeir et al. [MCA00] or Chan [CS96]). As pointed out in Section 2.2.3, faulty spreadsheet instances can have severe effects.

According to several studies (see Panko [PH96], Butler [But00] and Clermont et al. [CHM02]) in the average 3% of the cells on a specific spreadsheet instance were erroneous, and $60\% - 96\%$ of the surveyed spreadsheet instances contained faulty cells. These error rates seem rather high but they correspond to the human error rate that psychologists generally state for the solution of complex cognitive problems. Therefore, the same error rate as in conventional software should be expected. However, this is not the case because in conventional software development techniques like modularization reduce the overall complexity. Thorough testing and a defined software development process are applied and have a further influence on the overall software quality.

Due to the fact that spreadsheet users are end-users (see Nardi and Miller [NM90a] or Brown and Gould [BG87]), these techniques that impose restrictions on the spreadsheet programmers, cannot be directly applied to spreadsheet development. The beauty of spreadsheet programs is defined by the freedom of the users to either express their model without limitations on the spreadsheet program, or, as it is stated by Nardi and Miller [NM90b], to formalize their domain with the support of the spreadsheet.

The development process itself is generally a trial-and-error process. The user will make a first prototype of his domain and then modify or extend it until it fits his needs. As Parnas [Par94] states, software ages. Spreadsheets are software, consequently, they are also subject to aging. Hence, the spreadsheet program life cycle is often subdivided into short maintenance cycles. Each of the maintenance cycles tends to modify the initial spreadsheet program in order to re-adjust it for an evolved environment.

In addition, spreadsheet programs are usually not or only insufficiently documented. Hence, the high amount of modifications to the spreadsheet model tends to blur it. Maintenance will be based on assumptions of how the initial model looked like. Of course, these assumptions do match the maintainers current view of the problem, but they might not match the creators domain-model. Therefore, it is indispensable to find a way to support spreadsheet comprehension without making additional IT-knowledge necessary.

Effective auditing of spreadsheet programs has to be performed by a group of at least three IT and domain-specialists performing code inspection on a cell-by-cell level. This approach is described by Panko [Pan97]. Although it is reported that more than 90% of the errors can be found, the major drawback of this approach is that it is very expensive. Of course, the same is true for reviews in the conventional software development process. However, the problem awareness in software engineering motivates developers to spend time and money in order to increase the software quality. As end-users are not aware of the quality problems, they are not willing to take the overhead into account.

Modern testing tools (see Chapter 5 for an overview) are nearly as effective but easier to apply. Nevertheless, there are still two more drawbacks of testing approaches in general: firstly, although it sounds strange, testing of spreadsheet programs and the correction of errors tend to entail further inconsistencies of the spreadsheet model and hence, will increase the errors coming up in further maintenance operations (see the results of a field audit reported by Clermont et al. [CHM02]). As testing is performed on the numerical level, corrections are often also made on the numerical level only, i.e. a formula that produces an unexpected output is 'corrected' by overwriting it with the expected value or by introducing spurious *if-statements*.

The second drawback are costs. As business applications of spreadsheet programs tend to become large (20 columns and 200 rows are common), the testing on the required cell-by-cell level is too time-consuming to be taken into consideration. Hence, it should be possible to find *dangerous spots* on the spreadsheet and identify copies of groups of cells already tested. These copies do not have to be tested again.

1.2 History of Spreadsheet Systems

As spreadsheets are a helpful modeling tool and support problem formalization by the user (see Nardi and Miller [NM90a]), they have already existed before the advent of personal computers in offices. However, the decision to use computers in business organizations was strongly influenced by the availability of spreadsheet systems (see Browne [Bro02] or Power [Pow02]).

The time-line in Table 1.1 on the following page gives an overview of spreadsheet development and divides it into *the pre-PC area*, *the early spreadsheet systems*, *current spreadsheet systems* and *trends*. Each of these parts will be briefly described in the following subsections.

1.2.1 The Pre-PC Area

Before spreadsheet systems or even computers were available accountants already used sheets of papers that were divided into rows and columns in order to arrange their accounts and figures in a readable way. These sheets, so-called spreadsheets, have not only given the name to the modern, computer supported spreadsheet systems. In fact, the long lasting use of *traditional*, not computerized spreadsheets opened the offices' doors for spreadsheet systems.

	Era	**Year**	**Invention**
	Pre PC		Non computerized spreadsheets for accountants
		1961-1964	Spreadsheet implementations for simulations of financial models on mainframes. Implemented with `Fortran IV`.
		1978	*Software Arts*, the company that released `VisiCalc`, is founded.
	Early Spreadsheets	1979	`VisiCalc`, the first interactive spreadsheet system, is released for the `Apple II`. The system is developed by *Software Arts*, marketing is done by *VisiCorp*.
		1981	`VisiCalc` is ported to different platforms, among them the `IBM PC`.
		1983	Legal conflicts between *VisiCorp* and *Software Arts* favor the invention of `Lotus 1-2-3`.
		1985	Due to a lack of development *Software Arts* is sold to *Lotus*, although they won the law-suit against *VisiCorp*.
		1984	`Microsoft Excel` is released, originally for the `512K Apple`.
	Modern	1987	`Microsoft Excel 2.0` is released for `MS-DOS`. It is also one of the first applications that is ported to `Microsoft's` newly released `Windows Operating System`.
		1992	Spreadsheet systems from other companies are released for `Windows`.
		1995	`IBM` acquires Lotus Development. `Microsoft Excel` is market-leader.
	Trends	2000–	Numerous spreadsheet systems are freely available for the open-source operating system `LINUX`.

Table 1.1: Overview of spreadsheet development over the last fifty years

In the middle of the 1960s the first computerized spreadsheets for the simulation of financial models were implemented on mainframe computers by Richard Mattesich and his research group (see [Mat02]). The chosen programming language was `Fortran IV`. These simulations were mere first steps and the environment had nothing in common with the capabilities of modern spreadsheet systems. However, a baseline for the rapid development in the next forty years was laid.

1.2.2 Early Spreadsheet Systems

Although spreadsheets are often argued to be the obstetrician that enabled the breakthrough of the `IBM` personal computers, they were actually invented in 1979, i.e. two years before the PC. The first spreadsheet system for microcomputers, *VisiCalc*, was available for the `Apple-II` and later on for the `HP85` and `HP87` systems (see [Pow02]).

Modern spreadsheet systems have a user interface that is still similar to the one offered by *VisiCalc*. It has only been subject to minor changes in the last 20 years in order to add mouse-support and diagram capabilities (see Figure 1.1 on page 7).

After the release of the `IBM` PC in 1981 *VisiCalc* was immediately ported to the new platform and the software, sold for $100 was often the trigger for companies to buy PCs that were worth a multiple of this value. In the dawn of the PC-era *VisiCalc* dominated the market. However, after some company-internal conflicts (see Power [Pow02], Bricklin [Bri00] or [Fle02]), Lotus became the new market-leader.

The spreadsheet functionality of the *Lotus-1-2-3* spreadsheet system was a small increment to *VisiCalc's* functionality, but it adopted faster to new operating systems and the handling was more user friendly. Additionally, new features, like diagrams and database-integration were added. Brandel [Bra99] states that *VisiCalc* was slow in adopting the new capabilities of the PC, i.e. the greater memory and the 16 bit capabilities. Consequently, Lotus dominated the software market in the early and mid 1980s when Microsoft was still gathering strength with increasing sales of the `MS-DOS` operating system and their word-processor `MS-WORD`.

Although Microsoft was successful in selling software for IBM PCs, the initial release of the spreadsheet system *MS-EXCEL* was sold for the 512K *Apple* in 1984. Nevertheless, three years later *MS-EXCEL* was ported to the IBM-PC to run under `MS-DOS 2.0`.

1.2.3 Modern spreadsheet systems

The release of Microsoft's operating system with a graphical user interface, `Windows`, was at the same time the start of a new era of spreadsheet systems. The spreadsheet user interface was improved by adding new features, like mouse operation or drag and drop.

By company policy, Microsoft's *Excel* spreadsheet system was one of the first available spreadsheet systems for the new environment. The competitors of Microsoft have not been able to compensate this advantage and, consequently *Excel* has remained the standard spreadsheet system up to now.

However, only little development has taken place since *Lotus-1-2-3*. The most important feature that was added was the integration of an expressive macro-language. This macro-language enables the spreadsheet user to add arbitrary imperative functions to his spreadsheet. More sophisticated use of this extension will turn the spreadsheet into a kind of database or user interface, supporting the conventional program written in the macro-language.

Nevertheless, the main application area for spreadsheets is still end-user programming. In this field the macro-language is only scarcely applied and often deteriorates the understanding of the developed spreadsheets.

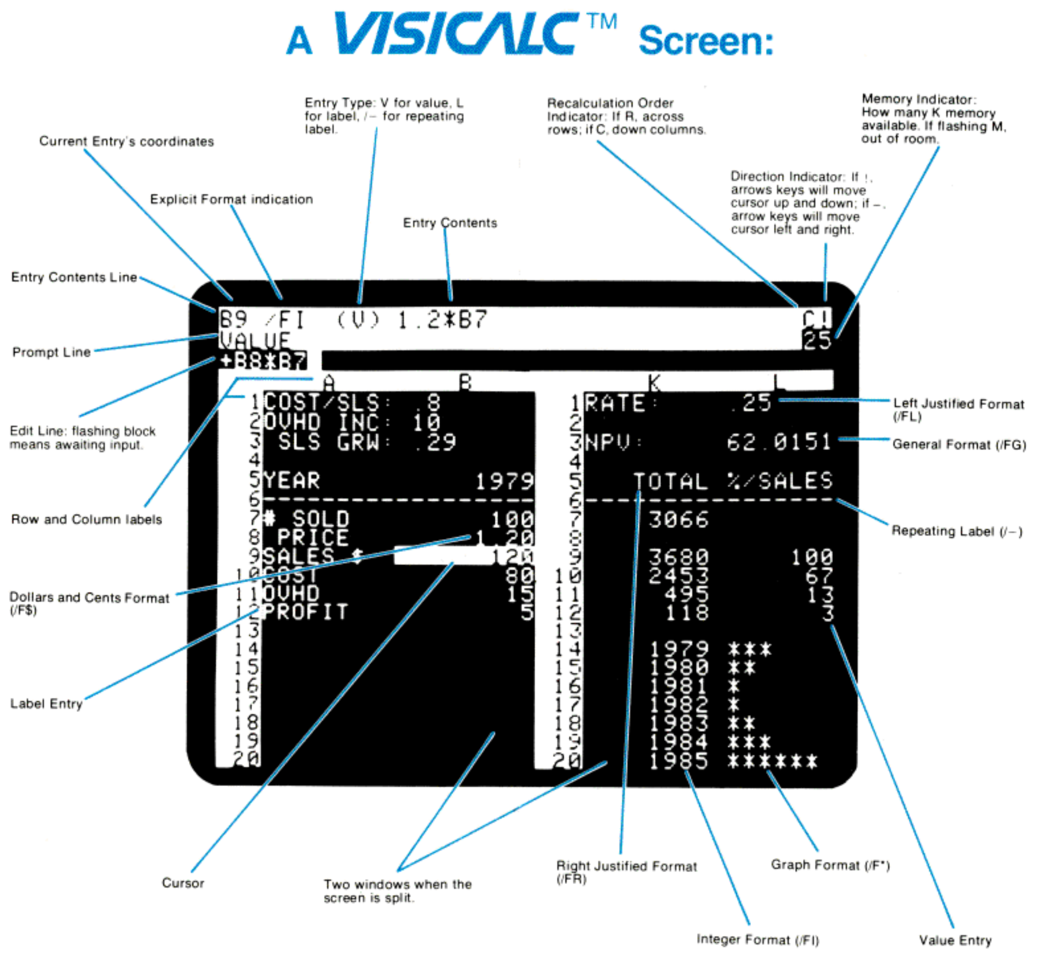
1.2.4 Trends

Most likely, *Excel* will remain the market-leader for the next years. However, with the propagation of new operating systems (like *Linux*), a new generation of spreadsheet systems is also likely to come up. Browne [Bro02] counts more than 20 different spreadsheet systems for different UNIX platforms. In contrast to spreadsheet systems currently dominating the market, these spreadsheet systems are more flexible due to the fact that most of them are open-source software. Therefore, new core functionalities can easily be implemented and will be part of the next release.

Hence, new ideas and development technologies can easily diffuse through these systems. This might be a way out of the lack of inventions in the last 20 years of spreadsheet system development.

One of the above mentioned spreadsheet systems is *Gnumeric*, which is part of some linux distributions. *Gnumeric* is subject to an open-source license and therefore, it is freely available and can be modified. The currently implemented functionality is similar to *Excel*'s. However, there are more innovative techniques used to achieve the same results. These techniques include the representation in `XML` (see Bradley [Bra98] for more details on XML) of spreadsheet programs.

New functionality can be added by plug-ins, a feature that is available for *Excel*, too, or by extending the source code. The latter is more efficient because the formula parser and several internal data-structures can be accessed. These possibilities were a prerequisite for the spreadsheet toolkit developed in this work to be integrated into the spreadsheet system (see Chapter 7).


A VISICALC™ Screen:
Current Entry's coordinates
Explicit Format indication
Entry Type: V for value, L for label, /– for repeating label.
Entry Contents
Recalculation Order Indicator: If R, across rows; if C, down columns.
Direction Indicator: If !, arrows keys will move cursor up and down; if –, arrow keys will move cursor left and right.
Memory Indicator: How many K memory available. If flashing M, out of room.
Entry Contents Line
Prompt Line
Edit Line: flashing block means awaiting input.
Row and Column labels
Dollars and Cents Format (/F$)
Label Entry
Left Justified Format (/FL)
General Format (/FG)
Repeating Label (/–)
Cursor
Two windows when the screen is split.
Right Justified Format (/FR)
Graph Format (/F*)
Integer Format (/FI)
Value Entry

```
B9 /FI  (V) 1.2*B7                          C!
VALUE                                       25
+B8*B7
     A         B               K         L
 1COST/SLS:  .8             1RATE:      .25
 2OVHD INC:  10             2
 3 SLS GRW:  .29            3NPV:    62.0151
 4                          4
 5YEAR          1979        5   TOTAL %/SALES
 6--------------------      6----------------
 7# SOLD         100        7    3066
 8 PRICE        1.20        8
 9SALES $        120        9    3680     100
10COST            80       10    2453      67
11OVHD            15       11     495      13
12PROFIT           5       12     118       3
13                         13
14                         14    1979 ***
15                         15    1980 **
16                         16    1981 *
17                         17    1982 *
18                         18    1983 **
19                         19    1984 ***
20                         20    1985 ******
```

Figure 1.1: The user interface of the initial *VisiCalc* spreadsheet system (see [Bri00] for more screenshots). The displayed version has already got the $A1$-notation for cell-addresses.

1.3 Roadmap of the Thesis

The next chapter is meant to give the reader an insight into different ways of using spreadsheet programs in business. Users who are already familiar with these issues might want to skip the first two sections of the chapter. However, the third section is essential for the further understanding of this thesis, as the vocabulary that will be used throughout this work is defined there.

Chapter 3 points out the main differences between spreadsheet programs and conventional software. It is highly recommended to go through this chapter in order to gain insight on why the conventional software engineering approach to quality assurance is doomed to fail with spreadsheets. Some theoretical issues which are of importance for the proper understanding of the background of this work are outlined in Section 3.3. Readers who are mainly interested in the application of this approach might want to skip this section.

In Chapter 4 the importance of quality assurance for spreadsheet programs is stated and supported by the results of some studies that examined spreadsheet programs that are in use by major companies. One of these studies was also used to demonstrate the efficiency of the auditing approach that is introduced in Chapter 6.

Chapter 5 discusses different methods and techniques for spreadsheet development and auditing that are currently discovered in the literature. The purpose of the chapter is to work out why these approaches are not generally used in practice, although most of them have promising concepts. Nevertheless, some of the ideas are also important for the visual spreadsheet auditing approach introduced in Chapter 6.

Chapter 6 introduces the visual spreadsheet auditing methodology developed in this work. It consists of 3 different approaches. Each of the individual approaches, its advantages, disadvantages and limits as well as possible auditing strategies for the approaches are discussed in detail and explained by an example. This chapter can be considered crucial both for readers interested in the application and in the theoretical background of the technology.

In Chapter 7, the auditing toolkit that was implemented as part of the work, is described from a technical point of view. The implementations of the most important algorithms as well as architecture and control flow of the individual components are briefly described.

Finally, the Chapter 8 discusses again the advantages and disadvantages of the introduced approach compared to conventional spreadsheet auditing and testing methodologies. Furhtermore, possible applications of this methodology, that might not be limited to spreadsheet auditing, and further improvements are outlined. In Figure 1.2 on the facing page a graphical visualization of the interdependencies between the chapters is shown.

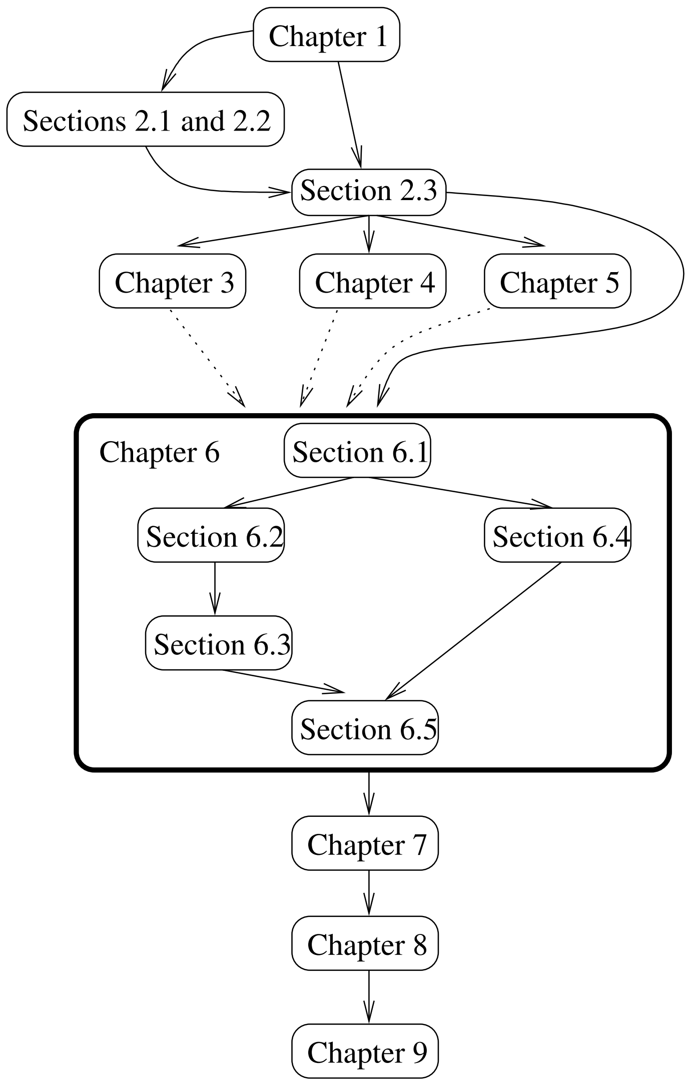


Figure 1.2: Roadmap of the thesis. Solid edges indicate crucial dependencies whereas dotted edges indicate dependencies that do not influence the further understanding. Chapter 6 that is boldly framed can be considered the main contribution of this thesis.

Chapter 2

Spreadsheets in Practice

As spreadsheets are a universal tool, there are also different kinds of spreadsheets that can be distinguished on basis of their application area. In Section 2.1 categories to apply to some classifications of spreadsheets are discussed. Consecutively, the importance of spreadsheets in business life is surveyed and some examples for spreadsheet errors are given that entailed law-suits or the bankruptcy of enterprises (see Section 2.2 and Section 2.2.3). An unambiguous terminology for dealing with spreadsheets is introduced in Section 2.3.

2.1 Applications of Spreadsheet Systems

As spreadsheets are a very powerful tool, the possible applications of spreadsheets are nearly unlimited. In this section the spreadsheet usage in three domains, i.e. *Science and Engineering*, *Business*, and *Programming*, is briefly described. Usually, all spreadsheets have the main task to gather data and perform some kind of analysis. However, typical business spreadsheets consist of simple calculations, like summing or calculating mean values, and these calculations tend to be repeatedly applied to different input data.

In contrast, scientific spreadsheets might also have a huge number of input parameters, but the calculations tend to be much more complex and are usually not repeatedly applied. These differences must be considered in spreadsheet development techniques, too. Whilst scientific spreadsheets have to be developed, maintained, and thus, tested on a cell-by-cell basis, the regular re-occurrence of certain patterns is an important characteristic of business spreadsheets.

The typical scientific spreadsheet is used for the modeling of mathematical equations or for the calculation and visualization of statistics. Orvis [Orv98] states that spreadsheets in physics and electronics engineering are mainly used to evaluate experimental data. As physicists need to solve complex equations that often cannot be solved analytically, they depend on numerical solutions. Rather than writing complex special purpose computer-programs, many of these equations can be solved easily by a spreadsheet. Together with their visualization and layout capabilities their solutions are more understandable for the user. Orvis [Orv98] argues that the presentation of intermediate results, which is one of the characteristics of spreadsheets that are discussed in Chapter 3, will increase the scientist's understanding of how his methods work and where the shortcomings are.

Although the examples presented in the literature, e.g. Orvis [Orv98], Gorni [Gor98], Julian [Jul98] or Kokol [Kok87], are computationally complex, they tend to consist of uniform rows or columns that contain parameters and formulas.

Another kind of numerical spreadsheets is used in science and engineering. Spreadsheets of this kind do generally not deal with the evaluation of experimental data but with the solution of certain differential equations (see e.g. Neuwirth [Neu98], Leroux and O'Brien [LRO98], Leharne [Leh98], and Shaw [Sha98]). These spreadsheets consist of an area to specify some input parameters and an irregular part where calculation is performed. Usually a formula is not copied into other cells.

2.1.1 Spreadsheets in Business Applications

Spreadsheets are a very important application in modern organizations. They are used to

- perform *what-if* analyses (see Teo and Tan [TT00]),
- make forecasts,
- calculate time-series (see Ragsdale [RP00]),
- calculate results of previous business periods (see Clermont et al. [CHM02], Subsection 4.3.2), and
- communicate with government agencies, i.e. the tax office (see Butler [But00]).

The above list makes it obvious that spreadsheets are of vital economic importance for organizations.

Moreover, the typical spreadsheet in business life is created by domain specialists. It consists of regular blocks that contain either equal formulas, or a certain pattern of formulas that is used repeatedly. Clermont et al. [CHM02] presents a field audit of business spreadsheets that examined 60, 000 cells in 78 spreadsheets. 34, 570 out of 36, 429 examined formulas were copies of the 1, 859 distinct originals.

Type	**User**	**Avg. Size**	Computational Complexity	Formula Patterns
Scientific Statistical	Scientist, Engineer	middle	high	regular
Scientific Numerical	Scientist, Engineer	small	very high	irregular
Business	Middle Management, Officers	large	low	highly regular
Algorithmic	IT Professional, Various	middle	low	regular

Table 2.1: Summary of different spreadsheet types in terms of users, average size, computational complexity and regularity

2.1.2 Spreadsheets in Programming

Some authors (see e.g. Casimir [Cas92], Orvis [Orv98]) state that spreadsheets are powerful programming tools because they unify the concepts of procedural and functional programming languages with a powerful user interface.

Considering these prerequisites spreadsheets should also be applicable for the solution of typical programming problems. Indeed, Casimir [Cas92] introduces spreadsheets for the calculation of *Fibonacci-Numbers*, the *factorial* of an arbitrary number, the simulation of *finite automatons* and the *game of life*, a *selection sort* algorithm, and, among others a solution for the *Towers of Hanoi* problem. As spreadsheets are not capable of expressing recursion, recursion is substituted by iterative solutions. In order to avoid circular references, for each iteration a new area is used on the spreadsheet (see Table 3.2 on page 38). Therefore, a regular geometrical pattern of formula-usage on the spreadsheet is generated.

Table 2.1 summarizes the different types of spreadsheet applications in terms of their users, average size, average computational complexity and the regularity of formula usage. This table states average values that are typical for representatives of each class.

2.2 Spreadsheets in Business

In order to introduce the thesis' subsequent focus on the application of spreadsheet systems in business, in the course of this section the importance of spreadsheets in organizations will be outlined.

2.2.1 Spreadsheets as Management Information Systems

Spreadsheet programs are considered strategic software (see Table 2.3 on page 16 and Tampoe and Taylor [TT96]). Therefore, they are widely used as a means of performing important analysis tasks and internal communication in the enterprise. Chan and Storey [CS96] found out that about 80% of spreadsheets created in companies are used as reports to the superiors in the organization; 14% find their way up to the desks of the vice-presidents.

These findings are somehow surprising considering the attitude of management towards the usage of IT as management tool. From 1991 to 1993, O'Brien and Wilde [OW96] interviewed 265 Australian upper and middle managers about their IT-usage. Generally, only 193 of them had a computer in their office. However, as will be shown later in this section, the remaining 72 are still confronted with spreadsheets or the results of spreadsheet programs.

On the other hand, O'Brien and Wilde [OW96] found out that amongst 261 representatives of 226 distinct organizations (this time, not at the management level) 238 have spreadsheet systems available on their desktop[1]. These findings were also confirmed by Chand and Storey [CS96].

European studies (see Vlahos et al. [VF92, VFK00]) showed similar figures, although the general level of IT-usage by managers is higher, which may be due to the fact that the interviewed managers where younger and had IT-education. However, spreadsheet applications are used by 62% (sample of [VF92]) and 71% (sample of [VFK00]) of the interviewed managers. This is remarkably high, bearing in mind that in the year 2000 only 59% made use of e-mail and only 24% of a web browser.

A study that was carried out in 1986 by Summer and Klepper [SK86] showed that the application of spreadsheet programs made up to 74% of the IT-usage of decision makers in companies. Although the study is not current, the experience that was gained during a large field audit (see Clermont et al. [CHM02]) supports these findings. Additionally, other remarkable findings of the study that are of qualitative nature and are still up to date, involve that

- the typical end-user application has more than 1 user.
- the scope of these end-user applications is either single- or multi-departmental.
- depending on the economic sector of the surveyed companies, the volume of processed data per application ranges between 100 and $1,000$ records for strategical operating companies or more than $1,000$ for manufacturing companies.
- the typical end-user program has a total lifetime of more than two years.

[1]Only the omnipresence of word-processors- available on 254 desktops- was superior to the spreadsheet applications

Cost range	Respondents	Percentage
$< \$1,000$	6	2.5%
$\$1,000 - \$10,000$	10	4.2%
$\$10,000 - \$100,000$	7	2.9%
$\$100,000 - \$1,000,000$	8	3.3%
$> \$1,000,000$	11	4.6%
unknown	141	58.8%
cannot disclose	57	23.8%
Total	240	

Table 2.2: spreadsheet users estimates of the cost of an error in the spreadsheet. Taken from Table 13 in [CS96].

Though, this study examined end-user programs in general, they still found out that spreadsheet programs are the most important end-user programs in the surveyed companies. Therefore, the above mentioned properties can be generalized to spreadsheet programs.

As the analytical capabilities of spreadsheets have increased since 1986 by adding diagram-features and equation solving, it is argued that the importance of spreadsheet programs has increased, as well. Summer and Klepper [SK86] assumed that spreadsheets can be used for *simple analysis*. Up to now, studies have reported that spreadsheets are employed for very complex analysis tasks, since the domain specialist is familiar with them (see Chan and Storey [CS96] or [Orv98]).

Although 41% of the spreadsheet users are generally aware of the importance their spreadsheets have for the organization (see Table 2.2), most of them are self-taught and have no formal spreadsheet training or IT skills. Additionally, the majority of 58.8% has no idea of the potential damage an erroneous spreadsheet might entail.

Another interesting result by Chan and Storey [CS96] was the fact that the better the spreadsheet expertise of a certain user is, the less this user esteems the value of his word-processor. Therefore, it is assumed that these users start to use the formating capabilities of the spreadsheet systems to lay out documents.

Despite some quality problems spreadsheets are still considered trustworthy. The reasons are different and depend on the age structure of the management:

1. The *old generation* of managers have used the paper-and-pencil 'spreadsheet' themselves. As they do not tend to have terminals in their offices at all (see O'Brien and Wilde [OW96] and Chan and Storey [CS96]), they are only confronted with printouts. Hence, they are not aware that their decisions are based on highly complex programs that are created by end-users. Often, the spreadsheet printout is considered the output of a word-

Strategy applications	**Packages used**
Mergers, acquisitions and divestments	Shareholder value model, Spreadsheets
Diversification	Shareholder value model, Spreadsheets
New product development	Hiview
Risk assessment	Spreadsheets, Risk, Hiview
Technology forecasting	None
Opportunity appraisal	Spreadsheets
Consensus building	Hiview and custom made software
Restructuring and turnaround	Spreadsheets
Strategic alliances	Spreadsheets
Investments in capital projects	Spreadsheets
Joint ventures	Spreadsheets
Assessing the business environment	Custom made software, Spreadsheets
Evaluation of alternatives	Custom made software, Spreadsheets
Simulations	Spreadsheets, Scenarios, P&L, Balance Sheet
Process modeling	IDEFF
Financial modeling	Spreadsheets
Creative thinking	Vensim, Blank sheet of Paper, Lotus AMIPRO, I-Think, Hiview
Voting on propositions	None
Making qualitative judgments	Custom made software, Spreadsheets
Forecasting	Spreadsheets, SPSS, Regression Models
Logic diagrams, decision trees or preference diagrams	Freelance

Table 2.3: "Strategy software" in companies. Taken from Figure 2 in [TT96].

processor.

2. The *new generation* starts to use spreadsheets at university[2]. Therefore, they take the use of spreadsheets for granted. However, they have no IT-training and are, therefore, not aware of the dangers of writing straightforward unstructured and complex spreadsheet programs.

The forthcoming generation change in management is supposed to make the second argument more important, while argument one will gradually vanish (see O'Brien and Wilde [OW96]).

2.2.2 Spreadsheets as the Organization's Interface

Another important task of spreadsheets is to support communication not only within the organizations, but also between subsidiary companies and their parent companies (see Clermont et al. [CHM02]), and between companies and official agencies, such as tax-offices (see Butler [But00]).

Hence, erroneous spreadsheets do not only bear strategic risks, they can even entail law-suits. The decision on whether a spreadsheet is 'only' erroneous or was deliberately manipulated is difficult to be made and often whithin the discretion of some auditors.

Therefore, the spreadsheet programmer must be able to assess the risk of his spreadsheets. Depending on the risk different quality assurance means have to be employed. Even non-IT experts get insight into the necessity of careful testing or checking of spreadsheets seeing consequences of failures such as the economic bankruptcy or a prison sentence for tax fraud.

2.2.3 Errors and Consequences

Table 2.2 on page 15 shows the user's estimate for potential damage due to spreadsheet errors. At this point, the question arises whether there is a safeguard against the damage likely to occur. By giving some famous examples from literature (see Panko's spreadsheet web site [Pan02a] and [But00]) in Table 2.4 on the following page this question can be answered with **No**!

The cited examples represent only a fraction of the spreadsheet errors that tend to occur. On the one hand, only a small number of spreadsheet errors is discovered at all. Spreadsheet users tend to blame the unpredictability of events for inaccurate forecasts rather than erroneous models. On the other hand, discovered spreadsheet errors are for the sake of the reputation of the spreadsheet

[2]In this university, business administration students are trained in spreadsheet usage since 1987. Alexander [Ale96], Kovar and Evans [KE01] and Janvrin and Morrison [JM00] give further examples for the teaching of spreadsheet programming or the bias spreadsheet training had on spreadsheet development studies.

user or the company itself rather silently corrected than published. Bankruptcy because of erroneous spreadsheets might have happened in several cases, but spreadsheets were seldom declared to be the cause.

The causes for the reported errors vary. Amongst others, common sources are modeling errors, rounding errors, typographic errors, or handling errors. At any rate, the result can be very unpleasant. Table 2.4 on the next page gives an overview of famous errors, their source, and the estimated damage.

2.3 Terms and Concepts

Obviously, this work deals with spreadsheet related issues. The terms used in this context are already overloaded with different meanings, that is why possible ambiguities have to be eliminated by defining a spreadsheet-related vocabulary. As spreadsheets are an end-user programming tool, this work will basically not distinguish between the programmer and the user. Generally, the term *user* will be used. Only in special situations the role of groups of people is stressed by calling them programmers or writers.

2.3.1 Basic Building Blocks

In this section the terms *cell*, *spreadsheet*, *formula* and the cell reference mechanism will be introduced. These terms form the baseline of spreadsheet understanding.

Static Structure

For the spreadsheet user, a cell is a rectangular area on the user interface that displays a value according to specific format rules. It has basic editing capabilities. As formatting issues are not important for our further definitions, we will neglect this issue in our definitions. As each cell is either empty or contains a value, the domain of cell values is outlined as follows:

Definition 1: Values

The set of values, V, that is referenced by the subsequent definitions, is defined by $V = \mathbb{Q} \cup \mathsf{Strings} \cup \{\texttt{Error}\} \cup \{\texttt{Undefined}\}$. $\square$

$\mathbb{Q}$ denotes the set of rational numbers, $\mathbb{Q} = \{\frac{a}{b} | a \in \mathbb{N}, b \in \mathbb{N} \setminus \{0\}\}$. Each cell has a unique cell address.

Definition 2: Cell Address (CA)

The cell address is an n-tuple $(c_1 : \mathbb{N}, \cdots, c_n : \mathbb{N})$. $\square$

In two-dimensional spreadsheets, the cell address is given by a pair (r, c), with r denoting the horizontal and c the vertical distance of a cell to the upper left corner of the spreadsheet UI. The reader might be familiar with the `A1`-notation for cell addresses. Nevertheless, we will use the `R1C1` style for addressing cells.

Description	**Damage**	**Category**
Old data source for automated ordering spreadsheet used. $30,000$ units at \$4 each instead of $1,500$ units ordered.	\$114,000	handling
Spreadsheets were used to project the market for CAD-equipment. Numbers were rounded to whole dollars. As the inflation multiplier was neglected, \$1.06 became \$1. The market was underestimated by \$36,000,000	crucial	rounding
Discounted cash flow was used to evaluate investment proposals. The spreadsheet had been implemented by a programmer who left the company long ago and did no documentation. Although the prime rate rose from 8% to 20% between 1973 and 1981, the spreadsheet was kept at 8%.	severe	modeling
A Florida construction company used `@sum` to total numbers in a range. The range was not changed when an item was added, so the added item was not summed. This made the company to underbid the project by a quarter of a million dollars. The company sued Lotus.	$>$ \$250,000	modeling
At the Fidelity mutual fund, a spreadsheet was used to report distributions for various funds. For the huge Magellan fund, a \$4.32 per share capital gains distributions was forecast in November, and investors were notified. However, in december the company announced that there would be no distribution. A clerical worker put the wrong sign in front of a \$1.2 billion ledger entry. This "created" a \$2.3 billion gain instead of the real \$0.1 billion loss.	severe	handling
In a North Carolina election, incorrect results of an election were about to be posted. Mr. Woodbury, using a calculator, detected an inconsistency. Examination revealed an incorrect cross-tabulation in the spreadsheet being used to post the results.	severe	modeling
In 1992, among 131 tested tax-calculation spreadsheets, that were sent to British tax officers 14 contained material errors. The examined spreadsheets were for rather simple calculations of value added tax.	5,000,000	various

Table 2.4: Reports of spreadsheet errors in practice (see [Pan02a, But00])

In the more common so-called `A1`-notation the letter specifies the column (`A` for 1, `B` for 2, and so on) and the number specifies the row address of the cell.

The `R1C1` address specifies the row-coordinate of the cell after the letter `R` and the column-coordinate of the cell after the letter `C`. The `R1C1` style is used in the internal formula representation of most of the spreadsheet systems.

The cell is the atomic unit of a spreadsheet user interface. We need to address the concept that a cell is the holder of all information of a spreadsheet.

Definition 3: Cell (C)
A cell is defined by a triple $(c : CA, v : V, f : (\cdots) \rightarrow V)$. c specifies the cell address of the cell. v is the value that is displayed in the cell. v can either be entered as a constant value or it is calculated by the formula that is specified by the formula f (see Definition 9 on page 22). $\square$

Other aspects of cells, e.g. editing or formatting rules, can usually be inspected only on a cell-by-cell basis via the standard interface.

The spreadsheet itself is a collection of cells. The user interface displays the cell values in the assigned positions. So, there are two views on the spreadsheet:

Definition 4: Spreadsheet (S)
Spreadsheet $S = \{C_i | i = 0, 1, \ldots\}$ is a set of cells. $\square$

Definition 5: Spreadsheet UI
Spreadsheet UI is an n-dimensional array of cells. The spreadsheet UI is the tabular user interface that renders the values of certain cells. $\square$

Generally, the term spreadsheet refers to Definition 4. Otherwise we will use the term *spreadsheet UI*.

Although our definition for the cell and the spreadsheet dealt with a general, n-dimensional spreadsheet, we will limit n to the value of 2 in our further considerations, because we assume that the typical spreadsheet tends to become very complex in two dimensions already. Moreover, only few more-dimensional calculations are performed. Furthermore, the UI is bound to two-dimensional displays, anyway.

Computational Structure

The function to calculate a cell's value is not limited to constant operands. The user can use the value of other cells as input to the function. The values will be looked up dynamically by the spreadsheet system, whenever the cell's value is calculated.
In the following definition $\oplus$ denotes the vector-addition that is defined by
$(a_1, b_1) \oplus (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$.

Definition 6: Cell-Referencing Function
The relative cell-referencing function $cref(src : CA, id : CA) \rightarrow V$ returns the value that is associated with the cell address $src \oplus id$. $cref(src, id) = v$, with $(src \oplus id, v, f) \in S$. *src* usually denotes the address of the referenced cell, *id* is considered the origin of the coordinate system. $\square$

Definitions 7 and 8 introduce two conceptually different ways to invoke the cell referencing function.

Definition 7: Relative Cell Reference
A call to the cell referencing function $cref(src, id)$ with $id \neq (0, \cdots, 0)$ is called a relative cell reference. $\square$

The cell address of the referenced cell is passed as *id* to the referencing function, and the source address *src* is specified by its distance along the coordinates of the spreadsheet relative to the referencing cell. Hence, the value that is returned by the cell-referencing function will depend on the cell address of the referencing cell.

The relative referencing mechanism has the advantage that the context of a formula is dependent on the cell address that it is attached to, thus it is *location dependent*. Therefore, copying or moving the cell will also update the addresses of the referenced cells.

Definition 8: Absolute Cell Reference
A call to the cell referencing function $cref(src, id)$ with $id = (0, \cdots, 0)$ is called an absolute cell reference. $\square$

Hence, an absolute cell reference will globally return the same value regardless of the cell address of the referencing cell, Therefore, it is *location independent*.

The cell address of the referencing cell does not have to be fully specified. In this case only certain parts of the cell reference behave relative. E.g., it is possible that the column address of the referenced cell is fixed, whereas the row address is relative, by specifying the arguments of the relative cell reference function $id = (0, c)$ and $src = (AbsRow - 0, AbsCol - c)$, with c denoting the column address of the referencing cell, *AbsRow* and *AbsCol* the row and column address of the referenced cell.

A partially relative cell reference shares the main characteristics of relative references, and it does not meet the definition for absolute cell references: the result depends on where the reference is stated. Therefore, partially relative cell references are considered relative cell references.

Example 1: Cell References
In the syntax of the most popular spreadsheet systems, i.e. Excel, the $ sign denotes a reference to be absolute. Thus, referencing `$A$1` will always yield the the value that is displayed in the upper-left cell of the spreadsheet UI, regardless of the cell, where the reference is stated. A reference to `A1` entered in a specific cell, for instance `B1`, will be adjusted to `A2` if it is moved or copied into the cell `B2`.

However, in the `R1C1`-style, it will always be `R0C-1`. A partial relative reference, e.g. `$A1` in cell `B1`, will always reference a cell in the first column. However, if it is stated in `B2`, `C2` or `K2`, it will always return the value displayed in the cell `A2`. $\Diamond$

Cell referencing is the fundamental construction mechanism for spreadsheets, because it enables the linkage between values and functions of different cells. A cell can be referenced regardless of its contents.

The function that can be specified to calculate the value that is associated with a cell is called the cell's formula if it contains either absolute or relative cell references.

Definition 9: Formula
A formula is an expression in the spreadsheet system's formula language. The formula is a function $f(cref^+) \rightarrow V$. f has to be specified by a string of finite length. $\Box$

Constants in the formula are not considered to be part of its input, as they are definitely static and cannot be changed without re-writing the formula. The result of the formula is directly bound to the cell. However, the displayed value might be different as it is subject to formatting rules. Nevertheless, a reference to the cell will always yield the exact result of the formula.

Most of the modern spreadsheet systems offer means to use simple control flow concepts in formulas. However, the control flow cannot be specified outside the formulas. Iteration or recursion are usually not supported, only `IF`-Statements are widely available.

If a cell is defined by $((c_1, \cdots, c_i), v, f)$ with $f \neq \emptyset$, $v = \mathsf{eval}(f)$, the cell's formula is evaluated and its result is the cell's value. If $f = \emptyset$, v is either specified by the user or $v =$ `undefined`. In the latter case the cell is considered empty and appears blank on the spreadsheet UI.

2.3.2 Spreadsheet Programs and Spreadsheet Instances

In addition to the basic building blocks we want to introduce the concepts of spreadsheet program and spreadsheet instance. Basically, a spreadsheet program consists of the calculation directive and constant values that are needed to properly specify the formulas. The spreadsheet instance is a spreadsheet program plus the expected input values.

The concept of the spreadsheet program corresponds to the term *program* in conventional programming, whereas the spreadsheet instance can be compared to the execution of a program, i.e. it is the program plus the input data.

However, there is a number of important differences between spreadsheet concepts and conventional programming concepts (see Section 3.2 for a more detailed discussion):

- There are no declared constants, as the user is free to enter constant values in any cell,
- each cell is a potential input **and** output cell,
- the program is stored together with the input,
- therefore, a formula can be part of the input, because the user can enter a requested input either as value or as a formula, and,
- there are no loops.

Hence, we have to employ heuristics to classify cells in a spreadsheet to the spreadsheet program or to the spreadsheet input data. Roughly speaking, it is assumed that each cell that contains a formula (see Definition 9 on the facing page), is part of the spreadsheet program. As absolute cell references are widely used to bind constants into formulas we consider any cell that is referenced by absolute cell references also part of the spreadsheet program. Pure input data is generally not absolutely referenced, because if formulas are copied throughout the spreadsheet, they have varying input data. Thus, any other cell in the spreadsheet that is referenced by a cell in the spreadsheet program belongs to the spreadsheet input data.

Apart from these heuristics, there are also some techniques to specify a spreadsheet program as a data flow program or by means of an object-oriented programming language (see Paine [Pai97a] Du and Wedge [DW90] and Ronen et al. [RPL89], further descriptions are given in Chapter 5).

Definition 10: Spreadsheet Program (SSP)
The spreadsheet program is a subset of the spreadsheet, containing all cells with non-empty formulas or cells that are absolutely referenced by cells in the spreadsheet program. Formally, the spreadsheet program is the union of cells containing formulas and cells that are absolutely referenced by these formulas, $SSP = FC \cup AR$, with $FC = \{c | c \in S \vee (c = (ca, f, v) \vee f \neq \emptyset\}$ and $AR = \{c_j | c_j \in S \vee \exists c_k = (ca_k, f_k, v_k) \in FC \bullet f_k \texttt{ absolutely references } c_j\}$. $\Box$

From Definition 10 follows that a cell that is absolutely referenced by a single formula is considered to be a part of the spreadsheet program, even if it contains only a constant value and is relatively referenced by an arbitrary number of formulas. Of course, this is not true for cells that are only partially absolutely, i.e. partially relatively referenced. They are only considered a part of the SSP if they either contain a formula, or if they are also the target of an absolute cell reference.

Definition 11: Spreadsheet Input Data (SID)
The spreadsheet input data $SID \subseteq S \setminus SSP$ is the set of all cells that are not part of the spreadsheet program, but referenced by cells in the spreadsheet program: $SID = \{c_j | c_j \in S, c_j \notin SSP, \exists c_k = (ca_k, v_k, f_k) \in SSP \bullet f_k$ `relatively references` $c_j\}$. □

The spreadsheet program is a finite set, since it contains only cells with formulas that are specified by the programmers. As each of the cells' formulas is programmer specified, too, it can only contain a limited number of calls to the cell referencing function. Consequently, SID is a finite set.

Definition 12: Spreadsheet Instance (SSI)
The spreadsheet instance $SSI = SSP \cup SID$ is the union of the spreadsheet program and the spreadsheet input data. □

Cells that are not empty but are neither referenced by any other cell nor reference any other cell, are called dead cells. Dead cells are in the spreadsheet but are not part of the spreadsheet instance.

The following conditions hold:

1. $SSP \cup SID \cup Dead\ Cells\ = S$
2. $SSP \cup SID \subset S$
3. $SSP \cap SID = \emptyset$.

Changes in the spreadsheet input data are considered to be a re-execution of a spreadsheet program with different input, whereas changes in the spreadsheet program are considered to be maintenance. Thus, two spreadsheets with different spreadsheet input data are considered to be equal if they have the same spreadsheet program.

2.3.3 Graph Representations

Cell-references represent dependencies between cells. The data dependency graph reflects the kind of inter-cell dependencies that have to be resolved in order to evaluate the cells' formulas.

Definition 13: Data Dependency Graph (DDG)
The *data dependency graph* (DDG) of a spreadsheet is a directed acyclic graph $DDG = (V, E)$, where each cell $c = (ca_i, v_i, f_i)$ in the spreadsheet is represented by a vertex $v \in V$, if $v_i \neq$ `Undefined` $\vee \exists (ca_j, v_j, f_j) \in S | f_j \mathsf{references} c$ There is an edge $(v_1, v_2) \in E$, if the formula in v_2 references v_1. □

Only those cells that do not have an `Undefined` value or are referenced by any other cell are taken into consideration in the DDG. This restriction is necessary, because the potential number of cells in a spreadsheet is not bound. However, only a finite number of cells will be part of the spreadsheet instance and, consequently, of interest.

If a cell c_1 depends on a cell c_2, and c_2 is transitively dependent on c_1, a so-called *circular reference* occurs. In this case the spreadsheet system cannot transform the dependency graph into an acyclic graph and will either remove some edges, as it is the case in many modern spreadsheet systems, or report an error (in Excel only for the first circular reference).

Definition 14: Set Relation Graph (SRG)
The set relation graph (SRG) of a spreadsheet is a directed acyclic graph $SRG = (V_s, E_s)$. V_s contains subsets of the node-set V of the DDG. Based on the $DDG = (V, E)$, the set of inter-set-edges E_s is defined by $E_s = \{(s_1 : \mathbb{P}V, s_2 : \mathbb{P}V) | \exists v_1 \in s_1 \mid \exists v_2 \in s_2 \bullet (v_1, v_2) \in E\}$. □

In the SRG each node n_s represents a set of nodes from the DDG. There is an edge between two nodes n_1 and n_2 in the SRG if there is an edge in the DDG between an elements of n_1 and an element of n_2. Thus, the SRG is a useful aid to handle the data dependencies in spreadsheet abstractions (see Chapter 6), as it gives a representation of relations between different parts of the spreadsheet program that is not based on a cell-by-cell resolution. The semantics of the nodes of the SRG is usually denoted by a suffix, e.g., SRG_{LA} denotes a SRG with logical areas as nodes, SRG_{SC} denotes a SRG with semantic classes as nodes and in the SRG_{DM} the nodes represent data modules (see Chapter 6).

2.3.4 Different Views on the Spreadsheet

Apart from the different formal representations of a spreadsheet program, there are also different ways to perceive a spreadsheet. If users limit their view to the spreadsheet UI, they see only the values that are rendered by cells at a given location. Thus, the perception of the spreadsheet is a geometrical one.

The *geometrical* or *spatial model* of the spreadsheet is based on this perception. To describe a geometrical model, it is sufficient to specify the values displayed at a specific location. Usually, the design of a spreadsheet program is based on this model.

The *algorithmical model* of the spreadsheet is in contrast based on the DDG only. Hence, the DDG specifies the algorithmical model. Obviously, the algorithmical model does not have to correspond to the geometrical model, i.e. a cell and its dependents in the DDG can be located in totally different places on the spreadsheet UI.

The *conceptual model* of the spreadsheet manifests in the way the spreadsheet users assume that a given spreadsheet program works. Factors like formatting styles and geometrical distances as well as labels generally have a strong influence on the conceptual model. Although the geometrical model has a strong influence on the conceptual model because it is easy to notice, the conceptual model is independent from both former models.

While the geometrical and the algorithmical model are easy to specify by means of cell coordinates and values and the *DDG*, the conceptual model is subject to the individual viewpoint of a user and, thus, not formally described. Nevertheless, the users can be supported in the process of building the conceptual model by offering comprehensive representations of the spreadsheet that are not totally influenced by the geometrical model.

2.3.5 Area concepts

The spreadsheet user tends to group cells into areas, either

1. by placing them next to each other on the spreadsheet UI, or
2. by using them as input for an aggregation function[3], or
3. by calculating their values with a similar formula.

A *physical area* is a set of cells located within a rectangular area on the spreadsheet UI. A physical area is specified by the absolute or relative coordinates of the cells in the upper-left and lower-right corner. All cells in the rectangular area thus specified are part of a physical area.

There are transient physical areas that arise during spreadsheet development, when a set of cells is selected by the user in order to perform an operation, e.g. copy and paste. Non-transient physical areas are usually the input for aggregation functions in formulas. As the segmentation of the spreadsheet into physical areas depend on formulas, the following properties are true:

1. Physical areas are spatial areas on the spreadsheet UI.
2. Physical areas can overlap, i.e. a cell can be part of more than one physical area.
3. A cell does not have to be in a physical area.

Apart from the physical areas there are also cells that are somehow related but not spatially adjacent. Therefore, we introduce the concept of *logical areas* (see Mittermeir et al. [MCA00] and Chapter 6 for a more complete discussion). A logical area is a set of cells that are related in the conceptual model of the user. Typical symptoms for this kind of relatedness are similar formulas, or a neighborhood on the spreadsheet UI. As there are different criteria for the partitioning, logical areas can overlap.

[3]An aggregation function is a function that will calculate a single scalar value for an arbitrary set of input values, e.g. `SUM`, `MAX` or `AVG`.

2.3.6 Further Terms

The terms that are subsequently defined are important for the understanding of most spreadsheet systems, but they are not used for the spreadsheet auditing technique presented in Chapter 6.

The *spreadsheet editor* supplies the user with important aids for creating and modifying spreadsheets. Among the common tools, there are *copying* and *pasting* formulas, applying format guidelines to cells, simple audit tools and the presentation of results of the spreadsheet execution. Usually, the spreadsheet editor is embedded in the spreadsheet UI and is invoked, whenever the user selects a cell.

The spreadsheet system consists of the spreadsheet editor, a specific spreadsheet formula language and spreadsheet UIs. The reader might be familiar with some spreadsheet systems, like *EXCEL*, *LOTUS-1-2-3* or *Gnumeric*.

Most spreadsheet systems support the creation and usage of *macros*. There are two kinds of macros that are used in different fields for different purposes:

1. The *recorded macro* is generated by recording the actions the user executes to achieve a certain goal. The recorded steps can be repeated for other cells or groups of cells. The result of the execution of a recorded macro is similar to copying the group of cells that were initially created for recording the macro. The usage of this kind of macro cannot be observed in the spreadsheet program.

2. The *procedural macro* is a small user defined subroutine that is written in the *spreadsheet macro-language*, which is usually an imperative programming language. The procedural macro accepts a list of parameters and will return a single result value to the caller. Procedural macros in formulas are treated like built-in functions of the spreadsheet system. A special kind of procedural macros does not produce a scalar result, but it can manipulate arbitrary cells of the spreadsheet. Thus, they can change the spreadsheet programm, too.

It is obvious that the second kind of macros has to be tested and examined carefully. However, testing procedural macros is similar to testing conventional software, and therefore not part of the work presented here (see the relevant software-engineering literature, e.g. Preston [Pre92], Beizer [Bei90] or Myers [Mye79]).

The expressive power of procedural macros is not limited to support-functions for spreadsheet programs. Procedural macros can even totally change the typical spreadsheet usage to data structures and data storages for conventional software. This programming approach is often put in practice by spreadsheet experts with little IT-training.

Summary

The following issues have been addressed in this chapter:

- The basic concept of spreadsheets is a very natural one (*paper, pencil, calculator*).
- Spreadsheets are used in different application areas.
- Therefore, there are, roughly speaking, two kinds of spreadsheets:
    1. the small, but computationally very complex one, consisting of some input data and a few mathematically complex formulas.
    2. the large one, consisting of uniform patterns of data and relatively simple formulas.
- Spreadsheets of the second kind are very common in business-use.
- Spreadsheets evolved from two-dimensional tables that were common in accounting before computers were used.
- Only superficial evolution has taken place since the introduction of computerized spreadsheet systems.
- Spreadsheets are an important tool for decision making and reporting in business.
- Spreadsheet errors are a threat to the economic survival of companies.
- In order to discuss spreadsheets, we need an accurate vocabulary.

Chapter 3

Characteristics of Spreadsheet Programs

This chapter aims to point out the differences between spreadsheets and other kinds of software. Due to these differences many software engineering techniques cannot be applied in a directly to spreadsheet development.

A distinction can be made between social differences, structural differences and evaluation strategy differences. Social differences (see Section 3.1) mainly arise because of the fact that spreadsheet programs are end-user programs.

The comparison of spreadsheet programs with other kinds of programs yields interesting differences in the program structure that are surveyed in Section 3.2. The evaluation strategy of a program generally influences error propagation and runtime performance. Hence, it is important to understand the evaluation strategy of spreadsheet programs. Intuitively, spreadsheet programs seem to correspond to data flow programs. Though, this is not as shown in Section 3.3, this is not true. Finally, this chapter will define some criteria that have to be fulfilled by successful spreadsheet development techniques.

3.1 Social Differences

The most important characteristic of spreadsheet programmers is that they do not consider themselves programmers at all. Although spreadsheet systems provide the most successful end-user programming environment, some of the main features of traditional end-user languages are not implemented, e.g. visual programming is not supported by spreadsheet systems. Users have to literally specify the arithmetic expression for each cell. However, as it is stated by Nardi and Miller [NM90a], despite this apparent drawbacks, spreadsheets are wide spread throughout the desktops of end-users. (see Brancheau and Brown [BB93] for a discussion of end-user computing).

Spreadsheet systems provide end-user programming, but not in the classical sense. Nardi and Miller [NM90a] point out that the typical features of end-user languages, like visual representation of the program, are not part of spreadsheet systems. According to them the success of spreadsheet systems is due to the absence of control flow *in the large* [NM90a]. Of course, each formula has a local control flow[1], as if-statements are part of most of the spreadsheet systems' formula languages. However, there is no way to explicitly specify control flow outside the formulas, since control flow is replaced by the more intuitive concept of global data flow.

3.1.1 Spreadsheets and end-user computing

As stated above, spreadsheet systems are very popular although they cannot be considered typical end-user languages. Nevertheless, Nardi and Miller [NM90a, NM90b] came up with the following properties that support the end-user.

The tabular user interface

The tabular user interface is not only common to the end-users in business (see Section 1.2), it also supports users in modeling the problem space. In general, making models has to start with the development of an initial model structure. In discussions with people who do not have any modeling experience it turned out that one of the difficulties is to find a starting point, which is how to structure the model. The spreadsheet system already provides a common and useful structure, i.e. the tabular grid. The end-users only have to deal with making the model of their domain.

Locality

The hiding of complexity is another important feature. Spreadsheet programmers consider each cell an independent unit of the spreadsheet. Therefore, a local view on each cell is sufficient to create a spreadsheet program. Nevertheless, the hidden complexity is still present and emerges in correcting, testing and maintaining spreadsheet programs. Additionally, the programmers do not have to care about the control flow in the spreadsheet program, as they only specify data flow. The order of evaluation is subject to the spreadsheet system.

High-level constructs

The high-level constructs of the spreadsheet language provide domain specific functionality. Widely used spreadsheet systems provide, for instance, functions for the calculation of interests. Hence, programmers do not have to break down complex functions into the low-level constructs of a programming language.

[1]The local control flow can also be interpreted as case differentiation.

Learning Rate

There is only a small number of both domain specific and spreadsheet specific constructs an individual spreadsheet writer has to be aware of in order to create successful spreadsheet programs. Therefore, only little initial training is necessary to begin with spreadsheet programming. As their experience of grows, programmers learn new constructs which fit their specific needs and interests.

3.1.2 Organizational matters

In spreadsheet development projects the domain knowledge of the spreadsheet programmer should be efficiently transformed into spreadsheet programs. So, to develop important spreadsheets, domain experts can often benefit from the support of IT-specialists. However, the survey reported by Nardi and Miller [NM90b] points out that this support is hardly accepted. Spreadsheet programmers prefer the help of other, more sophisticated spreadsheet programmers in their own domain.

Although spreadsheet systems generally do not support cooperative work, Nardi and Miller [NM90b] found out that the co-development of spreadsheet programs is the rule. Therefore, programmers share both their domain knowledge and their programming knowledge. Fuller et al. [DAFP93] outline a spreadsheet with change-management to support collaborative work.

The success of the spreadsheet paradigm might also have its roots in the fact that no IT-professionals have to be involved in order to create even very important spreadsheets. This is based upon the fact that

- in many organizations, IT and other departments do not tend to cooperate,
- spreadsheet programmers do not want to give away their domain knowledge,
- spreadsheet programmers do not have to explain their complex, domain specific requirements to *novices* and
- when requirements change, spreadsheet programmers have the freedom to modify their spreadsheet programs without limitations.

The main difference between spreadsheet and conventional programmers is that the former are domain experts and the latter IT-specialists. For spreadsheet programmers problems often seem clear and simple. That is why they do not understand why to put any *overhead* in the application of software-engineering techniques. Conventional programmers are methodologically trained and should know about the complexity of programming. However, they do not have the domain specialists' deep insight into the matter.

Category	**Concept**	**SP**	**IP**
Social	High Level Constructs	5	3
	Self-adjusted learning rate	5	2
	Given Framework for modeling	5	2
	Well founded analysis and design	2	5
Structural	Control Flow	2	5
	Interfaces	$\emptyset$	5
	Subroutine invocation	3	5
	Composite data types	$\emptyset$	4
	Variables, Constants, Functions	$\emptyset$	5
	Local Variables, Scoping	$\emptyset$	5
Layout	afferent, processing, efferent parts	$\emptyset$	5
	Influence of presentation on algorithm	5	2

Table 3.1: Summary of differences between spreadsheet and imperative programs. The importance of a fact or concept is rated from $\emptyset$ (=not present) to 5 (=very high). (*SP = Importance in spreadsheet programs, IP = importance in imperative programs*)

3.2 Structural Differences

Spreadsheet programs are usually end-user programs, whereas imperative programs are generally developed by IT-professionals. The resulting differences in the development have been summarized in the previous section. However, there are some fundamental differences in the paradigms used. Subsequently, only classical spreadsheet programs which do not include procedural macros, are taken into account. The most important differences between spreadsheet and imperative programs are summarized in Table 3.1.

Structural differences between imperative and spreadsheet programs are due to the fact that spreadsheet programs look like functional programs considering a cell reference to be a function call.

Wilhelm and Maurer [WM97] consider the following three characteristics for functional languages:

1. There is no separation between statements and expressions.
2. Names are identifiers for expressions, not for memory addresses.
3. Functions can be arguments and results of other functions.

The first two properties are true for most of the spreadsheet systems. Hence, the spreadsheet paradigm incorporates principles of the functional paradigm. Some special constructs and extensions to the classical spreadsheet paradigm which is surveyed here will fulfill the third requirement as well. However, there are also some fundamental differences. If a formula is considered a function, the spreadsheet paradigm does not support nested functions.

Additionally, a functional program has a well defined output that can be input for another function. If we consider the displayed values in the cells the output of a spreadsheet program, it can be argued, that another spreadsheet program can freely reference this output, i.e. use it as input. Thus, the requirement of a well defined output seems to be met on first sight. A closer look at the output vector will reveal that the values in the output vector are not mutually independent. As a consequence, the above stated requirement is not met.

Control Flow

The global control flow of a spreadsheet program is not specified by and generally, not known to the spreadsheet programmer. It is up to the spreadsheet system to determine an efficient way to evaluate the spreadsheet program. The spreadsheet programmer can only specify control flow on the cell-level. In imperative programs the programmer must specify the global control flow throughout the system.

No interfaces

The concept of defining interfaces in order to support modular design and data-abstraction in imperative programming languages is unknown to any spreadsheet system. Therefore, in order to make use of the functionality of a certain spreadsheet program, the program's implementation has to be shown to its user.

Subroutines by copying

In imperative programming the concept of subroutines is very important to reduce the complexity and to raise the understandability of programs. However, as spreadsheet programs pass arguments to cell formulas by specifying the relative distance from the cell, the cell's content has to be copied and pasted at another location in order to invoke its formula-calculation with different arguments a second time. In contrast to the concept of subroutines, the copy-paste mechanism does not reduce complexity at all, because the programmer is free to modify or overwrite each of the copies individually. Additionally, two identical formulas need not necessarily be the result of a copy-and-paste operation.

Subroutines have a scalar result

The result of the cells' formula is displayed in the cell and can be accessed by other formulas by referencing a cell. However, the fact that the result is displayed in the cell restricts the possible output to a displayable one. Therefore, only values (numbers, text, date or boolean values) are valid results of a formula. Composite data types, such as arrays or records, that are common to most imperative programming languages are not supported by any spreadsheet system whatsoever.

Data access via cells

In imperative programming, different ways to access and store data are common:

- Data can be the result of an expression, and thus results from a parametrized function call.
- Data can be read out of a variable and can be written into a variable.
- Data can be read out of a constant.

Spreadsheet programs do not support these different kinds of data sources. Actually, there is only a single data-source: the cell. The value can be computed by a formula or entered by the user. It is not possible to distinguish between program constants and values that are supplied by the user.

Only single assignment variables

For the user, each cell is a possible input variable, but from the viewpoint of formula cells any other cell is a constant, and thus, it must not be changed. Therefore, formulas cannot deliberately change any other cell on the spreadsheet, as there is no explicit assignment operation. A formula may only read out the values of other cells and process these values in its calculation. But then, certain cells will be changed because they depend on the result of the formula. Nevertheless, dependencies are controlled by the depending formulas, not by their data source.

Only global variables

In contrast to the mechanisms of blocks and scoping in imperative programming languages, spreadsheet systems do not support modularity. Accordingly, each cell's value can be accessed by any other cell. This is one of the reasons for the inherent complexity of spreadsheet programs.

Intermediate results are visible

Imperative programs transform a specific input into an output. Intermediate results and variables are not visible to the user. Opposingly, spreadsheet programs depend on the cell as the only possible data store. As cells are generally visible[2], all intermediate results of the spreadsheet program are visible. Hence, the

[2]Although columns or rows can be hidden, they are not absolutely invisible, as the user can still reference them or re-display them by commands in the spreadsheet system's menu.

cells can be accessed by other formulas and further decrease the modularity of a spreadsheet program. However, the visible intermediate results are of benefit to debugging. In convential software special tools that are more or less difficult to apply are necessary in order to trace the value of variables at run-time. For spreadsheet programs, this feature is part of the concept.

Layout influences calculation

In imperative programs there is no influence of the result presentation on the algorithm at all. Imperative programs usually consist of an afferent, a processing and an efferent part that is responsible for the presentation of the results. However, as intermediate results of spreadsheet programs are visible, the processing part is also efferent. Though this could be helpful in order to find erroneous parts of a certain algorithm, unfortunately the complexity of spreadsheet programs is usually increased by that. Actually, there are two ways in which visible intermediate results influence the calculation:

1. The spreadsheet programmer has a geometrical model of the spreadsheet program that deviates from the algorithmic model. Things that would belong to each other from an algorithmic point of view, are not related in the domain view. Thus, the corresponding formulas will be placed at different locations in the spreadsheet UI and a relation is only hard to discover.

2. The spreadsheet programmer complicates the calculation in order to obtain an additional intermediate result.

3.3 Evaluation Strategy Differences

From the viewpoint of IT-specialists spreadsheet programs share many features with data flow programs (see Kavi et al. [KBB86]). Nevertheless, some of the key concepts, such as the consumption of tokens, cannot be found in spreadsheet programs. Besides, it is to the same extent reasonable to consider spreadsheet programs graph reduction programs.

However, none of these concepts seem to totally cover the evaluation strategy of spreadsheet programs. This conceptual mismatch is due to the interactivity of spreadsheet programs. Both data flow and graph reduction programs are conventional software. Hence, they are fully specified and then executed. A spreadsheet program is subject to a permanent cycle of changes and re-evaluation of the concerned cells.

It is argued that the spreadsheet program is mainly a functional program with an eager evaluation strategy. The re-evaluation is propagated by a data flow mechanism through cells that are adjacent in the *DDG* to the changed cells.

3.3.1 Experiments on Spreadsheet Evaluation Strategy

This section has two objectives. On the one hand it should support the arguments about spreadsheet evaluation strategy in the following sections. On the other hand, it demonstrate the impact of the evaluation strategy on spreadsheet program testing.

Furthermore, the results of two experiments are to be reported; the first one is about change propagation through the spreadsheet, and the second one about the influence of circular references on the change propagation.

Change Propagation

This experiment was performed on a Linux desktop computer with the *Gnumeric* spreadsheet system. The open-source spreadsheet system is used because it has a built-in debug mode that prints out which cells are evaluated.

The experimental spreadsheet program consists of an if-statement in cell `A5`, that returns either the sum of `A1:A2` or `B1:B2`, depending on the value in the cell `A4`. The cell's `A6` and `B6` further process the result. The value in `A4` is 1, therefore `A5` will sum up `A1:A2`. In Figure 3.1 on the facing page, (a) and (b) show the experimental spreadsheet program. As the user changes the value in `B2` from 6 to 7, a re-evaluation of `B6` and `A6` is triggered (see (c) and (d) in Figure 3.1 on the next page), although `B2` has no direct influence on any of the cells' values.

Apart from the fact that the change propagation is obviously eager[3], i.e. all cells that could be affected by a change are re-evaluated, this experiment shows another interesting feature: the evaluation starts at `B6` and `A6`, which are both sinks in the spreadsheet programs *DDG*.

Furthermore, `A5` is evaluated when its result is requested for the first time by `B6`. The second time, when its result is needed by `A6`, the result is immediately passed on without re-evaluation.

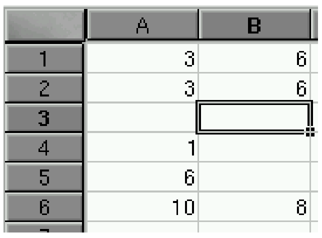
The calculation of a result when it is needed for the first time does not support the assumption that the spreadsheet program is a data flow program (see Subsection 3.3.4).

Circular references

Some authors claim that spreadsheets are capable of complex algorithms like quicksort or the *Tower of Hanoi* problem (see Casimir [Cas92]). However, these algorithms require either iteration or recursion, both concepts that are apparently not available in spreadsheet formula languages.

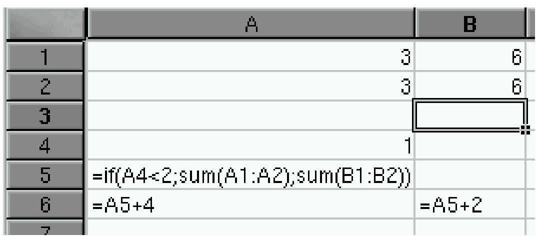
The obvious solution is loop-unfolding (see Figure 3.2 on page 38): instead of a multiple execution of the same block, the block is copied n times, where n is the number of required iterations. The output of the i^{th} computation is the input

[3]It cannot be but eager, since all intermediate results are visible.



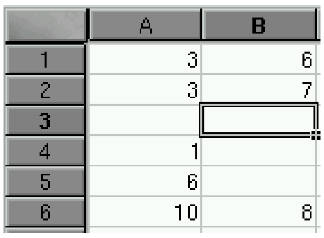
	A	B
1	3	6
2	3	6
3		
4	1	
5	6	
6	10	8

(a) Numerical View



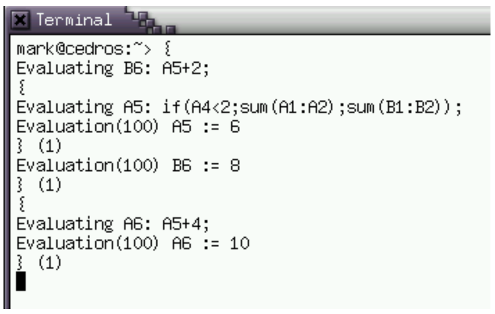
	A	B
1	3	6
2	3	6
3		
4	1	
5	=if(A4<2;sum(A1:A2);sum(B1:B2))	
6	=A5+4	=A5+2

(b) Formula View



	A	B
1	3	6
2	3	7
3		
4	1	
5	6	
6	10	8

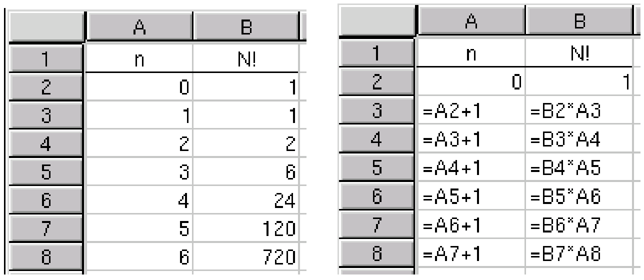
(c) Numerical View after change to `B2`



```
Terminal
mark@cedros:~> {
Evaluating B6: A5+2;
{
Evaluating A5: if(A4<2;sum(A1:A2);sum(B1:B2));
Evaluation(100) A5 := 6
} (1)
Evaluation(100) B6 := 8
} (1)
{
Evaluating A6: A5+4;
Evaluation(100) A6 := 10
} (1)
```

(d) Debug trace of gnumeric after changing `B2`

Figure 3.1: Screen shots of the change-propagation experiment in the *Gnumeric* spreadsheet system



	A	B
1	n	N!
2	0	1
3	1	1
4	2	2
5	3	6
6	4	24
7	5	120
8	6	720

	A	B
1	n	N!
2	0	1
3	=A2+1	=B2*A3
4	=A3+1	=B3*A4
5	=A4+1	=B4*A5
6	=A5+1	=B5*A6
7	=A6+1	=B6*A7
8	=A7+1	=B7*A8

(a) Numerical view (b) Formula view

Figure 3.2: Loop Unfolding is used in order to simulate iteration. In our example we calculate 6! by two formulas. In B the result of the preceding row is multiplied by the value in column A in the same row. The value in column A is calculated by adding 1 to the cell in the preceding row. Except for `A2` and `B2`, all formulas in one column are copies of the same source.

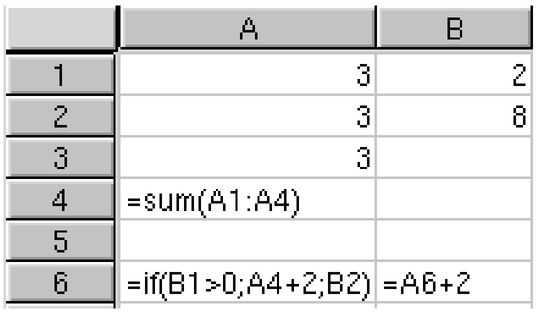
for the $i + 1^{th}$ computation. However, in order to calculate the result of $n + 1$ iterations, e.g. 7! in Figure 3.2, a further row has to be added to the spreadsheet program. Thus, loop-unfolding works only for a given number of iterations.

Loop-unfolding must not induce *loops*, i.e. circles, in the *DDG* of the spreadsheet program. Circular references, i.e. a formula is transitively dependent on its result, would induce circles in the *DDG*. Circular references are either erroneously introduced into a spreadsheet program, i.e. in an attempt to model iteration, or they result from a conceptual error.

The circular-reference experiment (see Figure 3.3 on the next page) was carried out with the *Excel* spreadsheet system, because different spreadsheet systems do not consistently deal with circular references. *Excel* issues an ignorable warning to the spreadsheet writer. However, as the *Excel* spreadsheet system evaluates the spreadsheet program on the basis of a dynamically constructed *DDG*, circular references in the not executed branch of an if-statement do not influence the evaluation order. As soon as the *DDG* becomes circular, e.g. because of a change in the pre-conditions of an if-statement, the formula that states the circular reference is considered a sink node of the *DDG*, and thus, the evaluation stops at this place. This behavior often confuses spreadsheet users, since no further warnings are issued.

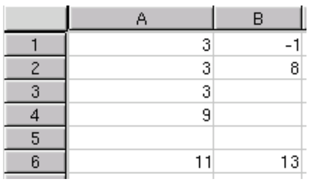
In the experiment with the *Excel* spreadsheet system (see Figure 3.3 on the facing page) an erroneous formula is specified for cell `A4`. Depending on the value in `B1` the error is either propagated into `A6` and further into `B6` or it does not influence any result. Obviously, the circular reference occurs only when the value in `B1` is above zero.

When the erroneous formula in `A4` was specified, *Excel* reported an ignorable



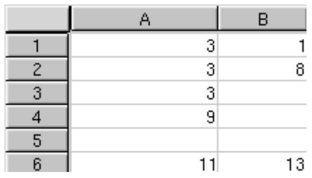
	A	B
1	3	2
2	3	8
3	3	
4	=sum(A1:A4)	
5		
6	=if(B1>0;A4+2;B2)	=A6+2

(a) Formula View



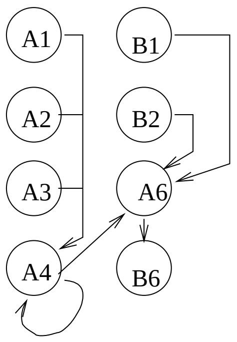
	A	B
1	3	-1
2	3	8
3	3	
4	9	
5		
6	11	13

(b) Numerical View, No Cycles



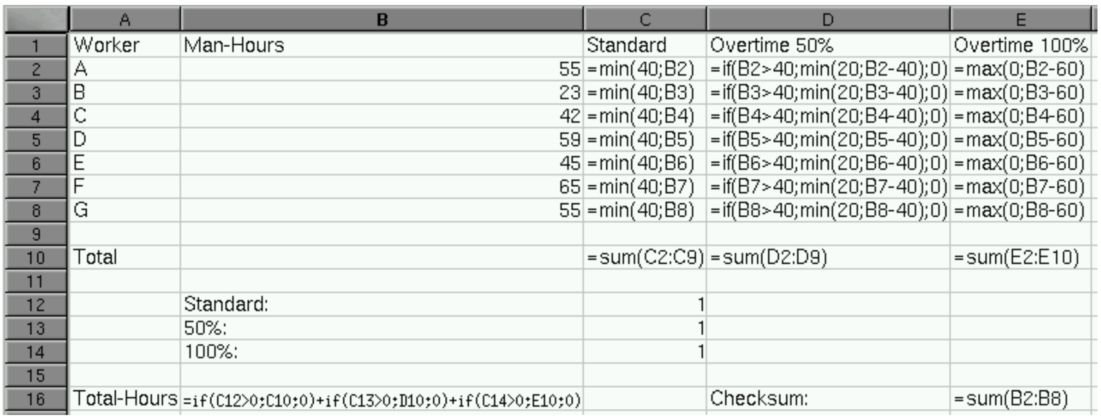
	A	B
1	3	1
2	3	8
3	3	
4	9	
5		
6	11	13

(c) Numerical View, Cycles



(d) Cyclic DDG

Figure 3.3: The circular reference experiment. Figure (a) is the formula view of the spreadsheet program. In Figure (b) `B1` is changed to -1. The formulas in `A6` and `B6` are not re-evaluated. (c) shows a correct output for `A6` and `B6`, because the loop in the *DDG* is bypassed by the *if*-statement in `A6`. Figure (d) shows the initial cyclic *DDG*.



	A	B	C	D	E
1	Worker	Man-Hours	Standard	Overtime 50%	Overtime 100%
2	A	55	=min(40;B2)	=if(B2>40;min(20;B2-40);0)	=max(0;B2-60)
3	B	23	=min(40;B3)	=if(B3>40;min(20;B3-40);0)	=max(0;B3-60)
4	C	42	=min(40;B4)	=if(B4>40;min(20;B4-40);0)	=max(0;B4-60)
5	D	59	=min(40;B5)	=if(B5>40;min(20;B5-40);0)	=max(0;B5-60)
6	E	45	=min(40;B6)	=if(B6>40;min(20;B6-40);0)	=max(0;B6-60)
7	F	65	=min(40;B7)	=if(B7>40;min(20;B7-40);0)	=max(0;B7-60)
8	G	55	=min(40;B8)	=if(B8>40;min(20;B8-40);0)	=max(0;B8-60)
9					
10	Total		=sum(C2:C9)	=sum(D2:D9)	=sum(E2:E10)
11					
12		Standard:	1		
13		50%:	1		
14		100%:	1		
15					
16	Total-Hours	=if(C12>0;C10;0)+if(C13>0;D10;0)+if(C14>0;E10;0)		Checksum:	=sum(B2:B8)

Figure 3.4: The salary-accounting spreadsheet in formula view. Circular reference in `E10`.

warning. As it is true for any end-user, the warning was ignored. At first, a value < 0 was entered in `B1`. Despite the ignored error message, correct results (see Figure (b) in Figure 3.3 on the page before) were displayed.

Then, the value in `B1` was modified to be above zero. Although no additional error was reported, the spreadsheet's result was erroneous, as the cells that depend on `A4` were not re-evaluated any more (see Figure (c) in Figure 3.3 on the preceding page).

Apparently, the spreadsheet system changed the *DDG* of the spreadsheet program in order to keep it free of cycles (see Figures (c) and (d) in Figure 3.3 on the page before).

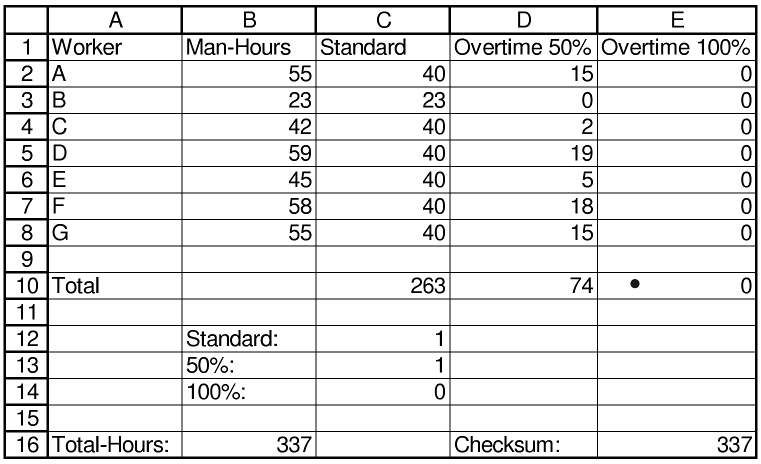
Effects of the Evaluation Strategy

These experiments were motivated by the analysis of an erroneous accounting spreadsheet. Although the spreadsheet system gave an initial *circular-reference* warning, the user did not recognize corrupt results.

After the precondition for an *if*-statement had changed, some checksums showed critical results. Although, the corrupt cells' formulas were not erroneous, the user found out that the formulas were not re-evaluated when values of cells they depend on changed.

It was found out by means of model visualization (see Chapter 6) that a sum-formula referenced itself. The sum's expected result was 0, so nobody recognized the fault, and only the failure puzzled the user.

The example spreadsheet consists of a regular part for data-accumulation and a computational part for salary calculation. In Figure 3.5 on the facing page the computational part (below row 10) has been simplified. In the first part, in column `A` the name of the worker and in column `B` his total working hours are



	A	B	C	D	E
1	Worker	Man-Hours	Standard	Overtime 50%	Overtime 100%
2	A	55	40	15	0
3	B	23	23	0	0
4	C	42	40	2	0
5	D	59	40	19	0
6	E	45	40	5	0
7	F	58	40	18	0
8	G	55	40	15	0
9					
10	Total		263	74	● 0
11					
12		Standard:	1		
13		50%:	1		
14		100%:	0		
15					
16	Total-Hours:	337		Checksum:	337

Figure 3.5: The salary-accounting spreadsheet. No error is visible

entered. In column `C` and `D` two categories of overtime are calculated. The first 20 hours of overtime have to be paid with a 50% surcharge, more than 20 hours of overtime lead to a bonus of 100%.

In the second part, users can select the categories of overtime they want to consider. As there are no entries in the 100% overtime category , it is legitimate to turn this feature off by entering the value 0 in the cell `C14`. However, in Figure 3.6 on the next page worker *F* has 5 hours of 100% overtime. In Figure 3.7 on page 43 the 100% feature is turned on. At this time the mismatch between the total-hours and the checksum cell is critical.

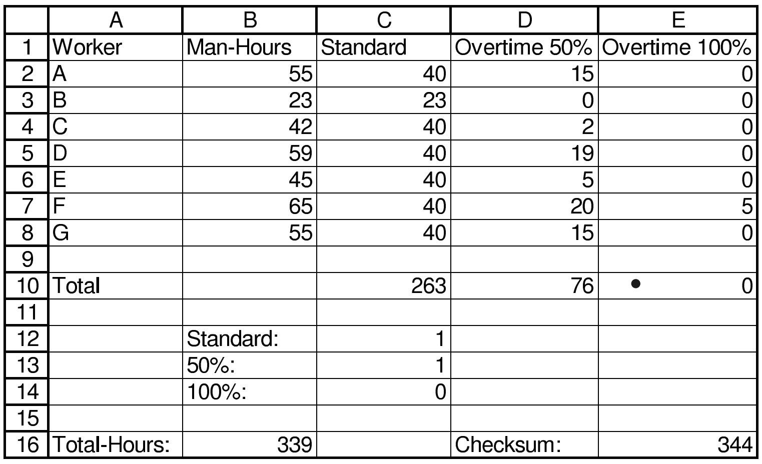
The mismatch occurs because of the circular reference in cell `E10`, where a sum-formula references itself. However, the circular reference influences the evaluation only when the 100% feature is turned on (see Figure 3.4 on the preceding page). No additional error is reported and the evaluations of all cells depending on the circular reference, i.e. reachable in the *DDG* from the circular reference, are frozen.

Although the impact of the evaluation strategy can be ignored unless errors occur, it is argued that it has a severe impact on understanding spreadsheet programs. Therefore, it is necessary to understand it.

3.3.2 Graph Reduction Semantic

Graph reduction is an efficient evaluation strategy for functional programs. For an in-depth discussion of the functional paradigm (see Winston and Horn [WH93] or Hudak [Hud89]). Treleaven [TBH82] discusses graph reduction as an evaluation strategy for demand driven programs, and Amamiya [Ama92] shows that any functional program can be specified as a graph reduction program.

Generally, a functional program is evaluated by performing reduction steps,



	A	B	C	D	E
1	Worker	Man-Hours	Standard	Overtime 50%	Overtime 100%
2	A	55	40	15	0
3	B	23	23	0	0
4	C	42	40	2	0
5	D	59	40	19	0
6	E	45	40	5	0
7	F	65	40	20	5
8	G	55	40	15	0
9					
10	Total		263	76	● 0
11					
12		Standard:	1		
13		50%:	1		
14		100%:	0		
15					
16	Total-Hours:	339		Checksum:	344

Figure 3.6: The salary-accounting spreadsheet. There are 5 hours of 100% overtime. An error is visible only in `E10`

until none of the symbols in the functional program can be reduced any further. A reduction step consists of the replacement of a function call and its arguments with the function itself. In the end, the generated expression consists of atoms and basic operations only and can easily be evaluated.

If this reduction is performed by literally replacing the function call with the function, string reduction is performed (see right-hand side of Figure 3.8 on page 44). String reduction is inefficient if we assume that the result of the reduced function is used more than once. In this case, the same subexpression occurs more than once and has to be evaluated at each occurrence.

To overcome these drawbacks graph reduction does not perform the literal replacement of function calls with the associated expression. A function call is replaced by a link to the function (see left-hand side of Figure 3.8 on page 44). When the function's result is requested for the first time, it is evaluated and the result is stored. When the result is requested the next time, the stored result is forwarded to answer the request.

3.3.3 Basic Differences to Reduction Semantics

In this section the assumption that a spreadsheet program is not a pure graph reduction program will be justified. Therefore, the two main differences between graph reduction and spreadsheet programs are discussed: *loops* and *change propagation.*

Sequential Loops

Sequential loops are not part of the functional concept (see Manna and Amir [MA70]) and, therefore, not part of graph reduction. However, recursion is native

	A	B	C	D	E
1	Worker	Man-Hours	Standard	Overtime 50%	Overtime 100%
2	A	55	40	15	0
3	B	23	23	0	0
4	C	42	40	2	0
5	D	59	40	19	0
6	E	45	40	5	0
7	F	65	40	20	5
8	G	55	40	15	0
9					
10	Total		263	76	• 0
11					
12		Standard:	1		
13		50%:	1		
14		100%:	1		
15					
16	Total-Hours:	339		Checksum:	344

Figure 3.7: The salary-accounting spreadsheet. There are 5 hours of 100% overtime. An error is noticed by matching checksums.

to functional programs and iteration can be expressed by means of tail-recursion.

Nonetheless, there is no means of implementing loops in spreadsheet programs, except for loop unfolding as shown in Figure 3.2 on page 38. Recursion or loops would violate the following properties of spreadsheet programs:

All intermediate results are visible: Recursion, i.e. re-evaluation of a cell until a stable state is reached, would entail multiple evaluation cycles of a cell. However, the intermediate results that were produced in this cell before reaching the stable state, cannot be displayed. So only the result of the cell 'survives' the recursion.

There is no global control flow: Recursion, or sequential loops, are a kind of control flow. The user has to deal with a stopping condition and has to have a mental concept of the order of evaluation.

As loops or recursion are a violation of the evaluation semantics of spreadsheet programs, the accidental usage entails odd results (see Subsection 3.3.1) that vary from one spreadsheet system to the other.

Change Propagation

Generally, a functional program has one defined result. A change to a function that influences the final result has no impact on the result until the program is re-evaluated. As stated above (see Figure 3.9 on page 47), the interactivity of spreadsheet programs implies a sophisticated change propagation technique. A change to any formula will automatically lead to

1. its re-evaluation,

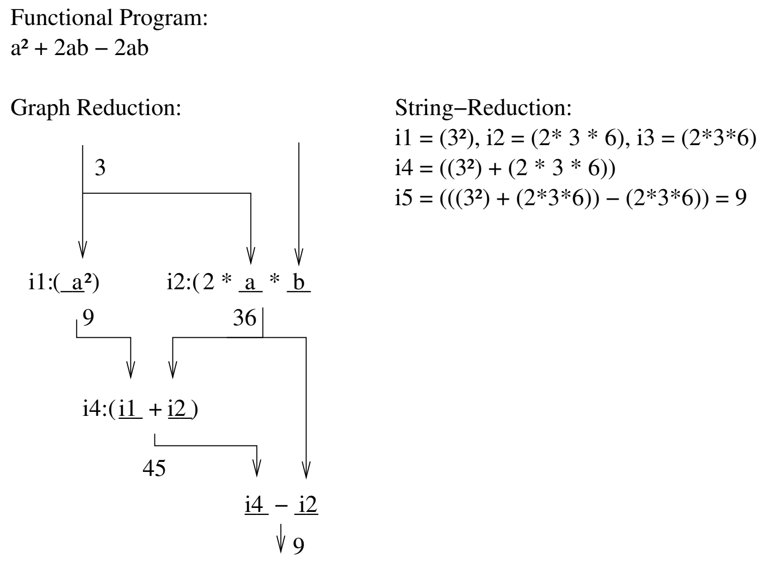


Figure 3.8: Evaluation of the expression $a^2 + 2ab - 2ab$ with $a = 3, b = 6$. Left-hand side uses graph reduction, right-hand side string reduction

2. the re-evaluation of depending formulas, and
3. the re-evaluation of transitively depending[4] formulas.

The change propagation mechanism is mainly data flow controlled.

3.3.4 Basic Differences to Data Flow Semantics

The preceding section pointed out a couple of differences between the spreadsheet evaluation strategy and graph reduction. These facts tempt to neglect the reduction semantics of spreadsheet evaluation and to classify a spreadsheet program as a data flow program.

As the foregoing considerations sketched a combined evaluation strategy for spreadsheet programs, it has to be pointed out, why a spreadsheet program is no pure data flow program. Data flow programs show different behavior in evaluation triggering, data flow specification and loops.

Before elaborating the differences between data flow evaluation and spreadsheet evaluation, the foundations of data flow semantics will be briefly discussed.

Data Flow Semantics

Data flow programs divide a complex computation up into a set of elementary calculations and their interconnections. As each of the calculations can be per-

[4]A formula f_1 is transitively dependent on a formula f_2, if f_1 uses the result of a formula f_3 that depends on f_2 (or is transitively depending on f_2)

formed as soon as all of the required input values are available, data flow programs offer a high level of parallelism. Data flow programs are usually specified through a data flow graph (see Treleaven [TBH82], Ang [Ang93] and Ackermann [Ack82]).

The term *data flow graph* denotes a directed graph. The nodes represent the computations of the data flow program. Each computation node can be the target of several edges, and it is the source of exactly one edge. Tokens are markings of edges that represent output values of their source node. When all of the edges that target a specific node, n, are marked, the computation of n is executed. The tokens are consumed and the edge that leaves n is marked with a token. If there is an edge (n, m) from n to m, n is called the *data source* of m.

Sources of the data flow graph are nodes that are not target of any edge. If a source represents a constant value, the edge that leaves the source node is permanently marked. Correspondingly, a sink node is a node that is not source of any edge. Sink nodes usually represent results of data flow programs.

There are additional, special kinds of nodes, namely *selector* and *multiplier* nodes, that are the source of more than one edge. The computation of a selector node s will evaluate a condition, and, depending on the condition, mark one of the edges that leave s with the token. A multiplier node m is the target of one edge. All edges that leave m are marked with the token that arrives at m.

These special nodes, that are considered extensions to the general data flow model, allow the implementation of sequential loops in data flow programs (see Ang [Ang93] and Kavi et al. [KBB86]).

Evaluation triggering

Roughly speaking, a calculation of a node n in a data flow program is performed as soon as all of the edges that point to n are marked with a token. The calculation will consume these tokens. In order to re-evaluate n, all of the edges have to be marked again.

In a spreadsheet program SP that is represented by a $DDG = (V, E)$, a node $v \in V$ is re-evaluated as soon as one of the edges $e = (x, v) \in E$ is marked. Hence, there is a marking of edges, but not with the data flow semantics. The tokens are rather evaluation-triggers than input values for a calculation.

Data Flow Specification

In a data flow graph, each computational non sink node is left by exactly one edge[5]. The nodes are not aware of their data sources, but only of the tokens that arrive.

In a spreadsheet program the data flow is specified bottom-up, i.e each node specifies its data sources. Each node can be the data-source of an arbitrary

[5]An exception of this rule is introduced by multiplier nodes, that are the target of one edge but the source of an arbitrary number of edges.

number of nodes. Conditional data flow is not part of spreadsheet programs. This shortcoming is patched by basic control flow in the nodes, which is not part of data flow concepts.

Loops

By means of extensions to the data flow concept, namely *selector* and *multiplier* nodes, sequential loops can be implemented by data flow programs.

As it was already stated in Subsection 3.3.3, loops are not part of the concept of classical spreadsheet programs. However, procedural macros or similar extensions, can implement loops.

3.3.5 Spreadsheet Evaluation Strategy

As it has been stated above and supported by experiments, spreadsheet programs are considered partially a graph reduction program, and partially a data flow program. Therefore, the evaluation of a spreadsheet program has to be described from two viewpoints:

Locally, a cell evaluates its formula by reducing it. As a cell's value can be accessed by more than one cell, graph reduction is performed (see Treleaven et al. [TBH82] and Amamiya [Ama92]. The reduction graph corresponds to the *DDG* of the spreadsheet (see Definition 13 on page 24).

Globally, it has to be decided, which cells have to be re-evaluated when a change occurs. Therefore, a change in a cell will produce *change tokens* that are distributed by a data flow graph that is similar to the *DDG*, too.

The evaluation strategy is illustrated by Figure 3.9 on the next page. Vertexes in the figure correspond to cells, thin edges represent the *DDG*-edges. Cell `C` is subject to a change by the user. It can be assumed that there is a formula in `C`, as `C` depends on two other cells. After the change has taken place, the value of `C` is recalculated by reduction (local viewpoint).

In order to propagate any changes, the successors of `C` have to be re-evaluated, too. Therefore, a change token is passed along the dashed line (global viewpoint). As soon as a change token arrives at a cell, the cell will be re-evaluated by reduction and pass the change token on to its successors.

However, as a graph reduction program is a functional program, its evaluation strategy is lazy evaluation, i.e. a node is only evaluated when its value is requested. In contrast, spreadsheet program evaluation is usually globally eager, i.e. a node is evaluated as soon as possible. Nevertheless, lazy evaluation is performed locally, because only those parts of a formula are evaluated that contribute to its result.

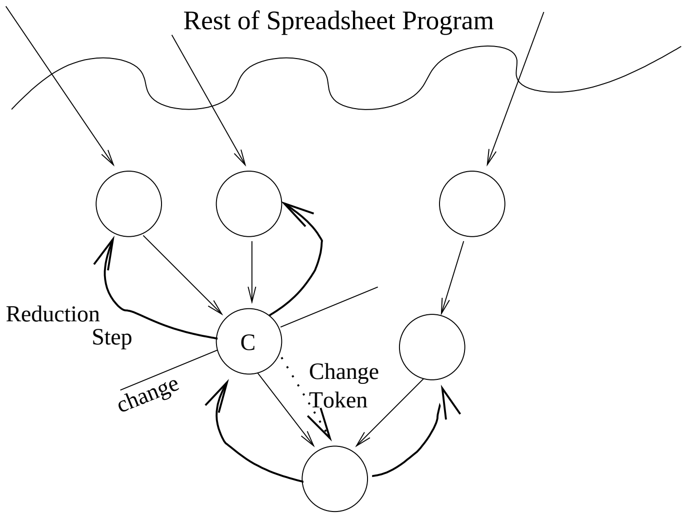


Figure 3.9: Spreadsheet evaluation strategy

This divergence is caused by the fact that a functional program is defined to have one specific result. Only those functions or nodes in the reduction graph that influence the result are evaluated.

As shown in Section 3.2, the spreadsheet program has no dedicated output. Each intermediate result is considered output, and, therefore, each potentially affected cell has to be evaluated.

Apparently the spreadsheet program unifies some concepts of data flow and graph reduction. However, the dualism in the classification is only true at first sight. Sharp et al. [Sha92] discuss similarities between data flow languages and functional languages. They state that

> many languages developed by data flow workers are in essence functional languages. This is not surprising, since such languages fit naturally on the data flow execution model. Data flow is in fact viewed by some as just another implementation technique for functional or applicative languages.

Hence, the classification of a spreadsheet program as a graph reduction program entails its classification as a data flow program.

3.4 Criteria for Successful Methodologies

The different evaluation strategy and a different group of users make conventional testing approaches unsuitable for spreadsheet programs. In this section some important criteria for successful spreadsheet auditing and testing methodologies will be introduced.

Reichwein, Rothermel and Burnett [RRB00] introduce three important constraints on spreadsheet-related testing methodologies:

Accompanying deployment: Spreadsheet development is not divided into different phases, i.e. programming, compiling, execution. Thus, it is suggested that spreadsheet programmers need to be accompanied by tool support.

No specialized vocabulary: As argued in Section 3.1, spreadsheet users are domain experts. Although they gradually build up spreadsheet expertise, they are generally not willing to study IT-related terms and concepts (see Nardi and Miller [NM90a]). Thus, a spreadsheet auditing or testing methodology should hide technical details and rely only on concepts the spreadsheet programmers are familiar with.

Minimal overhead: Spreadsheet systems offer immediate feedback to the spreadsheet users. Hence, a change in the input values is immediately reflected in new results. An integrated spreadsheet auditing or testing methodology must not hinder the interactivity of a spreadsheet instance.

Additionally, five further constraints, namely *fault localization, qualitative errors, scalability, integration*, and *freedom of layout* are worth looking at.

Fault Localization

Most of the spreadsheet testing methodologies have to deal with the localization of detected errors (see Ayalew [Aya01] or Rothermel et al. [RLDB98]). Quite often, the cell that displays a faulty value, does not contain an erroneous formula. Thus, it is argued that a successful spreadsheet auditing approach must be capable of detecting errors and showing their source. Otherwise, the burden of correcting errors in a consistent way will be left to the spreadsheet programmer.

As some field audits have shown (e.g. [CHM02]) users tend to perform the correction of errors with unknown source by overwriting the cell that displays the faulty value with a constant value. Obviously, this strategy will deteriorate further spreadsheet maintenance.

Qualitative Errors

Qualitative errors are errors that are not immediately visible by inspecting the output of a current spreadsheet instance (for a more detailed discussion, see Teo and Tan [TT00]). Nevertheless, qualitative errors are errors spreadsheet program that might turn up on the value level in future spreadsheet instances.

This kind of error is very dangerous for long-living spreadsheets, because ongoing modifications of spreadsheets tend to introduce qualitative errors and transform qualitative errors into quantitative errors, i.e. errors on the value level.

Hence, qualitative errors are an important issue in many large economic spreadsheets (see Clermont et al. [CHM02]). However, recognizing a qualitative error requires IT-skills.

Fault localization usually deals with the detection of qualitative errors that have already manifested in quantitative errors at another location of the spreadsheet UI. Thus, valuable support for the spreadsheet programmer includes the detection of qualitative errors.

Scalability

Spreadsheet auditing is often doomed to fail due to high costs and long duration. This is already true for medium-sized spreadsheets (see Panko [Pan97]), but much worse for large spreadsheets that are commonly used in business applications (see Section 2.2, Clermont et al. [CHM02]).

There are two common obstacles to the scalability of spreadsheet-related techniques. The first one is the employment of the spreadsheet UI as user interface. Thus, the visible clipping is limited by the size of the screen and the size of a cell on the spreadsheet UI.

The second obstacle is of conceptual nature: usually, the unit to be examined is the cell. Hence, most of the techniques fail for large spreadsheets with more than $1,000$ cells.

Integration

As spreadsheet development is an incremental process (see Reichwein et al. [RRB00], Nardi and Miller [NM90a] and Brown and Gould [BG87]) that is embedded into the spreadsheet system, the auditing or testing toolkit should also be integrated into the spreadsheet system. This integration should increase the availability of the toolkit and minimize the difficulty of employment.

Freedom of Layout

There is a couple of techniques and style guides that promise to increase the efficiency and decrease the error probability of spreadsheet development. A more detailed discussion is given in Chapter 5. Unfortunately, many of these techniques restrict the spreadsheet programmers' freedom of using the spreadsheet UI in the way they want to.

As a matter of fact, these techniques are not accepted by spreadsheet programmers, because they do not recognize the necessity of reducing the complexity of their programs. On the contrary, spreadsheet programs are often deliberately complicated in order to meet some layout requirements.

Summary

The following issues have been addressed in this chapter:

- Spreadsheet development is different from conventional software development.
- Spreadsheet programmers are end-users.
- There are conceptual differences between spreadsheet programs and conventional programs that require different testing and auditing approaches.
- Spreadsheet programs have no user specified control flow. The spreadsheet programmer specifies the data flow.
- Spreadsheet programs are neither pure data flow programs nor pure reduction programs. Indeed, they are a mixture of both of them.
- Evaluation is triggered by a data flow like mechanism and performed by graph reduction.
- Certain spreadsheet errors are due to ambiguities in the spreadsheet evaluation strategy.
- Successful auditing and testing techniques have to take these facts into account.

Chapter 4

Spreadsheet Error Studys

The importance of spreadsheets to organizations is obvious (see Section 2.2). Although some features of spreadsheet programs are undesirable from the software engineer's point of view, this does not imply that these programs are more error prone than other software artifacts.

In order to show the need for sophisticated spreadsheet auditing and testing techniques, this chapter summarizes the results of past spreadsheet audits and development experiments. To get a better understanding of these results, an approach to define a spreadsheet error taxonomy is introduced beforehand.

4.1 A Taxonomy of Spreadsheet Errors

The variety of sources for spreadsheet errors calls for a taxonomy of spreadsheet errors. As in natural sciences, where taxonomies have been invented by Carolus Linnaeus in 1737 [Lin37] (summarized by Asimov [Asi84]), taxonomies should support research in handling manifold phenomena.

A taxonomy for spreadsheet errors has to meet the fact that there are different ways to make errors and that a spreadsheet error can be caused either by corrupt input data or by an erroneous spreadsheet program. However, extensive categorizations of spreadsheet errors can be built with different intentions, and, thus, will give a different perspective on spreadsheet errors.

There are a couple of error categorizations (see Panko [Pan98b], Panko and Halverson [PH97], Teo and Tan [TT00] or Ayalew et al. [ACM00]), but none of them meets the requirements for a taxonomy to categorize all possible kinds of errors in a hierarchic system.

Rajalingham et al. [RKC00] present a quite complete taxonomy of spreadsheet errors[1]. They divide spreadsheet errors into 29 hierarchically organized categories that contain all work that has previously been done on categorization

[1]Some extensions are suggested in Section 4.2

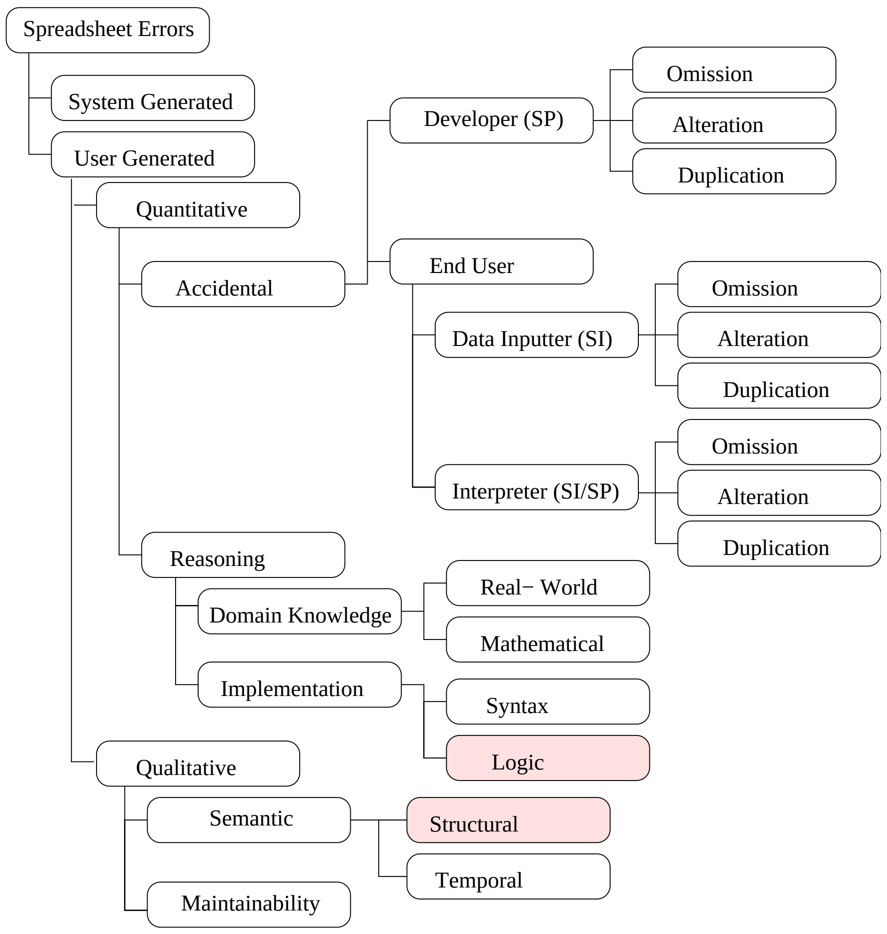


Figure 4.1: Taxonomy of spreadsheet errors (taken from [RKC00]). The categories with gray background are extended by area-related errors in Section 4.2.

(see Figure 4.1). For each leaf error category an example is given in [RKC00]. Subsequently, the most important error categories will be briefly discussed.

4.1.1 System Generated versus User Generated Errors

As in conventional software, the spreadsheet program is not the only potential error source. Of course, the spreadsheet system that evaluates the spreadsheet program or the floating point unit of the computer system can be erroneous. An example for system generated errors in the context of circular references is given in Figure 3.3 on page 39.

4.1.2 Quantitative versus Qualitative Errors

User generated errors are categorized into quantitative and qualitative errors. This distinction is made by most of the error categorizations. However, the meaning of the term quantitative error is not unambiguously defined. Panko [Pan98a] and Rajalingham [RKC00] define quantitative errors as errors that *produce an incorrect value in at least one bottom line value.*

This definition is obviously too restrictive, as two cells with erroneous output that nullify each other in a result cell, are not considered. As a matter of fact, other authors (e.g. Teo et al. [TT00], Clermont et al. [CHM02]) define a quantitative error as an error that produces an incorrect value in any cell of the spreadsheet.

In contrast, qualitative errors cannot be revealed by testing a spreadsheet on the value level, as they are usually caused by incorrect formulas that accidentally produce the correct result for a specific spreadsheet instantiation. However, when maintenance is performed, they usually turn into quantitative errors, i.e. they produce erroneous values.

Usually, qualitative errors are introduced by overwriting a cell's formula that produces an unexpected result with a constant value or a superficially modified formula. It is easy to see that a modification of cells the original formula was depending on, will cause a quantitative error.

In software engineering terms, qualitative errors can be considered faults that do not lead to failures, whereas quantitative errors are failures. As in conventional software, the fault that causes a failure can be at a different place. The same is true for quantitative errors that might be caused by a qualitative error at a different location.

Accidental versus Reasoning Errors

Accidental errors are a common phenomenon in human error theory, as it is shown by Rasmussen [RPG94], Gilmore [Gil90] and Ormerod [Orm90]. On his human-error research website [Pan02b] Panko states that programmers usually produce an error rate of 2 – 5% in their code. Empirical studies of program error rates show a range from 0.6% [BP93] to 15% [ZZ93], depending on the task and the environment of the development.

These errors happen, although people are aware of the risk of accidental errors. As it is stated not only by Reason [Rea90], but also by Cicero, for humans to err is inevitable- errare humanum est.

As their name suggests, accidental errors can be either typographic errors, misplaced formulas, or logical errors. The difference between the category of accidental and the category of reasoning errors that contains syntax and logical errors, too, is the way the errors are produced.

Rasmussen [RPG94] distinguishes between

Errors, i.e. the right conception is incorrectly put into practice, and

Mistakes, i.e. users have a wrong concept that is correctly put into practice.

The inevitable accidental errors fall into the first category, whilst reasoning errors can usually be considered mistakes. Accidental errors will occur automatically even if users carefully check their programs.

A similar distinction is also common in conventional software engineering, where the terms *bug, error, mistake* and *fault* are used. As it is stated by Beizer in [Doe99], in software there might be *programmer errors* that lead to faults. However, in the further discussion that is reported by Doernhoefer [Doe99], it is pointed out that there can be multiple reasons for faults, e.g. problems with requirements, hardware or test cases.

For the sake of simplicity, the term error is used throughout this thesis to denote errors and mistakes. Whenever a distinction has to be made, it is stated explicitly.

Reasoning Errors

Reasoning errors are usually the result of a mismatch between the programmers' perception of the real world and the problem they want to solve. Usually, programmers build models of the problems they want to solve (see Hoc [HNX90]). Errors in this model are usually the result of reasoning errors that can be caused by

- misunderstanding the Real-World
- wrong transformation of the real-world problem to a mathematical representation
- misunderstanding the spreadsheet's internal logic,
- misunderstanding the spreadsheet system's formula language.

As reasoning errors do not occur accidentally, they are usually made repeatedly, whenever the incorrect part of the model occurs in the spreadsheet program. A special kind of reasoning error is *overspecialization.*

Overspecialization is usually introduced as qualitative error when spreadsheet programmers do not consider absent special cases in their models. However, during maintenance these special cases tend to become important and result in arbitrary changes of the spreadsheet program. Thus, former regular structures, i.e. repeatedly used formulas, are torn apart.

It is easy to see that these maintenance operations cause the introduction of a special kind of qualitative errors that are summarized by the category *maintainability.*

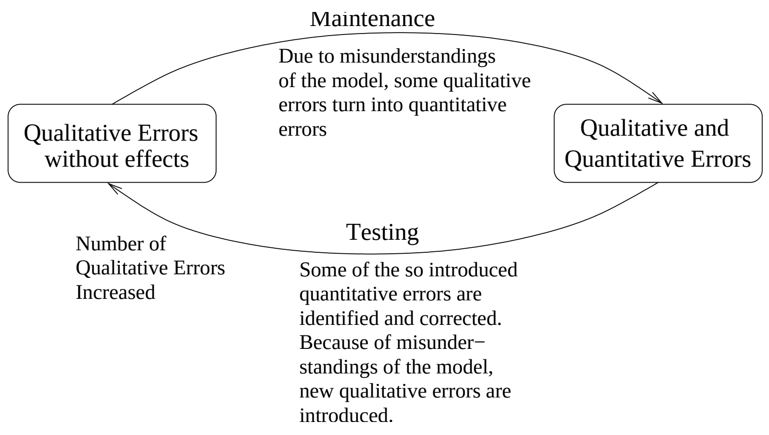

Maintenance
Due to misunderstandings of the model, some qualitative errors turn into quantitative errors
Qualitative Errors without effects
Qualitative and Quantitative Errors
Number of Qualitative Errors Increased
Testing
Some of the so introduced quantitative errors are identified and corrected. Because of misunder-standings of the model, new qualitative errors are introduced.

Figure 4.2: The vicious cycle of spreadsheet maintenance. The testing effort and the probability of remaining quantitative errors increases in each cycle.

4.1.3 Qualitative Errors

So far, the error-taxonomy has dealt with errors that have an immediate impact on the displayed values of cells. The definition of quantitative errors is consistent with errors in conventional software that are defined as a mismatch between the expected and the delivered output of a software system (a more detailed discussion of the testing process is given in the software engineering literature, e.g. Beizer [Bei90], Harrold [Har00], Kan [Kan95], Kaner [KFN93] or Perry [Per95]).

However, qualitative errors are also an important issue for spreadsheets, especially when they are subject to evolutionary changes. For conventional software the term *qualitative error* is not widely used, however, the aging of software due to evolutionary changes is subject to research (see Rajlich [BR00], Parnas [Par94]). As stated above, changing the spreadsheet instance might make formerly correct spreadsheet programs deliver incorrect output when qualitative errors turn into quantitative errors.

Field audits have shown that qualitative errors tend to be introduced when a spreadsheet is adapted to a changed environment or when errors are corrected. In a large field audit it has been shown that most of the errors in real world spreadsheets are qualitative errors [CHM02]. As they accumulate throughout the maintenance cycles, the testing effort increases, too. Unfortunately, the correction of errors thus discovered will usually introduce new qualitative errors (see Figure 4.2).

Hence, qualitative errors can be considered a time-bomb hidden in the spreadsheet. They do not show on the value level until they turn into quantitative errors; and sometimes quantitative errors do not even show in the same cell where the qualitative error is located. Therefore, it is rather a symptom of the error that has to be traced back to the error. Most of the sophisticated spreadsheet testing techniques (see Ayalew [Aya01], or Rothermel et al. [RRB00, RLDB98]) employ certain error-tracking heuristics.

Reasons for Qualitative Errors

In the previous sections it was pointed out that quantitative errors are the product either of inevitable accidental errors or of reasoning errors. Qualitative errors are sometimes reasoning errors, too, but as we have stated above, they are often invented during maintenance or by superficial error corrections.

Qualitative errors are supported by the following shortcomings of the spreadsheet development process (see [CHM02]):

1. Tests are performed on the value level, and error corrections take place mainly on the value level, too.
2. Spreadsheet writers do strongly identify with *their* spreadsheets and are heavily overconfident in their personal spreadsheet programming skills. Thus, documentation is considered unnecessary.
3. Maintenance cycles date between one month and one year.

The first property obviously supports the introduction of qualitative errors by

- overwriting formulas with constant values, or,
- correcting only one instance of multiple copies of an incorrect formula.

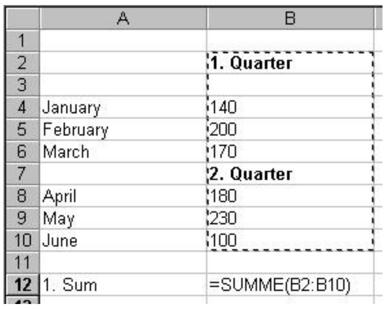
The latter two properties of the development process lead to an ongoing deterioration of the spreadsheet model's clarity. Spreadsheet writers often consider documentation work to be useless overhead [CHM02], so it is often neglected. Thus, maintenance has to take place based on assumptions of how the spreadsheet works.

However, even the creators of a spreadsheet often forget the internal logic of a spreadsheet program. If someone who is considered to be responsible for a certain spreadsheet leaves the company or merely changes the department, the spreadsheet is often useless for the successors.

4.2 Area Related Errors

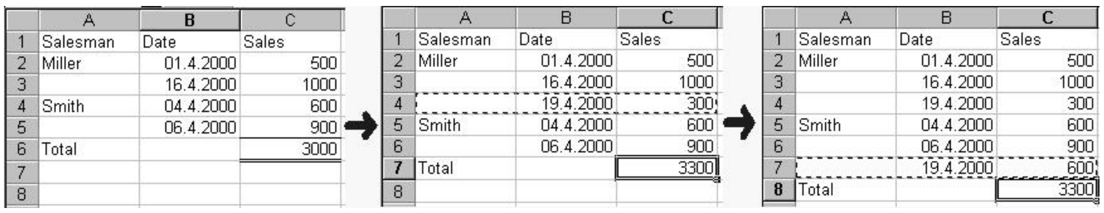
The taxonomy that was discussed in the previous section can be stated more precisely in the category of reasoning errors. As it is mentioned in Subsection 4.1.2, reasoning errors can show as quantitative or as qualitative errors[2]. However, they are always the result of a misunderstanding or a misconception.

[2]We are aware of the fact that this behavior is not consistent with the taxonomy of spreadsheet errors shown in Figure 4.1 on page 52.



	A	B
1		
2		**1. Quarter**
3		
4	January	140
5	February	200
6	March	170
7		**2. Quarter**
8	April	180
9	May	230
10	June	100
11		
12	1. Sum	=SUMME(B2:B10)

Figure 4.3: Reference to a blank/wrongly typed cell.



	A	B	C
1	Salesman	Date	Sales
2	Miller	01.4.2000	500
3		16.4.2000	1000
4	Smith	04.4.2000	600
5		06.4.2000	900
6	Total		3000
7			
8			

	A	B	C
1	Salesman	Date	Sales
2	Miller	01.4.2000	500
3		16.4.2000	1000
4		19.4.2000	300
5	Smith	04.4.2000	600
6		06.4.2000	900
7	Total		3300
8			

	A	B	C
1	Salesman	Date	Sales
2	Miller	01.4.2000	500
3		16.4.2000	1000
4		19.4.2000	300
5	Smith	04.4.2000	600
6		06.4.2000	900
7		19.4.2000	600
8	Total		3300

Figure 4.4: Physical area specification error.

The powerful concepts of physical and logical areas (see Subsection 2.3.5) is often the source for reasoning errors. If the internal logic of the spreadsheet has already been blurred by previous maintenance operations, the danger of reasoning errors will further increase.

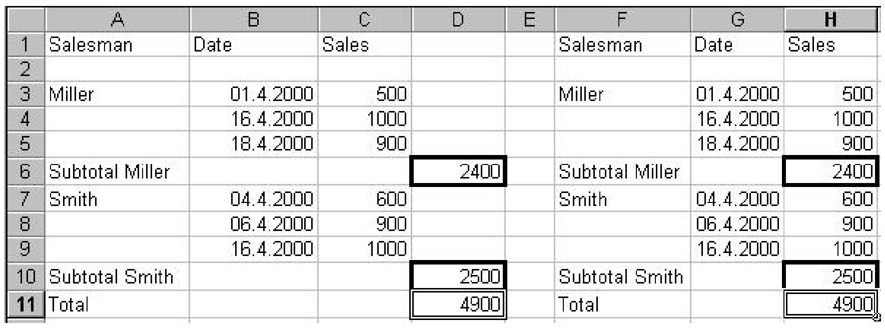
In [ACM00] five categories of errors that are caused by misuse of physical or logical areas were defined:

1. Physical area related errors

    Reference to a blank/wrongly typed cell occurs if a physical area includes cells of different types, e.g. empty or text cells in a sum-formula's area. As long as these cells are not accidentally filled or the label does not change to a numeric label, this error is a qualitative error (see Figure 4.3).

    Physical area specification errors can occur if the user defines a physical area over a rectangular area of cells. If new cells are inserted into the rectangular area, the physical area is automatically re-adjusted. However, if the new cells are prepended or appended, the area is not readjusted. The new cells are part of the geometrical area but not of the physical area. (see Figure 4.4).

    Physical area mix-up problems occur if two different physical areas interfere (see Figure 4.5 on the next page). This is often the case for subtotals and causes at least one of the involved physical areas to be



	A	B	C	D	E	F	G	H
1	Salesman	Date	Sales			Salesman	Date	Sales
2								
3	Miller	01.4.2000	500			Miller	01.4.2000	500
4		16.4.2000	1000				16.4.2000	1000
5		18.4.2000	900				18.4.2000	900
6	Subtotal Miller			2400		Subtotal Miller		2400
7	Smith	04.4.2000	600			Smith	04.4.2000	600
8		06.4.2000	900				06.4.2000	900
9		16.4.2000	1000				16.4.2000	1000
10	Subtotal Smith			2500		Subtotal Smith		2500
11	Total			4900		Total		4900

Figure 4.5: Physical area mix-up problem. On the left hand side the areas are separated. On the right hand side, because of the overlapping physical areas, the grand-total must not be calculated by the built-in sum-function of the spreadsheet system.

removed[3].

2. Logical area related errors

Overwriting a formula with a constant value is a common qualitative error that is usually caused by superficial error corrections (see Subsection 4.1.3).

Copy Misreference is another qualitative error. A constant value or an absolute cell reference is specified in a formula instead of a relative reference. This error turns into a quantitative error, when the formula is copied into another cell. The same kind of error occurs vice-versa if a constant value is erroneously specified by a relative cell reference.

Depending on the effects, these errors can be categorized either as *structural* or as *logic* errors, and thus, they may or may not show on the value level.

4.3 Results of Field Audits and Experiments

This section gives a brief summary of the results of field audits and so-called laboratory experiments. Field audits usually analyze spreadsheets that are already in use (or shortly before deployment), whereas laboratory experiments are carefully designed exercises that aim to find out the influence of certain environment conditions, e.g. previous experience, time pressure, training, education, on the total error rate of developed spreadsheet programs.

Moreover, for experiments a solution of the problem is known, and thus, the developed spreadsheets can be checked against a given solution. Hence, all errors can be revealed in development experiments, whereas spreadsheet programs

[3]Only as physical area - the conceptual unit still remains.

that are analyzed in field audits behave like conventional software. This means, *Program testing can be used to show the presence of bugs, but never to show their absence! [DDH72]* [Har00].

4.3.1 Metrics

Although many audits have taken place (see [Pan02a, Pan98a] or Tables 4.1 and 4.2 for a summary of results), there is no general agreement on a metric for spreadsheet errors. The following metrics are usually applied (the first three are described in detail by Panko and Halverson [PH96], the latter ones by Clermont et al. [CHM02]).

As erroneous input data is not an issue in this thesis, all metrics deal with measuring corrupt formulas. The first two metrics are rather coarse-grained and do not consider individual errors, but deal with the overall impact of errors on a spreadsheet program. *Cell error rate*, *formula error rate* and the *absolute number of errors*, make statements about the error ratio or the number of errors in a spreadsheet program, whereas the last two metrics are developed to make statements about the number of errors compared to the number of different formulas in a spreadsheet.

Subsequently, each of the metrics is briefly described:

Percentage of erroneous models specifies the relative amount of erroneous spreadsheet programs that have been analyzed.

Error magnitude is a measure for the impact of spreadsheet errors on the model. Panko and Halverson [PH96] define the error magnitude as the percentage of correct bottom line values. Another interpretation offered deals with the question whether decisions that are based on the spreadsheet are influenced by the errors. However, the error magnitude is usually not reported in spreadsheet field audits.

Cell Error Rate (CER): The CER is calculated as ratio of none empty cells to erroneous cells (see Panko and Halverson [PH96]). Of course, the cell error rate is only a relative statement on the spreadsheet program's quality. One wrong formula that is copied very often, will raise the CER over-proportionally, although it will be easy to find the errors. Additionally, cells that are part of the spreadsheet instance are usually considered for the calculation of the CER. Thus, the number of input values can influence the CER.

Formula Error Rate: The formula error rate is the ratio of the number of formula cells to erroneous cells. This measure is an improvement of the CER, as input values are not taken into account anymore.

Absolute Number of Errors: This metric is easily available. However, it is not suitable for comparing the quality of spreadsheets of different size.

Absolute Number Of Error Classes: Error classes consider the fact that formulas are often copied multiple times during spreadsheet development. Thus, errors are multiplied by copying erroneous formulas. Errors that occur multiple times because of such a copy-operation, are grouped into one error class.

Error Classes per Copy Equivalence: This measure is a normalization of the formula error rate, as the ratio of error classes to unique formulas is calculated. A copy equivalence class consists of identical formulas (see Chapter 6 for a more precise definition).

Unfortunately, none of these metrics can be used to give either an objective statement about the quality of a spreadsheet or to compare the quality of different spreadsheet. The only metric that takes the severeness of errors into account is error magnitude. However, there is no objective way to evaluate the error magnitude of a given spreadsheet.

4.3.2 Findings

Apart from the field audits and spreadsheet development experiments that are summarized in Tables 4.1 and 4.2, a more recent spreadsheet field audit is described in [CHM02]. In the remaining section the main findings of the latter audit will be briefly summarized.

Auditing Environment

Auditing was performed from April until August 2001 by a third-year computer science student supervised by the author and one of the supervisors of this thesis. The auditor was assigned to the accounting department of an international cooperation with headquarters in Vienna where he could work desk-to-desk with the spreadsheet producers. The contact with the tool developers was kept by e-mail and by regular visits. He examined three voluminous Excel workbooks that are mainly used for consolidation. The three workbooks consisted of 78 worksheets with 60,446 non-empty cells.

Before the audit started, the auditor who had only little bookkeeping experience, discussed the basic idea and functionality of each workbook with the respective author. Additionally, the author was interviewed about the lifespan of the workbook, the usual maintenance cycle and the number of users.

Then, for each spreadsheet in the workbook, the following characteristics were documented: dimension, number of non-empty cells, number of formulas, constants, and literals. At first, the correctness of the displayed values was checked.

Study	**Year**	**# Sheets**	**% w. Errors**	**CER**
Davies & Ikin [DI87]	1987	19	21%	
Cragg & King [CK93]	1992	20	25%	
Hicks	1995	1	100%	1.2%
Butler [But00]	1992	273	11%	
Coopers & Lybrand	1997	23	91%	
KPMG	1997	22	91%	
Lukasic	1998	2	100%	2.5%
Butler [But00]	2000	7	86%	0.38%
Total		367	24%	
Total since 1997		54	91%	

Table 4.1: Results of field audits in the last 15 years, summarized by Panko [Pan00]. *# Sheets* denotes the number of examined spreadsheets, *% w. Errors* the relative number of spreadsheets with at least one error, and *CER* is the cell error rate.

Study	**# Sheets**	**% w. Errors**	**CER**
Brown & Gould [BG87]	27	63%	
Olson & Nilsen [ON88]	14		21%
Lerch	21		9.3%
Hassinen	92	55%	4.3%
Janvrin & Morrison (1) [JM00]	61		7%–14%
Janvrin & Morrison (2) [JM00]			8%–16%
Kreie et al. [KCPR00]	73	42%	2.5%
Teo & Tan [TT00]	168	42%	2.1%
Panko & Halverson [PH96]	42	79%	5.6%
Panko & Halverson	35	86%	4.6%
Panko & Sprague [PS99]	26	35%	2.1%
Panko & Sprague [PS99]	17	24%	1.1%

Table 4.2: Results of development experiments, summarized by Panko [Pan00].

Special attention was paid to wrong sums, wrong formatting and errors that were reported by Excel.

After these routine checks in the value domain a prototype of the toolkit described in Chapter 7 was applied. The irregularities thus discovered were then discussed with the spreadsheet authors to find out if the detected irregularities were deliberately introduced or whether they had to be corrected and counted in the error statistics.

Thus, the auditor had a lot of discussion with the domain specialists who created the spreadsheets. All documented errors have been acknowledged by the spreadsheet creators. The identified errors were collected in an error database. For each error information about the location, the kind of error and its impact was gathered. Additionally, a short description was stored.

Examined Spreadsheets

The audit examined three large Excel workbooks. Each of them was used to gather data from various departments of the company and to calculate different financial ratios at the corporate level. These financial ratios are an important base for strategic decisions. The workbooks analyzed served the following purpose.

RAT-2001 calculates a financial statement. Data is aggregated from sub-sheets that correspond to the enterprise's organization. Hence, there are worksheets for different business-units and corporate sectors. These worksheets are aggregated to calculate the financial statement of each division. The spreadsheet has been in use for one year so far. There is extensive maintenance each month. The company's annual budget processed by these spreadsheets is about $150,000,000.

TP-Report was in use for three months when examined. The lifespan of the spreadsheet was considered to be unlimited. When audited, the author was the only user, but it was planned to delegate maintenance of particular worksheets to other employees. The sheet accumulates data from four other workbooks that are maintained by four different persons. During our study the workbook has been fundamentally changed, so we re-audited it. The results reported only reflect the latest version audited.

AB-Market performs material costs analysis. It has been in use since 1999 and was modified each year, before budgeting is done. A copy of the workbook is sent to each branch office where its input cells are filled in by at most three employees. The completed/updated workbooks are sent to the author again, who then merges the copies into a single workbook. The data obtained by this procedure is used to analyze cost of raw material of the various factories. For the analysis, additional information, such as current and fore casted volume, costs, price per unit, and average prices are added

WB	**#Cells**	**#Occ.**	**#Form.**	**#CE**	**#Lit.**	**#EC**	**Errors**
RAT-2001	56, 485	19, 444	12, 382	7, 062	814	21	257
TP Report	69, 835	23, 502	16, 873	6, 629	950	83	1, 561
AB-Market	66, 385	17, 500	7, 174	10, 326	95	5	14
Total	192, 705	60, 446	36, 429	24, 017	1, 859	109	1, 832

Table 4.3: Absolute error distribution. WB denotes Workbook, Occ. Occurences, Form. Formula, Lit. Literal, EC error-classes, CE copy-equivalence classes.

WB	**#Cells**	**#Occ.**	**#Form.**	**#Lit.**	$\frac{CE}{Formula}$	**#EC**	**Errors**
RAT-2001	56, 485	34.00%	64.00%	36.00%	66%	21	1.30%
TP Report	69, 835	34.00%	72.00%	28.00%	56%	83	6.70%
AB-Market	66, 385	26.00%	41.00%	59.00%	13%	5	0.08%
Total	192, 705	31.37%	60.27%	39.73%	51%	109	3.03%

Table 4.4: Error distribution, relative (Error-% is given relative to occupied cells). Abbreviations are the same as in Table 4.3.

to the workbook. This information is extracted from the companies' SAP-based information system. The workbook calculates a budget target for each factory that can be compared to the planned budget. The calculated budgets' values are about $13,000,000 each.

Overview of Results

In 78 audited spreadsheets, 109 error classes with 1,832 occurrences were identified (see Table 4.3 on the facing page). As the workbooks usually consisted of similar spreadsheets, the occurrence of one error class is not limited to one spreadsheet. We identified several error classes that were copied into different spreadsheets of the same workbook.

The workbook TP-Report was still under construction when the study finished and so many of the identified problems were immediately corrected. This explains why so many error classes were detected in this workbook. The workbook AB-Market was re-designed a short time before the audit took place. Hence, there was only a small amount of errors in the model.

The distribution of errors in the audited workbooks is given in absolute numbers in Table 4.3, whereas Table 4.4 gives the relative distribution as a percentage.

By classifying the errors and error classes into quantitative and qualitative errors, we obtained the distribution given in Table 4.5. The classification into the error-categories listed in Section 4.2 is given in Table 4.6 on the following page. The category *Others* consists of a wide diversity of error classes with patterns more or less unique for the individual instances.

Workbook	**Error Category**	**Error Classes**	**Errors**
RAT-2001	qualitative	7	84
	quantitative	14	183
TP Report	qualitative	73	1,503
	quantitative	10	58
AB-Market	qualitative	5	14
	quantitative	0	0
Total	qualitative	85	1,591
	quantitative	24	241

Table 4.5: Error classification into qualitative and quantitative errors

Error Category	**Error Classes**	**Errors**
Constant instead of formula	16	1222
Constant instead of reference	8	78
Reference to empty cell	8	78
Formula copied to far	24	215
Other	53	239

Table 4.6: Error distribution by error category

WB	**#Form.**	**#CE**	**EC**	**CE/Form.**	**EC/CE /**	**Err./Form.**
RAT-2001	12.382	811	21	6,6%	2,6%	2,07%
TP-Report	16.837	950	83	5,6%	8,7%	9,25%
AB-Market	7.174	95	5	1,3%	5,2%	0,19%
Total	36.429	1.859	109	5,1%	5,9%	5,02%

Table 4.7: Error class (EC) distribution, relative to copy-equivalence classes (CE)

The Copy-Equivalence to Formula ratio, i.e. the average size of each copy equivalence class was calculated. In the average, each copy-equivalence class contains 5.1 formulas. Thus, only every fifth formula cell of the spreadsheet had to be checked in detail. Of course, this measure is blurred, as there are certain formulas, e.g. check-sums or other validation formulas, that occur only once, whilst others occur more than 20 times. For multiple occurrences of the same formula it had only to be checked if they are used in the right place.

The frequency of occurrences of error classes relative to copy-equivalent classes is obviously related to the frequency of errors relative to formulas. This seems to support our assumption that errors are likely to be multiplied by copy & paste. However, as it is shown by Table 4.7, the workbook AB-Market does not follow this trend. It is argued that this is due to the 'youth' of this workbook. The errors detected seem to be mainly in check-sums and thus, not copied over many cells.

Discussion of the Audit's Result

The quality of the company's spreadsheet was surprisingly good at first sight. The audit did not reveal tremendously wrong results. This might be due to the fact that the spreadsheets have been properly tested in the developers' perspective, i.e. on the value level. However, the corrections were usually made on the value level, which made the spreadsheet model inconsistent. This bears the danger of spectacular errors to turn up in future evolution steps. However, the audit still discovered 241 quantitative errors in the spreadsheets.

The company's representatives were very concerned about the audit's result. They stated that better spreadsheet development practices are going to be introduced. The representatives were also interested in guidelines to decide whether a specific application should be realized by a spreadsheet or by a database application. Among of the suggested improvements were better documentation and the application of systematic testing and auditing approaches.

Summary

The following issues have been addressed in this chapter:

- There are various kinds of spreadsheet errors.
- Accidental errors cannot be avoided.
- Mistakes are caused by wrong concepts, whilst
- Errors are typical slips of the human brain.
- There are quantitative and qualitative errors/mistakes.

- Development experiments showed cell error rates between 1% and 14%
- As qualitative errors are not recognized on the value level,
- They accumulate during evolution cycles.
- Superficial error correction and lack of documentation are the main reasons for qualitative errors.
- Qualitative errors usually turn into quantitative errors in future evolution cycles.

Chapter 5

Survey of Spreadsheet Development, Auditing and Testing Methods

So far it was pointed out that spreadsheet errors are as widespread as spreadsheet applications and that they can have remarkable impact. In order to get the attention of managers, it is important to provide some drastic examples of the impact of errors and show the statistical evidence[1]. As a matter of fact, the negative arguments as such are usually not sufficient to convince decision makers.

In order to improve spreadsheet error rates, a way out of the chaos must be shown. Some researchers who deal with spreadsheet related issues, like Grossman [Gro02], call for a new discipline, the so-called spreadsheet engineering, to develop an integrated approach to support spreadsheet development.

It is considered important to stress the positive impact of a professional approach to spreadsheet development, i.e. shorter development time, less errors and a higher overall productivity. As stated by practitioners and researchers[2], these advantages of a defined process are even more suitable for gathering the attention of the decision-makers in commercial enterprises than the threat of errors.

Although there has been no widespread discipline of spreadsheet engineering yet, the remaining chapter will discuss spreadsheet development techniques that have been proposed up to now. Therefore, a categorization into techniques considering the development process as a whole (Section 5.1), layout-design guidelines (Section 5.2, logical design and implementation (Section 5.3), testing (Section 5.4) and visual spreadsheet auditing techniques (Section 5.5) is made.

[1]The same scene could be observed in the early days of software engineering, too [BBL76, Bro95, CK02]

[2]during a panel discussion at EUSPRIG 2002

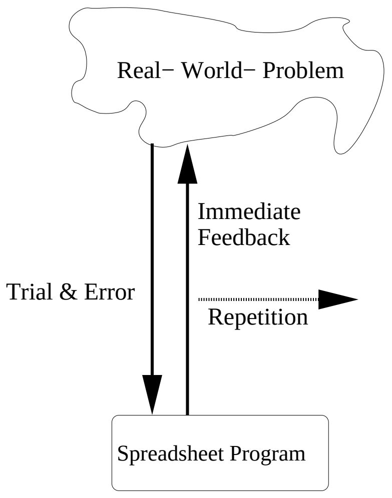


Figure 5.1: The spreadsheet development process as described by Nardi and Miller [NM90a] is similar to a code & fix life cycle of conventional software (described by Boehm [Boe88]).

5.1 The Spreadsheet Development Process

In Chapter 3 the differences between spreadsheets and conventional software were summarized. Putting the focus on the development process and trying to divide the process into the steps of a waterfall model (see Boehm [Boe88] for a discussion of software life cycle models) the following assumptions can be made:

Analysis: Apart from the fact that typical spreadsheet users are not aware that they are writing programs at all, they are also often not aware of the actual real-world problem to solve. Nardi and Miller [NM90a] state that the abstraction step that is performed by implementing a spreadsheet program is generally an important step for the users to get a better understanding of the real-world problem. As there is no explicit analysis step in the spreadsheet development process (see Figure 5.1), the problem is analyzed and understood by an iteration of trial and error steps.

Design: According to software-engineering literature (see e.g Budgen [Bud94]) designing conventional software deals with identification of components, their functionality and their interfaces. The design phase of spreadsheets is often driven by the desired geometrical design, not by the conceptual design (see e.g. Nevison [Nev87]). There is no specification of the spreadsheet program nor of the formulas of the cells. The spreadsheet writers try to find an allocation for values in a way that corresponds to the geometric model they have in mind.

Implementation: The implementation of a spreadsheet program differs from the implementation of conventional software, because of the trial-and-error iterations that we have already mentioned. This approach is also supported by the fact that the users immediately see the impact of their modifications to the spreadsheet. The missing design and specification also influences the implementation part. Furthermore, the techniques for reducing the complexity of programming conventional software, e.g. modularization, local variables or parameters, are not supported by spreadsheet systems. Multiple worksheets in a workbook are usually not suitable to introduce modularization. It turned out that the error rate of spreadsheets consisting of multiple worksheets increases (see Janvrin et al. [JM00]), because spreadsheet users are not forced to define interfaces. However, in the hands of a disciplined spreadsheet *programmer*, multiple worksheets can decrease the overall complexity of a spreadsheet program.

Testing: Considering conventional software testing gets a lot of attention. Testing spreadsheet programs deals with some extra problems. There is no specification, the people who test the spreadsheet are often those who implemented it. Consequently, there is no risk-awareness although a spreadsheet program is generally astonishingly complex. Long-term studies (see Chapter 4) stated that errors were found in a high number of the examined spreadsheets. The conclusion is that the explicit testing phase in the spreadsheet development process is either neglected or not performed with enough care, even if crucial decisions are based on the results of a spreadsheet.

Maintenance: Maintenance of spreadsheet programs is a rather tricky task. There are no design documents and normally there is also no documentation. Nevertheless, spreadsheets are often modified to solve problems that are related to the one they were designed for. Even when these modifications are done by people who are aware of the conceptual model behind the spreadsheets, this is a very error prone task.

Summarizing the statements above, the spreadsheet development process seems to consist of a design phase that deals with the layout of the spreadsheet program, a combined analysis and implementation phase and a neglected testing phase. Maintaining spreadsheets is very difficult but at the same time a common task.

The spreadsheet life cycle outlined so far corresponds with the findings of most of the studies that have been introduced in Chapter 4. However, it has also been shown that many errors can be avoided by a better planned process. Therefore, a structured spreadsheet development approach, the so-called R.A.D.A.R life cycle, is suggested by David Chadwick in [CR98, CRKE99].

5.1.1 The R.A.D.A.R Spreadsheet life cycle

The R.A.D.A.R Spreadsheet Life Cycle, developed by the University of Greenwich, divides spreadsheet development into five phases that sound familiar for software engineers:

1. Requirements
2. Analysis
3. Design
4. Acceptance
5. Review

However, the similarity is only superficial. Having a closer look at each of the phases reveals that some important differences appear that reflect the differences between spreadsheets and conventional software.

Requirements Phase

The requirements phase of the R.A.D.A.R life cycle model is very similar to the initial stage of the various life cycle models for conventional software. This phase is meant to answer the following questions:

- What is the purpose of the spreadsheet?
- Which staff are to be consulted?
- How long to do? What cost to do?

Although this phase is the only stage to collect requirements, these questions do not deal with requirements in the first place- in conventional software development cycle, this phase resembles the feasibility study.

However, the lack of an explicit requirements analysis is not as negative as it sounds considering the fact that spreadsheet writers are usually domain experts and write spreadsheets for their special needs. Nevertheless, later phases of this life cycle suggest the idea that the spreadsheet program is developed by a professional programmer for a customer (see the *Acceptance* and *Review* phases). In this case, the requirements phase appears to be insufficient.

Analysis Phase

The analysis phase that is attached to the requirements phase contains the following analysis, design and implementation tasks:

- Gather Data: interviews, documents
- Define Row and Column Titles,
- Define Modules: Data, Simple and Complex functions
- Identify Title Links
- Create Functions
- Create Lookup Data Modules
- Identify Data Links

Again, the analysis tasks are not what would be expected in the analysis stage of a life cycle for conventional software. The term *modules* that is used by Chadwick [CR98] refers definitions that are given by the *structured methodology for spreadsheet design*, a spreadsheet development technique that was developed at the University of Greenwich by Knight and Rajalingham [KCR00, RCKE00] (see Subsection 5.3.3). Although the life cycle model recommends this development technique, it is not restricted to it.

Linked to the structured spreadsheet design approach, the tasks of the analysis phase of the R.A.D.A.R life cycle are rather design and implementation than analysis tasks in the classical sense. However, spreadsheet writers usually do not comprehend the term *design* the way software engineers do. For spreadsheet writers, design rather refers to the geometrical layout of the spreadsheet. Thus, to avoid any ambiguities the term design is not used at this stage.

Design Phase

The design phase deals with finding the final layout for the spreadsheet program. The layout restrictions of the structured methodology for spreadsheet design (see Subsection 5.3.3) have to be considered. Surprisingly, additional tasks of this phase test the spreadsheet instance for correctness and develop a user guide.

Obviously, the spreadsheet program is instantiated in the design phase and, therefore, the first chance for testing it is at this stage. However, if the program is developed separately from the geometrical design (see the corresponding tasks in the analysis phase), the opportunity should be used for testing the program independently of the layout, as well.

Nevertheless, considering spreadsheet writers end-users, it is obvious that they do not distinguish between the spreadsheet program and the spreadsheet UI.

Acceptance and Final Review Phase

The last two phases are similar to deployment and maintenance phases in life cycles for conventional software. In the acceptance phase, the users should be trained, another testing phase, this time together with the spreadsheet users[3], is carried out, and eventually, the handover to the client takes place.

The final review phase had better be called the *operation and maintenance phase*, because it contains the following two steps:

- Let it operate for a while
- Does it satisfy the client? If not, then start again.

Evaluation and discussion of the R.A.D.A.R model

The R.A.D.A.R model has been taught at university courses for business students at Greenwich since 1998. Chadwick [CR98, Cha02] claims that the overall quality of the spreadsheets that are developed in graduate courses has significantly improved since then. As students have to write spreadsheet programs as main part of their exams, Chadwick argues that average increments between 3 and 6 points in the examination results are a proof for the efficiency of the R.A.D.A.R life cycle model. Chadwick [CR98] argues that common influences on examination grades, like well-motivated and well-trained teaching staff or a higher number of students attending the course, do not apply for this specific experiment, since

- only one of several courses was directed by a person involved in the development of the R.A.D.A.R life cycle. The other staff had to make themselves familiar with the technique during teaching. Thus, advantages in motivation might have been compensated by less experience of the teachers, and
- statistics have not shown higher student numbers in the courses.

However, the results have to be read with care, as there are many implicit influences on course results and an increase between 3 and 6 points, i.e. 3 and 6 percent, is, though significant, not revolutionary high.

The approach has also remarkable drawbacks. At first, the findings of Nardi and Miller [NM90a] on the nature of spreadsheet writers are ignored (see Section 3.1). In the R.A.D.A.R approach it is assumed that they are willing to submit to a rigorous life cycle and development methodology.

Additionally, the testing phase that should be paid specific attention to, becomes a sub-task of the geometrical design. This proceeding will encourage superficial testing with all its drawbacks (see Section 4.3). After all, having a look at the tasks in the requirements and acceptance phase of the process, it is obviously assumed that a client is involved in the development process. There are a few

[3]Chadwick[CRKE99] literally uses the term *user/client*

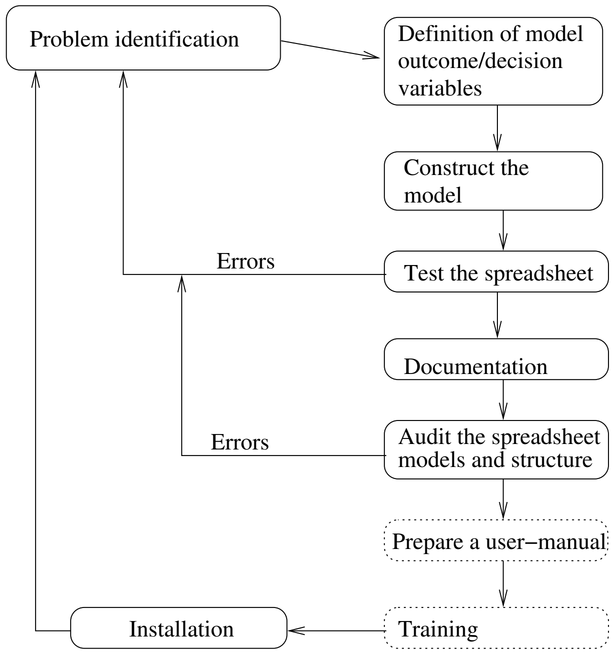


Figure 5.2: The spreadsheet development life cycle, suggested by Ronen [RPL89]. The dashed phases are optional.

spreadsheets that are developed by professional programmers for some clients, but as empirical research (see Chapter 2 and Chapter 4) has pointed out, most of the spreadsheets are developed by the person in charge in the organization.

5.1.2 Spreadsheet Analysis and Design

An alternative to the life cycle model described above is presented by Ronen et al. [RPL89]. Again, the suggested spreadsheet development life cycle considers many particularities of spreadsheet programs and their creators (see Figure 5.2).

Nevertheless, the life cycle model is intertwined with a spreadsheet development technique that imposes restrictions on the layout and forces the spreadsheet writers to introduce a design of the spreadsheet program by means of an extended data-flow graph. Ronen et al. [RPL89] recommend different approaches for the development of spreadsheet programs that depend on the context of use and the complexity. For spreadsheets that are intended for personal use only, an ad-hoc approach is suggested, whereas complex spreadsheets that are meant as company wide information sources should be implemented by using formal methods.

As a matter of fact, Ronen et al. do not pay enough attention to the evolution of spreadsheets. Although the life cycle model basically supports multiple iterations through the life cycle, the change of the importance and usage of spreadsheets is not taken into account. Not all the parameters that are required to make a decision about a spreadsheet program's importance are known in advance. Many spreadsheet programs are initially intended for short life cycles and personal usage, and eventually become important for the organization. Hence, the suggested development technology will not be the optimal one.

A more detailed description of the specific phases of the life cycle model will not be given at this point. However, these phases incorporate tasks that are very similar to the R.A.D.A.R model. Indeed, the R.A.D.A.R model seems to be another formulation of Ronen's ideas, replacing the specific data-flow-oriented design approach with other restrictions.

5.2 Layout-Design Guidelines

This section summarizes the guidelines for good spreadsheet design that are given in the literature. The guidelines presented here consider only the geometrical layout of the spreadsheet and thus, have to be separated from layout restrictions based on the formula structure that will be surveyed in Subsection 5.3.3.

Subsequently, different style guides, i.e the *10 Tips to Improve Spreadsheet Style* guide [Pfa01], *Spreadsheets with Style* [Nev87] and the recommendations of Ronen[RPL89] will be briefly discussed.

5.2.1 Improving Spreadsheet Style

Pfaffenberger [Pfa01] gives 10 rather simple recommendations to make spreadsheets more readable. Although they are rather simple-minded at the first glance, to obey these recommendations will effectivley increase the readability of spreadsheets. Nevertheless, as spreadsheet programs are often based on predetermined layouts, spreadsheet writers are not willing to follow the recommendations. Additionally, some of the given recommendations obviously contradict other style-guides. So, Nevison's style-guide [Nev87] recommends to make blocks for constants, input and formulas, whereas Pfaffenberger recommends to keep related things locally together.

The following recommendations are given:

Write a spreadsheet like a report

Thus, write from the left to the right, and from bottom to top. Indeed, there are also some spreadsheet auditing tools (e.g. Chan [CC00, Cha01]) that bank on such regularities of spreadsheet programs.

Put formulas and related constants together

Pure input-blocks should be avoided. It is argued that this improves the spreadsheet user's understanding of the context of a formula. This is surely the case.

However, the maintenance of the spreadsheet program, and even the (re-)instantiation of a spreadsheet program becomes more difficult, as the spreadsheet user has to be aware whether he is allowed to overwrite a formula or not.

Protecting the formulas is no solution, as it has been observed that the spreadsheet users tend to override cell-protections that *they* consider useless. Additionally, some of the formulas might actually be input-data and only used for convenience.

The three crayon rule

It is argued that people cannot remember the meanings of more than three different colors. This suggestion seems reasonable. However, psychologists argue that the short-term memory is able to store 4–5 different colors (see Lee and Chun [LC01]).

The glowing formula rule

This rule suggests that the used formulas should be marked by a special color. Although this might increase the understanding of where formulas are, this rule is surely not applicable in domains where the meaning of figures depends on their color (e.g. bookkeeping).

Put it all on one sheet

It is argued that the usage of multiple worksheets significantly increases the overall error rate of spreadsheet programs (see Janvrin [JM00]). Additionally, it is stated that the user still has a chance to comprehend the structure of a spreadsheet program by using the built-in zoom function if it is presented on a single sheet.

However, very large spreadsheets, that are often found in commercial applications, consist of up to $70,000$ cells (see Clermont et al. [CHM02]). On a single spreadsheet, this would take at least 270×270 cells without a single blank cell in between. Thus, it is questionable if this suggestion is applicable for large spreadsheets.

The remaining tips are more or less common sense, e.g. unambiguous labels should be used, leave the grid as it is, cells with a formula must not look blank, be concise, and so on.

Although each of the suggestions can be put in question either by empirical, psychological or domain-specific findings, following them will definitley improve the spreadsheet's readability. As readability influences the spreadsheet's quality in the long term, many maintenance errors that are introduced due to misunderstandings of the model can be avoided.

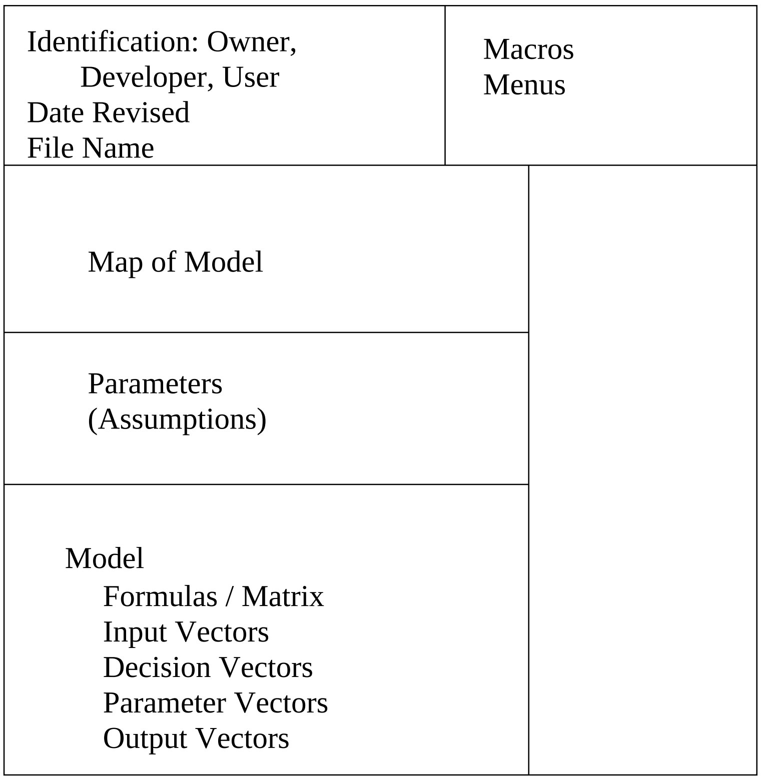


Figure 5.3: Spreadsheet structure recommended by Ronen et al. [RPL89]

5.2.2 Further Style Guides

Many other style guides (see e.g. Nevison [Nev87], Ronen [RPL89] and Berry [Ber86]) recommend a different arrangement of the spreadsheets' contents (see Figure 5.3). The spreadsheet is split up into different parts with different tasks. In contrast to Pfaffenberger's recommendation, the geometric distance between input values, constants and the concerned formulas can be rather large, as there is a section for the input values, one for formulas and another one containing the constants.

Although the self-explanatory powers of such a spreadsheet are usually not very high, such an arrangement ressembles much to conventional software that is structured similarly. Thus, maintenance and re-instantiation of a given spreadsheet program can be performed far more efficiently and less error prone. The locations of formulas are obvious, and the spreadsheet users are aware of which cells can be provided with input data (even in the form of simple formulas).

Nevertheless, the connection between input values and the corresponding formulas is not clear anymore and the spreadsheet users might not be aware of the overall influence that changes in the input can have on the output of the spreadsheet program (yet, this is similar to conventional software, too).

In order to increase the spreadsheets' comprehensability, a large part of the spreadsheet (the *map of model* in Figure 5.3) is dedicated to explain how the model works and where on the spreadsheet UI a certain part of the model is implemented. The *parameter*-block contains the constant values that are used in the formulas. Finally, in the *model*-block, the input, formula and output vectors are placed. It is suggested to consider whole columns or rows to be such vectors.

In fact, many users intuitively come to a spreadsheet layout that conforms to such a style guide. However, the clear separation tends to blur over time, as some of the entries in the input vector become formulas themselves, or new parameters are introduced (or some others replaced by formulas) without adjusting the spreadsheet programs documentation.

5.3 Logical Spreadsheet Design

As spreadsheet writers often sense a spreadsheet as a modeling tool (see Nardi and Miller [NM90a]), the layout-design guidelines aim to reduce their models' complexity by restricting the number of possible layouts. Similar to software engineering there are also some approaches that try to reduce the complexity of spreadsheet programs even further by introducing a certain design paradigm that can be data-flow-oriented (e.g. Ronen [RPL89]), layer-oriented (e.g. Isakowitz [ISL95]), structure-oriented (e.g. Knight and Rajalingham [KCR00, RCKE00, CRKE99] or constraint-based (e.g. Stadelmann [Sta93] and Wilde [Wil93]).

Generally, each of these approaches goes beyond simple limitations of the layout. They introduce an explicit design[4] and modeling step into the spreadsheet development lifecylce. Hence, the tasks of implementing, testing and maintaining a spreadsheet program will be supported by the explicitly stated design. Nevertheless, following these approaches also imposes restrictions on the spreadsheet layout.

Subsequently, each of the above stated spreadsheet design paradigms will be presented and discussed.

5.3.1 Data Flow Oriented Spreadsheet Design

In Subsection 3.3.5 the spreadsheet evaluation strategy was widely discussed. It has been shown that a spreadsheet program appears at the first glance as a data flow program. Data-flow-oriented spreadsheet design techniques build up upon this view on the spreadsheet. Usually, the spreadsheet *designer* is offered an enriched data flow modeling technique in order to visually model the DDG (see Definition 13 on page 24) and the formulas.

For larger spreadsheets the DDG is a complex graph. Thus, a visual model of the pure DDG is not of much gain for the spreadsheet's maintainers. Therefore, Ronen [RPL89] offers a more compact modeling technique. Instead of modeling

[4]This time, the term design does not denote geometrical or layout design, but logical design

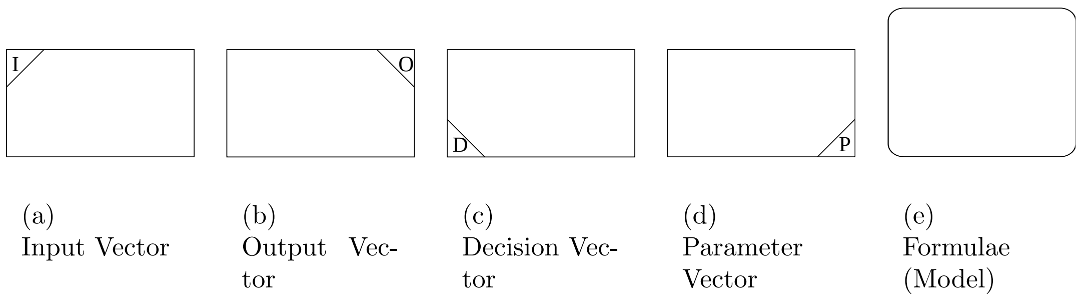


Figure 5.4: Elements of the spreadsheet flow diagram notation, introduced by Ronen [RPL89].

the data flow on a detailed level, they suggest to model relationships. Therefore, the so-called *Spreadsheet Flow Diagram* (SFD) (see Figure 5.4) is defined. In this notation, formulas with equal semantics are represented by a single node, which also applies for their input and output.

In order to visually model a spreadsheet with the spreadsheet flow diagram, five different node types are available: *input vector*, *output vector*, *Decision Vector*, *parameter vector* and *formulae*. Each of these nodes is briefly described in the following.

Formulae

Formulae are the place where data manipulation is performed. Thus, they are targeted by edges from all node types. An arbitrary number of formulas with the same semantic are represented by one formula node. Each of these formulas may consume only one value from each targeting node and supply only one output.

A formula node can be further decomposed into less complex formula nodes. Thus, the formula node might have more than one corresponding cell on the spreadsheet UI for each of the elements (see Example 2).

Input Vector

The input vector models cells that are referenced by a formula. In the graphical notation, an edge is placed between the input vector and formula node. Although the input vector contains a set of semantically related cells, only one element of the vector serves as the input for one formula. It is suggested to place cells that are modeled by the same input vector adjacent as rows or columns on the spreadsheet UI.

Only user supplied input is modeled by the input vector. If the results of another formula serve as input for a computation, this is modeled by an edge between the two formula nodes.

Output Vector

The notion of an output vector is not clear and not further defined by Ronen et al. It seems that they distinguish between intermediate results, that are only represented by a formula node, and final results. Cells that belong to a result that is obviously interesting for the user, are represented by such an output vector. To model the origin of the result, an edge is placed from the corresponding formula to the output vector.

Decision Vector

The decision vector is similar to the input vector. However, cells in a decision vector are only used for conditions in *if-statements* in the formula vector.

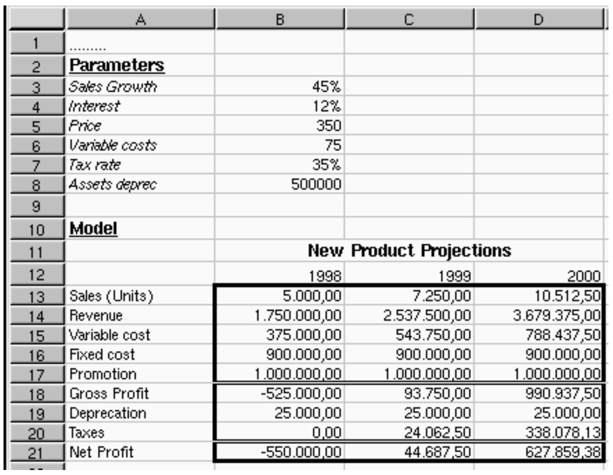
Parameter Vector

The parameter vector contains cells that contain variables the calculation depends on, but are not supplied by the user, e.g. tax-rates. Parameters are often used by more than one formula. As opposed to inputs, the same parameter can be used by more than one formula.

In [RPL89] the following example for the illustration of spreadsheet flow diagrams is given.

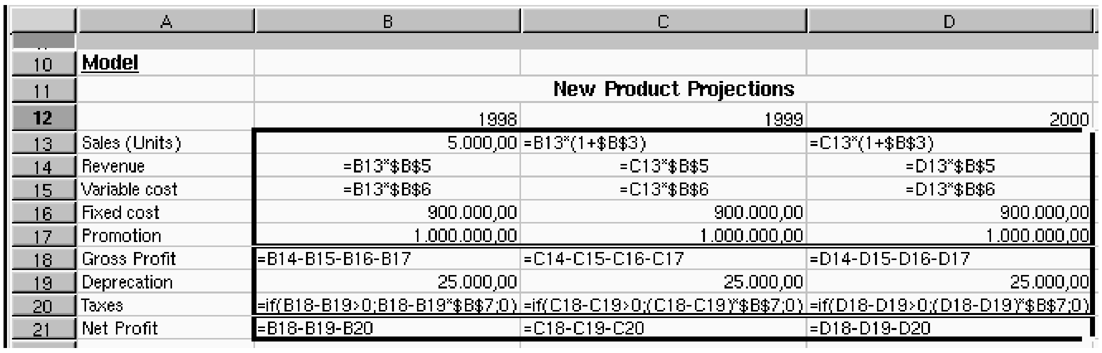
Example 2: Spreadsheet Flow Diagram
In Figure 5.5(a) on the following page a spreadsheet for predicting sales of a new product is introduced. The upper part, containing information about release, author as well as brief documentation of the spreadsheet's model has been skipped. The supplied parameters are used by the formulas of the model via absolute cell references, whereas all other references are relative ones (the formula view of the spreadsheet's model part is given in Figure 5.5(b) on the next page).
In Figure 5.6(a) on page 81 a high level spreadsheet flow diagram is given. The input vector has a subscript $_t$ to indicate that the model is predicting values over a time period. In Figure 5.6(b) on page 81 the formula is exploded into a lower level of detail. ◊

Obviously, the SFD of a spreadsheet is a helpful means of understanding a spreadsheet program. The logic of the spreadsheet program is explicitly stated, and thus, easier to comprehend. However, the SFD notation is not widely accepted amongst spreadsheet programmers and no further evaluation was done. One of the drawbacks of the technique is that it forces the user to make a data-flow-oriented (or, as Ronen argues, a relationship-oriented) model of their problem, before they start writing the spreadsheet program. Thus, the great value of the spreadsheet as modeling tool (see [NM90a]) is neglected.



	A	B	C	D
1	....			
2	**Parameters**			
3	*Sales Growth*	45%		
4	*Interest*	12%		
5	*Price*	350		
6	*Variable costs*	75		
7	*Tax rate*	35%		
8	*Assets deprec*	500000		
9				
10	**Model**			
11			**New Product Projections**	
12		1998	1999	2000
13	Sales (Units)	5.000,00	7.250,00	10.512,50
14	Revenue	1.750.000,00	2.537.500,00	3.679.375,00
15	Variable cost	375.000,00	543.750,00	788.437,50
16	Fixed cost	900.000,00	900.000,00	900.000,00
17	Promotion	1.000.000,00	1.000.000,00	1.000.000,00
18	Gross Profit	-525.000,00	93.750,00	990.937,50
19	Deprecation	25.000,00	25.000,00	25.000,00
20	Taxes	0,00	24.062,50	338.078,13
21	Net Profit	-550.000,00	44.687,50	627.859,38

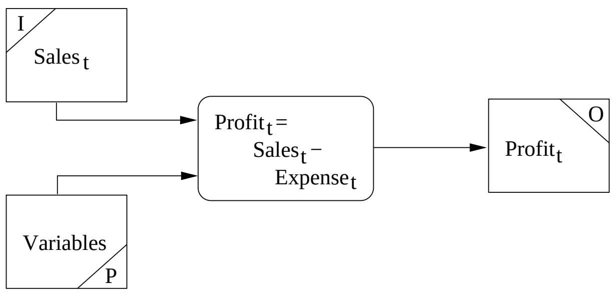
(a) The spreadsheet



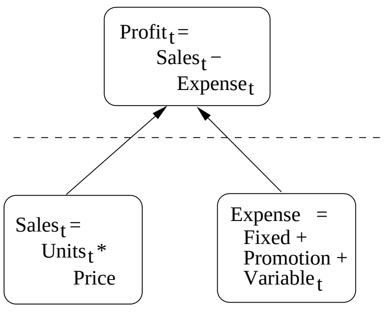
	A	B	C	D
10	**Model**			
11			**New Product Projections**	
12		1998	1999	2000
13	Sales (Units)	5.000,00	=B13*(1+$B$3)	=C13*(1+$B$3)
14	Revenue	=B13*$B$5	=C13*$B$5	=D13*$B$5
15	Variable cost	=B13*$B$6	=C13*$B$6	=D13*$B$6
16	Fixed cost	900.000,00	900.000,00	900.000,00
17	Promotion	1.000.000,00	1.000.000,00	1.000.000,00
18	Gross Profit	=B14-B15-B16-B17	=C14-C15-C16-C17	=D14-D15-D16-D17
19	Deprecation	25.000,00	25.000,00	25.000,00
20	Taxes	=if(B18-B19>0;B18-B19*$B$7;0)	=if(C18-C19>0;(C18-C19)*$B$7;0)	=if(D18-D19>0;(D18-D19)*$B$7;0)
21	Net Profit	=B18-B19-B20	=C18-C19-C20	=D18-D19-D20

(b) Formula view of the model part

Figure 5.5: Example spreadsheet for the SFD modelling technique.



(a) High-level SFD for the model


Profit$_t$ = Sales$_t$ − Expense$_t$
Sales$_t$ = Units$_t$ * Price
Expense = Fixed + Promotion + Variable$_t$

(b) More detailed SFD for the profit-formula

Figure 5.6: Example SFD model of a simple spreadsheet (see Figure 5.5(a) on the preceding page)

Additionally, the mapping from the SFD to a specific spreadsheet program is not as clear as it seems. Elements of a specific vector have to be mapped to a spatial area of the spreadsheet. Although the membership in a specific vector is due to semantic criteria, it does not explicitly require a sameness of the formula. In Example 2, $\mathsf{Sales_t}$ is represented by the formula $\mathsf{Sales_t} = \mathsf{Units_t} * \mathsf{Price}$. However, having a look at Figure 5.5(b) on page 80, it is recognized that the units for 1998 are a constant value, whereas those for 1999 and 2000 are calculated by a formula.

Another ambiguity of the suggested modeling technique is the assumed classification of cells into input, formula and output cells. Although this structure seems reasonable at first sight, the nature of spreadsheet programs does not allow an unambiguous classification of cells into one of these categories. Each cell is displayed, hence each cell might be an output cell. Each cell can be referenced by formulas, and the user is free to supply an input by specifying a formula, thus each input cell can be a formula cell and vice-versa. The criticized ambiguities are also shown in Example 2. In Figure 5.6(a) on the page before $\mathsf{Sales_t}$ is modeled as an input vector, whereas a look at the model in more detail reveals that $\mathsf{Sales_t}$ is a formula.

Although the nature of spreadsheet programs and spreadsheet writers makes it very hard to introduce an explicit design step in the spreadsheet development life cycle, the SFD notation obviously supports the comprehension of a given spreadsheet program. In Chapter 6, an approach is introduced that can reverse engineer spreadsheet programs into a notation that is very similar to the SFD.

5.3.2 Relational Spreadsheet Modeling

Usually the software's logical model is treated separatley from the input data. However, spreadsheet's are different in this request, too. The spreadsheet writers are encouraged by the user interface of the spreadsheet system to intertwine the spreadsheet program nearly inseparably with the input data. Although the approaches described so far postulate the introduction of any kind of logical design for a spreadsheet, no distinction between model and input data can be made after the spreadsheet has been instantiated.

Therefore, Isakowitz et al. [ISL95] suggest to view a spreadsheet separated into a logical and a physical view. The physical view of a spreadsheet instance is defined as a set of triples that relates cells with values and formating rules.

Definition 15: Physical View
The physical view of a spreadsheet instance is a set of triples: $SSI = \{(CA, \mathsf{Definition}, \mathsf{formatting\_rules}\}$. CA denotes a cell's address, the definition specifies the value that is displayed at the given address, and can be either a calculated or a constant value. □

Thus, Isakowitz et al. [ISL95] argue that the spreadsheet UI can be constructed by rendering the physical view of the spreadsheet instance. As only values are rendered, formulas are not part of the physical view. However, this does not capture the essence of spreadsheet programs. Therefore, the logical view is introduced.

The logical view of a spreadsheet consists of so-called *functional relations*. Functional relations are defined similarly to relations in relational databases, as they consist of attributes and can have one or more tuples (see [EN94, Vos00] for a discussion of relations and their use in relational database systems), but they can contain either *data* attributes, i.e. constants, or *functional attributes* that are bound to functions that are calculated as needed.

Isakowitz et al. denote the set of functional relations that are stated to build the skeleton of a spreadsheet[5] with the letter S. As a spreadsheet also consists of some constant values and editorial remarks, e.g. labels, the data property D holding all constant values and an editorial property E for labels and formatting issues are introduced. In order to map such a logical view of a spreadsheet to a physical view, a binding property B is specified, too. Thus, a spreadsheet can be defined in the following way:

Definition 16: Spreadsheet
A Spreadsheet is the (symbolic) sum of $S + D + E + B$. □

Although this definition is not similar to Definition 4 on page 20, there is no contradiction. On the one hand, we argue that the spreadsheet is a set of cells, on the other hand, in [ISL95] some implicitly given semantics of these set of cells are factored out.

In order to clarify the terms and concepts, Example 2 is resumed.

Example 3: Extracting the logical model from a spreadsheet
The spreadsheet that is presented in Figure 5.5 on page 80 consists of two separate relations, one for the parameters and the second for holding the model for prognosis. While the parameter section contains only constants, in the model section several cell values are calculated by formulas. Thus, the functional relation S (see Figure 5.7(a) on the next page) also contains the two relations. In the relation *parameters* only references to numeric entries are made, whereas relation *New Product Projections*, i.e. the model part, contains both formulas that are stated in the functional relation, and references to numeric entries.

The data relation D (see Figure 5.7(b) on the following page) contains all the remaining numeric values. Calculated values are represented by the letter 'c'. ♢

It is remarkable that the representations of S and D are stated independently from the spreadsheet UI. In [ISL95] it is argued that the data for D can be easily read out from any relational database. It is only a matter of the binding B to map the so stated logical model to a spreadsheet.

[5]This corresponds to our findings in Definition 10 on page 23 where the spreadsheet program is defined to be essentially formed by the cells that contain formulas.

relation parameters**alias** a **type vector**

SalesGrowth:	numeric
Interest:	numeric
Price:	numeric
VariableCosts:	numeric
TaxRate:	numeric
AssetsDeprec:	numeric

relation New Product Projections **alias** p

_Unnamed$_1$:	numeric key
Sales(Units):	`if` $n = 1$ `then` numeric `else` $p_{n-1}.Sales(Units) * (1 + a.SalesGrowth)$
Revenue:	$p_n.Sales(Units) * a.Price$
VariableCost:	$p_n.Sales(Units) * a.VariableCosts$
FixedCost:	numeric
Promotion:	numeric
GrossProfit:	$p_n.Revenue - p_n.VariableCost - p_n.FixedCost - p_n.Promotion$
Deprecation:	numeric
Taxes:	`if`$(p_n.GrossProfit - p_n.Deprecation) > 0$ `then` $(p_n.GrossProfit - p_n.Deprecation) * a.TaxRate$ `else` 0
NetProfit:	$p_n.GrossProfit - p_n.Deprecation - p_n.Taxes$

(a) Schema S

a					
SalesGrowth	Interest	Price	VariableCosts	TaxRate	AssetsDeprec
0.45	0.12	350	75	0.35	500000

p									
_U$_1$	SU	RV	VC	FC	Promo	GP	Deprec.	Taxes	NP
1998	5000	c	c	900000	1000000	c	25000	c	c
1999	c	c	c	900000	1000000	c	25000	c	c
2000	c	c	c	900000	1000000	c	25000	c	c

(b) Data D

Figure 5.7: Factorization of the spreadsheet given in Figure 5.5 on page 80 into functional (Schema) and data relations. In Figure 5.7(b) the names of the attributes were abbreviated.

The so stated logical model supports maintenance as well as testing, since the complexity of the spreadsheet is reduced, because formulas become visible and the data flow is stated in a more explicit way. However, this approach is based very much on mapping a spreadsheet instance to relations. But spreadsheet writers who are usually end-users (see the detailed discussion in Section 3.1) lack background knowledge about the relational paradigm, that is nowadays taught to every computer science student. Thus, it is questionable if they benefit from a logical, relational representation of a spreadsheet.

Additionally, the separation between formulas and data, although highly appreciated by software engineers[6], might make it harder for end-users to comprehend the spreadsheet. A certain kind of spreadsheet writers, namely IT-professionals or consultants that specialize in spreadsheets, are of course supported by this approach.

Testing of formulas becomes much easier, as the spreadsheet program can be tested independently of the layout. In constrast, many of the spreadsheet testing methodologies that will be presented in this chapter only support the testing of a specific spreadsheet instance.

Forward and Reverse Engineering

For most of the spreadsheet programs that are in use today, the S, D, B and E relations or properties are not explicitly stated. Thus, Isakowitz also presents a set of *Excel*- macros that allow users to visually select parts of the spreadsheet and supply some extra information in order to automatically extract the S, D, B and E and the FRL representation from a spreadsheet instance. Therefore, in a first step a map is extracted. The map is a triple of the logical address, the physical address and either a formula, a constant value or a string. For an example of a map, see Table 5.1 on the next page.

Having the map and the bounds of the relations, D and S have to be extracted[7]. Therefore, a factoring algorithm with the following steps is introduced by Isakowitz et al. Subsequently, the abbreviation CA denotes the cell address (see Definition 2 on page 19)

1. For each constant map entry of the form $r[i].x : CA, \mathsf{constant}$, rewrite the entry as $r[i].x : CA, \mathsf{data\_type}$, where data_type denotes the constant's data type.

2. For each formula map entry of the form $r[i].x : CA, formula$, rewrite the entry as $r[i].x : CA, formula'$, where $formula'$ results from replacing the addresses of all referenced cells in $formula$ with their map label. Absolute

[6]However, the object-oriented paradigm breaks this separation.

[7]As the extraction of B and E is more or less straightforward, a detailed description is not considered necessary. A brief description is given by Isakowitz in [ISL95].

$p_1.Unnamed_1$	B12	1998	$p_1.Promotion$	B17	1000000
$p_2.Unnamed_1$	C12	1999	$p_2.Promotion$	C17	1000000
$p_3.Unnamed_1$	D12	2000	$p_3.Promotion$	D17	1000000
$p_1.SalesUnits$	B13	5000	$p_1.GrossProfit$	B18	=B14-B15-B16-B17
$p_2.SalesUnits$	C13	B13 ∗ $B$3	$p_2.GrossProfit$	C18	=C14-C15-C16-C17
$p_3.SalesUnits$	D13	C13 ∗ $B$3	$p_3.GrossProfit$	D18	=D14-D15-D16-D17
$p_1.Revenue$	B14	B13 ∗ $B$5	$p_1.Deprecation$	B19	25000
$p_2.Revenue$	C14	C13 ∗ $B$5	$p_2.Deprecation$	C19	25000
$p_3.Revenue$	D14	D13 ∗ $B$5	$p_3.Deprecation$	D19	25000
$p_1.VariableCost$	B15	B13 ∗ $B$6	$p_1.Taxes$	B20	`IF` ···
$p_2.VariableCost$	C15	C13 ∗ $B$6	$p_2.Taxes$	C20	`IF` ···
$p_3.VariableCost$	D15	D13 ∗ $B$6	$p_3.Taxes$	D20	`IF` ···
$p_1.FixedCost$	B16	900000	$p_1.NetProfit$	B21	B18-B19-B20
$p_2.FixedCost$	C16	900000	$p_2.NetProfit$	C21	C18-C19-C20
$p_3.FixedCost$	D16	900000	$p_3.NetProfit$	D21	D18-D19-D20

Table 5.1: The map for the *New Product Projection* (**p**) relation in Figure 5.7(a) on page 84.

$p[1].SalesUnits$	numeric
$p[2].SalesUnits$	$p[n-1].SalesUnits * a[0].SalesGrowth$
$p[3].SalesUnits$	$p[n-1].SalesUnits * a[0].SalesGrowth$

Table 5.2: Map entries for relation p, attribute $SalesUnits$. The factoring algorithm has been applied up to step 5.

cell references are marked by writing a \$ sign before the tuple index in $formula'$.

3. Remove CA from all map entries.

4. Collate map entries $r[i].x$ in clusters,
so that each cluster contains entries that have identical relation names (r) and attribute names (x). Whithin each cluster, sort the entries by the tuple index (i).

5. Convert absolute tuple references into relative tuple references.
For each map entry $r[i].x$, with an associated formula containing a reference to $r[j].y$, i.e. within the same relation, let $d = i - j$.

   (a) If $d < 0$, replace j with $n - d$,

   (b) if $d > 0$, replace j with $n + d$,

   (c) if $d = 0$, replace j with n.

   where n denotes literally the character 'n'. Thus, the target of intra-relation references is now given by the attribute name and the distance within the relation.
For each attribute reference of the form $r[\$j].z$, i.e. for each absolute attribute reference, rewrite the reference as $r[j].z$.

6. Contract the map.
Subsequently, the term cluster denotes a set of map entries with an equal right-hand side of the entry. In order to aggregate a set of map entries into a cluster, they must

   - denote the same attribute within the same relation, and
   - have consecutive tuple indices.

   E.g. in Table 5.2 on the preceding page, $p_2.SalesUnits$ and $p_3.SalesUnits$ are a cluster. $p_1.SalesUnits$ has a different right-hand side, hence it is not in the same cluster. Having constructed the clusters,

   (a) for each cluster eliminate all but the first entry (i.e. the entry with the lowest tuple index) from the map. The map label of the first entry is subsequently denoted *cluster entry label* and

   (b) compute tuple index ranges for each cluster. Let cluster entry labels be $r[k_1].x, r[k_2].x, \cdots, r[k_m].x$, with $k_j \in \mathbb{N}$. For each cluster, rewrite the entry labels as $r[k_1 \leq n < k_2], r[k_2 \leq n < k_3], \cdots, r[n \geq k_m]$.

   (c) If a rewritten entry becomes $r[j \leq n < j + 1]$, rewrite it again as $r[n = j].x$.

(d) If a cluster consists of only one entry, rewrite its single label as $r[n].x$.

7. Consult the user supplied information about the spreadsheet's structure to

   - get the name for all relations,
   - find out the relations' cardinality,
   - identify the keys.

This algorithm results in a representation of D and S, independent from the spreadsheet UI (see Figure 5.7 on page 84). Given such a representation, as well as the B and E properties of a spreadsheet, forward engineering from the logical towards the physical model of a spreadsheet can be done by a synthesis algorithm that is also proposed in [ISL95]:

1. Let the logical schema of relation r consist of one or more lines of the form $(x : definition)$, where x is an attribute name.

   (a) If x's definition contains no case structures, rewrite the line as $(r[n].x : definition)$.

   (b) If x's definition contains linews of the form $condition_j \rightarrow definition_j$, rewrite the lines as $(r[condition_j]).x : definition_j$.

2. For each attribute reference that occurs in the definition part of each line, rewrite the references in an extended `FRL`-syntax[8].
   If an attribute reference becomes $r[j].x$ for some constant j, rewrite it further as $r[\$j].x$.
   Eliminate the header r from the schema.

3. Let m be the number of tuples in r, i.e. the cardinality of r.

   (a) Replace each line of the form $(r[n].x : definition)$ with a series of m lines of the form $(r[1].x : definition), \cdots, (r[m].x : definition)$.

   (b) Replace each line of the form $(r[k_1 \leq n \leq k_2].x : definition)$ with a series of $k_2 - k_1 + 1$ lines of the form $(r[k_1].x : definition), (r[k_1 + 1].x : definition), \cdots, (r[k_2].x : definition)$.

   (c) Replace each line of the form $(r[n \geq k].x : definition)$ with a series of $m-k+1$ lines of the form $(r[k].x : definition), \cdots, (r[m].x : definition)$.

   (d) Replace each line of the form $(r[n = j].x : definition)$ with a single line of the form $(r[j].x : definition)$.

[8]FRL is a formula-relational-language. A detailed definition is beyond the scope of this work and can be found in the appendix of [ISL95].

4. Let $r[i].x$ be the label of a line, and $r[n+j].y$ is a related (i.e. within the same relation) attribute reference in the line's definition. For each such line, rewrite its related references as $r[d].y$, with $d = i + j$.

5. Construct the physical map. For each logical map entry of the form $(r[i].x :$ $definition)$ and a matching binding entry, create a physical map entry of the form $(r[i].x : \ CA, definition)$.

6. Convert relational references to cell addresses. For each map entry of the form $(r[i].x : \ CA, formula)$, replace $formula$ with $formula'$ which is the same as $formula$, except that each attribute reference $p[j].y$ that appears in formula is subsituted with the cell address of the map, whose entry label is $p[j].y$.

7. Add editorial entries. Merge the list of map entries produced by now with the list of editorial entries from E.

Discussion

This approach offers many promising features for spreadsheet programmers. Amongst them are

- a logical model that is independent of the physical layout of the spreadsheet instance,
- a neat separation between data and formulas,
- easy integration of spreadsheet programs and relational databases,
- data dependencies become obvious through the `FRL` language for formulas, and
- the overall complexity of the spreadsheet might be reduced, because repeatedly used formulas are collapsed into so-called clusters.

The last issue supports testing, maintenance and the evolution of spreadsheets.

However, the suggested methodology did not break through in practice, because of numerous reasons. As it is already stated above, spreadsheet writers are not familiar with the relational paradigm this approach builds upon. As spreadsheet writers are end-users, the abstraction of the spreadsheet into relations tends to puzzle them.

The abstraction that can be generated from a given spreadsheet instance is indeed helpful, but not sufficiently coarse-grained to support the comprehension of large spreadsheet instances. Relations that are repeatedly used throughout a spreadsheet are not recognized as being *of the same kind.* Thus, for each instance of a relation, a separate `FRL` representation is calculated. Isakowitz et al. are aware of this problem and argue that an algorithm operating on the `FRL` program might help.

In the given context the term *relation* covers rather semantic relations, i.e. what the user considers to be related. In fact, each tuple in such a relation can have another structure of (potentially different) formulas and constants. They are grouped into relations by the user who selects areas on the spreadsheet UI and assigns names. Hence, finding similar relations is obviously very hard, when the internal structure of the tuples in the same relation is very heterogeneous. To sum up, in order to have an advantage of this approach, the spreadsheet writer must

- understand the relational paradigm,
- be familiar with the `FRL` language, and
- be disciplined in order to generate homogenous relations.

Obviously, these requirements will be hard to meet for end-users.

A Spreadsheet Compiler

Isakowitz et al. did not resume their work. However, others have picked up the thread. Paine [Pai97b, Pai97a, Pai01] introduced an approach to specify a spreadsheet program via an object-oriented language, thus supporting inheritance and specialization. Hereby, the relations that are described in [ISL95] are the objects. However, Paine does not primarily aim to generate spreadsheets. What he really wants is to take advantage of a spreadsheet like user interface. However, some features of spreadsheet systems are not supported by ModelMaster, which is the official name of the spreadsheet compiler.

ModelMaster compiles the program stated in the spreadsheet specification language to HTML-code that can be viewed with a web browser. The interactivity of spreadsheet programs is, of course, not present any more. Spreadsheet users cannot browse through the program and have a look at formulas or change them. Of course, sometimes this is exactly what spreadsheet writers desire.

Additionally, a spreadsheet specification can also be extracted from a spreadsheet. Therefore, an algorithm that is similar to Isakowitz's factoring algorithm is applied. The so generated specification can then be presented as a read-only, numbers-only document on the world wide web. An on-line demonstration of ModelMaster is available at [Pai02], for the fundamental functioning of ModelMaster see Figure 5.8 on the preceding page.

The improvement compared to [ISL95] is obvious: no user interaction is necessary to translate a spreadsheet to an HTML-page. The partitioning into relations is rather based on heuristics that are based on the geometrical orientation and neighborhood of cells, as well as on equality of the cell's formulas.

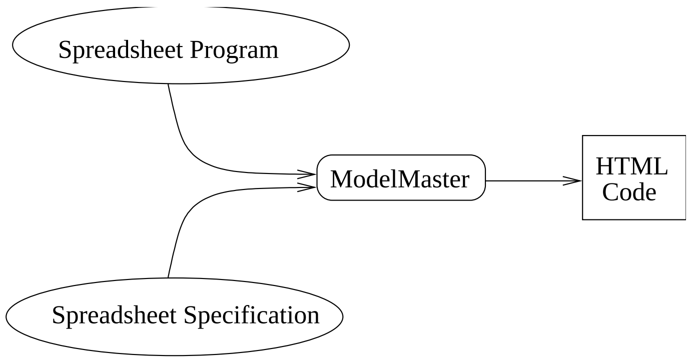


Figure 5.8: ModelMaster compiles either a spreadsheet specification in a specific language into a html-page, or extracts the spreadsheet specification from a given spreadsheet instance

5.3.3 Structured Spreadsheet Modeling

Obviously, Isakowitz et al. [ISL95] relate the logical design of spreadsheets to databases and therefore, suggest a relational approach. Of course, spreadsheets can also be seen as software that manipulates data rather than a data store. Thus, relating the logical design of spreadsheets to structured program design, as it was described by Jackson [Jac75] is legitimate.

Rajalingham et al. [RCKE00, RCK02, KCR00] suggest to design spreadsheets like programs. They embedded their design approach into the analysis stage of the R.A.D.A.R life cycle model for spreadsheets that has been described in detail in Subsection 5.1.1.

The structured spreadsheet design approach is top-down oriented, i.e. the design starts by defining the desired output of the spreadsheet program. Each of the identified outputs are then decomposed into parts that are again analyzed and disassembled until the spreadsheet writer does not consider a further fragmentation to be useful. If a specific unit is used by more than one superordinate unit, it is duplicated. In order to keep the design simpler, only the unit is duplicated, but not its subordinates.

Each of the parts is then analyzed in terms of whether it is iteratively repeated, one part of an alternative decision or one step in a sequence. Thus, the outcome of this design-step is a tree structure, representing the relationship of cells.

Implementation

Once the logical design is completed, it is suggested to figure out the formulas that correspond to each node in the structure tree. In order to keep things clear, there is a predetermined rule of how to map the logical design to a geometrical design:

- For each row, only 2 columns are filled in.

- A label has to be filled in into the outer left column.
- Each level of the structure tree is assigned to a specific column on the spreadsheet UI. The formula of the structure tree's root has to be placed in the second column in the first row[9], the formulas of the subordinate levels emerge from left to right in the consecutive rows.
- Each formula may only reference inputs that are placed in a specific spatial area, and cells that are geometrically:
  - exactly one column to the right, and
  - below the referencing formula, and
  - there must not be another entry in the column of the referencing formula that is above the referenced cell.
- Nodes in the structure tree that were created by a duplication of a specific unit are resolved by a formula, referencing the result of the source formula.

This very strict mapping of the logical design to a geometrical design has the advantage that the logical design is not buried anyhow in the layout but still transparently present. However, for the spreadsheet users the predetermined layout might not conform to the way they want to have the results presented. Thus, Rajalingham et al. recommend to distribute a spreadsheet program over three different sheets, the first one consisting of only input cells, the second one of the spreadsheet program itself and the third one properly formatting the desired output.

In Example 4, a structure tree for the spreadsheet program that was introduced in Example 2 is developed. For the sake of the example, it is assumed that the spreadsheet writer also wants to calculate the overall profit or loss.

Example 4: Structured Spreadsheet Design

For the analysis only the *model* part of the spreadsheet has to be considered, as the *parameters* section consists only of user supplied values (see Figure 5.5(a) on page 80). Assuming the desired output to be *overall profit* and each year's *net profit*, the following considerations reveal the spreadsheet program's internal structure: the overall profit is calculated by summing the net profit of three years.

The net profit, again, can be split up into *gross profit*, *taxes* and a user supplied input. As a matter of fact, the calculation of tax debit references the gross profit twice: once to evaluate a condition, the second time only optionally depending on this condition. Hence the unit *gross profit* will be duplicated. Gross profit itself is split up into *revenue*, *variable cost* and two user inputs. Having a look at column `B` of Figure 5.5(b) on page 80, no further meaningful decomposition can be done.

[9]The first column already contains the labels

However, columns `C` and `D` are different, because the *sales(units)* entry is not user supplied. It results from a formula. Thus, the *sales(units)* part of column `C` has to be tripled, because it is used by the *revenue* and *variable cost* parts of column *C* and by the *sales(units)* part of column `D`. The *sales(units)* part of column `D` has to be doubled, as it is used for calculating revenues and variable costs.

The result of this considerations is the structure chart presented in Figure 5.9 on the following page. Although the structure of the three *net profit* nodes seems very similar, there is no iteration because of the differences in calculating each year's revenues and variable costs that are due to the running adaption of the sales per units figure. The spreadsheet program that is generated from the structure chart is shown in Figure 5.10 on page 95.

◊

The example gives insight into the main advantage of structured spreadsheet design: the layout reflects the underlying data flow. It can even be stated that the semantic relations between the distinct formulas and figures can be found out by examining the geometrical layout of the spreadsheet.

Nevertheless, the way this implementation methodology is designed pleases software engineers and programmers, but not end-users. Apart from the importance of the spreadsheet as modeling tool, the approach offers the following advantages:

- As intermediate results are not visible anymore, the spreadsheet program can be clearly separated into an afferent part, a processing, and an efferent part.
- By projecting the spreadsheet program into a more or less one-dimensional user interface, complexity is taken out of the representation. The remaining second dimension can be used to reflect the spreadsheet program's structure.
- Taking input cells out of the program makes at the same time clear which parameters are allowed to be adjusted. Spreadsheet users are prevented from overwriting formulas with constant values.
- The spreadsheet program itself is clearly separated from cells that are necessary for its instantiation. Thus, maintenance and testing the program is supported.

However, as every coin has two sides, the advantages bear also significant drawbacks:

- Intermediate results are not visible anymore. They are virtually blinded out on a separate spreadsheet. The visual presentation of intermediate results is often a very desirable feature of spreadsheet programs.

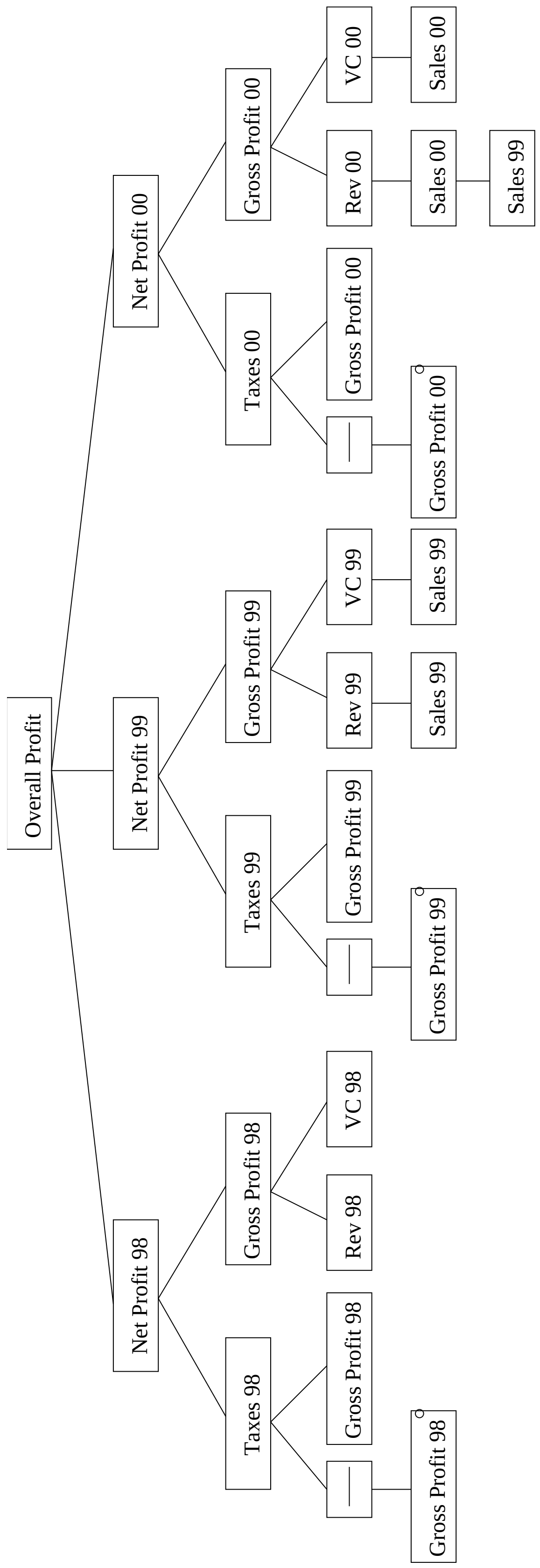
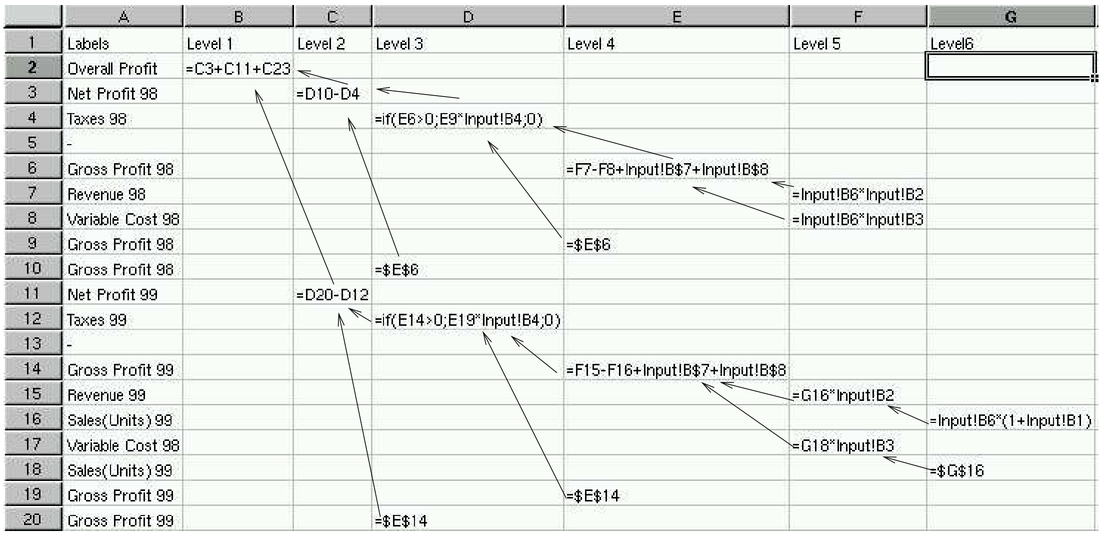


Figure 5.9: Structure chart of a spreadsheet program



	A	B	C	D	E	F	G
1	Labels	Level 1	Level 2	Level 3	Level 4	Level 5	Level6
2	Overall Profit	=C3+C11+C23					
3	Net Profit 98		=D10-D4				
4	Taxes 98			=if(E6>0;E9*Input!B4;0)			
5	-						
6	Gross Profit 98				=F7-F8+Input!B$7+Input!B$8		
7	Revenue 98					=Input!B6*Input!B2	
8	Variable Cost 98					=Input!B6*Input!B3	
9	Gross Profit 98				=$E$6		
10	Gross Profit 98			=$E$6			
11	Net Profit 99		=D20-D12				
12	Taxes 99			=if(E14>0;E19*Input!B4;0)			
13	-						
14	Gross Profit 99				=F15-F16+Input!B$7+Input!B$8		
15	Revenue 99					=G16*Input!B2	
16	Sales(Units) 99						=Input!B6*(1+Input!B1)
17	Variable Cost 98					=G18*Input!B3	
18	Sales(Units) 99						=$G$16
19	Gross Profit 99				=$E$14		
20	Gross Profit 99			=$E$14			

Figure 5.10: Fragment of the spreadsheet program resulting from Figure 5.9 on the preceding page. Input and dedicated output data is placed on different worksheets.

- By projecting the spreadsheet program into a one-dimensional user interface, spreadsheet development does no longer benefit from the ingenious user interface, i.e. the tabular grid that spreadsheet writers are familiar with.
- Moreover, the tabular grid is still there, but has a different semantic as it is not domain specific but implementation specific[10].
- The strict separation between input cells and formulas has certain advantages. However,
  - it tends to conceal the context of formulas (see Subsection 5.2.1).
  - surveys, e.g. by Janvrin and Morrison [JM00], have shown that spreadsheet programs that are distributed over more than one worksheet tend to be more error prone.

Additionally, spreadsheet writers have to deal with concepts as modules, iteration and alternatives that are usually not common in their application domain. For large business spreadsheets that have been reported to consist of nearly 17, 000 formula cells, this approach would yield spreadsheets consisting of 17, 000 rows and up to 10, 000 input cells (see Table 4.1 on page 60). It is questionable if the so generated spreadsheet program would be any clearer than the original ones.

Thus, although the methodology's name suggests so, structured spreadsheet engineering is not able to significantly reduce the complexity of size for a given

[10]In fact, [RCK02] denotes columns in the spreadsheet program as *virtual columns*.

spreadsheet program. In Example 4, the minimal differences in calculating the number of the sold units prevent the application of iteration. Thus, although all formulas for calculating the net profits are textually equal, except for the formulas for revenues and variable costs, they have to be copied and information about their sameness is consecutively lost.

It is argued that the sameness can still be detected from the similar row labels throughout the years. However, this is not really true, as revenues or variable costs have the same row label, but different formulas for each year. In this case, the maintenance and error correction in formulas is not supported.

5.4 Testing Methodologies

So far this chapter has dealt with techniques and methodologies to increase the spreadsheet quality by preventing the introduction of errors. Subsection 4.1.2 pointed out that an overall error rate of 2–5% has to be expected in software due to human errors. Testing software, and thus, testing spreadsheet programs, deals with the identification and correction of errors. Although a controlled development process, accurate design and documentation clearly increase the overall software quality, they are no substitute for testing.

Conventional software is usually tested by supplying test cases and comparing the system's actual output to the expected output, that is generated by a test oracle. However, as it is pointed out by Huang [Hua75] or Dijkstra [DDH72], the large set of possible test cases calls for a carefully selected sample of test cases in order to obtain statistically significant test results.

Currently, two approaches for spreadsheet testing are documented in literature, i.e. *spreadsheet testing using interval analysis* (see Ayalew [Aya01]) and the *what you see is what you test (WYSIWYT)-* approach (see Rothermel et al. [RLDB98, RCB$^+$00, RRB00]).

The latter approach requires the spreadsheet writer to supply a sufficient number of relevant test-cases for each formula. Furthermore, for each formula a *degree* of testedness is calculated. Therefore, spreadsheet writers have to state for each formula cell that it is considered tested. Additionally, the degree of testedness of formulas that the current cell depends upon is also taken into account.

The degree of testedness for each cell is shown on the spreadsheet UI by changing the cell's border color from red (untested) to blue (tested). Thus, the more blue the cell's border is shaded, the higher is its degree of testedness. Although this testing technique is tightly embedded in the spreadsheet system that the end-users are familiar with, it has significant drawbacks, as it depends on the selection of *good* or *relevant* test cases. End users usually do not have a mental model of the control flow in the formula, so they tend to select test cases that are relevant only for the application domain, and thus, the overall coverage[11] of the

[11]Coverage is discussed by Zhu et al. [ZHM97], in terms of *statement coverage*, *branch*

test cases will be too small. These findings correspond with the fact, that the empirical evaluation for the testing approach by Reichwein et al. [RRB00] has a sample group of professional programmers.

Interval testing, as an alternative approach, does not force the spreadsheet writer to supply test cases. For each cell, that is subject to testing, spreadsheet writers supply a range that they assume to be correct. Ayalew [Aya01, ACM00, MAC00] denotes this range as the *expected interval* E. This interval is meant to embrace many test cases of a conventional testing approach and thus, increase the overall coverage of the testing approach. For each formula a bounding interval E is calculated by applying the formula to the expected intervals of its input values with the rules of interval arithmetic.

Finally, some cross checks are performed to ensure that the expected interval E is contained within the interval of possible values B and that the cell's value is within E. A toolkit for interval testing is introduced in [Aya01]. Compared to the WYSIWYT approach interval testing is more user centered. Coverage is not in the users' responsibility but can be increased by executing the formulas as interval-formulas.

Of course, testing large spreadsheet programs with either approach is expensive. This fact is true for conventional software, too. Perry [Per95] states: *Too little testing is crime- too much testing is sin.* Obviously, the impact and the probability of errors has to be considered when deciding upon the right amount of testing. As shown in Table 2.2 on page 15, many spreadsheet writers are aware of the cost of errors, and thus, should be willing to take the extra effort of testing into account.

5.5 Visual Spreadsheet Auditing

In order to overcome the problems of testing spreadsheets, i.e. the lack of sufficient test data and the lack of testing skills, visual auditing has become a popular review method for spreadsheet programs. The rationale of spreadsheet visualization is to reduce the inherent complexity of spreadsheet programs to a magnitude that is easier to understand for the human auditor (see e.g. Nixon et al. [NO01]). Usually visual auditing tools color cells that share certain characteristics, i.e. similar formulas, a certain data type or spatial neighborhood, with the same color on the spreadsheet UI. Thus, the auditor does not have to check the spreadsheet on a cell-by-cell basis any more but can check larger units.

5.5.1 Commercial Visualization Toolkits

Commercially distributed spreadsheet visualization toolkits, such as *SpACE* [But00], the *Spreadsheet Detective* [Sof02b] or the *Operis Analysis Kit* [Sof02a], are avail-

coverage and *path coverage.*

able as Excel-Plug-ins. The basic functionality of these auditing tools is the same:

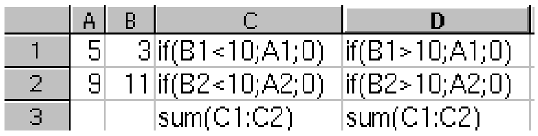
- The user selects a certain grouping criterion, e.g. data type, a unique formula, no dependents.
- All cells that correspond to the grouping criterion are shaded in a specific color.
- The spreadsheet auditor looks for inconsistencies.

Although the visualization is a worthwhile help in understanding and debugging spreadsheets, important issues are neglected. To use the spreadsheet UI to display the colored cells corresponds to the requirement of a maximal integration of the methodology into the spreadsheet system (see Section 3.4), but also imposes crucial limitations: the spreadsheet auditors' view on the spreadsheet is limited by the resolution of the display. Thus, they can check only a specific part of the spreadsheet program at a time. For large spreadsheets, this visualization often turns out to be inconvenient, as there are usually connections between cells that are spatially scattered.

Thus, a look at the spreadsheet program as a whole is not supported by these techniques. However, as Nixon et al. [NO01] or Davis [Dav96] pointed out, they offer a better support for the spreadsheet writers as the built-in auditing tools of a spreadsheet system. The built-in auditing tools can usually be used to visualize the data dependencies in a spreadsheet program on a cell-by-cell basis, and thus, are helpful for locally tracking bugs. *Excel* and *Gnumeric*, for instance, offer a basic functionality for showing the successors and the precedents of a given cell in the DDG.

5.5.2 S2 and S3 Visualization

Sajaniemi [Saj00] introduces two technique for the visualization of spreadsheet programs based on the sameness of the cell's formulas, the *S2* and *S3 Visualization*. Cells that are recognized as related are grouped into an area (see Subsection 2.3.5). In contrast to the previously described approaches, the grouping criteria are not exactly similar formulas and spatial neighborhood, but some evolution of the cells' formulas in the identified areas is allowed. The *S2* and *S3 Visualization* require that each area has an orientation, depending on the data flow in the area; either top-down, bottom-up, left-right or right-left. As the formulas in the areas have to be strictly similar, only minor modifications that are explained in the next paragraph, are allowed. Additionally, there is a unique orientation of the data flow in the area. An area can only grow in one dimension, i.e. a vertically-oriented area has a width of one cell and a horizontally-oriented area a height of one cell on the spreadsheet UI. Thus, there is also an ordering of the cells in the area, as in a top-down area the uppermost cell is the 1^{st} one, and so on.



	A	B	C	D
1	5	3	if(B1<10;A1;0)	if(B1>10;A1;0)
2	9	11	if(B2<10;A2;0)	if(B2>10;A2;0)
3			sum(C1:C2)	sum(C1:C2)

Figure 5.11: Example for calculating a guided-sum.

The expressions in cells in the same area do not have to be exactly equal, i.e. they are allowed to evolve. However, it is necessary that the formula in the n^{th} cell contains the expression in the $(n-1)^{th}$ cell either as suffix or as prefix. This feature is meant to support counting, e.g. in the first cell the expression $= 1$ is stored in order to initialize the counter, in the further cells the formula $= \text{c}ell\ before + 1$ automatically increases the counter.

In order to belong to the same area, the cells' formulas have to meet the required similarity criterion. Additionally, cells are required to be spatial neighbors on the spreadsheet UI. There are only minor exceptions to this rule, i.e. if two areas intersect, there is one cell that should belong to each of the areas. In this case, one of the areas is interrupted and the intersecting cell is added to the other area. However, this interference does not tear any areas apart.

This technique offers a better abstraction of the spreadsheet program as it will decompose the spreadsheet into abstract units, consisting of cells with equal formulas. Thus, the auditor can check the spreadsheet by analyzing the interactions and patterns of occurrence of these abstract units. Apparently, the effort for auditing is decreased compared to other visualization techniques that can only trace the occurrence of specific formulas, one at a time.

A drawback of this technique is that it considers only physical areas with a fixed homogeneity criterion. If such an area is separated by an inconsistency, this will be highlighted. However, the approach cannot distinguish between an inconsistency and a regular substructure, such as a sub-total. It will be broken up into unrelated pieces, and the connection between the subtotals and the overall sum will be difficult to identify since the subtotals are not necessarily arranged in a contiguous physical area.

The very strict homogeneity criterion can lead to further problems. Imagine a spreadsheet program where column A holds a value, column B holds another value that should control if A's value should be copied either into column C or into column D (see Figure 5.11). Although columns C and D appear to have a very similar formula, and they operate on the same data, the *S3-Visualization* that is introduced in [Saj00] will not recognize them as related.

5.5.3 Data Flow Based Visualization

Chan et al. [CC00, Cha01] introduce another interesting visualization and debugging approach. It is mainly data flow based and offers support for *local* and *global* debugging. However, both debugging strategies are again tied to the spreadsheet as visualization tool. Therefore, the user can only audit/debug a section of the spreadsheet that corresponds to the size of their screen at a time. In brief, it is assumed that the data flow in a spreadsheet program should correspond to a text: the data should flow from cells that are situated on the upper left corner of the spreadsheet UI to cells on the bottom-right corner of the spreadsheet UI. Data flow that does not correspond to this rule is considered dangerous and will thus be reported to the auditor by colorizing the concerned cells.

However, as the spreadsheet UI is the user interface of the auditing toolkit, the linkage between spatially widespread parts of the spreadsheet is still very hard to understand, and zooming can only be done by adjusting the display size.

Summary

The following issues have been addressed in this chapter:

- There are systematic approaches to spreadsheet development.
- Spreadsheet life cycle models are suitable for spreadsheet development by well-trained users.
- Techniques, such as logical spreadsheet design, relational spreadsheet modeling, and structured spreadsheet modeling introduce conceptual design into spreadsheet development.
- None of these technologies leads to a break-through, although development is simplified and error rates may be reduced.
- Systematic testing of spreadsheet programs suffers from problems: lack of test data, large and complex programs, localization of errors.
- Spreadsheet visualization reduces the complexity of spreadsheet programs and thus can improve the comprehension of spreadsheet programs.
- Visual spreadsheet auditing toolkits are already commercially distributed and widely used.
- Still there are some major drawbacks, as the visualization is tied to the spreadsheet UI and the criteria for grouping cells usually contain spatial constraints.

Chapter 6

Model Visualization

In the previous chapter spreadsheet visualization was characterized as a successful technique for reducing the overall complexity of spreadsheet programs that have to be tested and comprehended. The visualization techniques introduced so far have turned out to work sufficiently well for rather small spreadsheet programs, or for the decomposing of large spreadsheet programs into smaller parts. However, they do not scale up to very large spreadsheets.

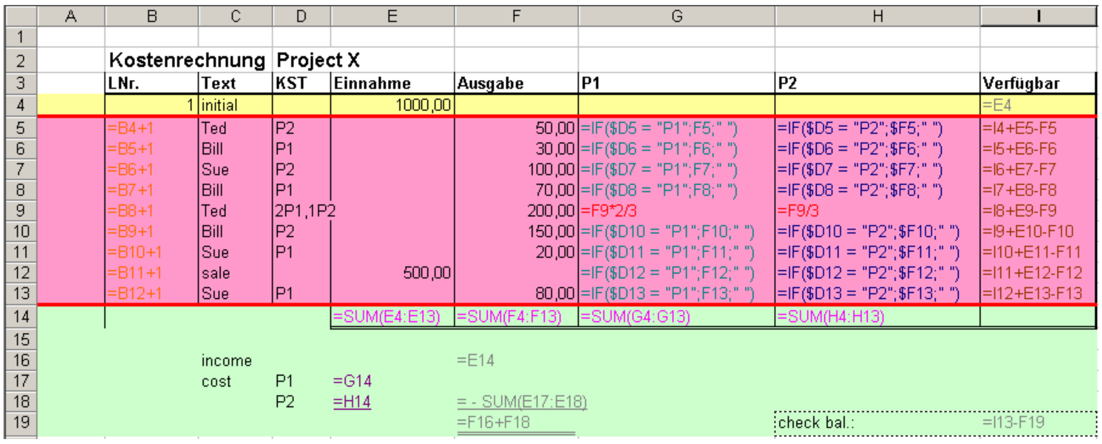
This chapter will present alternative approaches for the visualization of large spreadsheet programs. These approaches are more flexible as the techniques described in Section 4.3, as there are different grouping criteria for cells depending on the desired degree of similarity between cells in the same area or the data flow between cells. Furthermore, the visualization is not exclusively tied to the spreadsheet UI. Similar cells are aggregated into logical areas (see Section 6.2), semantic classes (see Section 6.3) or data modules (see Section 6.4) and the SRG (see Definition 14 on page 25) is offered as a comprehension aid.

6.1 Rationale

Spreadsheet programmers are assumed to have a conceptual model of the spreadsheet programs they create. This conceptual model usually determines how certain cells can be grouped into units. The fact that spreadsheet programmers are not forced to indicate these units complicates the reconstruction of the spreadsheet model.

Important tasks in reconstructing the spreadsheet model are to detect homogeneous areas in the spreadsheet [Saj00], to analyze the data flow in order to find regularities, and to find logical relationships between cells that are not physically adjacent. When the spreadsheet model is reconstructed, it must be presented to the auditor in a comprehensive way.

Considering the widespread application of spreadsheet programs (see Section 2.1) and the different kinds of spreadsheet programs in different application areas,



	A	B	C	D	E	F	G	H	I
1									
2		Kostenrechnung		Project X					
3		LNr.	Text	KST	Einnahme	Ausgabe	P1	P2	Verfügbar
4		1	initial		1000,00				=E4
5		=B4+1	Ted	P2		50,00	=IF($D5 = "P1";F5;" ")	=IF($D5 = "P2";$F5;" ")	=I4+E5-F5
6		=B5+1	Bill	P1		30,00	=IF($D6 = "P1";F6;" ")	=IF($D6 = "P2";$F6;" ")	=I5+E6-F6
7		=B6+1	Sue	P2		100,00	=IF($D7 = "P1";F7;" ")	=IF($D7 = "P2";$F7;" ")	=I6+E7-F7
8		=B7+1	Bill	P1		70,00	=IF($D8 = "P1";F8;" ")	=IF($D8 = "P2";$F8;" ")	=I7+E8-F8
9		=B8+1	Ted	2P1,1P2		200,00	=F9*2/3	=F9/3	=I8+E9-F9
10		=B9+1	Bill	P2		150,00	=IF($D10 = "P1";F10;" ")	=IF($D10 = "P2";$F10;" ")	=I9+E10-F10
11		=B10+1	Sue	P1		20,00	=IF($D11 = "P1";F11;" ")	=IF($D11 = "P2";$F11;" ")	=I10+E11-F11
12		=B11+1	sale		500,00		=IF($D12 = "P1";F12;" ")	=IF($D12 = "P2";$F12;" ")	=I11+E12-F12
13		=B12+1	Sue	P1		80,00	=IF($D13 = "P1";F13;" ")	=IF($D13 = "P2";$F13;" ")	=I12+E13-F13
14					=SUM(E4:E13)	=SUM(F4:F13)	=SUM(G4:G13)	=SUM(H4:H13)	
15									
16			income			=E14			
17			cost	P1	=G14				
18				P2	=H14	= - SUM(E17:E18)			
19						=F16+F18		check bal.:	=I13-F19

Figure 6.1: A block of a typical business spreadsheet

there is obviously not a only a single effective strategy to increase the overall quality of spreadsheet programs. However, field audits (see Section 4.3) pointed out that spreadsheet programs in business applications can become very large and usually consist of sets of *unique* formulas that are copied throughout the spreadsheet and then often slightly modified.

Very often, the structure of these spreadsheets consists of multiple blocks that start with certain cells that contain some initialization values and procedures, a large block of repeatedly used formulas, that might be slightly modified, and a final part where some overall figures and checksums are calculated (see Figure 6.1). Usually, the middle part tends to grow and become very large over time, whereas the top and bottom parts are stable in size.

Thus, it becomes important to reduce the number of formulas in the large middle-part (shaded in dark gray in Figure 6.1) that have to be tested and comprehended. Apparently, spreadsheet visualization reduces the testing effort, as only one instance of multiple copies of a formula have to be tested, whilst the remaining copies have only to be checked for whether they are placed at the right location in the spreadsheet UI. If copying was done properly, visual auditing tools identify and highlight spatial areas with homogeneous formulas by changing the color of the cells in the spatial area on the spreadsheet UI.

These approaches do not take into account that equal formulas are often not placed in a self-contained spatial area but repeated in a regular pattern throughout the whole spreadsheet program. Thus, with very large spreadsheets, visualization becomes an important issue. The limits that are imposed by the spreadsheet UI hinder the presentation of inter-cell relations that span over large geometrical distances on the spreadsheet. But even with smaller sheets, the internal complexity might warrant to display ”recoverable design information”, i.e. information that might be considered important for modularization in conventional software systems.

Hence, the approach presented in this section aims to support the spreadsheet programmer in revealing the initial conceptual model of a spreadsheet program by

1. reducing the overall size-complexity of the spreadsheet program,
2. identifying cells that users considers semantically related,
3. grouping them into units,
4. finding other units that are related by the spreadsheet programs data flow,
5. finding regular geometrical patterns of occurrences of cells in the same unit, and
6. presenting the identified units and patterns in a comprehensive way to the programmer.

The second point deals with recovering what the spreadsheet programmer considers to be related. One heuristic to identify relationships is to compare formulas. Most auditing toolkits available so far are capable of grouping adjacent cells with equal formulas into a unit. However, as it has been stated above, this grouping criterion is not sufficient. Spreadsheet programmers often copy formulas throughout the spreadsheet but slightly modify them afterwards.

In fact, formulas that are meant to fulfill the same task have to be considered similar. These formulas tend to be the result of a copy and paste operation, usually carried out by the built-in copy and paste functionality of the spreadsheet system, sometimes by simply retyping the same formula into different cells. Sometimes the formulas are slightly modified, in order to change constants or adjust relative references.

Alternatively, formulas are not copied into adjacent cells. The spreadsheet programmer may consider a block of geometrically adjacent cells to be a unit and, thus, copy the whole unit. Hence, there will not be a self-contained geometrical area of cells with similar formulas, but rather a regular pattern of re-occurrences of these cells. Thus, an efficient visual auditing technique

- has to recognize the relatedness of *similar* formulas, and
- must not take the geometrical distance between related cells into account, but
- has to figure out regular geometrical patterns of related cells and highlight disruptions of that pattern.

As it is stated by Mittermeir et al. [MC02], the core idea behind the definition of logical areas rests on the spreadsheet development process. Computations to be performed within a cell are specified by filling in a formula. It was stated above that if computations in other cells are to follow the same "logic", spreadsheet programmers will usually copy the formula over those cells. This copy operation usually extends over a physical (dense, rectangular) area. However, once done, this step in the development process is recorded nowhere. Thus, to economize typing spreadsheet programmers can copy a complex formula over a large area and than break this area by redefining some of the formulas that were in this physical area initially. Another spreadsheet writer might be less concerned about typing, and in consequence re-type the formulas in these physically unconnected parts of the sheet. The effect on the final sheet is identical.

Considering logical areas, the assumption is that having the same formula at different portions of the sheet is not something that happens by accident. It is rather due to a virtual *conceptual model* an application expert has about the problem to be solved. Hence, such patterns of regularity should be recovered. Distortions in such patterns of regularity indicate hot-spots that have to be subject to accurate testing.

However, it has to be considered, how "having the same formula" is to be understood in a spreadsheet context. Spreadsheet formulas do not use variables in the way variables are used in either mathematics or in conventional programming. Instead of $res := x + y$ with res, x and y being variables, one rather writes $\_ = a1 + b2$ with $a1$ and $b2$ being the addresses of cells containing the values supposed to be bound to the variables x and y and "_" is not given explicitly. It stands here for the address of the cell into which this expression is written, i.e. the cell where the computed value for res will eventually show. Thus, the "sameness" of a formula has to be seen in the different ways copy-operations or copy-and-modify-slightly-operations are to work.

Consecutively, three approaches to spreadsheet visualization are presented. The first two, namely logical areas and semantic classes, focus on the similarity of formulas and support auditors to group similar cells into more abstract units. Therefore, a logical area consists of similar cells, that are distributed over the spreadsheet. Semantic classes are an advancement of logical areas, as they identify similar areas on the spreadsheet. Thus, they are more scalable. Logical areas are discussed in Section 6.2, semantic classes in Section 6.3.

The third approach, i.e. data modules, do not focus on the similarity of formulas but figure out cells that are related by data flow. Broadly speaking, a data module is a set of cells that contribute to the same result of the spreadsheet. Data modules are discussed in detail in Section 6.4.

6.2 Logical Areas

In order to cover the relationship between formulas that have a similar functionality in the spreadsheet program, but are not physically related or not exactly equal, the concept of the logical area was introduced by Ayalew et al. [ACM00] and Mittermeir et al. [MCA00, MC02]. Briefly, a logical area is a set of cells with similar formulas. In contrast to all the other visualization techniques presented in the previous chapter, the positions of cells on the spreadsheet UI are not considered. As formulas can be slightly modified, there are different degrees of similarity that are introduced in the next section.

6.2.1 Terms and Concepts

A logical area is a set of formula cells with pairwise equivalent formulas. Subsequently, the suggested equivalence criteria are defined. Definitions 17 to 19 focus on the structure of formulas, i.e. the order of operators and operands. As these are properties of cells, i.e. of nodes in the data dependency graph, they are referred to as *node equivalence classes*. The notion of node equivalence is basically based on a static perception of the spreadsheet. Hence, in order to partition a spreadsheet program into logical areas that are based on a node equivalence class, structural patterns in the formulas are looked for.

Likewise, one might assume a dynamic perception. This dynamic perception will only examine the data flow between different cells. As the data flow is reflected in the edges in the *DDG*, this class of equivalence criteria is called *link equivalence classes*. It contains source, sink and aggregation equivalence, as presented in definitions 20 to 23. Again, it can be stated that there is a certain pattern that is looked for, but this time it is not a pattern which is directly in the formula, but in the data dependency graph.

It is assumed, as it is true for patterns in general (see e.g. Coplien [Cop00] or Appleton [App97]), the re-occurrence of a pattern is used to solve the same or at least a related problem. Hence, finding formulas that conform to a specific pattern one can assume that these formulas are somhow related.

Additionally, there are two special equivalence criteria that are needed in order to deal with empty cells and computationally dead cells, i.e. cells that do not partake in the spreadsheet programs calculation, presented by definitions 24 and 25.

Node Equivalence Classes

Definition 17: Copy Equivalence
Cells c_1 and c_2 are copy equivalent ($ce(c_1, c_2) = true$) if their formulas are identical. Relative references are compared in the `R1C1`-notation (see Example 1). □

As defined in [Saj00], two cells are copy-equivalent, if they have the same format, and their formulas are equal. This is the strongest form of equivalence. A common concept behind copy equivalent cells is assumed, irrespective of whether

they are topologically adjacent or not.
Example 5: Copy Equivalence
Let the formula $\_ = A1 + A2$ be in the cell $A3$ and the formula $\_ = B1 + B2$ in the cell $B3$. In the `R1C1` notation both formulas can be written as $\_ = \texttt{R0C} - 2 + \texttt{R0C} - 1$, and thus they are copy equivalent. $\Diamond$

Definition 18: Logical Equivalence
Cells c_1 and c_2 are logically equivalent ($le(c_1, c_2) = true$) if their formulas differ only in constant values and absolute references. $\Box$

This relaxation allows for several patterns that are generally repeated, but exhibit different fixed parameters. Categorizations of input values might be as good an example as computations of sales tax with different tax rates.
Example 6: Logical Equivalence
If $\_ = A1 + \$A\$2$ is placed in the cell $A3$ and $\_ = B1 + \$F\$4$ is the formula in cell $B3$, the `R1C1` notation is $\_ = \texttt{R0C} - 2 + \texttt{R\$1C\$2}$ in $A3$ and $\_ = \texttt{R0C} - 2 + \texttt{R\$4C\$6}$ in $B3$. Because the relative references are still equal in the `R1C1` notation, $A3$ and $B3$ are considered logically equivalent. $\Diamond$

Definition 19: Structural Equivalence
Cells c_1 and c_2 are structurally equivalent ($se(c_1, c_2) = true$) if their formulas contain the same operations in the same order. $\Box$

Two formulas which are structurally equivalent can have different constant values as well as different absolute and relative references. Nevertheless, the same operators and functions are applied in the same order to different data. This is a direct extension of the concept of logical equivalence. In general, structural equivalence can be seen as proxy for macro insertion in low level procedural programming languages. In more elaborate spreadsheet programs it might even amount to something close to a subroutine concept.
Example 7: Structural Equivalence
The formulas $\_ = A1 + B4 - 5$ and $\_ = B7 + B2 - B3$ are structurally equivalent. $\Diamond$

Link Equivalence Classes

Definition 20: Source Equivalence
Cells c_1 and c_2 are source equivalent ($src(c_1, c_2) = true$), if all their relative references have equal coordinates. Absolute cell references are not considered. $\Box$

Here, the concrete operation used in the respective formulas is immaterial as long as the formulas have the same arity and the relative distance to the arguments of the operation remains stable. It is important to note, that logically equivalent nodes would satisfy this criterion, but not vice-versa. Here, the focus is on links.

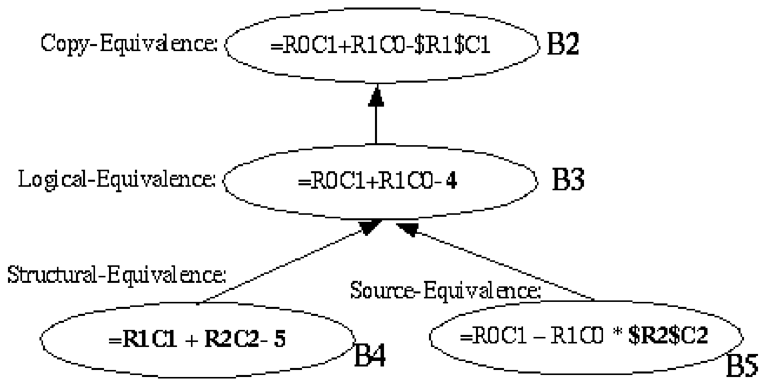


Figure 6.2: Formulas that are copy-, logical, structural and source equivalent to the formula R0C1 + R1C0 − $R1$C1. The cell addresses on the right hand side of each entry correspond to cell addresses in Example 8.

Hence, the concrete operations might be different. Definitions 18 to 20 obviously loosen the strict equivalence criterion of copy equivalence, as copy equivalence cells are also logical, structural and source equivalent. Thus, there is a partial order within the equivalence criteria (see Figure 6.4 on page 110).

By means of this partial order, the difference between node and link equivalence classes can be demonstrated by the following example (see also Figure 6.2).

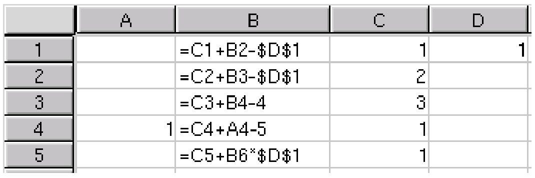
Example 8: Node vs. Link Equivalence

Having the formula R0C1 + R1C0 − $R0$C4 in cells B1 and B2, the formula R0C1 + R1C0 − 4 in cell B3, the formula R0C1 + R1C − 1 − 5 in B4, the formula R0C1 + R0C1 ∗ $R0$C4 in B5, the following equivalence criteria will hold:

- $Eq_{ce}(\texttt{B1}) = \{\texttt{B1}, \texttt{B2}\}$
- $Eq_{le}(\texttt{B1}) = \{\texttt{B1}, \texttt{B2}, \texttt{B3}\}$
- $Eq_{se}(\texttt{B1}) = \{\texttt{B1}, \texttt{B2}, \texttt{B3}, \texttt{B4}\}$
- $Eq_{src}(\texttt{B1}) = \{\texttt{B1}, \texttt{B2}, \texttt{B3}, \texttt{B5}\}$

The formula in B5 is source equivalent to B1, because the relative cell references have the same coordinates. Thus, source equivalence can be determined by analyzing only the edges that point to a node in the *DDG*. In contrast, B4 references totally different cells. However, it is structural equivalent to B1, as it contains the same operators in the same order. B3 is logical equivalent, as the only difference to B1 is the replacement of an absolute cell reference by a constant value. ◊

In order to define sink equivalence, the function $reldist : Cells \times Cells \rightarrow \mathbb{Z} \times \mathbb{Z}$ that is explained in Section 6.3 on page 122, is used. In brief, *reldist* computes the relative distance between two cells.



	A	B	C	D
1		=C1+B2-$D$1	1	1
2		=C2+B3-$D$1	2	
3		=C3+B4-4	3	
4	1	=C4+A4-5	1	
5		=C5+B6*$D$1	1	

Figure 6.3: The formulas from Example 8 in the `A1` notation.

Definition 21: Sink Equivalence
Cells c_1 and c_2 are sink equivalent($sink(c_1, c_2) = true$), if there exist two cells c_3 and c_4, with $src(c_3, c_4) = true \wedge \textsf{relatively references}(c_3, c_1) \wedge \textsf{relatively references}(c_4, c_2) \wedge c_3 \neq c_4 \wedge reldist(c_3, c_1) = reldist(c_4, c_2)$. $\square$

Sink equivalence is the mirror image of source equivalence. Again, the focus is on the link and the cells related by sink equivalence need not have anything in common on their own right. Their communality stems from the fact that they are used by source equivalent cells. Hence, from the users' perspective they are related since their values are used by other cells in a related way.

Definition 22: Aggregation Equivalence
Cells c_1 and c_2 are aggregation equivalent ($agg(c_1, c_2) = true$), if there exists a cell c_3 that references c_1 and c_2 in an aggregation operation. $\square$

Aggregation equivalence is a variation of the idea expressed in sink equivalence in so far as it groups cells according to their usage pattern. However, the common usage is not stereotypically spread out in the tabular plane but aggregated to an individual cell. The physical area(s) over which summation operations range are typical examples of aggregation equivalent cells.

For further considerations, a third category of relationships between nodes is defined.

Definition 23: Data Equivalence
Cells c_1 and c_2 are data equivalent ($de(c_1, c_2) = true$), if $\forall c_i : \textsf{Cell} | \textsf{references}(c_1, c_i) \leftrightarrow \textsf{references}(c_2, c_i)$. $\square$

Two formulas are data equivalent if they reference identical cells. This is the converse of structural equivalence. Not the same macro or subroutine is executed with different data, but the same data is treated by different subroutines. What-If analysis might serve as typical pattern for this case. The general pattern will be: compute an intermediate result and use it in various computations.

Definition 24: Empty Cells
Cell c_1 is in the equivalence class empty cells ($ec(c_1) = true$, if it neither contains a formula nor a value. $\square$

Definition 25: Computationally Dead Cells
Cell c_1 is in the equivalence class computationally dead cells ($cd(c_1) = true$), if it is neither referenced by any other cell nor contains references to other cells. $\square$

Computationally dead cells do not partake in the calculations of the spreadsheet. They are generally used as descriptive labels for values computed in neighboring cells or cells in the column below or in the respective row. For the definition of empty cells or dead cells, singletons suffice. However, in real sheets this equivalence class is quite populated.

Computationally dead cells do not necessarily have to be empty cells, e.g. cells containing labels or cells containing some input data that is not referenced because of errors in other formulas. An empty cell, in contrast, does not have to be computationally dead. It has been shown by field audits of spreadsheet programs (see [CHM02]), that referencing blank cells is a common error in real-world spreadsheet programs[1]. However, empty cells that are not computationally dead have to be considered a symptom of an error, whereas computationally dead cells that are not empty are not that suspicious.

In the context of equivalence classes the equivalence class generator is defined as a function:

Definition 26: Equivalence Class Generator
The equivalence class generator is a function $Eq_{ind}(c) = \{c_j | ind(c, c_j)\}$, with $ind \in \{ce, le, se, src, sink, agg, de\}$. It returns the set of cells satisfying the specified binary equivalence relation with cell c. □

As it can be seen by Definition 26, the equivalence class generator returns a set of cells that are in the same logical area with a given cell and with respect to a certain equivalence criterion. The equivalence class generator is an important aid for to define semantic units in the subsequent section.

6.2.2 Examples for Logical Areas

In this section, the relationships between the different equivalence criteria are explained by means of examples.

Example 9: Node Equivalence Classes
In Figure 6.5 on the following page $Eq_{ce}(\texttt{B1}) = \{\texttt{B1}, \texttt{B2}\}$ and $Eq_{ce}(\texttt{D1}) = \{\texttt{D1}, \texttt{D2}\}$. $Eq_{le}(\texttt{B1}) = \{\texttt{B1}, \texttt{B2}, \texttt{D1}, \texttt{D2}\}$, because `D1` and `D2` absolutely reference `D4`, whereas `B1` and `B2` absolutely reference `B4`. The difference in the relative references, `E1` and `E2` instead of `C1` and `C2`, would not violate copy equivalence, as the `R1C1`-notation of the address of the referenced cells is `R0C1` in all cases.

At last, $Eq_{se}(\texttt{B1}) = \{\texttt{B1}, \texttt{B2}, \texttt{C1}, \texttt{C2}, \texttt{D1}, \texttt{D2}\}$, as logically equivalent cells are also structurally equivalent. ◊

[1]Indeed, referencing blank cells is often not considered an error, as formulas that reference blank cells often do not show any results. For a more detailed discussion of the problem, see [ACM00].

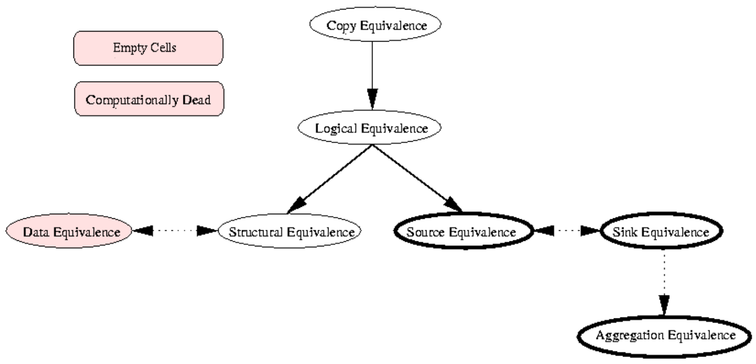
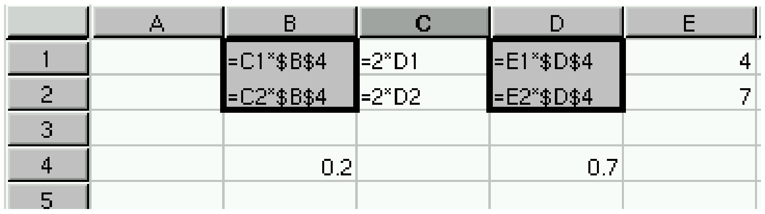


Figure 6.4: Partial order of equivalence criteria. Solid edges denote a relation due to the partial order, dotted edges represent a conceptual relation.



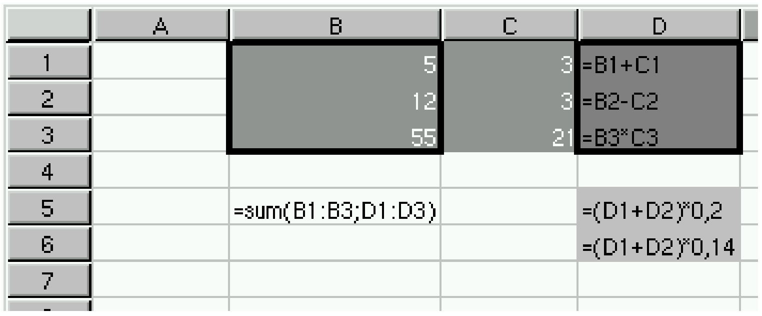
	A	B	C	D	E
1		=C1*$B$4	=2*D1	=E1*$D$4	4
2		=C2*$B$4	=2*D2	=E2*$D$4	7
3					
4		0.2		0.7	
5					

Figure 6.5: The thick borders (B1 and B2, and D1 and D2) indicate two logical areas based on copy equivalence. B1 and B2 are not copy equivalent to D1 or D2, because of the absolute cell reference to B4 or D4. Nevertheless, the grey shaded area (B1, B2, D1 and D2) is a logical area of logical equivalent cells. Cells B1, B2, C1, C2, D1 and D2 are structural equivalent.

Example 10: Link equivalence classes
Figure 6.6 on the next page illustrates link equivalence classes. Although there is no node equivalence between any cells (except for structural equivalence between D5 and D6), link equivalence classes can be used to create an abstraction of the spreadsheet program. The sum-formula in B5 aggregates the cells B1, B2, B3, D1, D2 and D3. Thus, $Eq_{agg}(\texttt{B1}) = \{\texttt{B1},\texttt{B2},\texttt{B3},\texttt{D1},\texttt{D2},\texttt{D3}\}$.

The cells D5 and D6 are data equivalent, as they reference exactly the same cells. It is important to note, that the relative coordinates of the references differ: in the R1C1-notation, the formula in D5 is $(R-4C0+R-3C0)*0.2$, whereas D6's formula is $(R-5C0+R-4C0)*0.14$. However, it is the same cells that are the target of the reference, and thus $Eq_{de}(\texttt{D5}) = \{\texttt{D5},\texttt{D6}\}$.

In contrast to data equivalence, source equivalence requires cells to have equal relative references, not equal targets for the references. The formulas in the cells D1, D2 and D3 relatively reference $R0C-2$ and $R0C-1$. As none of the cells' formulas contains any other relative reference, $Eq_{src}(\texttt{D1}) = \{\texttt{D1},\texttt{D2},\texttt{D3}\}$. The order of the relative references in the formula is taken into account neiter by



	A	B	C	D
1		5	3	=B1+C1
2		12	3	=B2-C2
3		55	21	=B3*C3
4				
5		=sum(B1:B3;D1:D3)		=(D1+D2)*0,2
6				=(D1+D2)*0,14
7				

Figure 6.6: The areas that are surrounded by a thick border are taken together a logical area of aggregation equivalent cells that are aggregated by the sum-formula in `B5`. The light-grey shaded cells `D5` and `D6` are data-equivalent. The dark-grey shaded cells `D1, D2` and `D3` are source-equivalent. As they reference the cells with white text-color, the latter are sink-equivalent.

source equivalence, nor by data equivalence.

Whenever a source equivalence class can be identified, a logical area of sink equivalent cells can be identified, too. In the example, `D1, D2` and `D3` relatively reference `B1, B2, B3, C1, C2` and `C3`. $Eq_{sink}(\texttt{B1}) = \{\texttt{B2}, \texttt{B3}, \texttt{C2}, \texttt{C3}\}$, because these cells are referenced by `D2` and `D3`, i.e., cells that are source equivalent with `D1`, the cell that references `B1`. ♢

In both examples stated above, logical areas are often also spatial areas on the spreadsheet UI. However, this does not have to be the case, as it can be seen for $Eq_{le}(\texttt{B1})$ in Figure 6.5 on the facing page or for $Eq_{agg}(\texttt{B1})$ in Figure 6.6.

6.2.3 Auditing Strategies for Logical Areas

The abstraction of the spreadsheet program that is generated by logical areas can be used in different ways to understand and debug a given spreadsheet program. The spreadsheet auditor can either focus on a data driven view of the spreadsheet (i.e., by examining the link equivalence classes) or rely on the static structure of the formulas. As it can be seen in Figure 6.4 on the facing page, the partial order of equivalence criteria is more complete for the node equivalence classes. Hence, there is a higher potential for generating hierarchic representations of the spreadsheet program with node equivalence classes.

Thus, the auditor can start to inspect a spreadsheet program on the most abstract level of structural equivalence classes. If a certain structural equivalence class attracts the interest of the auditors, they can examine the contents by having a look at its members that can be either cells, logical equivalence classes or copy equivalence classes.

The auditing itself can be

- pattern driven,

- SRG_{LA} driven and
- structure driven.

Usually, a combination of all three approaches will be employed for auditing a real word spreadsheet program. In the next part, each of the approaches will be explained based on an example that was first published by Panko [Pan97]. The example spreadsheet (see Figure 6.7 on the next page and Figure 6.8 on page 114) can be used by students to do some annual accounting tasks and consists of 108 non-empty cells, including 57 cells with formulas. However, the spreadsheet contains a couple of seeded errors. Thus, auditing this spreadsheet is an acknowledged measure for the efficiency of a spreadsheet auditing approach.

Although the spreadsheet program looks correct at first sight, the following errors are hidden:

E19: + should be -

D20: 20 should be `B20`.

F5: Cells in sum should be `F14` – `F20`

B17: should be $12 * 30$

F8: should reference `F5` instead of `E5`.

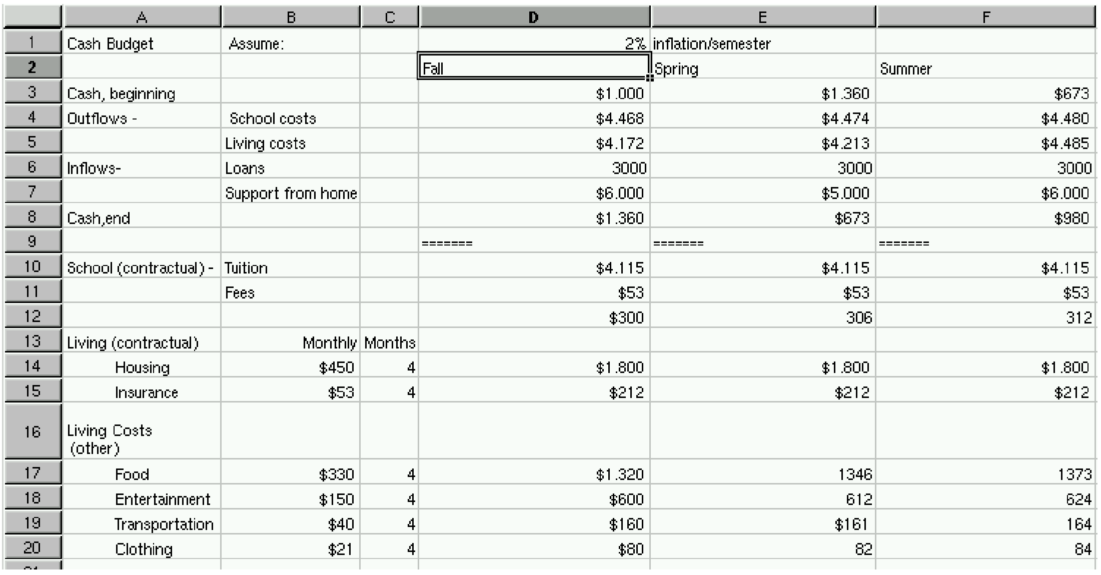
There are further omission errors, e.g. *parental gifts* and *parking expenses*, both being relevant factors for a students budget, which are not taken into account. However, these errors are not obvious on the value level, and a written specification of the spreadsheet's task has to be available in order to find them.

Pattern Driven Auditing Strategy

For the pattern driven auditing strategy the following steps are performed:

1. Identify logical areas,
2. map logical areas back to the spreadsheet UI,
3. find geometrical patterns for the members of each logical areas, and
4. take irregularities in the pattern as hint for a hot-spot in the spreadsheet program, i.e. a place that needs to be accurately tested or changed.

Coloring cells in the same logical area with the same background color will result in Figure 6.9 on page 115. As the number of available grey-scales is limited, not all logical areas are colored. In fact, the spreadsheet auditor will not color all the logical areas at the same time, but focus on one at a time.



	A	B	C	D	E	F
1	Cash Budget	Assume:		2%	inflation/semester	
2				Fall	Spring	Summer
3	Cash, beginning			$1.000	$1.360	$673
4	Outflows -	School costs		$4.468	$4.474	$4.480
5		Living costs		$4.172	$4.213	$4.485
6	Inflows-	Loans		3000	3000	3000
7		Support from home		$6.000	$5.000	$6.000
8	Cash,end			$1.360	$673	$980
9				=======	=======	=======
10	School (contractual) -	Tuition		$4.115	$4.115	$4.115
11		Fees		$53	$53	$53
12				$300	306	312
13	Living (contractual)	Monthly	Months			
14	Housing	$450	4	$1.800	$1.800	$1.800
15	Insurance	$53	4	$212	$212	$212
16	Living Costs (other)					
17	Food	$330	4	$1.320	1346	1373
18	Entertainment	$150	4	$600	612	624
19	Transportation	$40	4	$160	$161	164
20	Clothing	$21	4	$80	82	84

Figure 6.7: Example spreadsheet for evaluation of auditing approach, value level.

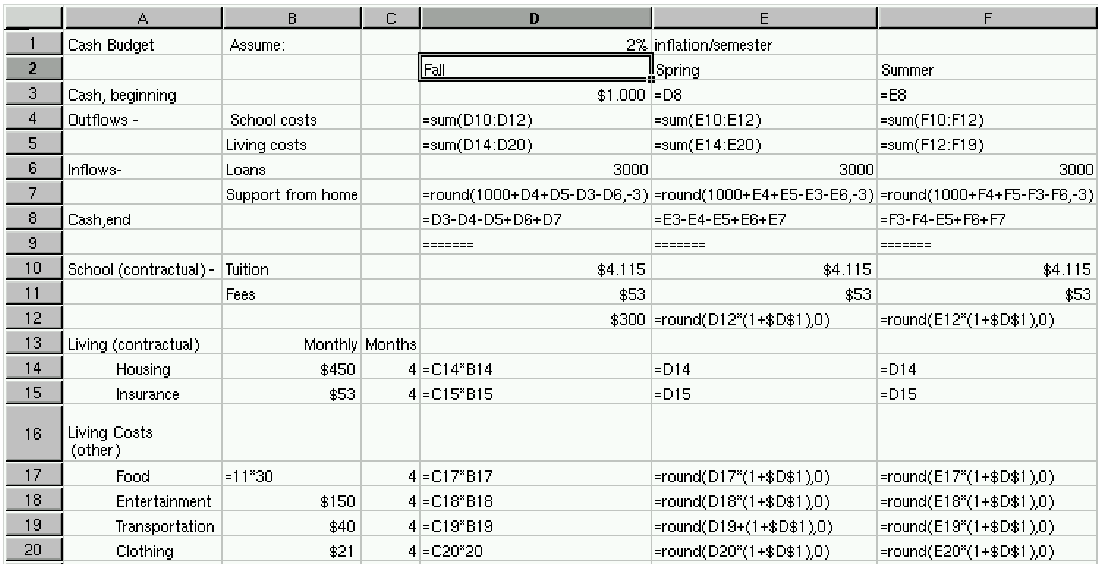
Nevertheless, it is obvious that D20 and E19 break an otherwise regular pattern of cells. The same is true for F5 and F8, as the other cells in column F in the upper half of the spreadsheet program are copy-equivalent with cells in column E. Thus, 4 of 5 areas can easily be spotted as irregularities in the geometrical pattern of cells in logical areas.

The formula in B17 only occurs once, and thus is isolated in a logical area. In this case, the pattern driven auditing strategy is not helpful.

Another drawback of this strategy is that the spreadsheet auditor has to see all the cells in a logical area. This is not possible for large spreadsheets, where not all of the cells in a logical area fit on the screen. In this case, the auditor might not be aware of irregularities in the pattern. If the spatial distance between cells in a logical area is that large, that only one of the cells can be shown on the screen at a time, the auditor will not be able to recognize any pattern.

SRG_{LA} Driven Auditing Strategy

A *SRG* driven auditing strategy banks on inspecting the spreadsheet programs SRG_{LA}, i.e., an abstraction of the data dependency graph of the spreadsheet program, with logical areas representing nodes (see Definition 14 on page 25 for a formal definition of the *SRG*). This auditing strategy is based on the assumption that the spreadsheet program is a kind of data flow program. As a data flow program can be inspected by inspecting the data flow graph, the same is possible for a spreadsheet program, too. However, spreadsheet programs tend to become very large, and thus the data flow graph might be too complex to be comprehended by a human.



	A	B	C	D	E	F
1	Cash Budget	Assume:		2%	inflation/semester	
2				Fall	Spring	Summer
3	Cash, beginning			$1.000	=D8	=E8
4	Outflows -	School costs		=sum(D10:D12)	=sum(E10:E12)	=sum(F10:F12)
5		Living costs		=sum(D14:D20)	=sum(E14:E20)	=sum(F12:F19)
6	Inflows-	Loans		3000	3000	3000
7		Support from home		=round(1000+D4+D5-D3-D6,-3)	=round(1000+E4+E5-E3-E6,-3)	=round(1000+F4+F5-F3-F6,-3)
8	Cash,end			=D3-D4-D5+D6+D7	=E3-E4-E5+E6+E7	=F3-F4-E5+F6+F7
9				=======	=======	=======
10	School (contractual) -	Tuition		$4.115	$4.115	$4.115
11		Fees		$53	$53	$53
12				$300	=round(D12*(1+$D$1),0)	=round(E12*(1+$D$1),0)
13	Living (contractual)	Monthly	Months			
14	Housing	$450	4	=C14*B14	=D14	=D14
15	Insurance	$53	4	=C15*B15	=D15	=D15
16	Living Costs (other)					
17	Food	=11*30	4	=C17*B17	=round(D17*(1+$D$1),0)	=round(E17*(1+$D$1),0)
18	Entertainment	$150	4	=C18*B18	=round(D18*(1+$D$1),0)	=round(E18*(1+$D$1),0)
19	Transportation	$40	4	=C19*B19	=round(D19+(1+$D$1),0)	=round(E19*(1+$D$1),0)
20	Clothing	$21	4	=C20*20	=round(D20*(1+$D$1),0)	=round(E20*(1+$D$1),0)

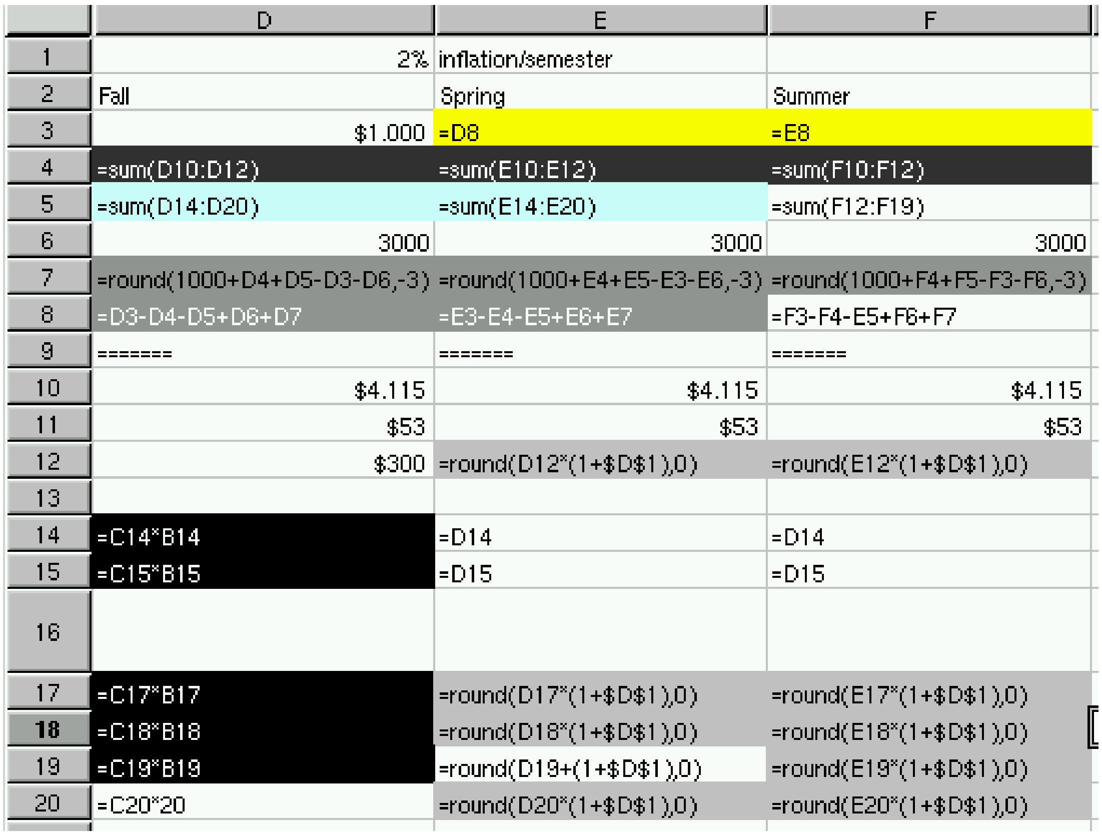
Figure 6.8: Example spreadsheet for evaluation of auditing approach, formula level.

In order to reduce the complexity of the data flow graph (the DDG), cells in the same logical area are collapsed into a single node. Thus, the spreadsheet auditor has to deal with a reduced number of nodes and edges in the SRG_{LA}, and thus, incorrect nodes and edges will attract the attention of the auditor much faster.

In this auditing strategy selecting an appropriate level of abstraction is an important issue. In the example (see Figure 6.10 on page 116 and Figure 6.12 on page 119) there is an SRG_{LA} with logical areas that are based on structural equivalence classes and another one based on copy equivalence classes. Whilst the first one is inconspicous, the second one reflects all errors that were built into the spreadsheet program.

SRG_{LA} driven auditing does not bank on the spreadsheet UI for the visualization of the program. Thus, the spreadsheet auditors' capabilities are not limited any more by the size of the screen and auditing can be performed in the context of the whole spreadsheet program.

However, the SRG_{LA} for a large spreadsheet program still tends to become complex, e.g. in [CHM02] SRG_{LA}s with more than 30 nodes and up to 100 edges were inspected. Although this representation is much less complex than the DDG and can be handled by a professional auditor, it is too complex to be comprehended by an end-user.



	D	E	F
1	2%	inflation/semester	
2	Fall	Spring	Summer
3	$1.000	=D8	=E8
4	=sum(D10:D12)	=sum(E10:E12)	=sum(F10:F12)
5	=sum(D14:D20)	=sum(E14:E20)	=sum(F12:F19)
6	3000	3000	3000
7	=round(1000+D4+D5-D3-D6,-3)	=round(1000+E4+E5-E3-E6,-3)	=round(1000+F4+F5-F3-F6,-3)
8	=D3-D4-D5+D6+D7	=E3-E4-E5+E6+E7	=F3-F4-E5+F6+F7
9	=======	=======	=======
10	$4.115	$4.115	$4.115
11	$53	$53	$53
12	$300	=round(D12*(1+$D$1),0)	=round(E12*(1+$D$1),0)
13			
14	=C14*B14	=D14	=D14
15	=C15*B15	=D15	=D15
16			
17	=C17*B17	=round(D17*(1+$D$1),0)	=round(E17*(1+$D$1),0)
18	=C18*B18	=round(D18*(1+$D$1),0)	=round(E18*(1+$D$1),0)
19	=C19*B19	=round(D19+(1+$D$1),0)	=round(E19*(1+$D$1),0)
20	=C20*20	=round(D20*(1+$D$1),0)	=round(E20*(1+$D$1),0)

Figure 6.9: Example spreadsheet with colorized logical areas

Structure Driven Auditing Strategy

Finally, spreadsheet auditing can be performed based on the static structure of the formulas. This auditing strategy exploits the hierarchy of equivalence criteria that has been introduced in Figure 6.4 on page 110. It is assumed that certain errors occur because of minor modifications in the targets of relative references. Thus, cells that were copy or logical equivalent before, become structural equivalent due to an error.

As there is a hierarchy between the equivalence criteria, a logical area based on structural equivalence could be further decomposed into logical areas based on logical equivalence or copy equivalence, or into unique cells. Thus, logical areas can be organized as a tree, starting with structural equivalent cells that are further grouped into logical equivalent cells on the next level. The logical equivalent cells can then be grouped into copy equivalent cells. Logical areas of copy equivalent cells can only be decomposed into unique cells.

Of course, a structural equivalence class can contain only one logical equivalence class that contains only one copy equivalence class. In the latter case, it is assumed that the structural equivalence class cannot be further decomposed. The

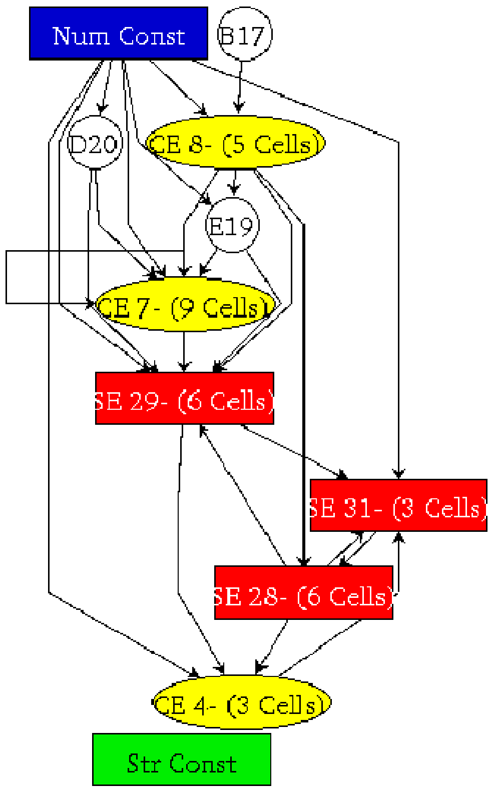


Figure 6.10: SRG_{LA} of the example spreadsheet. Structural equivalent cells are represented by a single node. The examination of the SRG together with the information given in the structure browser (see Figure 6.11 on the facing page) will not reveal any irregularities.

same is true for logical equivalence classes that contain only one copy equivalence class.

For structure driven auditing, the auditors examine the tree of logical areas. If a structural equivalence class contains a high number of logical or copy equivalent cells with a few outliers, these outliers have to be carefully examined. The same is true for logical areas based on logical equivalence. If they consist of logical areas and a few unique cells, these unique cells might be outliers because of some modifications that erroneously occurred.

The assumption is that spreadsheets usually have a regular structure, and thus, a formula is usually copied into several places. Only a limited number of formulas, e.g. check-sums, will not be copied throughout the spreadsheet program. The same is true for modifications of formulas that take place after they being copied. In both cases, there will be no copy equivalent (and maybe, also no structural equivalent) cells. Thus, the cells modified in this way will be outliers on a high level of the tree of logical areas.

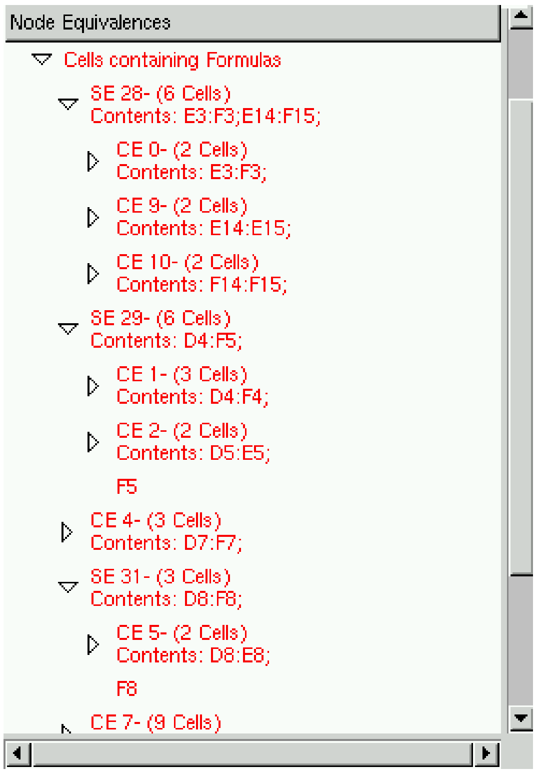

Node Equivalences
Cells containing Formulas
SE 28- (6 Cells)
Contents: E3:F3;E14:F15;
CE 0- (2 Cells)
Contents: E3:F3;
CE 9- (2 Cells)
Contents: E14:E15;
CE 10- (2 Cells)
Contents: F14:F15;
SE 29- (6 Cells)
Contents: D4:F5;
CE 1- (3 Cells)
Contents: D4:F4;
CE 2- (2 Cells)
Contents: D5:E5;
F5
CE 4- (3 Cells)
Contents: D7:F7;
SE 31- (3 Cells)
Contents: D8:F8;
CE 5- (2 Cells)
Contents: D8:E8;
F8
CE 7- (9 Cells)

Figure 6.11: Hierarchy of logical areas in the example spreadsheet

In Figure 6.11 the hierarchy of logical areas is shown for the example spreadsheet (see Figure 6.7 on page 113). Logical areas with only a single member are skipped in the tree. Logical areas based on structural equivalence are labeled with *SE* followed by a unique number, copy equivalent cells are labeled with *CE* followed by a unique number. *SE 28* consists of 6 cells that can be assigned to three different copy equivalence classes. However, there are no outliers in *SE 28*. In contrast, *SE 29* contains two logical areas and one unique cell, i.e. `F5`. Indeed, `F5` is one of the erroneous cells in the example spreadsheet. The same is true for `F8` in *SE 31*.

`E19` and `D20`, the other erroneous cells, are not structural equivalent with any other in the spreadsheet program. As they are located immediately beneath the root of the tree of equivalent cells, they will also attract the attention of the auditor.

6.2.4 Discussion

Obviously, logical areas are a promising approach to support spreadsheet users in auditing and understanding spreadsheet programs. Each of the auditing strategies presented was tested and found hot-spots that the authors of the example spreadsheets have deliberately introduced. In practice, spreadsheet programs are, of course, less clear to the auditors. Hence, they will have to combine the three auditing strategies to obtain reasonable results. However, the example spreadsheet has also certain properties that favor logical areas:

1. It is highly regular: there were only 6 logical areas, and 3 single cells.
2. The whole spreadsheet fits the screen.
3. There is no check-summing and data analysis part.

Because of the small number of logical areas, an advanced auditor was able to check the spreadsheet and find all the errors in less than 5 minutes. As the whole spreadsheet fits on the screen, the search for irregularities in the geometrical pattern of cells in the same logical area is supported. Checksum and data analysis parts of a spreadsheet program usually contain complex formulas that only occur one or two times. Thus, they tend to increase the number of logical areas with only little gain for the spreadsheet auditor.

Additionally, the geometrical extension of logical areas is opposed to the conceptual model of the spreadsheet user. When the user builds the spreadsheet program by copying the contents of a row into the next one, or by copying a column into the next one, as in the example, the logical areas will extend row-wise (if columns are copied) or column-wise (if rows are copied). Thus, in a small example with a limited number of logical areas, the auditor can reconstruct the conceptual model by means of logical areas. However, if there are many logical areas, or if the conceptual units of the spreadsheet programmer cannot be displayed on a single screen any more, this reconstruction is hindered.

Hence, a higher level of abstraction that corresponds to the users' conceptual model is called for. In Section 6.3 the concept of semantic classes is introduced. Semantic classes are based on logical areas but offer a higher level of abstraction, and the resulting model of the spreadsheet program consists of entities with a geometrical extension that corresponds to a coarser perspective on the users conceptual model.

Semantic classes fit large spreadsheets. However, as they offer an abstraction of logical areas, good results will require an analyzed spreadsheet program with a regular structure. Spreadsheets with many different formulas and no regular usage of similar formulas call for an other approach, that does not examine formulas in order to generate abstract units but is based on the spreadsheet programs *DDG*. The technique supporting this feature is presented in Section 6.4.

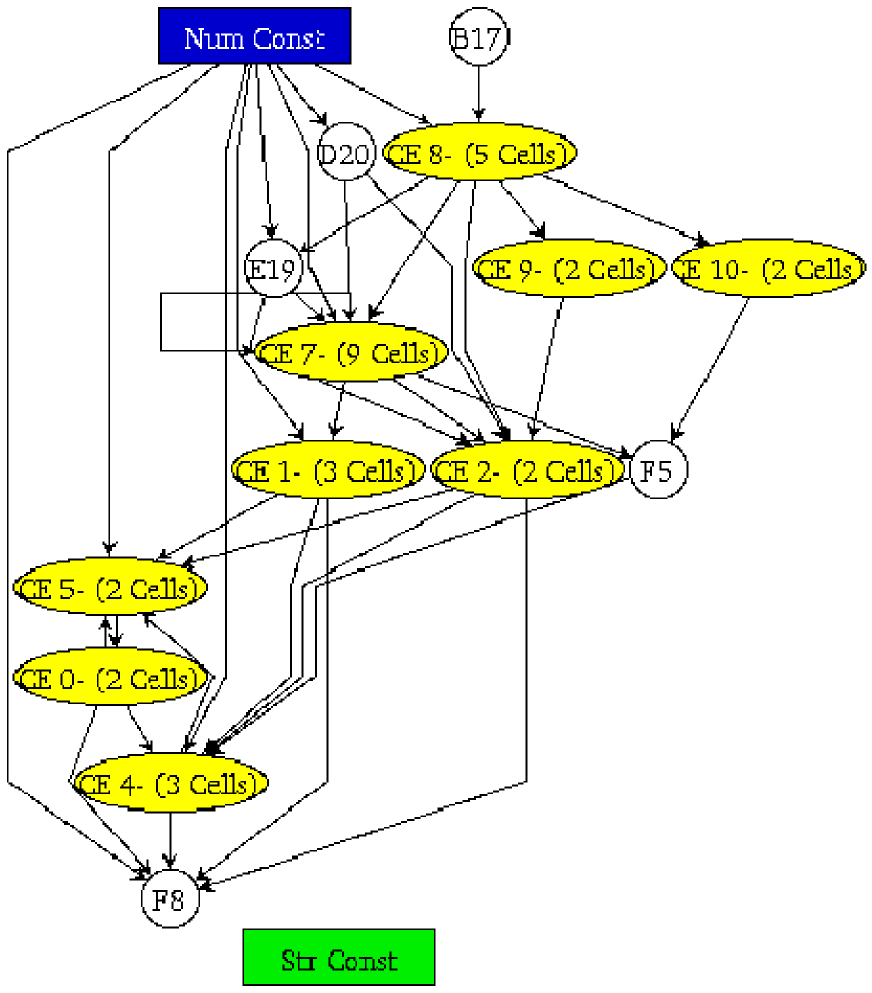


Figure 6.12: *SRG* of the example spreadsheet. Copy equivalent cells are represented by a single node. Together with the information given by the structure browser (see Figure 6.11 on page 117), for instance, the following irregularity can be revealed: there is an edge from `CE2` to `F8`, i.e. the living costs in either fall or spring are taken into account for the calculation of the cash at the end of the summer (although they already influenced the cash at the beginning of the summer, as there is a connection from `CE2` to `CE0` via `CE5`).

6.3 Semantic Classes

As stated above, logical areas reach their limits when very large spreadsheets are analyzed. To cope with them, the principle of fully automatic structure recognition is left and users are allowed to specify whether related areas are spread out column-wise, row-wise, or in patterns taking full advantage of the (two-)dimensional nature of a sheet. Thus, instead of exclusively focusing on the content of formulas to define logical equivalence classes, now spatial situations are taken into account and it is checked, whether the semantic content of these areas is of repetitive nature.

Thus, it is called for a way to identify groups of cells whose member cells are at most a given, user defined distance apart and that form (irrespective of the actual number of cells involved) a repetitive pattern. The cells within such a weakly contiguous group are considered candidates for *semantic units.* If such groups are replicated on the sheet, these replications are identified and grouped into a common *semantic class.*

Conceptually, the notion of semantic class is related to the notion of node equivalence dealt with in Subsection 6.2.1. Thinking about the spreadsheet development process it might be assumed that the semantic class results from copying not a single cell but a whole spatial area instead. But again, there is no information about the development process, and "relatedness" is as vague a concept as "sameness" was when comparing individual formulas. There, the problem was solved by defining node equivalence in terms of three concepts, rooted in copy equivalence which was then successively relaxed to logical equivalence and structural equivalence. Now, a *unit generator* is postulated to formalize "relatedness".

To grasp the idea, one might assume that the unit generator demands copy equivalence among different spatial areas, i.e. cells located on identical relative position within the areas are copy equivalent. If so, those areas could be collapsed into a common semantic class. As will be shown through the definitions, the actual concept is less rigid. For practical reasons, it allows even different relationships between the origin, i.e. the upper left cell of a semantic unit, and the rest of the cells in semantic units forming a class.

As there is no access to the spreadsheet writers' presupposed conceptual model, semantic units are only built upon an assumed semantic relatedness. Hence, the term "semantic" might be slightly too optimistic. As will be seen later, the algorithm to identify semantic classes just attempts to locate the largest possible patterns where replication or relatedness can be postulated. It seems fair to assume that such replications in general do not occur by chance. At least it can be stated that the probability of identifying spurious large semantic classes is far less than the probability of finding unrelated iterators ($\_ = neighbor + 1$) that would be grouped into the same copy-equivalent logical area, irrespective of what they are iterating over.

This approach extends the concept of logical areas by taking the users' view of the spreadsheet more explicitly into account. Therefore, it is necessary to consider the way users mentally group cells (row-, column- or block-oriented). Thus, semantic classes are defined by combining geometrical constraints with the notion of node equivalence.

Detailed definitions of semantic units and semantic classes are given in Subsection 6.3.1. Informally, a semantic class consists of semantic units satisfying the following constraints:

1. All semantic units in a semantic class satisfy the same geometric constraints.
2. All cells in semantic units of a given semantic class residing on positions with the same relative distance to the upper left corner of their semantic unit are in the same logical equivalence class.

The size of the semantic unit is given by the number of cells it contains. Semantic units of size 1 will generate semantic classes that correspond to logical areas. If the size of the semantic unit (number of cells encompassed) increases, the size of the generated semantic classes (number of units encompassed) tends to decrease. However, up to a certain size b the decrease of the class size is not significant. b is a measure for the size of the semantic blocks the spreadsheet programmer had in mind and hence a measure of the size of semantic units to identify as recovered semantic blocks. Although b is not considered by the definitions stated in the subsequent section, it is an important parameter for the algorithm that identifies the semantic units.

6.3.1 Formal Definition

The geometrical constraints users can impose on the unit define the direction and maximal distance of cells that might partake in the same semantic unit. Hence, semantic units can be forced to have an extent only in one dimension, by specifying 0 as the maximal distance for all other dimensions. Moreover, to be consistent with the notion of multi-dimensionality, distances can be indicated over several dimensions. For the two-dimensional case this implies three distance parameters: d_h, d_v and d_m with d_h denoting the maximal horizontal distance allowed between two cells in a semantic unit, d_v the maximal vertical distance allowed, and d_m the maximum Manhattan distance between two cells in the unit without cells not belonging to the unit in-between. By adjusting these parameters, users can restrict the semantic units to consist of cells in the same row ($d_v = 0$) or in the same column ($d_h = 0$). Distance constraints greater one would allow gaps. E.g. a distance vector $(d_h, d_v, d_m) = (2, 3, 4)$ would allow cells in a semantic unit to be separated by at most one empty column or by a column containing labels. Furthermore, blocks might be separated by two lines. However, blocks must not be broken by both, a foreign (say empty) column and two foreign rows. In the rest of this chapter, (d_h, d_v, d_m) will be referred to as the distance vector $\vec{d}$.

Definition 27: Cells

Cells denotes the set of non-empty, non-label cells in a spreadsheet program. □

The difference between empty cells and label cells, i.e. computationally dead cells, is discussed in Definition 24 on page 108 and Definition 25 on page 108. For the formal definition of semantic units and classes on two-dimensional spreadsheets the following functions are introduced:

- $absPos : Cells \rightarrow \mathbb{N} \times \mathbb{N}$ returns the absolute coordinates of the cell on the spreadsheet.

- $relDist : Cells \times Cells \rightarrow \mathbb{Z} \times \mathbb{Z}$ returns the distance between two cells. It is given by $relDist(c_1, c_2) = absPos(c_2) - absPos(c_1)$, with " $-$" representing the subtraction of the respective address vectors.

- $top : \mathbb{P}Cells \rightarrow \mathbb{N} \times \mathbb{N}$ returns the absolute coordinates of the upper-left cell in a set of cells.

- $near : Cells \times Cells \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \{True, False\}$ is true if the two argument cells are not separated by a distance larger than $\vec{d}$ in the respective distance category. Formally, this is:
  $near(c_1, c_2, \vec{d}) \leftrightarrow (relDist(c_1, c_2) = (h, v) \wedge$
  $\vec{d} = (d_h, d_v, d_m) \wedge h \leq d_h \wedge v \leq d_v \wedge h + v \leq d_m)$
  Semantic classes are sets of sets of mutually near cells. Therefore, the reflexive transitive closure of *near* to identify semantic units as such sets of mutually near cells is defined as:

- $dense : \mathbb{P}Cells \times Cells \times Cells \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \{True, False\}$ is a function checking whether the set of cells cs (first argument) contains only cells c_i and c_j that are not separated by a distance larger than $\vec{d}$ in the respective distance category without containing also all cells c_k needed for bridging this gap. Thus $dense(cs, c_1, c_2, \vec{d}) \leftrightarrow$
  $(near(c_1, c_2, \vec{d}) \vee \exists\ c_3 \in cs \mid near(c_1, c_3) \wedge dense(cs, c_3, c_2, \vec{d})$
  Figuratively, one could say that all cells in cs can be placed on a graph with edges defined along the coordinate system of the spreadsheet UI. If the cells are mapped to nodes of this grid, the cells are considered to be *dense* with respect to each other, if there is a sequence of nearest neighbors in which each neighbor can be reached by crossing at most d_r edges in the respective direction r. It should be noted, though, that this sequence has no defined relationship whatsoever with the data flow graph introduced earlier.

With the help of these functions, the definitions for semantic units, semantic classes, and unit-generators can be given. The first definition deals with the semantic support, i.e. the set of densely located cells, out of which semantic units are to be selected.

Definition 28: Semantic Support

The maximal set of cells satisfying the spatial constraints $\vec{d}$ is referred to as semantic support. It is defined as the set

$$\begin{aligned} SS_{\vec{d}} = \{ & cs : Cells | \\ & \forall c_i, c_k \in cs \bullet (dense(cs, c_i, c_k, \vec{d}) \wedge \\ & \qquad \forall c_j \in S \setminus cs | \neg dense((cs \cup \{c_j\}), c_i, c_k, \vec{d}))\} \end{aligned}$$

S denotes the spreadsheet. $\square$

Thus, the non-empty, non-label cells of a given spreadsheet program can be partitioned into a set of semantic supports. Each of the semantic supports is dense and maximal with respect to its geometrical extent. However, the semantic supports are only blocks that bear nothing but geometrical semantics. In the next step each of the semantic supports is examined on whether there are dense subsets of the semantic supports that can be founded on more than one place in the spreadsheet program. This leads to the definition of *semantic units*.

Example 11: Semantic Support

In Figure 6.13 on the following page, the spreadsheet introduced in Subsection 6.2.3 is partitioned into semantic supports, with $\vec{d} = (1, 0, 1)$. In Figure 6.14 on page 125, $\vec{d} = (0, 1, 1)$. Obviously, the cells' content is not taken into account, it is only checked that all the cells have to be non-empty and do not contain labels. $\diamondsuit$

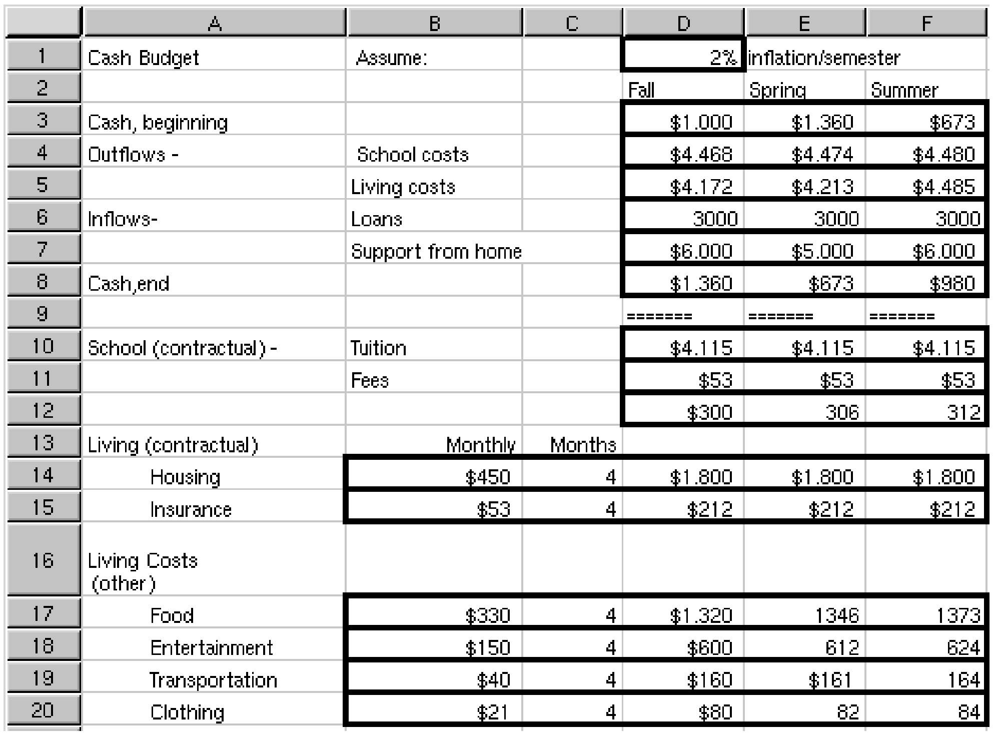
A *semantic unit* is a set of spatially related cells. This set is further checked for whether it is replicated in the spreadsheet program, i.e. whether it exhibits a pattern of replication so that each replication satisfies the semantic constraints of a *unit generator*.

Definition 29: Semantic Unit

A semantic unit U_i satisfying the spatial constraints $\vec{d}$ is defined as a dense subset of its semantic support, generated by its unit generator.

$$\begin{aligned} & U_i \subseteq SS_{\vec{d}} \wedge \\ & \quad (\exists X : \mathbb{P}\, Cells |\ Gen_{SS, Eq_{Start}, Eq_{Rest}} = (ld, X) \wedge |X| > 1 \wedge U_i \in X) \wedge \\ & \quad \forall\ c_i, c_k \in U_i \bullet dense(U_i, c_i, c_k, \vec{d}). \end{aligned}$$

$\square$

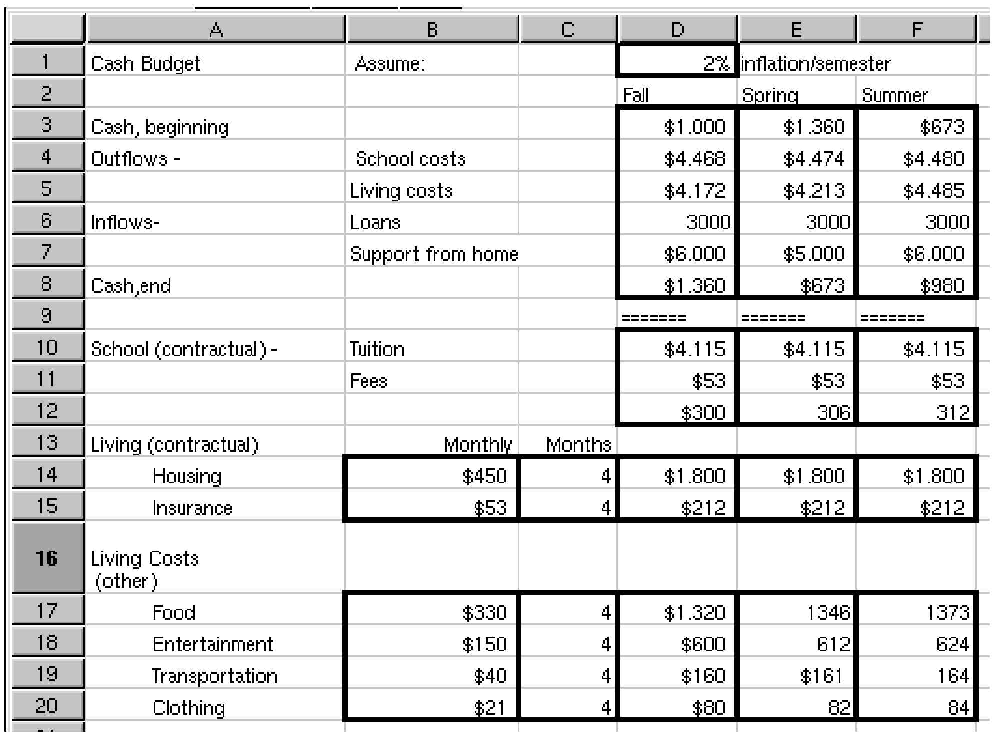


	A	B	C	D	E	F
1	Cash Budget	Assume:		2%	inflation/semester	
2				Fall	Spring	Summer
3	Cash, beginning			$1.000	$1.360	$673
4	Outflows -	School costs		$4.468	$4.474	$4.480
5		Living costs		$4.172	$4.213	$4.485
6	Inflows-	Loans		3000	3000	3000
7		Support from home		$6.000	$5.000	$6.000
8	Cash,end			$1.360	$673	$980
9				=======	=======	=======
10	School (contractual) -	Tuition		$4.115	$4.115	$4.115
11		Fees		$53	$53	$53
12				$300	306	312
13	Living (contractual)	Monthly	Months			
14	Housing	$450	4	$1.800	$1.800	$1.800
15	Insurance	$53	4	$212	$212	$212
16	Living Costs (other)					
17	Food	$330	4	$1.320	1346	1373
18	Entertainment	$150	4	$600	612	624
19	Transportation	$40	4	$160	$161	164
20	Clothing	$21	4	$80	82	84

Figure 6.13: Example spreadsheet with semantic supports framed, $\vec{d} = (1, 0, 1)$

As mentioned above, the *semantic support* comprises a set of cells that have just from their spatial proximity the *potential* to form a meaningful semantic unit. If they actually do that, will depend on whether replications satisfying the unit generator Gen, which is defined next, can be found[2]. It is to be noted that the denseness criterion defining the support needs to hold also within the unit itself. Thus, *near*-relationships have to be maintained in building the respective subset. Only in this case, one of the units making up the class might be labeled as base unit of the semantic class. One should also note that a given support might furnish several different, non-overlapping units. Subsequently, Eq_{Start} denotes an equivalence relation that has to hold among the upper left cells of semantic units in the same semantic class, and Eq_{Rest} the equivalence relation among the remaing cells in the semantic units (see Definition 30 on page 126).

[2]There has to be more than one such set of cells X in the powerset of cell-sets cs.

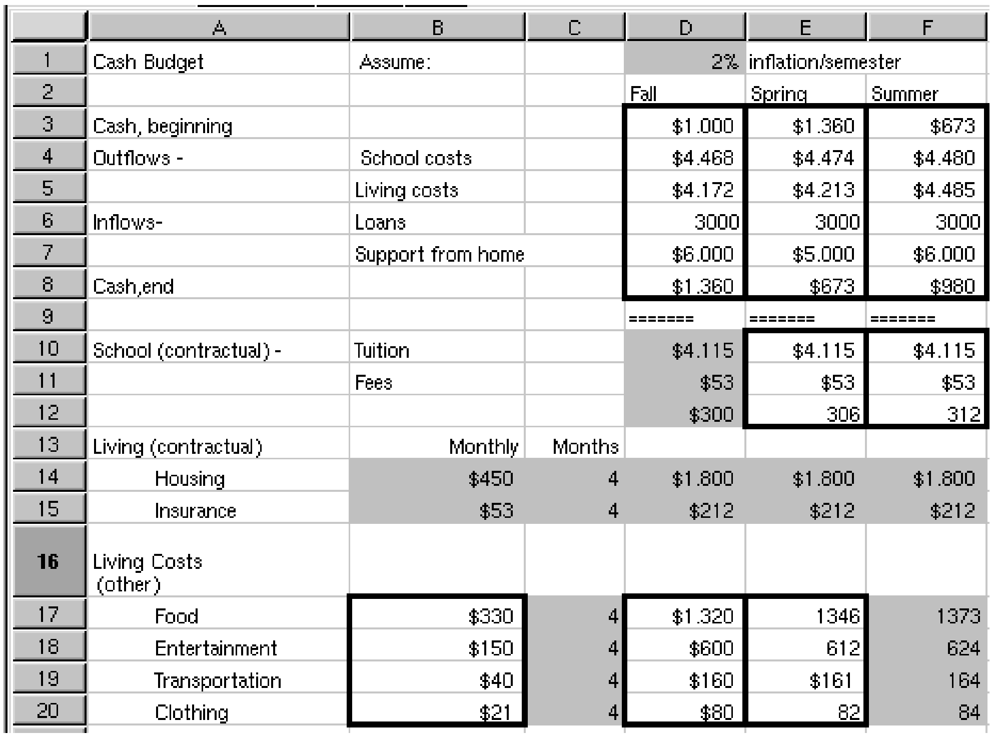


	A	B	C	D	E	F
1	Cash Budget	Assume:		2%	inflation/semester	
2				Fall	Spring	Summer
3	Cash, beginning			$1.000	$1.360	$673
4	Outflows -	School costs		$4.468	$4.474	$4.480
5		Living costs		$4.172	$4.213	$4.485
6	Inflows-	Loans		3000	3000	3000
7		Support from home		$6.000	$5.000	$6.000
8	Cash,end			$1.360	$673	$980
9				=======	=======	=======
10	School (contractual) -	Tuition		$4.115	$4.115	$4.115
11		Fees		$53	$53	$53
12				$300	306	312
13	Living (contractual)	Monthly	Months			
14	Housing	$450	4	$1.800	$1.800	$1.800
15	Insurance	$53	4	$212	$212	$212
16	Living Costs (other)					
17	Food	$330	4	$1.320	1346	1373
18	Entertainment	$150	4	$600	612	624
19	Transportation	$40	4	$160	$161	164
20	Clothing	$21	4	$80	82	84

Figure 6.14: Example spreadsheet with semantic supports framed, $\vec{d} = (0, 1, 1)$

Example 12: Semantic Units
Assuming the semantic supports according to $\vec{d} = (0, 1, 1)$ (see Figure 6.14), a possible decomposition of the spreadsheet program into semantic units is given by Figure 6.15 on the following page. For Eq_{start} and Eq_{rest} structural equivalence has been chosen. The grey-shaded cells are semantic units that consist of single cells. Areas with a thick border are semantic units consisting of multiple cells. $\diamondsuit$

A *unit generator* consists of a set of local coordinates that identifies the set of cells to be related according to the relatedness-criterion mentioned in the introduction of this section and of the set of semantic units generated. The strictest form of relatedness would be to require copy equivalence to hold among all cells assuming identical relative positions within the semantic units to be compared. In higher order abstractions, less rigid constraints seem to be desirable, though. Therefore, any of the criteria defined by the equivalence class generator (see Definition 26) are allowed. Since the origins of the semantic unit might play a special role, a distinction is made between the equivalence classes for $top(Cells)$, denoted by Eq_{Start}, and Eq_{Rest}, the equivalence relation to hold among cells on other positions in related semantic units.



	A	B	C	D	E	F
1	Cash Budget	Assume:		2%	inflation/semester	
2				Fall	Spring	Summer
3	Cash, beginning			$1.000	$1.360	$673
4	Outflows -	School costs		$4.468	$4.474	$4.480
5		Living costs		$4.172	$4.213	$4.485
6	Inflows-	Loans		3000	3000	3000
7		Support from home		$6.000	$5.000	$6.000
8	Cash,end			$1.360	$673	$980
9				=======	=======	=======
10	School (contractual) -	Tuition		$4.115	$4.115	$4.115
11		Fees		$53	$53	$53
12				$300	306	312
13	Living (contractual)	Monthly	Months			
14	Housing	$450	4	$1.800	$1.800	$1.800
15	Insurance	$53	4	$212	$212	$212
16	Living Costs (other)					
17	Food	$330	4	$1.320	1346	1373
18	Entertainment	$150	4	$600	612	624
19	Transportation	$40	4	$160	$161	164
20	Clothing	$21	4	$80	82	84

Figure 6.15: One possible decomposition of the example spreadsheet into semantic units, $\vec{d} = (0,1,1)$

Definition 30: Unit Generator
For a unit $U_i \subseteq SS_{\vec{d}}$ the unit generator $Gen_{SS_{\vec{d}},Eq_{Start},Eq_{Rest}}$ is defined as

$$\begin{aligned}
Gen_{SS_{\vec{d}},Eq_{Start},Eq_{Rest}} \;=\; & \{(ld : \mathbb{P}\,(\mathbb{N}\times\mathbb{N}),\ X : \mathbb{P}\ Cells) \mid \\
& \exists U_i \subseteq SS_{\vec{d}} \bullet (\\
& \quad \forall c_i, c_j \in U_i \bullet dense(U_i, c_i, c_j, \vec{d}) \wedge \\
& \quad \exists T : Cells \mid (T = Eq_{Start}(top(U_i))) \wedge \\
& \qquad \forall cs \in X \bullet top(cs) \in T) \wedge \\
& \quad \forall dd \in (ld \setminus \{(0,0)\} \mid \exists\ R_{dd} :\ Cells \mid \\
& \qquad dd = relDist(c_{dd}, absPos(top(U_i))) \wedge \\
& \qquad R_{dd} = Eq_{Rest}(c_{dd}))\wedge \\
& \qquad (\forall cs \in X \mid \exists c_j \in cs \bullet \\
& \qquad\quad relDist(c_j, absPos(top(cs))) = dd\ \wedge c_j \in R_{dd}) \wedge \\
& \quad (\textstyle\bigcup_j (U_j\ \bullet\ U_j \in X) \subseteq ((\bigcup_{dd} R_{dd}) \cup T))\ \}.
\end{aligned}$$

The complexity of this definition is due to the fact that it maps the sets of elements defined on the basis of some node equivalence criterion on sets defined on the basis of some spatial criterion in a way that the involved “transposition” covers all elements of the respective sets. The definition also highlights that Definition 28 provides only the framework out of which actual semantic units are to be isolated, by finding sets of cells that match according to the equivalence criterion Eq_{Start}. This minimal unit defines but a mere logical area restricted by additional spatial constraints. It can be extended when cells of equal relative distance dd to the start cells can be grouped into sets R_{dd} according to equivalence criterion Eq_{Rest} so that the union of the top-set with all rest-sets cover the sets contained in X related by the unit generator. Due to this construction, any of the sets contained in X forms a semantic unit and X itself is the semantic class from which those units are drawn.

Example 13: Unit Generator
Assuming the decomposition of a spreadsheet into semantic units shown by Example 12, and the assignment of cells to logical areas that is given by Figure 6.11 on page 117, the tabular presentation for the unit generator for the semantic unit with the cells `E3` to `E8`, with structural equivalence for Eq_{Start} and Eq_{Rest}, is as follows:

Coordinates	Set of Members
$(0,0)$	$\{E3, E14, E15, F3, F14, F15\}$
$(0,1)$	$\{D4, E4, F4, D5, E5, F5\}$
$(0,2)$	$\{D4, E4, F4, D5, E5, F5\}$
$(0,3)$	{Numeric constants}
$(0,4)$	$\{D7, E7, F7\}$
$(0,5)$	$\{D8, E8, F8\}$

A more formal notation for the unit generator specifies the unit generator by two sets, with the first set denoting relative coordinates within each semantic unit and the second set consisting of sets of cells that are in a specific equivalence relation. In the table above, the first column contains all possible relative coordinates, and each row in the second column is a set of equivalent cells. It can be seen that another semantic unit with cells `F3` to `F8` has the same unit generator. Thus, both units share a semantic class. $\diamondsuit$

To relate semantic units back to the conceptual notion of a set of spatial blocks spreadsheet writers might have had in mind when designing the sheet, the *semantic class* is defined as the set of semantic units that have the same unit generator.

Definition 31: Semantic Class
Let SC_{U_i} be the semantic class that contains the semantic unit $U_i \subseteq SS_{\vec{d}}$.

$$\begin{aligned} SC_{U_i} = \{U_j | & Gen_{U_j, Eq_{Start}, Eq_{Rest}} = Gen_{U_i, Eq_{Start}, Eq_{Rest}} \\ & \wedge (U_i = U_j \vee U_i \cap U_j = \emptyset\} \end{aligned}$$

$\square$

It is easy to see that SC_{U_i} is exactly the set X constructed by the generator. The semantic unit in the topmost, leftmost position can be particularly distinguished by referring to it as *base unit* of the class.

It is to be noted that these three definitions are interrelated and indeed still contain a degree of freedom not yet bound. Semantic units are drawn from an arbitrary set of cells meeting the specified spatial constraints. Hence, a straightforward partitioning of the spreadsheet into semantic units will not result in a helpful abstraction. To produce a useful classification of cells into semantic units, the quality of the resulting semantic classes has to be considered.

A semantic unit with a singleton as unit generator is not useful for abstraction. Therefore, semantic units should be defined in a way that the resulting semantic class contains a high number of other related semantic units. Admittedly, it could be argued that breadth of the base unit is as important a characteristic of powerful semantic units as depth of replication.

The algorithm to identify unit generators has to consider both issues. It offers two options to be controlled: the user can influence the breadth versus depth issue to a certain extent by properly specifying the distance vector. On the other hand, the very construction of the algorithm rests on the availability of logical areas.

Indeed, logical areas can be considered semantic classes with cells as semantic units. Thus, the algorithm has to merge portions of different semantic classes, satisfying the spatial constraints defined with the semantic support, until no further merging can be done that will not result in semantic classes that consist only of single semantic units.

This is not a strict merge though, since a merger can take place only if the local distances ld match. Therefore, merging involves also filtering on intersecting local distances. An additional parameter b is introduced, indicating the cutoff-percentage that halts the merging process in case the filtering part of the operation would drop more than $(100 - b)$ % of the set serving as base for the merger. To avoid unnecessary complexity, b is not included in definition 30. However, this additional explanation might indicate that T has algorithmically a distinct role in comparison to the adjungated sets R_{dd}.

6.3.2 An Algorithm for the Identification of Semantic Units

As already mentioned above, those semantic units have to be found that lead to a partitioning of the spreadsheet that is usable for the spreadsheet auditor.

Heuristics

Of course, there are many possible ways to partition the spreadsheet program into valid semantic units, but most of them do not match the users' model. However, it is assumed that a helpful abstraction of the spreadsheet can be generated by means of the following heuristics:

1. Build large semantic units, i.e. semantic units with a maximum number of cells in them.

2. Prefer unit generators that yield semantic classes with a high number of semantic units in them.

The criteria given above are partially contradictory, and thus have to be weighted. The second criterion is necessary to avoid the pathological case of one big semantic unit that covers all non-empty cells in the spreadsheet. This would be possible, if users allowed semantic units to extend on the horizontal and the vertical dimensions of the spreadsheet UI.

The first point suggests that abstraction can be increased by merging small semantic units to larger ones. Thus, the users have to deal with a smaller number of independent units. However, it is assumed that the number of replications of a small unit will be higher than the number of occurrences of larger units, as it will be subject to noise, and accidental replications of large units are rather seldom.

However, it is assumed that the influence of noise and accidental replications is not too severe, as long as the size of the identified semantic units is beyond the block size in the users' conceptual model. Indeed, the merging algorithm selects two semantic classes, with the highest number of semantic units that are reachable. Up to a certain size of the participating semantic units, the merging will involve most of the semantic units in the concerned semantic classes.

Depending on the regularity of the spreadsheet, the spreadsheet auditors will want to stop the merging process when less than $b\%$ of the semantic units in one of the affected semantic classes can be merged with the semantic units in the second class. b has to be specified by the auditor.

If it is common for formulas to co-occur with specific other formulas in a given spreadsheet program, a high value for b is recommended in order to filter out accidental co-occurrences between formulas. If there are some semantic units that occur many times, and in combination with different formulas, such as what-if analysis, a low value for b will still guarantee that the concerned semantic units are merged.

Thus, the first heuristic drives the merging of small semantic units to bigger units. As it is stated above, in typical real-world spreadsheets the number of the semantic classes will increase when smaller semantic units are merged to a larger one, as the larger the pattern becomes, the smaller the number of the replications will be. Obviously, a trade-off between the two criteria is necessary.

Currently, two semantic units U_1 and U_2 are considered candidates for a merge, if

- $Gen_{U_1,start,rest} \neq Gen_{U_2,start,rest}$, and

- $U_1, U_2 \in SS_{\vec{d}} \rightarrow U_1 \cup U_2 \in SS_{\vec{d}}$, i.e. the merged semantic unit still satisfies the geometric conditions, and

- at least $b\%$ of the semantic units in SC_{U_1} or at least $b\%$ of the semantic units in SC_{U_2} can be merged. b is the user specified parameter discussed above.

The first rule will guarantee that the merge is performed only between semantic units in different semantic classes. It is assumed that there is no additional level of abstraction introduced if semantic units in the same class are merged. For the spreadsheet auditor they are already observably related, because they are in the same semantic class.

The third rule ensures that occasional co-occurrence of formulas in the same logical area is not sufficient for the formation of a semantic unit.

The rules are iteratively applied on the set of semantic units, until no further merging is possible. As the order of the merge is relevant for the validity of the result, semantic classes with a higher co-occurrence should be merged first. It appears that the bigger the units in a given semantic class become, the higher the cohesion toward other semantic units will be. When the size of the border of the semantic unit grows, the number of candidates that become reachable obviously increases as well.

Rationale of the Algorithm

The selection algorithm that is outlined in Table 6.1 will at first select semantic classes with a higher probability of co-occurrence.

As Definition 29 implies, each cell is a semantic unit on its own, and definitions 30 and 31 will consider a logical area to be a unit generator and the cells in a logical area to form a semantic class. Hence, the input to the algorithm is a set of semantic units, i.e. single cells, with logical areas being their semantic classes.

The selection algorithm selects the two semantic classes with the maximal co-occurrence and passes them to the merging algorithm. Co-occurrence between semantic classes is coarsely defined as the percentage of the semantic units in the first semantic class that are in the neighborhood (i.e. reachable) from semantic units in the second class. This selection is iteratively repeated on the newly generated set of semantic units until no further merge is possible. Merging stops either because no co-occurring semantic units can be found, or because the percentage of co-occurrences is beyond the parameter b. See Table 6.3 on page 134 for the algorithm that calculates the co-occurrence between two semantic classes.

SC_s denotes the semantic class of the semantic unit s and $Gen_{s,start,rest}$ is the unit generator of the semantic unit s, with *start* and *rest* with denoting the equivalence class generators Eq_{Start} and Eq_{Rest}. *Merge_Candidates* calculates the absolute number of semantic units in two semantic classes that could be merged, according to the disctance vector $\vec{d}$.

The merging algorithm (see Table 6.2 on page 133) is invoked with the selected semantic classes to merge, the distance vector and the so far identified semantic units. For each semantic unit in the first semantic class, a *reachable* semantic unit in the second semantic class has to be found. The two units can then be merged, and the result is added to the set of semantic units.

Obviously, not all of the semantic units in two semantic classes can be merged, as for some of them there will be no reachable counterparts. To avoid loss of the not-merged semantic units, the merging will usually invent a new semantic class containing all the newly merged semantic units, and it will still keep the two semantic classes, containing the semantic units that were not merged.

The given algorithm is computationally rather complex, as it contains three nested levels of loops. However, as the input consists only of a small number of semantic units (in the average spreadsheet 30-50 logical areas can be identified), computational complexity is not a crucial issue here.

6.3.3 Auditing Strategies Based on Semantic Classes

As semantic classes are an abstraction of logical areas, the same auditing strategies are supported. Nevertheless, due to the higher abstraction that is offered by semantic classes, the auditor has to care for more parameters. The application of the pattern driven, *SRG*- driven and structure driven auditing strategies that have been introduced for spreadsheet auditing with logical areas in Subsection 6.2.3 will be consecutively discussed in terms of semantic classes.

Pattern Driven Auditing

Pattern driven auditing is a helpful aid to understand spreadsheet programs intuitively and will support the spreadsheet auditor in finding errors. Using a pattern driven auditing approach with semantic classes will involve the following steps:

- identify semantic units and semantic classes,
- interactively map semantic classes and semantic units back to the spreadsheet UI,
- find geometrical patterns for the occurrences of the member units of a given semantic class, and
- irregularities in the geometrical patterns are indicators for hot-spots.

However, this time the auditor does not look for regular patterns in the distribution of single cells, but in the distribution of the building blocks of the spreadsheet. The so generated pattern might be a regular occurrence of a semantic unit that is a whole row or a large part of a column.

Thus, a single erroneous cell will hinder the membership of the corresponding building block in a semantic class and will lead to a severe irregularity in the geometrical pattern that is easy to spot.

```
Algorithm Select($Set\ of\ SemanticUnit\ S$, $Set\ of\ Cells\ Cells$,
                 $Integer\ d_h$, $Integer\ d_v$, $Integer\ d_{Man}$,
                 $index\ start$, $index\ rest$,
                 $Integer\ b$)
return $Set\ of\ SemanticUnits$
declare
   $Boolean\ found$
   $Real\ maxCo = 0$
   $Cell\ c, c'$
   $Integer\ max\_cand$
   $SemanticUnit\ s, s', m_1, m_2$
begin
   repeat
      $found$ = false
      $max\_cand = 0$
      for $s \in S$ do
         for $s' \in S$ do
            if CoOccurence($SC_s$,$SC_{s'}$) $>$ $maxCo$ and
              $Gen_{s,start,rest} \neq Gen_{s',start,rest}$ and
              Co-Occurrence($SC_s$,$SC_{s'}$) $> b$ and Co-Occurrence($SC_{s'}$,$SC_s$) $> b$ and
              #Merge_Candidates($SC_s$, $SC_{s'}, d_h, d_v, d_{Man}$) $>$ $max\_cand$
            then
              $m_1 = s$
              $m_2 = s'$
              $maxCo$ =Co-Occurrence($SC_s, SC_{s'}$)
              $found$=true
              $max\_cand$ = #Merge_Candidates($SC_s$, $SC_{s'}$)
            end if
         end for
      end for
      if found then
         Merge($SC_{m_1}$, $SC_{m_2}$, $S$, $d_h$, $d_v$, $d_{Man}$)
      end if
   until not $found$
   return $S$
 end
```

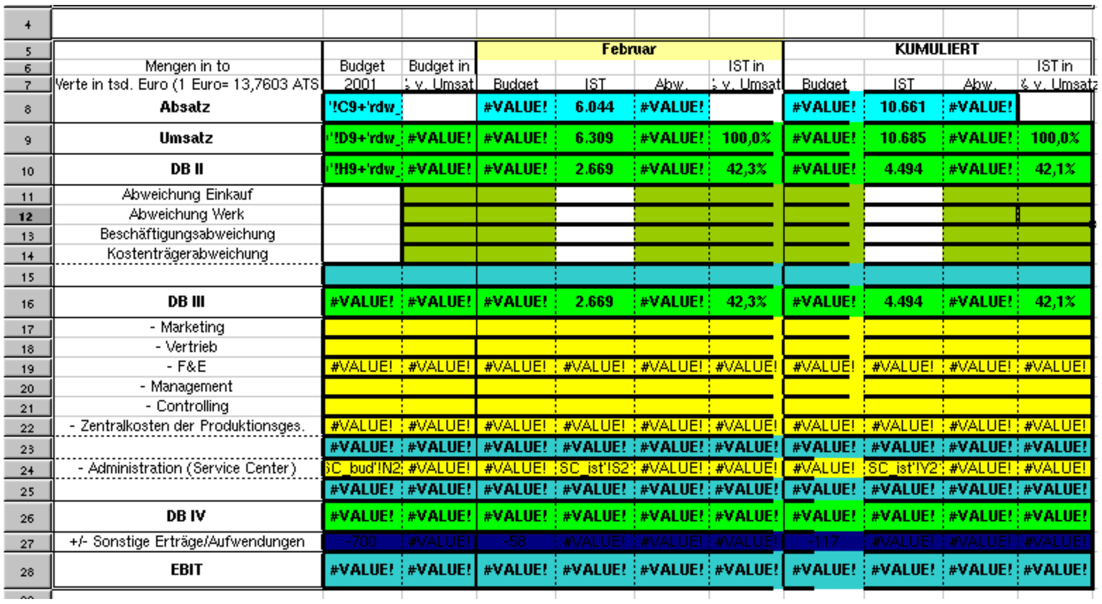
Table 6.1: Selection Algorithm

```
**Algorithm** Merge (*Semantic Class* $SC_1$, *SemanticClass* $SC_2$,
                     *Set of SemanticUnit* $S$,
                     *Integer* $d_h$, *Integer* $d_v$, *Integer* $d_{Man}$)
**return** *Set of SemanticUnit*
 **declare**
 *SemanticUnit* $s, s'$
 *Cell* $c, c'$
 **begin**
    $S = S \setminus (SC_1 \cup SC_2)$
    **for** $s \in SC_1$ **do**
          **for**$s' \in SC_2$ **do**
             **if** $\exists c \in s \mid$
                 $\exists c' \in s' \bullet reachable(SC_1 \cup SC_2, c, c', d_h, d_v, m_{Man})$
             **then**
               $S = S \cup \{s \cup s'\}$
               $SC_1 = SC_1 \setminus \{s\}$
               $SC_2 = SC_2 \setminus \{s'\}$
               **break**
             **end if**
          **end for**
    **end for**
    $S = S \cup SC_1 \cup SC_2$
    **return** $S$
  **end**
```

Table 6.2: Merging algorithm

```
**Algorithm** Co-Occurence (*Semantic Class* $SC_1$, *SemanticClass* $SC_2$,
                            *Integer* $d_h$, *Integer* $d_v$, *Integer* $d_{Man}$)
return double
declare
   Integer $sc_1\_cnt$, $sc_2\_cnt$
   Semantic Unit $S_1$, $S_2$
   Integer $cooc = 0$
   Set of Semantic Units Used
begin
   for $S_1 \in SC_1$ do
         $sc_1\_cnt = sc_1\_cnt + 1$
         for$S_2 \in SC_2$ do
             if$S_2 \notin Used\ \wedge$reachable$(S_1, S_2, d_h, d_v, d_{Man})$
             then
               $Used = Used \cup \{S_2\}$
               $cooc = cooc + 1$
             end if
         end for
   end for
   if$sc_1\_cnt > sc_2\_cnt$ then
         return $cooc/sc_2\_cnt$
   else
         return $cooc/sc_1\_cnt$
   end if
 end
```

Table 6.3: Calculating the co-occurence



4											
5				Februar				KUMULIERT			
6	Mengen in to	Budget	Budget in				IST in				IST in
7	Verte in tsd. Euro (1 Euro= 13,7603 ATS	2001	% v. Umsat	Budget	IST	Abw.	% v. Umsat	Budget	IST	Abw.	% v. Umsatz
8	**Absatz**	'!C9+'rdw_		#VALUE!	6.044	#VALUE!		#VALUE!	10.661	#VALUE!	
9	**Umsatz**	'!D9+'rdw_	#VALUE!	#VALUE!	6.309	#VALUE!	100,0%	#VALUE!	10.685	#VALUE!	100,0%
10	**DB II**	'!H9+'rdw_	#VALUE!	#VALUE!	2.669	#VALUE!	42,3%	#VALUE!	4.494	#VALUE!	42,1%
11	Abweichung Einkauf										
12	Abweichung Werk										
13	Beschäftigungsabweichung										
14	Kostenträgerabweichung										
15											
16	**DB III**	#VALUE!	#VALUE!	#VALUE!	2.669	#VALUE!	42,3%	#VALUE!	4.494	#VALUE!	42,1%
17	- Marketing										
18	- Vertrieb										
19	- F&E	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!
20	- Management										
21	- Controlling										
22	- Zentralkosten der Produktionsges.	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!
23		#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!
24	- Administration (Service Center)	SC_bud'!N2	#VALUE!	#VALUE!	SC_ist'!S2	#VALUE!	#VALUE!	#VALUE!	SC_ist'!Y2	#VALUE!	#VALUE!
25		#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!
26	**DB IV**	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!
27	+/- Sonstige Erträge/Aufwendungen	-700	#VALUE!	-58	#VALUE!	#VALUE!	#VALUE!	-117	#VALUE!	#VALUE!	#VALUE!
28	**EBIT**	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!	#VALUE!

Figure 6.16: Pattern of semantic units projected to the spreadsheet UI. Semantic units are framed with a thick border, units in the same cell are shaded with the same color. Empty cells are white. This partitioning is result of the merging algorithm, invoked with the distance vector $\vec{d} = (2, 0, 2)$ and $b = 1$.

Additionally, the pattern driven auditing strategy will also support the comprehension of a spreadsheet program, as building blocks with equal functionality are highlighted. In the following example, the pattern of semantic units in a part of a spreadsheet is discussed.

Example 14: Pattern of Semantic Units
In Figure 6.16 a part of a spreadsheet program is shown that has been in use in a company. It was examined in a field audit (see Clermont et al. [CHM02]). The spreadsheet deals with the calculation of the returns of a business branch, and is not trivial at all, although a clear structure is recognizable. There is a specific calculation that reoccurs in the *Umsatz*, *DB II*, *DB III* and *DB IV* rows. However, row 24 looks peculiar at first sight, as it interrupts an otherwise regular block of semantic units in the same class. Additionally, the semantic units in rows 11 to 14 have to be checked carefully. Although there are no values displayed, most of the cells have formulas- only the blank cells are empty, and, thus, not part of any semantic unit. ◊

However, there is an additional dimension the spreadsheet auditor has to consider: It is not only the pattern of occurrence of semantic units in a semantic class, but also the extent of each of the semantic units. If there are *reachable* semantic units in distinct semantic classes that the auditor considers related, it has to be checked, why no merging has happened. Usually, this occurs if the pattern of co-occurrence is distorted, or the parameters for the merging algorithm have not been chosen carefully.

SRG_{SC} Driven Auditing Strategy

The SRG_{SC} driven auditing strategy banks on an abstraction of the DDG, again. In analogy to the SRG_{LA}, nodes in the SRG_{SC} are semantic classes, an edge between two nodes n_1 and n_2 is drawn if a cell in the semantic class represented by n_1 references a cell in the semantic class represented by n_2.

Although this auditing strategy is at first sight very similar to the SRG-driven auditing with logical areas, there is a fundamental difference. The nodes present units that are supposed to fulfill a given self-contained task, whereas with logical areas the nodes in the generated SRG represented sets of cells with equal formulas. Thus, this time the data flow between different tasks is observable instead of dependencies on the formula level.

Again, the number of nodes and edges in the SRG is expected to be significantly smaller as in the DDG. In the example spreadsheet that has been discussed in Subsection 6.2.3 the SRG consists of 10 nodes and 20 edges (see Figure 6.17 on the next page), whereas the DDG consists of some 57 nodes and 91 and edges.

Although the SRG looks complicated at first sight, the auditor has to deal with a graph representation with a significantly reduced number of nodes and edges. Additionally, the visualization can be improved by supporting zooming into certain semantic classes, i.e. replacing the node of the semantic class in the SRG with a node for each semantic unit in the semantic class. This visualization technique, also known as fish-eye view, has already been used by approaches for the visualization and comprehension of conventional software systems, e.g. in the *RIGI*- Project (see Mueller et al. [MOTU93, MWT94]).

Part of the *RIGI*-Project is a visualization tool that visualizes each module of a software as a node, *def-use*[3] relations or procedure calls will generate an edge between two modules. However, as this level of abstraction is too high, there is a possibility for the user to zoom into given nodes and replace them either by their control flow graph or by the source code.

Nevertheless, in order to keep the complexity of the graph in an order of magnitude that is still comprehensive for a human reader, the other nodes in the graph are left at the higher level of abstraction. Thus, the user can concentrate on the details of a given module without neglecting the context of the module in the whole system. A detailed discussion of the software visualization toolkit is given by Tilley et al. [TMO92] the application of fish-eye views for the visualization of data is outlined by Furnas [Fur86].

[3]A *def-use* relation between two statements s_1 and s_2 is assumed, if s_1 assigns a value to a variable v and that value is read out by the statement s_2.

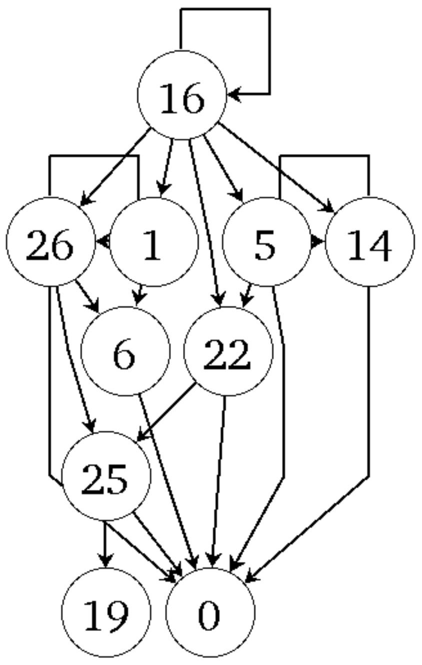


Figure 6.17: *SRG* of the spreadsheet presented in Figure 6.7 on page 113. The numbers the node represent semantic classes. The class assignment of specific semantic units and cells is shown in Figure 6.18 on the next page. The merging algorithm was invoked with $\vec{d} = (0, 1, 1)$, $b = 70\%$, $Eq_{start} = se$, $Eq_{rest} = se$.

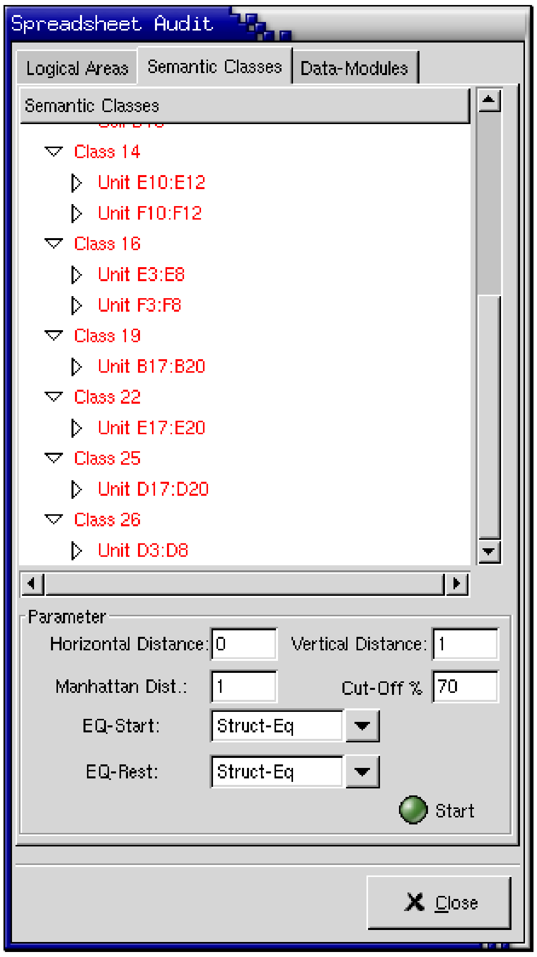
Applying fish-eye views in the *SRG*-driven auditing for spreadsheet programs has the advantage that the spreadsheet auditor can choose an individual level of abstraction for each part of the spreadsheet program. Thus, the three levels of abstraction,

1. Semantic Class,
2. Semantic Unit, and
3. Cell

can be individually assigned. E.g., the spreadsheet auditor can choose to replace a certain semantic class sc_1 in the *SRG* with the units $su_1, \cdots, su_n$ that form the class. Now, the auditor can decide again, to replace su_i, with $1 < i < n$ in the *SRG* with the cells $c_1, \cdots c_m$ that form su_1. Thus, the *DDG* of a given semantic unit can be examined in the context of the *SRG*.

Structure Driven Auditing Strategy

There are more ways to vary in the strictness of the similarity criteria for semantic units and classes, as there are for logical area. Consequently, structure driven


Spreadsheet Audit

Logical Areas | Semantic Classes | Data-Modules

Semantic Classes
- Class 14
  - Unit E10:E12
  - Unit F10:F12
- Class 16
  - Unit E3:E8
  - Unit F3:F8
- Class 19
  - Unit B17:B20
- Class 22
  - Unit E17:E20
- Class 25
  - Unit D17:D20
- Class 26
  - Unit D3:D8

Parameter

Horizontal Distance: 0 Vertical Distance: 1

Manhattan Dist.: 1 Cut-Off %: 70

EQ-Start: Struct-Eq

EQ-Rest: Struct-Eq

Figure 6.18: Semantic classes and semantic units of the example spreadsheet in the structure browser.The merging algorithm was invoked with $\vec{d} = (0, 1, 1)$, $b = 70\%$, $Eq_{start} = se$, $Eq_{rest} = se$.

auditing becomes more complicated. Hence, the structure browser for semantic classes (see Figure 6.18) cannot grasp the hierarchy of different combinations of Eq_{Start} and Eq_{Rest} any more (see Figure 6.19 on the next page).

However, the spreadsheet auditor can still calculate a partitioning of the spreadsheet program into semantic classes with a stricter and a weaker combination of Eq_{Start}, Eq_{Rest} and b. This allows to compare the outcome by coloring the result of each partitioning on the spreadsheet program, as shown in the following example:

Example 15: Structure driven auditing with Semantic Classes

In Figure 6.20 on page 140, the partitioning of the example spreadsheet into semantic classes and semantic units with $Eq_{Start} = se$ and $Eq_{Rest} = se$ is shown, where $Eq_{Start} = ce$ and $Eq_{Rest} = ce$ in Figure 6.21 on page 141. Comparing the colored spreadsheets, the following inconsistencies can be detected:

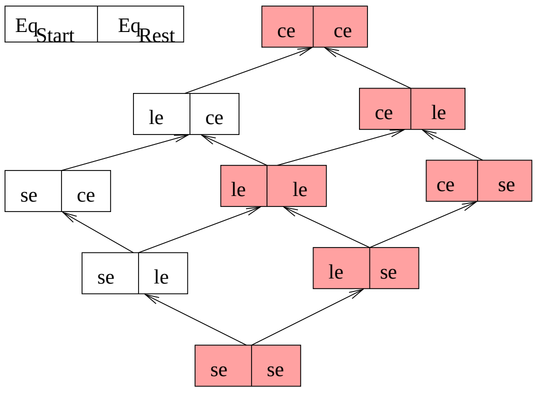


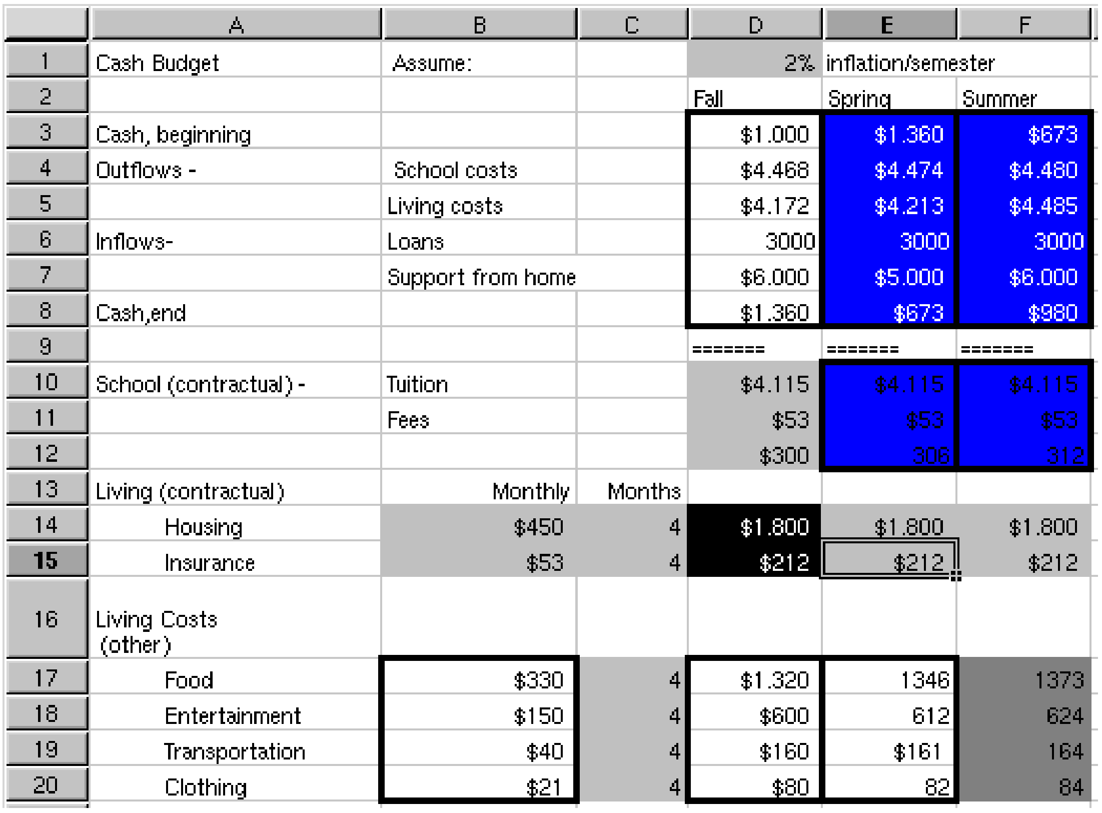
Figure 6.19: Hierarchy of strictness for similarity criteria Eq_{Start} and Eq_{Rest} for semantic units. Link equivalence classes and b, the boundary parameter, are neglected here. The gray-shaded combinations have Eq_{Start} stricter Eq_{Rest}, according to the hierarchy of logical equivalence criteria.

- Obviously, column D is distinct from E and F. Having a closer look at the logic and the formulas of the spreadsheet (see Figure 6.8 on page 114), this can be explained by the fact that in D some initialization is performed.
- In the columns E and F the upper parts are structural equivalent, the middle part consists of cells that are mutually copy-equivalent, and thus not merged into a semantic unit. The following questions have to be answered:
  - Why is there no similarity relation between the bottom of the two columns?
  - Why is the middle part copy-equivalent, whereas the upper part consists of two semantic units that have the same extent, but are only structural equivalent?

Examining the spreadsheet will reveal 4 of the 5 errors that are hidden in the spreadsheet. For a correct spreadsheet the coloring of semantic classes with $Eq_{Start,Rest} = ce$ is shown in Figure 6.22 on page 142. $\Diamond$

6.3.4 Discussion

Semantic classes introduce a higher level of abstraction into the spreadsheet auditing process, and thus, can potentially increase the understanding of complex spreadsheet programs or enable the auditor to spot irregularities without going into detail. Again, the irregularities found do only indicate hot-spots that need further investigation. Some of the identified problems might turn out to be introduced on purpose.



	A	B	C	D	E	F
1	Cash Budget	Assume:		2%	inflation/semester	
2				Fall	Spring	Summer
3	Cash, beginning			$1.000	$1.360	$673
4	Outflows -	School costs		$4.468	$4.474	$4.480
5		Living costs		$4.172	$4.213	$4.485
6	Inflows-	Loans		3000	3000	3000
7		Support from home		$6.000	$5.000	$6.000
8	Cash,end			$1.360	$673	$980
9				=======	=======	=======
10	School (contractual) -	Tuition		$4.115	$4.115	$4.115
11		Fees		$53	$53	$53
12				$300	306	312
13	Living (contractual)	Monthly	Months			
14	Housing	$450	4	$1.800	$1.800	$1.800
15	Insurance	$53	4	$212	$212	$212
16	Living Costs (other)					
17	Food	$330	4	$1.320	1346	1373
18	Entertainment	$150	4	$600	612	624
19	Transportation	$40	4	$160	$161	164
20	Clothing	$21	4	$80	82	84

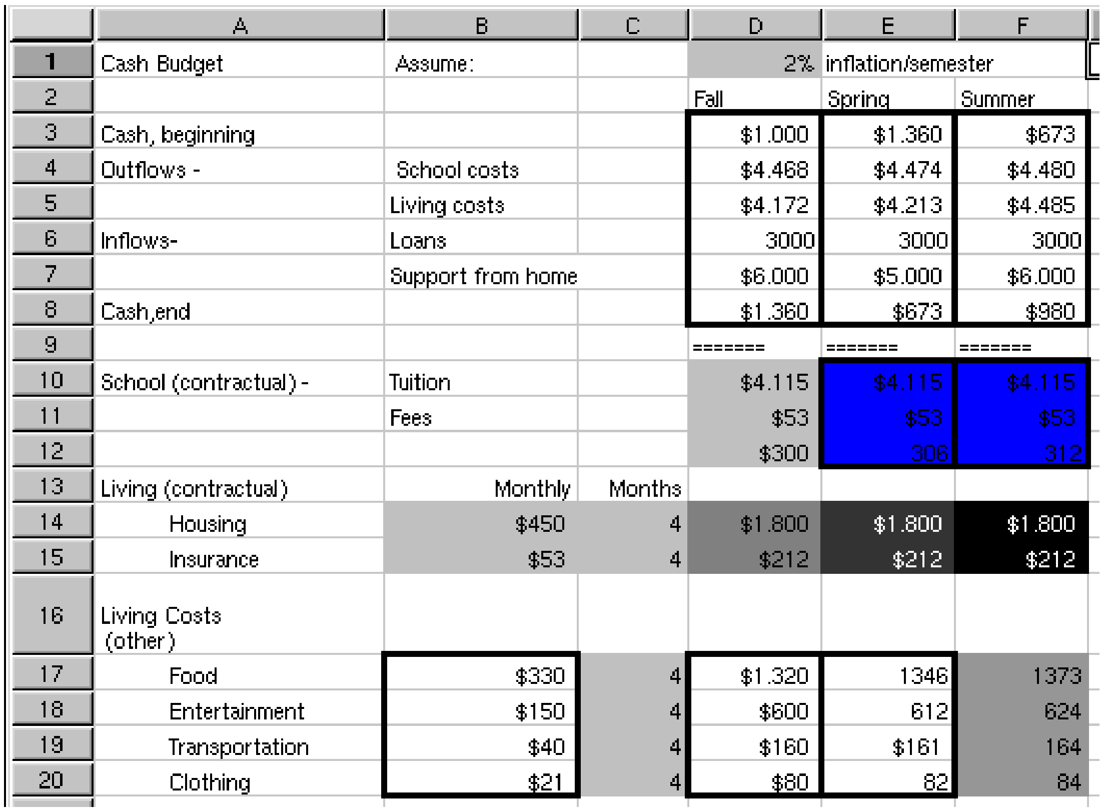
Figure 6.20: Visualization of semantic classes with $Eq_{Start,Rest} = se, \vec{d} = (0, 1, 1)$. Semantic units are framed with a thick border. Semantic units in the same class are shaded in the same gray-scale. Semantic units with singleton generators are not shaded at all. Not framed cells are in semantic units consisting of single cells.

If it were not for spreadsheet auditing, semantic classes are also a powerful tool for the maintenance of spreadsheet programs. Spreadsheet programmers can be supervised and warned, whenever a change to the spreadsheet program will change the structure of semantic classes and units, i.e. when the structure of the semantic building blocks of the spreadsheet program is likely to become blurred.

Although this technique promises to increase spreadsheet comprehension, it is not applicable without any special training any more. The spreadsheet auditor has to be aware of the impact of the decision for a certain distance vector and of the influence of the boundary parameter on the result.

Nevertheless, no special IT-terminology is necessary in order to train people, a thorough explanation of the geometrical terms is sufficiemt. Additionally, the result of the partitioning is well suited to be discussed with untrained domain experts, as it should correspond to their model of the spreadsheet program.

Thus, this auditing technique violates the claim for *no special vocabulary for spreadsheet auditing technologies* that has been stated in Section 3.4 (see Reich-



	A	B	C	D	E	F
1	Cash Budget	Assume:		2%	inflation/semester	
2				Fall	Spring	Summer
3	Cash, beginning			$1.000	$1.360	$673
4	Outflows -	School costs		$4.468	$4.474	$4.480
5		Living costs		$4.172	$4.213	$4.485
6	Inflows-	Loans		3000	3000	3000
7		Support from home		$6.000	$5.000	$6.000
8	Cash,end			$1.360	$673	$980
9				=======	=======	=======
10	School (contractual) -	Tuition		$4.115	$4.115	$4.115
11		Fees		$53	$53	$53
12				$300	306	312
13	Living (contractual)	Monthly	Months			
14	Housing	$450	4	$1.800	$1.800	$1.800
15	Insurance	$53	4	$212	$212	$212
16	Living Costs (other)					
17	Food	$330	4	$1.320	1346	1373
18	Entertainment	$150	4	$600	612	624
19	Transportation	$40	4	$160	$161	164
20	Clothing	$21	4	$80	82	84

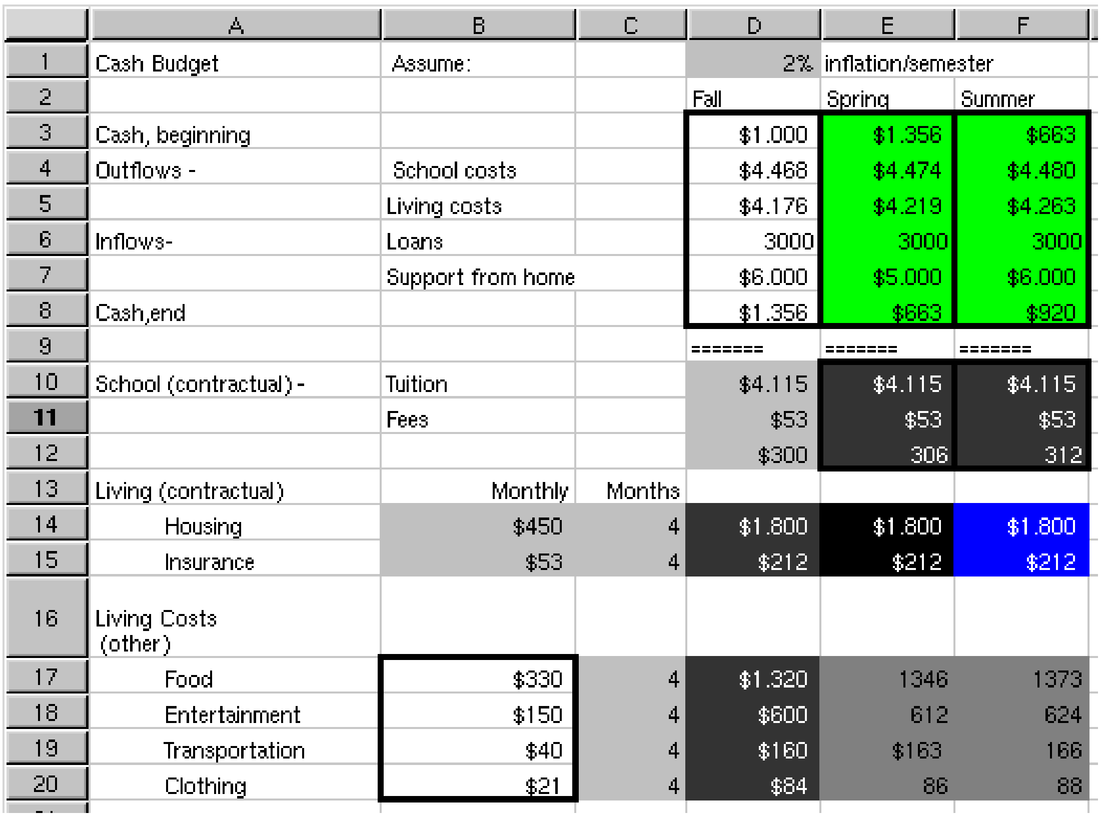
Figure 6.21: Visualization of semantic classes with $Eq_{Start,Rest} = ce, \vec{d} = (0, 1, 1)$.

wein et al. [RRB00]). For a more sophisticated spreadsheet user who creates and audits large spreadsheet programs, the support that is gained by the usage of semantic classes might be a motivation to have a look at the necessary extra vocabulary, that is still **non-IT** vocabulary.

However, the other criteria that have been summarized in Section 3.4, namely seamless integration and minimal overhead, are met by the prototype that implements the analysis technique suggested and is presented in Chapter 7.

As semantic classes build upon the concept of logical areas, they inherit the characteristic, that they are well suited for large, but regular spreadsheets, or for the regular parts of a given spreadsheet program. But they will not perform well with spreadsheets that have a rather irregular formula usage. In sheets like the one presented in Figure 6.1 on page 102, only the dark-grey shaded part will be efficiently analyzed with logical areas or semantic classes.

However, the bottom part (shaded in light-grey in Figure 6.1) is also very often subject to severe errors. Usually, due to the irregularity many unrelated logical areas are generated and the merging algorithm will not produce a useful result for this part. In order to overcome this drawback, a second abstraction technique that does not rely on the cells formulas at all, but operates directly on



	A	B	C	D	E	F
1	Cash Budget	Assume:		2%	inflation/semester	
2				Fall	Spring	Summer
3	Cash, beginning			$1.000	$1.356	$663
4	Outflows -	School costs		$4.468	$4.474	$4.480
5		Living costs		$4.176	$4.219	$4.263
6	Inflows-	Loans		3000	3000	3000
7		Support from home		$6.000	$5.000	$6.000
8	Cash,end			$1.356	$663	$920
9				=======	=======	=======
10	School (contractual) -	Tuition		$4.115	$4.115	$4.115
11		Fees		$53	$53	$53
12				$300	306	312
13	Living (contractual)	Monthly	Months			
14	Housing	$450	4	$1.800	$1.800	$1.800
15	Insurance	$53	4	$212	$212	$212
16	Living Costs (other)					
17	Food	$330	4	$1.320	1346	1373
18	Entertainment	$150	4	$600	612	624
19	Transportation	$40	4	$160	$163	166
20	Clothing	$21	4	$84	86	88

Figure 6.22: Visualization of semantic classes with $Eq_{Start,Rest} = ce, \vec{d} = (0, 1, 1)$. The spreadsheet program is the same as in Figure 6.21 on the page before, but this time the errors are corrected.

the spreadsheet program's *DDG*, is presented in the next section.

6.4 Data Modules

In order to get a useful abstraction of parts of a spreadsheet program where no regular usage of formulas occurs, a technique has to be introduced that does not rely on the similarity between formulas. An alternative approach is described in this section. This approach is meant for spreadsheet programs, that consist of complex calculations, but still show a regular data flow. Figure 6.1 on page 102 is structured like many other spreadsheet programs. Apart from a large part of repeated formulas, it consists of parts where formulas are used only once in order to calculate some final figures.

Obviously, patterns of formula usage are no suitable abstraction criteria in those described cases. However, in each formula there is still information about the data flow, or the data dependencies, in the spreadsheet program available. Subsequently, an abstraction technique that operates on the data flow graph of a spreadsheet program, i.e. the *DDG*, is introduced.

As it was discussed in Section 3.3, a spreadsheet program has some basic characteristics of data flow programs and of graph-reduction programs, too. Thus, the *DDG* of a spreadsheet program has an important role for its execution. As stated by Definition 13 on page 24, in the *DDG* each cell of the spreadsheet program is represented by a node, and there is an edge from node n_1 to node n_2, if the cell represented by n_2 references the cell represented by n_1. As the *DDG* is a directed, acyclic graph, there are some nodes, that are not sources of further edges, i.e. sink nodes.

Broadly speaking, a data module is a subgraph of the *DDG*, that has only a single sink node (see Definition 32). The sink node of such a data module is either a sink node of the *DDG*, i.e. a result cell of the spreadsheet program, or a node that is connected to more than one data module. To grasp the idea, one can assume that a data module is a set of cells that has a distinguished result cell, that is transitively dependent on all cells in the data module. Cells that are outside the data module may only reference its result cell.

However, before the *DDG* can be partitioned into such data modules, the result cells have to be identified. Obviously, not all sink nodes of the *DDG* have the semantics of a result of the spreadsheet program, e.g. check-sums.

Subsequently, data modules and their required properties will be formally defined. The identification of result cells and an algorithm for partitioning the *DDG* into data modules are discussed. Additionally, some auditing and visualization strategies will be briefly described and explained by an example.

6.4.1 Formal Definition

As stated above, a data module is a set of cells in the spreadsheet program that contribute to a specific result or intermediate result.

Definition 32: Data Module
Let (V, E) denote the sets of vertices and edges of a *DDG* of a given spreadsheet program. A data module d is a triple (V_d, E_d, n), with $V_d \subseteq V$, $E_d \subseteq E$ and $n \in V_d$ that fulfills the following requirements:

1. $domain(E_d) \subseteq (V \setminus \{n\})$
2. $domain(E \setminus E_d) \cap (V_d \setminus \{n\}) = \emptyset$
3. $n \in range(E_d) \vee \{n\} = V_d$
4. $range(E_d) \subset V_d$

□

In the above definition, a data module is defined as a triple of its member nodes, the edges and a result node. As a data module is defined as a part of the *DDG*, the nodes represent cells of a spreadsheet program. The four specified properties of a data module guarantee that

1. the result cell of the data module is not referenced by any other cell in the data module,
2. all edges with a source inside a data module are also part of the data module, only edges having their source in the result node are excluded,
3. the result of the data module is target of an edge in the data module, or the data module is a single cell, and
4. a cell in a given data module is only referenced by cells in the same data module.

Obviously, there are arbitrary subgraphs of the *DDG* that fulfill these requirements, e.g. each single cell can be considered a data module. However, in order to introduce an abstraction step, maximal data modules have to be identified.

Definition 33: Maximal Data Module
A data module $d = (V_s, E_s, n)$ in a $DDG = (V, E)$ is called a maximal data module, if $\nexists v \in V \cap V_s$ that can be added to d without violating any of the conditions that are required for a data module. $\square$

Hence, a data module is considered maximal if no other cell of the spreadsheet program (or no other node of the corresponding *DDG*) can be added.

Nevertheless, so far only the properties of individual data modules have been described. Additionally, there are some requirements that a valid partitioning of a given *DDG* into data modules has to fulfill. On the one hand, it has to be guaranteed that the union of all data modules is the original *DDG*, i.e. all nodes and edges of the *DDG* have to take part in one data module. On the other hand, no node of the *DDG* is allowed to be in more than one data module. As the *DDG* on its own, with cells as singleton data modules, already fulfills these properties, the data modules have to be maximal.

For each *DDG* there is one partitioning into data modules that fulfills the above stated requirements. This kind of partitioning is called a *valid partitioning.*

Definition 34: Valid partitioning
The set D_{Mod} of data modules over a $DDG = (V, E)$ of a given spreadsheet program is a valid partitioning if it fulfills the following requirements:

1. $\bigcup\{V_s | \exists n, E_s \bullet (V_s, E_s, n) \in D_{Mod}\} = V$
2. $\bigcup\{E_s | \exists n, V_s \bullet (V_s, E_s, n) \in D_{Mod}\} = E$
3. $\forall (V_1, E_1, n_1) \in D_{Mod}, (V_2, E_2, n_2) \in D_{Mod} | V_1 \neq V_2 \bullet V_1 \cap V_2 = \emptyset$
4. $\forall d \in D_{Mod} | d$ is maximal

$\square$

The first and the second property ensure that all nodes and edges of the DDG are assigned to at least one data module in D_{Mod}. The third requirement prevents a node from being assigned to two data modules. The last requirement ensures that all the data modules in a valid partitioning have to be maximal data modules.

6.4.2 Identifying Data Modules

The identification of the data modules in a given DDG starts with the removal of unnecessary sink nodes, to eliminate check-sums and other calculations that do not have the semantics of a result of the spreadsheet program. This step is necessary, because very often all sections of a spreadsheet are connected by some calculations that either yield a final sum or a check-sum. However, in this case each cell of the spreadsheet program is transitively referenced by the final sum and there will be only one data module.

Thus, the sink nodes of the DDG have to be checked by the users who decide if a sink node should be removed or not. The users have to prune the DDG until all the sink nodes are results of the spreadsheet program. All further consideration in this section concern the pruned DDG.

Of course it can be argued that each cell in the spreadsheet program is visible and therefore a result of the spreadsheet program. However, it is important to distinguish between intermediate results that might be only introduced in order to get insight into the calculation or to rewrite a formula in a simpler way, and the final result, that is placed on the spreadsheet because the spreadsheet users really want to see it. Obviously, all sink nodes of the DDG, i.e. cells that are not referenced by any other cell any more, are placed on the spreadsheet because the spreadsheet users want to have them there.

The algorithm for the identification of data modules is very straightforward. It operates on the pruned DDG and starts at the sink nodes. Each sink node is considered a data module. Nodes that are the source of edges that target only to one data module are merged with the specific data module. A node that is a source of edges into more than one data module is considered a data module on its own. These steps are repeated until all nodes in the *DDG* are assigned to a data module (see Table 6.4 on the facing page).

As a data module consists of nodes and edges, all edges targeting to a node in the data module have to be added. The result node of a data module is not obvious after the application of the algorithm, but the identification is very simple, as it is the only sink node in a data module.

```
Algorithm PartitionDDG: (pruned $DDG$ d)
return Set of Data Modules
declare
    nodeset $V,\ DM,\ JOIN$
    node $v,\ v',\ cur$
    edgeset $E$
    edge $e$
    $\mathbb{P}$ node result
    Integer amount
    Boolean $found$ = true
begin
    $result = \emptyset$
    $(V, E) = d$
    while ($found$)
            $found$ = false
            for $v \in V$
                    if $\nexists v' \in V \bullet (v, v') \in E$
                      $cur = v$
                      $found$=true
                      break
                    end if
            end for
            $amount = 0$
            $\forall DM \in result$
                    if $\exists (cur, v) \in E \bullet v \in DM$
                      $JOIN = DM$
                      $amount = amount + 1$
                    end if
            end for
            if $amount = 1$
                    $result = result \setminus JOIN$
                    $JOIN = JOIN \cup \{cur\}$
                    $result = result \cup JOIN$
            else
                    $result = result \cup \{cur\}$
            end if
            $V = V \setminus cur$
    end while

    return $result$
end
```

Table 6.4: Partitioning of a pruned *DDG* into node sets of data modules

6.4.3 Auditing Strategies Based on Data Modules

In this section, two auditing strategies and a strategy for fault tracing that employs data modules are discussed. The first auditing strategy, *pattern driven auditing*, examines the spatial pattern of cells in a data module. There is also an SRG_{DM} driven auditing strategy that assumes that an SRG_{DM} with data modules as nodes is easier to comprehend than the DDG. The fault tracing strategy relies on the property of data modules that they have only one distinguished result node.

Each of the strategies is explained by means of the example spreadsheet that has already been used in the previous section. The DDG of the example spreadsheet is shown in figure Figure 6.23 on page 154 and consists of 57 nodes and 91 edges. The sink nodes are shaded gray.

Pattern Driven Auditing

Spatial auditing is similar to the pattern driven auditing strategies that can be used together with the logical area and semantic class approaches. The identified data modules are presented to the spreadsheet auditors in a way that is similar to the structure browser. The selection of a data module in the structure browser will highlight the member cells in the spreadsheet.

As it was pointed by Chan et al. [CC00, Cha01], the data flow in a spreadsheet program will correspond to the geometrical layout of the spreadsheet, and major deviations between data flow and layout often indicate errors.

As there is no check-sum in the example spreadsheet, the DDG is not pruned. The data modules identified are shown in Figure 6.24 on page 155. In Figure 6.25 on page 156 the cells that are in the same data module are colored in the same shade of gray.

Obviously, the whole spreadsheet is member of the data module, the result cell of which is `F8`, except for the last row. Major irregularities can be identified in the last row:

- `B20` is not referenced by any data module, although all the other cells in the column are. This is a symptom of an erroneous formula in `C20`.
- `F20` is a sink node. This is a symptom of an erroneous formula in `F5`.

In the second case, an important feature of data modules is exploited for auditing. If a planned cell reference is not part of a formula, the data module will split up in two different modules. In the opposite case, a cell reference that should not be part of a formula, might lead to the merge of to unrelated data modules.

Hence, auditors have to watch out for superfluous data modules. Subsequently, the cell where the result of the superfluous data module should have been referenced has to be identified and corrected. The opposite case is more difficult. If an expected data module is not part of the visualization, auditors have to look for the cell where the missing data module is erroneously referenced. Although fault tracing is more troublesome, the presence of an error can be easily detected.

The other errors in the example cannot be identified by this approach. Wrong operators, e.g. in `E19` or mis-references to cells in the same data modules, will influence the result of a data module, but not the assignment of cells to a data module, as only the data dependencies are taken into account.

SRG_{DM} Driven Auditing

The SRG_{DM} driven auditing assumes that the SRG_{DM} of a given spreadsheet program is more comprehensive than its DDG. In the SRG_{DM} each data module is represented by a single node, that is labeled with the cell address of its result cell.

However, the SRG_{DM} of large spreadsheet programs is still large and it takes some effort to comprehend and test it. Additionally, data modules are not appropriate for all kinds of spreadsheet programs. Hence, some analyzed spreadsheet programs, e.g. the example spreadsheet discussed here, generated very compact and comprehensive SRG_{DM}, e.g. the SRG_{DM} of this spreadsheet has 5 nodes and 7 edges (see Figure 6.26 on page 156), whereas other spreadsheet programs yield very large SRG_{DM}s, e.g., an experimental spreadsheet program with an DDG with 255 nodes and 332 edges was decomposed into data modules resulting in a SRG_{DM} with 169 nodes and 246 edges. In the example spreadsheet SRG_{DM} indicates two result nodes, i.e. `F20` and `F8`. As the spreadsheet should have only a single result in `F8`, `F20` is obviously an superfluous data module. The auditor has to check, why the clothing (`F20`) does not influence the living costs (`F5` in the data module `F8`). The other data modules might be generated because of the superfluous data module `F20`.

A superfluous data module often entails other superfluous data modules, as the definition of a data module defines a data module as a set of cells that influences only one result or intermediate result. Hence, the presence of a broken link in the spreadsheet program due to a missing cell reference, will lead to other superfluous data modules. In the case of the example, the cell `F20` is a result cell due to a missing reference. `E20` will be identified as a data module on its own, because it is referenced by `F20` and transitively referenced by `F8`. Hence, the correction of the error that generated the data module `F20` will also wipe out all the other data modules, except `F8`.

This is also a drawback of the SRG_{DM} driven auditing technique, as it was only able to find a single error out of five. Again, wrong operators or mis references within a data module cannot be detected. However, it supports the finding of broken and superfluous links between otherwise unrelated cells on the spreadsheet, as even a single misreference, e.g. not referencing `F20`, has severe consequence in the visualization, e.g. four superfluous data modules.

Fault Tracing

Fault tracing is a very common problem in spreadsheet programs, as the symptoms of errors often do not occur at the same place as the faults that cause the wrong results. Hence, most testing techniques also involve techniques for fault tracing that are usually based on the calculation of error probabilities for the predecessors of the faulty cell in the spreadsheet programs *DDG* (see e.g. Ayalew [Aya01] or Reichwein et al. [RRB00]).

The generation of data modules and the usage of the SRG_{DM} are powerful helps for fault tracing. If an error is detected in the result cell of a data module, it is not necessary to check all the predecessors in the *DDG* until the error is found. If the spreadsheet auditor is aware of the data module where the symptom of the error occurred, there are only two possibilities:

1. The error occurred inside the data module where it is detected, or
2. the error occurred in a predecessor module in the SRG_{DM}.

It is not difficult to decide on which case applies: the spreadsheet auditor has to check the result cells of the predecessor data modules in the SRG_{DM}. If they are correct, the error is buried in the module where the failure occurred. Else it is assumed that the error is propagated from the erroneous module.

For the first case, the *DDG* of the data module where the failure occurred has to be checked by one of the techniques that are suggested in [Aya01, RRB00]. Nevertheless, as a piece of extra information, the auditors are aware that the error must be in the currently examined subgraph of the *DDG*, and the bug tracing can stop at the module boundaries.

In the second case, the same process is repeated: it has to be checked, whether the fault occurred inside the data module, or in one of its predecessor modules. Depending on the error source, either the module is checked, or the search continues upward in the SRG_{DM}.

Obviously, also a combination of error sources is possible, as errors can be hidden inside the module as well as in several predecessor modules. Nevertheless, an iteration of several testing and correction phases will finally find all the errors.

6.5 Discussion

In this chapter two orthogonal approaches for auditing spreadsheet programs have been introduced. The first approach is based upon logical areas that can be used to group similar formulas into abstract units. The advantages of logical areas compared to existing spreadsheet visualization techniques (see Section 5.5), that often group equal formulas into abstract units, are:

- The are varying degrees of similarity. Copy-equivalence, i.e. equality, is only the strongest one.

- The spatial relation among the cells in a logical area is not taken into account at first. Thus, also patterns of similar cells that are spread throughout the spreadsheet UI can be exploited for auditing.

A more powerful abstraction technique, based on logical areas, are semantic classes that group similar cells with similar neighbors into an abstract unit, the so-called semantic unit.

Nevertheless, in a field audit (see Section 4.3) it was shown that both logical areas and semantic classes are only applicable to spreadsheets or parts of spreadsheets with regular formula usage. As this is the case for at least large areas of vast business spreadsheet applications, this constraint is not a very severe limit for the approach presented.

Logical areas are very straightforward and can be easily communicated to the application experts that are often also the spreadsheet programmers. Generally, no special IT-skills are necessary to understand the concept of logical areas. The more sophisticated concept of semantic classes requires some special skills from the auditors in order to choose helpful values for the parameters. However, the technique can still be explained without IT-vocabulary only in terms of the two-dimensional tabular user interface.

It is assumed that spreadsheet users who notice that the concept of logical areas is not sufficient for auditing their vast spreadsheet programs, often will be willing to take some extra effort into account. Additionally, there are many spreadsheet consultants who are domain experts with special skills in spreadsheet testing and auditing. For them, special training is no obstacle.

An additional approach that is based on the data dependencies of a spreadsheet program is introduced with data modules. Data modules do not require any regular formula usage in the regions of the spreadsheet to be analyzed, as they are only based on the *DDG*. Thus, the strength of data modules is to find *broken links* in a spreadsheet program, i.e. parts of the spreadsheet programs that are not referenced by a final result, although they should be.

For very large spreadsheet programs, none of the approaches presented can generate a comprehensive abstraction of the spreadsheet program in isolation. Nevertheless, the combination of two approaches, or by iterative re-applications of an abstraction technique, even more abstract representations can be generated. Subsequently, some possible combinations of the approaches are presented:

Iterative Semantic Classes: The input for the partitioning of a spreadsheet program into semantic classes is the assignment of cells to logical areas. According to the definitions in Subsection 6.3.1, a cell is a singleton semantic unit and a logical area is a semantic class. Thus, the partitioning can also be performed with semantic units that consist of more than one cell. In this case, the number of semantic classes will steadily decrease, until the desired level of abstraction is reached.

Iterative Data Modules: The input for the partitioning of a spreadsheet program into data modules is the DDG. The output will yield a SRG, i.e. a directed, acyclic graph. Obviously, the same partitioning as with the DDG can be performed with the SRG, too.

Data Modules and Semantic Classes: A spreadsheet program often consists of several large parts, each of them having its own (intermediate) result. By generating data modules at first, the spreadsheet program can be separated into smaller and more comprehensive units. These units can than be further analyzed by the application of logical areas or semantic classes.

Each of these discussed approaches assumes that it is possible, to open an abstract unit and access its member units for further analysis, i.e. to access the cells in a data module for the application of semantic classes or the data modules in an iteratively generated data module in order to see the member cells on the spreadsheet UI. This resembles the application of fish-eye views that have already been discussed in Subsection 6.3.3.

Summary

The following issues have been addressed in this chapter:

- There is a large number of spreadsheet programs that consist of a large part of repetetive formula usage and an irregular part for the calculation of check-sums.
- Three different abstraction or auditing approaches have been presented: Logical Areas, Semantic Classes and Data Modules.
- A logical area is a set of cells with similar formulas.
- There are varying degrees of similarity that are either based on the operators (node equivalence classes) or on the referenced cells (link equivalence classes).
- There are no spatial constraints for the assignment of cells to logical areas.
- A semantic class is a set of similar cells with similar neighbors.
- There are three auditing and visualization strategies for logical areas and semantic classes:
  - pattern driven,
  - SRG-driven, and
  - structure driven auditing.

- A data module is a set of cells that is referenced only by one (intermediate) result of the spreadsheet program.
- There are two auditing and visualization strategies for data modules:
  - Pattern driven, and
  - SRG-driven.
- Data modules support fault tracing and testing of spreadsheet programs.
- Combinations of the abstraction steps, or iterative application of one abstraction step are possible.

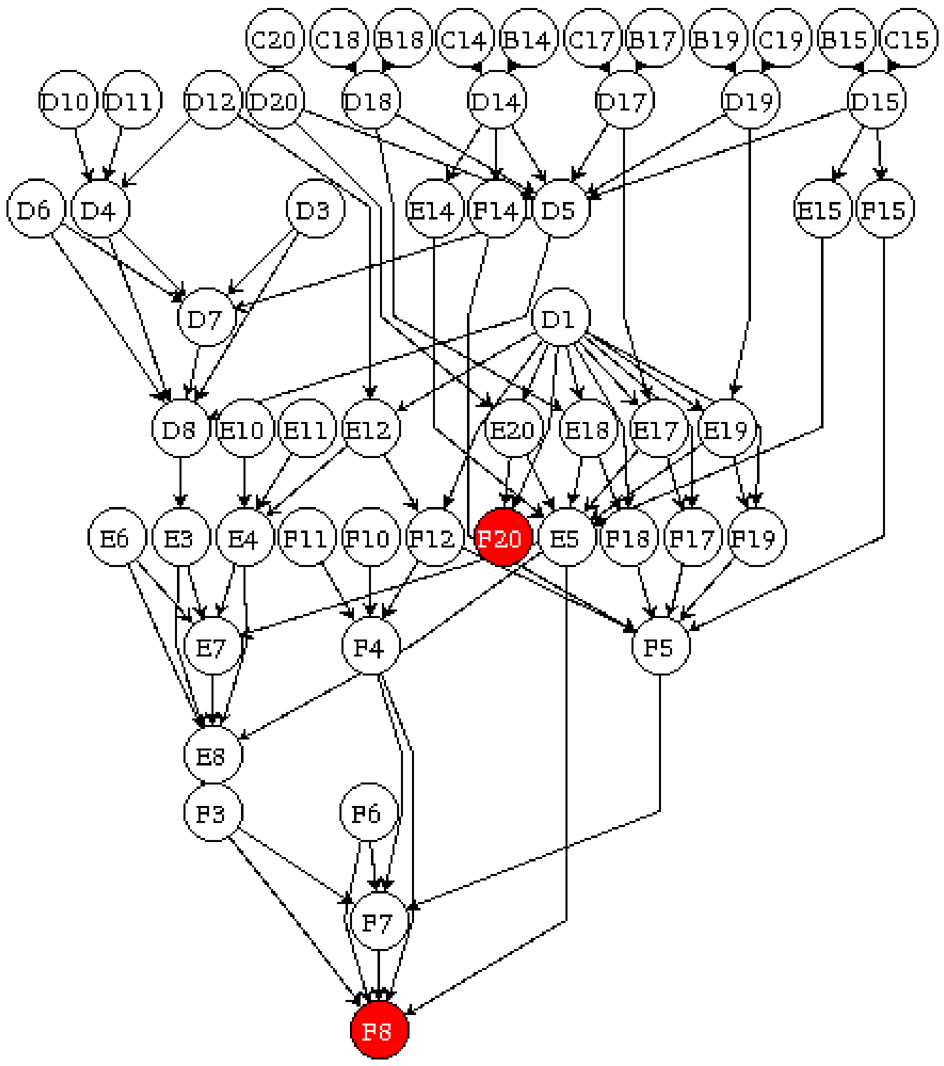
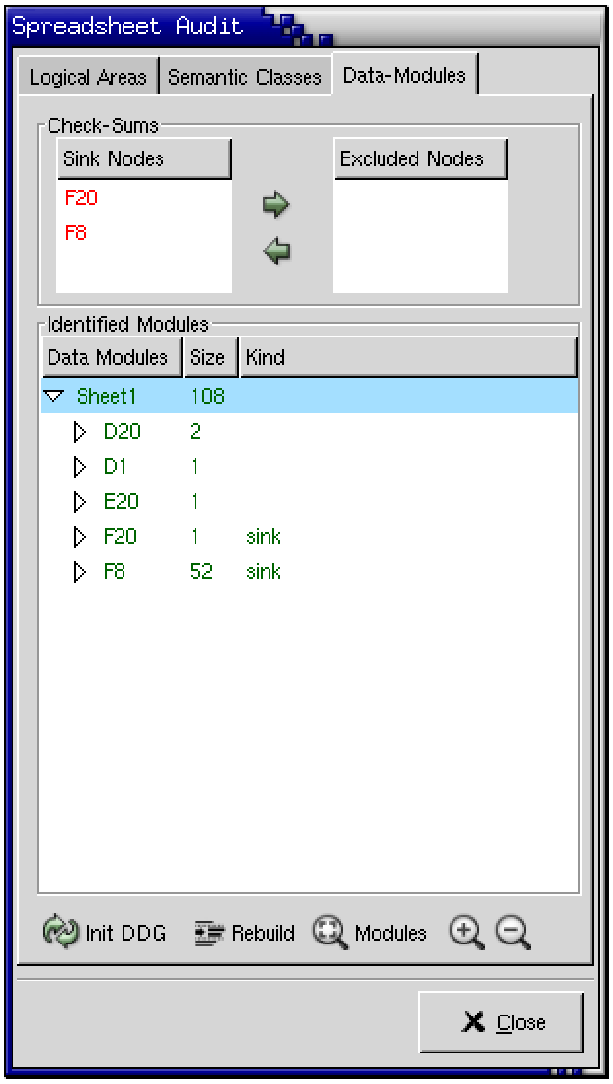


Figure 6.23: *DDG* of the example spreadsheet

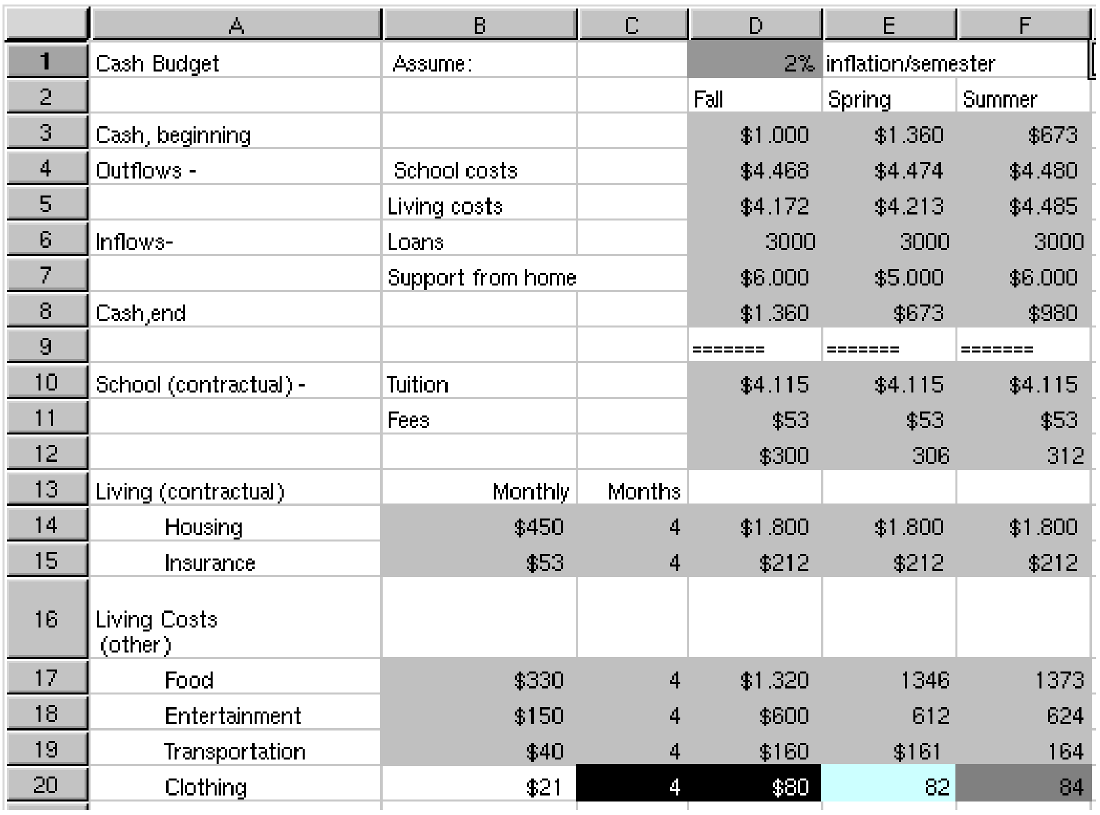


Sink Nodes	Excluded Nodes
F20	
F8	

Identified Modules

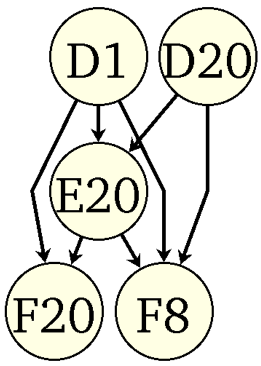
Data Modules	Size	Kind
Sheet1	108	
D20	2	
D1	1	
E20	1	
F20	1	sink
F8	52	sink

Figure 6.24: Valid partitioning in the structure browser



	A	B	C	D	E	F
1	Cash Budget	Assume:		2%	inflation/semester	
2				Fall	Spring	Summer
3	Cash, beginning			$1.000	$1.360	$673
4	Outflows -	School costs		$4.468	$4.474	$4.480
5		Living costs		$4.172	$4.213	$4.485
6	Inflows-	Loans		3000	3000	3000
7		Support from home		$6.000	$5.000	$6.000
8	Cash,end			$1.360	$673	$980
9				=======	=======	=======
10	School (contractual) -	Tuition		$4.115	$4.115	$4.115
11		Fees		$53	$53	$53
12				$300	306	312
13	Living (contractual)	Monthly	Months			
14	Housing	$450	4	$1.800	$1.800	$1.800
15	Insurance	$53	4	$212	$212	$212
16	Living Costs (other)					
17	Food	$330	4	$1.320	1346	1373
18	Entertainment	$150	4	$600	612	624
19	Transportation	$40	4	$160	$161	164
20	Clothing	$21	4	$80	82	84

Figure 6.25: Valid partitioning of the *DDG* projected back to the spreadsheet UI by coloring cells in the same data module in the same gray-shade



$$\begin{tikzcd}
D1 \arrow[d] \arrow[dd, bend right] \arrow[ddr] & D20 \arrow[dl] \arrow[dd] \\
E20 \arrow[d] \arrow[dr] & \\
F20 & F8
\end{tikzcd}$$

Figure 6.26: SRG_{DM} of the example spreadsheet program

Chapter 7

The Model Visualization Toolkit

The spreadsheet visualization techniques that have been introduced in the previous chapter offer valuable support for spreadsheet auditing and spreadsheet comprehension. However, partitioning a spreadsheet into semantic classes or data modules is very time consuming if it has to be done manually. Thus, a tool that is integrated in a spreadsheet system and that is able to identify and visualize logical areas, semantic classes and data modules is necessary.

In this chapter the prototype implementation of a spreadsheet visualization toolkit is discussed. The toolkit has been developed as a plug-in for the *Gnumeric* spreadsheet system that is integrated into the `GNOME`-Desktop Environment for *Linux*. *Gnumeric* has been chosen as the host spreadsheet system because the source code is available, and thus a very smooth integration of the visualization toolkit and spreadsheet system was feasible.

In the first section, the development and run time environment of the visualization toolkit are introduced and the advantages of the *Gnumeric* spreadsheet system compared to *Excel* as the host spreadsheet system for the visualization toolkit are discussed. In the next section, the implementation and the functionality of the prototype are explained in detail. Finally, the limits and desirable improvements for the prototype are outlined. A more detailed description of the required runtime environment and installation guidelines for the visualization toolkit and the *Gnumeric* spreadsheet system are given in Appendix A.

7.1 Environment

Although *Excel* is currently the most popular spreadsheet system, *Gnumeric* was chosen as target system for the prototype of the spreadsheet auditing toolkit. Due to the fact that *Gnumeric* is subject to the *GNU*-public license (see [gnu03] for the details), the source code is public domain, and can thus be modified and used freely.

In contrast, the *Excel* source code is not available at all. *Visual Basic for*

Applications (VBA) offers extensive functionality to increase the functionality of *Excel*. It turned out that a first prototype of the auditing toolkit that was written with *VBA* had massive performance problems, because in order to assign cells to logical areas, the abstract syntax trees of the attached formulas have to be compared. As the *Excel* formula parser is not accessible from *VBA*, each formula had to be parsed again by a parser that used to be part of the prototype initially.

7.1.1 Advantages of *Gnumeric*

Gnumeric plug-ins can be developed in any programming language that seems appropriate. They just have to register themselves to the spreadsheet systems by means of a specific plug-in API. As *Gnumeric* loads plug-ins dynamically into its address space at runtime, plug-ins can access all internal functions and data structures of the spreadsheet system[1]. Thus, not only the integration of a parser into the auditing toolkit becomes superfluous, but also the parsing of individual cells for comparing abstract syntax trees needs not be done by the add-on, because each formula is parsed as soon as it is entered and the abstract syntax tree is stored in an internal data structure. Hence, the already stored abstract syntax tree can be used for comparing formulas.

The performance of the prototype had been heavily improved by reimplementing it in the *Linux* and *Gnumeric* environment, because

1. runtime performance of `C` is superior to VBA and

2. formulas do not have to be parsed at analysis time.

Further advantages concerning the accessible internal functions of the *Gnumeric* spreadsheet system have supported the decision. These internal functions, e.g. parsing cell references of the `A1` style and converting them to the `R1C1` style, managing ranges and evaluating specific cells, saved a lot of development time and eliminated several potential sources of errors.

7.1.2 Architecture of the Prototype

The prototype has been implemented as a plug-in for the gnumeric spreadsheet system. Therefore, part of it is also an `XML`-interface required by *Gnumeric*, and that is loaded when *Gnumeric* is started. In the `XML`-interface file, the position of a menu item or a button in the *Gnumeric* main menu or task bar is specified. Additionally, the location of a shared library and the name of an initialization routine are specified that are loaded and invoked whenever the corresponding menu item or button is selected by the user.

[1]Another important side-effect of this technique is that plug-ins also have to be subject to the *GNU*- public license.

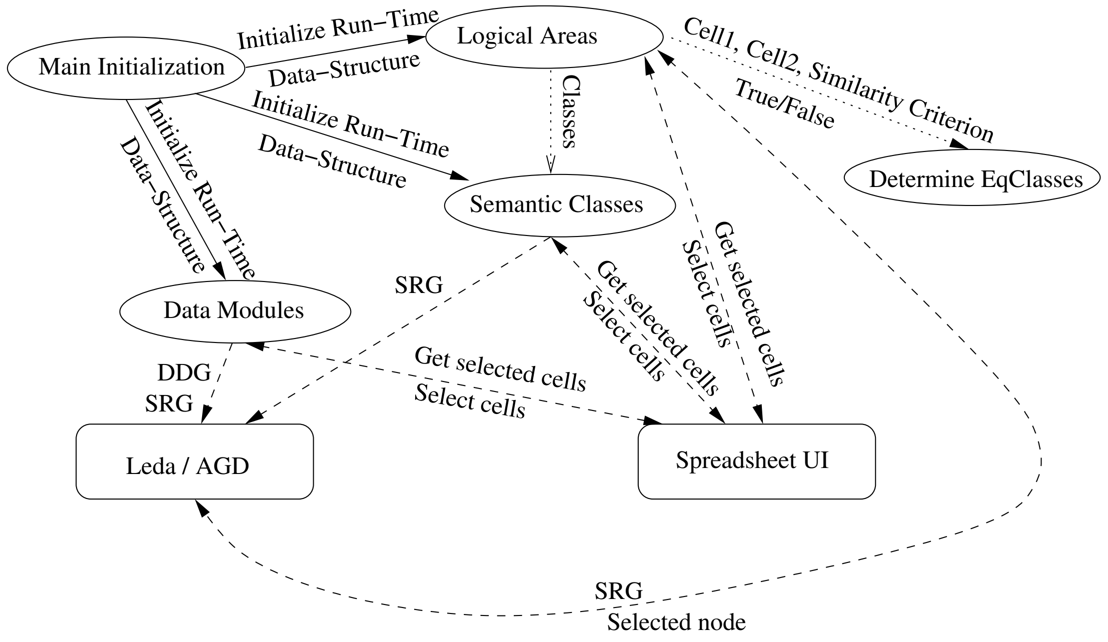


Figure 7.1: The architecture of the prototype

The prototype itself consists of four components that are only loosely connected. Each of the components uses functionality that is supplied by the spreadsheet UI, e.g. retrieving the selected cells or selecting cells. The components also use functionality that is provided by *LEDA* and *AGD* for visualizing graphs.

The initialization routine runs the plug-in's main dialog and connects the buttons with callbacks that will invoke specific analysis functionality. For each callback routine, a data structure is initialized that contains references to the active spreadsheet instance and all dialog elements that are needed to properly display the results of the analysis. Further initialization, e.g. defining callbacks for the tree-views, is performed when a given analysis functionality is invoked.

In Figure 7.1, the main components of the visualization toolkit and their interconnections are shown. The component that partitions the spreadsheet into logical equivalence classes is supported by a component that is capable of determining similarities between cells. This component is labeled *Determine EqClass*, but some additional auxiliary functions for the administration of logical areas are implemented there, too. The component *Logical Areas* controls the user interface, invokes the graph visualization of a *SRG* and is also able to highlight certain cells in the spreadsheet UI.

The selection of a node in the graph window of *AGD*[2] will invoke a callback in this component that leads to the selection of the corresponding entry in the tree-view and the spreadsheet UI. In order to improve the run time performance this

[2]As *LEDA* and *AGD* have been chosen for the visualization of *SRG*s and *DDG*s, the *AGD* graph window is denoted by the term *graph window* through the remaining chapter.

components also maintains several indices, e.g. to efficiently track the assignment of a given cell to a logical area and vice-versa.

These indices are also accessed by the *Semantic Classes* component. The component controls the user interface, invokes a graphical visualization of the *SRG*, and implements the merging algorithm that is presented in Subsection 6.3.2. The result of the merging algorithm is presented in a tree-view, semantic units and semantic classes that are selected in the tree-view are highlighted in the spreadsheet UI. As this component accesses the runtime indices of the *Logical Areas* component, they must be initialized, i.e. the *Logical Areas* component has to be invoked prior to the *Semantic Classes* component.

The *Data Modules* component is independent of the previous components. It uses only the spreadsheet UI for highlighting selected cells and several graph functions that are implemented by *LEDA*[3] for partitioning the *DDG* into data modules. At first, the *DDG* is visualized in the graph window. The user can delete sink nodes in the graph window before the data modules are computed. Additionally, this component supports *fish-eye views*.

In the next section, each of the components that have been introduced here are discussed in detail.

7.2 Functionality and Implementation

As the plug-in consists of three more or less independent main components (see Figure 7.1 on the preceding page) and some support libraries, the main components *logical areas*, *semantic classes* and *data modules* will be discussed. For each component the initial data structure that is passed by the plug-in's main function to the callback that invokes the component is explained, a brief insight into implementation specific details is given and some special features are presented.

7.2.1 Component *Logical Areas*

The component for partitioning a spreadsheet into logical areas is stored in files named `visualization.h` and `visualization.c`. The initial data structure is shown in Table 7.1 on the next page.*sheet* is a reference to the currently opened spreadsheet instance, *tree_structure* is a reference to the tree-view[4] in the user interface (see Figure 7.2 on page 163). The plug-in's initial routine connects the callback function `buildEquivalence` to the button labeled *Analyze*, `cb_expand` is the callback routine of the *Graph*-button.

[3]Originally, the open-source graph visualization toolkit DOTTY (see Ganser et al. [GN99]) was used for graph visualization. Due to insufficient integration and performance problems with DOTTY, *LEDA* was chosen. *LEDA* is not subject to the GNU public license.

[4]In the description of the auditing technique given in Chapter 6 the tree-view is identified by the term *structure browser*.

```
typedef struct {
    Sheet *sheet;
    GtkTreeView *tree_structure;
} CB_Data;
```

Table 7.1: Initial data structure of *Logical Areas* Component

Implementation Specific Details

The identification of logical areas is performed bottom-up. At first empty cells, label cells and cells with a numerical constant are filtered out and are assigned to separate logical areas. This reduces the remaining base of cells to be a traceable small set of cells. Consecutively, it is tried to merge individual cells to copy equivalence classes.

Therefore, the following steps are performed:

```
1. For each cell $c$
   (a) For each existing copy equivalence class $ce$:
       • If $c$ is copy equivalent to the formulas of cells in $ce$:
         – Add $c$ to $ce$.
         – Break.
   (b) If $c$ has not been assigned to any copy equivalence class:
       • Create a new copy equivalence class $ce_n$
       • Add $c$ to $ce_n$
       • Add $ce_n$ to the set of existing copy equivalence classes.
```

Obviously, this algorithm will create copy equivalence classes for cells with unique formulas, too. However, this side effect is desirable, as

- cells with unique formulas will be the only member of their copy equivalence class, and
- further analysis does not consider individual cells, but only classes of copy equivalent cells.

Consecutively, a similar algorithm is used to merge copy equivalence classes into the weaker logical equivalence classes. As mentioned above, the comparisons are not made on a cell-by-cell level, but on the level of logical areas. As the comparison algorithm has a worst-case complexity of $O(n^2)$, this additional layer of hierarchy drastically reduces the number of necessary comparisons between abstract syntax trees. In the example spreadsheet discussed in the previous chapter a reduction from 3,136 (on a cell-by-cell level) to 225 comparisons was observed.

```
typedef enum {
  NONE,
  COPY_EQUIV,
  LOGICAL_EQUIV,
  STRUCTURAL_EQUIV,
  SOURCE_EQUIV,
  SINK_EQUIV,
  STRING_CONST,
  NUM_CONST,
  BLANKS,
  ALL,
  FORMULA,
  CELL} EClass;

typedef struct {
  EClass type;
  GnmExpr *pattern;
  EvalPos  *pos;
  GPtrArray *members;
  GPtrArray *ranges;
  gint  id;
} EquivClass;
```

Table 7.2: The data type *EquivClass* conveniently handles logical areas.

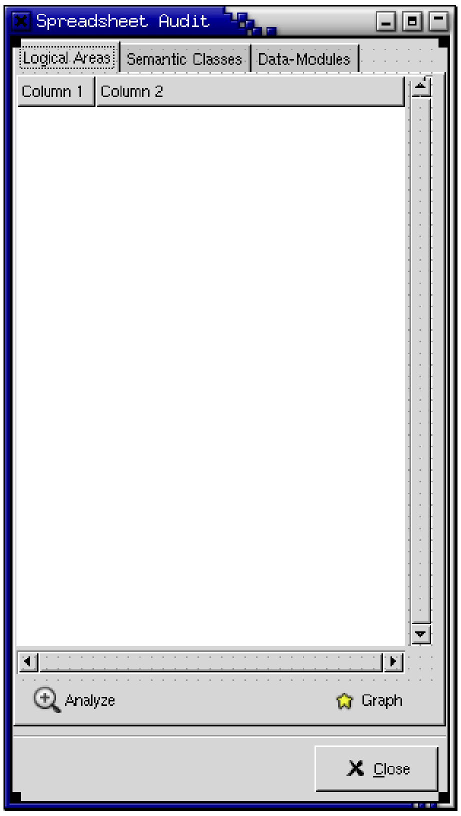


Figure 7.2: The user interface of the *Logical Areas*-Component

A third pass is performed in order to merge the logical equivalence classes into the even weaker structural equivalence classes.

Thus, the process of identifying logical areas will at first filter out all non-formula cells to reduce the base of cells that have to be considered and increase the runtime performance of the algorithm. The next steps will further reduce the number of elements the merging algorithm has to consider, as individual cells are merged to copy equivalence classes and in the next step, these are again merged into logical equivalence classes.

In order to handle logical areas efficiently, a data type *EqivClass* is defined (see Table 7.2 on the preceding page). This data type stores a unique identifier, the member cells, a pattern, and the kind of a logical area.

Depending on the degree of similarity, each logical area is stored in a data structure, e.g. there is a list of all structural equivalence classes.

As logical areas consist of cells that are pairwise equivalent regarding a specific equivalence criterion, it is sufficient to compare a formula to one member of the logical area to decide whether a formula can be added to a logical area. Therefore, for each logical area a pattern, i.e. the formula-expression of an arbitrary member cell[5] is stored. When a logical area is compared to a cell, the algorithm is invoked with the pattern and the cell's formula-expression. The comparison of two logical areas is done by a comparison of their patterns.

In detail, the algorithm recursively descends through the nodes in the abstract syntax tree of the two compared expressions. If the two nodes deviate, the algorithm will return `false`, otherwise the children of the nodes are pairwise compared. If it is checked for logical equivalence, the algorithm will ignore the children of nodes that represent constants or absolute cell references. If it is checked for structural reference, the children of all nodes, that are of type absolute or relative cell reference or constant value are ignored.

In contrast to the partitioning of the spreadsheet into logical areas, the visualization of the logical areas in the tree-view is generated top-down. The first and the second level of the tree-view are statically generated, with the root item representing the whole spreadsheet instance. On the second level, three special logical areas are inserted in order to increase readability:

Numerical Cells represents all cells that contain a numerical value,

Label Cells represents cell that contain a string value, and

Formula Cells contains all cells that contain a valid formula expression.

Numerical and label cells are identified irrespective of their usage, i.e. they can be computationally dead or not. The **Formula Cells** node is the root node for further items that correspond to logical areas. The list of all structural equivalence classes contains the child items of the *Formula Cells* node. For each structural equivalence class, those logical equivalent logical areas that it is built up from will be the child items. The same process is repeated for logical equivalence classes and copy equivalence classes and for copy equivalence classes and single cells.

The aggregation algorithm generates equivalence classes with only a single member (e.g. a formula that is not structurally equivalent with any other formula in the spreadsheet program will lead to the generation of a copy-, a logical- and a structural equivalence class, that have only one member), too. As these uniquely occurring equivalence classes will only puzzle the users without bearing any extra information they can be considered as noise and have to be filtered out in the visualization.

Hence, only equivalence classes with more than one member[6] are left out in the tree-view.

In order to be able to determine quickly the membership of a given cell in a logical area, or the logical areas that a cell is member of, there are two indices:

[5]In the current implementation, the pattern is the formula of the first cell that has been added to the logical area.

[6]In this context a *member* is a logical area with a stricter similarity criterion. Members of copy equivalence classes are cells.

Reverse index is a balanced tree, with the cell coordinates as key and a list of the logical areas, that the cell is member of as entry, and

Visible nodes is a balanced tree, with the *id* of a logical area as key and an array of the member cells as entry. Visible nodes contains only those logical areas that are displayed in the tree-view. The index is maintained by a callback function that is attached to the *item-expanded* event of a tree-item.

The indices are mainly used to give the user feedback of the partitioning of cells into logical areas by generating a *SRG* and by selecting the member cells of a logical area in the spreadsheet UI. The second task is performed by attaching a callback function to the *selection-changed* event of the tree-view. To each tree-item a data structure that contains reference to the represented cells is attached as so-called *user-data*. Whenever the user makes a selection, all cells in the pointer array attached to the selected tree-item are selected.

The user can also request a *SRG* with logical areas as nodes. Therefore, the logical areas that are currently visible in the tree-view as leafs will be nodes. The assembling of the *SRG* is complicated, as it consists of several look-ups in the index data structure:

1. The *visible nodes* index has to be traversed for leaf-nodes, i.e. nodes that do not have any children that are currently visible in the tree-structure. As the index consists of nodes that are logical areas, their *id* and the member cells can be accessed.

2. To generate the edges in the *SRG*, for each member of the logical areas that were selected in step one,

   (a) their dependents have to be figured out,

   (b) for each dependent,

      i. the logical areas that it is member of have to be looked up in the *reverse index*,

      ii. the logical area that is a node in the *SRG* has to be selected by checking the *visible nodes* index again,

      iii. an edge is inserted between the two corresponding *SRG*-nodes.

Integration from the Users' Point of View

In order to meet the postulated requirement of *seamless integration* of an auditing tool into the spreadsheet system, the following features that are meant to improve the cooperation between the visualization toolkit and the spreadsheet system have been integrated into this component:

Selection propagation: The selection of a logical area in the tree-view influences the spreadsheet UI as well as the graph-window:

- The cells that are a member of the logical area that users select in the tree-view will be immediately selected on the spreadsheet UI, and
- the leaf nodes of the tree-view will be the nodes of an SRG_{LA}, whenever the *Graph*-button is pressed.

Cell Restriction: If users select cells on the spreadsheet UI before they start the analysis, only the selected cells will be further considered.

Graph Feedback: Users can trace back the cells that an SRG_{LA}-node contains as well as the logical area that corresponds to a given SRG_{LA} node, by selecting the node in the graph-window.

In this component, the highest level of integration between auditing toolkit, spreadsheet system and the graph window is implemented. However, due to some problems with the integration of the graph window, there is no immediate change of the displayed SRG_{LA} when the user changes the displayed items in the tree-view. The SRG_{LA} is only generated when the users push the *Graph*-button.

This technique supports fish eye-views, as the user can expand or collapse logical areas in the tree-view in order to see their members. As the SRG has the leaf nodes of the tree-view as nodes, the users can decide, whether they want to look inside a specific logical area, or whether they want to examine it on a higher level of abstraction.

7.2.2 Component *Semantic Classes*

The component *Semantic Classes* is stored in the files `semclass.c` and `semclass.h` in the plug-in-directory. As the algorithm for detecting the semantic classes of a spreadsheet program (see Table 6.1 on page 132 and Table 6.2 on page 133) depend on the parameters d_h, d_v, d_{Man}, Eq_{Start}, Eq_{Rest} and b, the cutoff percentage, the corresponding fields are also part of the component's user interface (see Figure 7.3 on page 169) and of the initial data structure (see Table 7.3 on the next page). The initialization of the data structure is performed by the plug-in's main routine. Further callback functions, e.g. when the selection in the tree control changes, are added by the *on_btn_start* start function of the component that is called to invoke the component.

Control Flow in the Component

The component is invoked by calling the *on_btn_start* function. In the function, the data model of the tree-view is initialized by calling the *init_model* function. Consecutively, the parameters are read out of the text fields in the dialog.

Depending on the selections made for Eq_{Start} and Eq_{Rest}, pointer arrays that contain the corresponding equivalence classes are retrieved from the *Logical Areas* component. Therefore, the *Logical Areas* component must have been invoked

```
typedef struct {
  Sheet *sheet;
  GtkEntry *hordist;
  GtkEntry *vertdist;
  GtkEntry *cutoff;
  GtkEntry *mandist;
  GtkCombo *eqstart;
  GtkCombo *eqrest;

  GtkTreeView *result;
  gboolean first ;
} SC_CB_Data;
```

Table 7.3: Initial data structure of the *Semantic Classes* component

already. In a next step, the information has to be transformed to serve as initial data for the merging algorithm.

The *Logical Area* component retrieves a set of logical areas that contains cells. The input for the merging algorithm has to be a list of semantic classes and the parameters. As it has been discussed in the previous chapter, a logical area is a trivial semantic class, with singleton cells as semantic units. Therefore, the assignments of cells to logical areas (of equivalence criterion Eq_{Start}) is rewritten as assignment of semantic units to semantic classes.

Consecutively, the function *merge* that is a `C` implementation of the merging algorithm discussed in Subsection 6.3.2, is invoked. The function is shown in Table 7.4 on the following page.

The function consists of two major parts that are repeatedly executed in a loop:

Selection: From line 13–32, all semantic classes that are known so far are examined for co-occurence. Therefore, each pair of semantic units in two semantic classes is examined on whether they are *reachable* according to the distance vector. For each pair of semantic classes, a %-value is calculated that indicates the minimal percentage of semantic units in either of the classes that would be absorbed by a merged semantic class. The two semantic classes with a maximal co-occurence will be selected for a merge. Pairs of reachable semantic units that are members of the candidate classes are stored in the list *merge_cands*. If the maximal co-occurence between two semantic classes is below the cutoff-percentage, the function's main loop will terminate.

Merging: The actual merging is performed by lines 33–46. Therefore, a new semantic class is created (line 37), and the pairs of semantic units are merged

```
GList* merge(GList **units, gint dh, gint dv, gint dm, gint cutoff , eqinfo eq) {
  gboolean found;
  double maxCo;
  semunit *s1, *s2;
  gint m1, m2;
  do {
    guint i;
    guint maxsize = 0;
    GList *merge_cands = NULL;
    gint maxid = search_maxid(*units);

    found = FALSE;
    for (i = 0; i < maxid; i++) {
      guint j;
      for (j = i + 1; j < maxid; j++) {
        double aktocc = 0;
        if (i != j) {
           GList *test_cands = NULL;
           aktocc = cooccurence(*units, &test_cands, i , j ,  dh, dv, dm);
           if (aktocc > cutoff / 100 &&
               maxsize < g_list_length(test_cands)) {
             maxsize = g_list_length(test_cands);
              my_list_free (merge_cands);
             merge_cands = test_cands;
             m1 = i;
             m2 = j;
           } else {
              my_list_free (test_cands);
           }
           if (aktocc > cutoff / 100) {
             found = TRUE;
           } } } }
    if (found) {
      gint newmax = search_maxid(*units);
      guint i,j;

      merge_units(units, merge_cands);
       my_list_free (merge_cands);
      for (i= 0; i < g_list_length (*units ); i++) {
        semunit *su_1 = (semunit*) g_list_nth_data(*units , i );
        if (su_1->semclass_id == newmax) {
          for (j = 0; j < g_list_length (*units ); j++) {
            semunit *su_2 = (semunit*) g_list_nth_data(*units , j );
            if (su_2->semclass_id != su_1->semclass_id && is_in_class(su_1, su_2, eq)) {
              su_2->semclass_id = su_1->semclass_id;
            } } } } }
  } while (found);
  return *units;
}
```

Table 7.4: The C implementation of the merging algorithm

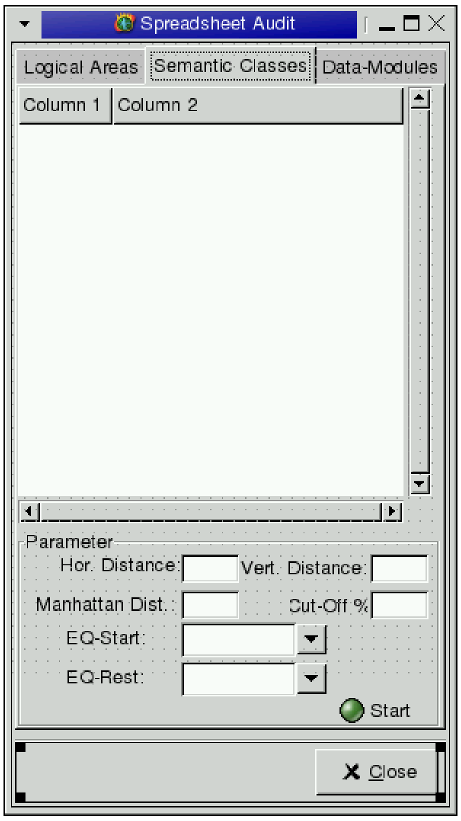


Figure 7.3: User interface of the *Semantic Classes* component

into one semantic unit that becomes a member of the new semantic class. The function *merge_units* performs this step and contains some overhead for memory management. Consecutively it is checked if one of the already defined semantic classes has become spurious, e.g. if all of the member units were absorbed by a new semantic class.

The selection is based on the result of the function *cooccurence* (see Table 7.5 on the next page). The function is invoked with the ID of two semantic classes sc_1 and sc_2 and the distance vector $\vec{d} = (d_h, d_v, d_m)$. It will return a list with pairs of semantic units (u_1, u_2) with $u_1 \in sc_1, u_2 \in sc_2$ and $\forall c_i, c_j \in u_1 \cup u_2 \mid dense(u_1 \cup u_2, c_i, c_j, \vec{d})$. The size of the semantic classes[7] sc_1 and sc_2 is stored in the variables `sc1_cnt` and `sc2_cnt`. For each semantic unit in sc_1 a semantic unit in sc_2 is searched that

1. satisfies the denseness criteria, and
2. is not already selected for merge with another semantic unit in sc_1.

[7]The size of a semantic class is given by the number of semantic units it contains.

```
double cooccurence(GList *units, GList **merge_cands, int sc1, int sc2,
                   int dh, int dv, int dm) {

  GList *consumed = NULL;
  unsigned int i,j;
  int sc1_cnt;
  int sc2_cnt;
  int cooc;
  double result;

  cooc = 0;
  sc1_cnt = 0;
  sc2_cnt = 0;
  for (i = 0; i < g_list_length (units); i++) {
    semunit *su1 = (semunit*) g_list_nth_data(units, i);
    if (su1->semclass_id == sc1) {
      sc1_cnt++;
      for (j = 0; j < g_list_length (units); j++) {
         semunit *su2 = (semunit*) g_list_nth_data(units, j);
         if (su2->semclass_id == sc2
             && g_list_find (consumed, su2) == NULL
             && g_list_find (consumed, su1) == NULL) {
           sc2_cnt++;
           if (reachable(su1, su2, dh, dv, dm)) {
             merge_cand *mc = malloc(sizeof(merge_cand));
             mc->s1 = su1;
             mc->s2 = su2;
             cooc++;
             *merge_cands = g_list_append(*merge_cands, mc);
             consumed = g_list_append(consumed, su1);
             consumed = g_list_append(consumed, su2);
           } } } } }
  if (sc1_cnt == 0 || sc2_cnt == 0) {
    result = -2;
  } else if (sc1_cnt > sc2_cnt) {
    result = (1.0 * cooc) / sc2_cnt;
  } else {
    result = (1.0 * cooc) / sc1_cnt;
  }
   g_list_free (consumed);
  return result;
}
```

Table 7.5: The C implementation of the co-occurence calculation

If such a unit is found, both semantic units are stored in the result list and are marked as consumed. Therefore, they are stored in a list of consumed semantic units. Finally, in order to calculate the percentage of co-ocurring semantic units, the size of the result list is divided by the size of the smaller semantic class.

Thus, the function returns a list of semantic units that could be merged and the percentage of semantic units in the smaller class that would be absorbed by a new semantic class. It has been decided to calculate the percentage with reference to the smaller class, because the new semantic class can only have a maximal size that is equal to the number of semantic units in the smaller semantic class, i.e. a value of 100% means that all of the possible merges are made. Obviously, this strategy will preferably select to merge small semantic classes that occur only in conjunction with a large semantic class with the affected units in the larger semantic class.

After the merging algorithm, the data is processed for presentation in the user interface. Currently, three kinds of output are supported:

1. Textual output is provided by the function *dump*. The textual output generates a report, with the id of a semantic class as heading and the member units as detail data.

2. *SRG* output is provided by the function *draw_graph*. A *SRG* is generated with nodes that represent semantic classes.

3. Tree-structured output is provided by the function *output*. In the tree control, there is a root node that represents all formula cells. The child-nodes represent semantic classes. Each semantic class has child nodes that represent semantic units that constitute the semantic class. A semantic unit has child nodes that represent its member cells. Similar to the *Logical Areas* component, the selected cells that are represented by the selected node in the tree view will be selected in the spreadsheet UI.

Limitations

The *Semantic Class* component has been developed to demonstrate the capabilities of spreadsheet visualization and auditing with *semantic classes*. Other development aims, e.g. the feasibility of fish-eye based visualization techniques and the integration of graph-based and structure-based visualization with the spreadsheet UI that were already met in the *Logical Areas* component, had a low priority in this component. Thus, a suggested fish-eye visualization has not been realized and there is no connection between the graph-window and the tree view or the spreadsheet UI.

Furthermore, in order to increase the usability the rather rudimentary user interface needs further improvement. Currently, the cells that the analysis should operate on, are read out from the *Logical Area*-component. Thus, if the users want

```
typedef struct {
  Sheet *sheet;
  GtkTreeView *tree_structure;
  GtkTreeView *cand;
  GtkTreeView *excl;
  void *mod_graph;
} Mod_CB_Data;
```

Table 7.6: Initial data structure of the *Data Modules* component

to analyze the spreadsheet or a region of the spreadsheet with this component, the corresponding region of the spreadsheet UI has to be analyzed at first with the *Logical Area*-component.

7.2.3 Component *Data Modules*

The component *Data Modules* is stored in the files `datamod.h` and `datamod.c`. The implementation of this component is straightforward as no specific algorithm is applied. The user interface has to support the possibility for users to preprocess the *DDG*,in order to remove sink nodes that do not correspond to spreadsheet results. Therefore, two list fields are supported, i.e. one that contains the current sink nodes and one that displays the sink nodes so far removed (see Figure 7.4 on the facing page). Therefore, the two tree views, i.e. `cand` and `excl` are stored in the initial data structure (see Table 7.6) that is initialized by the plug-in's main function.

In contrast to the other modules where only a callback that corresponds to the dialogs' start button is added by the main function, the main function also adds callbacks to the *left* ($\leftarrow$) and *right* ($\rightarrow$) buttons of the *DDG* that will move list items from the list of sink nodes to the list of excluded sink nodes. The same callback routine is added as `on-node-delete` callback to the graph-window menu. Hence, whenever the user deletes a sink node in the graph-window, it will be automatically added to the list of deleted nodes in the dialog.

7.2.4 Control Flow in the Component

Analysis is usually started with pruning the *DDG*. Therefore, the component generates a graphical representation of the *DDG* and highlights the sink nodes. Consecutively, the user is offered opportunities to remove sink nodes. Afterwards, the partitioning of the *DDG* into data modules is invoked by the `build_dmod` function that is a callback attached to the *Module*-button.

The partitioning itself is performed by the function `create_modules` that is stored in the file `graphs.c`. The function has been transfered to an extra file

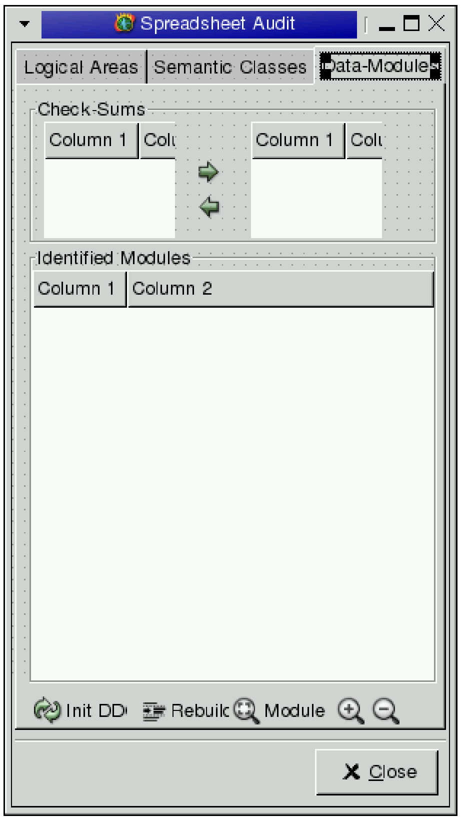


Figure 7.4: User interface of the *Data Modules*-component

and even an extra shared library, because it is written in C++. The change of the programming language was necessary, because mainly functions of the *LEDA* toolkit have been used to perform the analysis.

In detail, the implementation of the analysis is shown in Table 7.7 on the following page.

The analysis ranks the nodes in the *DDG* by a longest path ranking (see Alberts et al. [AGMN97] for a detailed discussion of graph layout algorithms). The longest path ranking will assign each node a rank, i.e. a unique ordinal number, that is calculated by the rank of the predecessors[8] +1. Thus, it can be assumed that there is at least one sink node that has the highest rank and, if nodes are ordered by their rank, sink nodes, i.e. leaf nodes, will come before the nodes they depend on. For the analysis, nodes are visited in the order of their rank. For each node it is checked for if it is referenced by exactly one other module. If this condition is true, the node will join the module. Else a new module containing only this node is created (see lines 37–60 in the listing).

Consecutively, the information of the initial *DDG* is used to reconstruct the edges of the *SRG* with data modules as nodes (see lines 61–81). Finally, some

[8]A predecessor of a node v in the *DDG* is the source of an edge, heading to v.

```cpp
void* create_modules() {
    graph& G = gw->get_graph();
    GraphWin *g_new = new GraphWin("Data_Modules");
    graph& GN = g_new->get_graph();
    node_array<int> rankN(GN);
    LongestPathRanking L(true);
    GraphWinInterface A(*g_new);
    SugiyamaLayout sugiyama;
    int i;
    node_array<int> rank(G);
    h_array<node, int> *assignments = new h_array<node, int>;
    node v;
    edge e;
    int current_rank = -1;
    int next_modid = 0;
    const char* cap;

    g_new->set_animation_steps(0);
    free(mod_ass);
    free(modules);
    free(module);

    text = new h_array<int, const char*>; //Indexing modul# with label
    rep = new h_array<int, node>; //Index von Modul# auf rep. knoten
    mod_ass = new h_array<const char*, list<const char*>* >;
    modules = new list<const char*>;
    module = new node_array<int>(G);

    LongestPathRanking(true).call(G, rank);

    //Search highest rank, in order to start collapsing bottom-up
    forall_nodes(v, gw->get_graph()) {
      if (rank[v] > current_rank) {
        current_rank = rank[v];
    } }
    //Go through all nodes, start at bottom
    for (i = current_rank; i >= 0; i--) {
     forall_nodes(v, gw->get_graph()) {
      //Is node to be examined now?
      if (rank[v] == i) {
        int curr_mod = -1;
        int mod_cnt = 0;
        forall_out_edges(e, v) {
          node w = gw->get_graph().target(e);
          //How many nodes in how many different modules reference
          //the current node?
          if (curr_mod != (*module)[w]) {
             curr_mod = (*module)[w];
             mod_cnt++;
        } }
        // Only one -> add this node to the dependent's module.
        if (mod_cnt == 1) {
           (*module)[v] = curr_mod;
        } else {
          //else, lets create a new module, because we
          //have to deal either with a sink, or an intermediate node.
          (*module)[v] = next_modid++;
          //Set the new module's text to the intermediate node's label.
          (*text)[(*module)[v]] = (const char*) gw->get_label(v);
    } } } }
    for (int i = 0; i < next_modid; i++) {
        point p;
        list<const char*> *l = new list<const char*>;
        (*rep)[i] = g_new->new_node(p); //Create, for each module, a node that represents the module.
        g_new->set_label((*rep)[i], (*text)[i]);
        modules->push((*text)[i]); //put the label into the module-member list.
        (*mod_ass)[(*text)[i]] = l;
    }
    forall_nodes(v, gw->get_graph()) {//Now, lets have a look at the edges!
      (*mod_ass)[(*text)[(*module)[v]]]->push(gw->get_label(v));
      forall_out_edges(e, v) {
        node w = gw->get_graph().target(e);
        //If the module of the target is not equal to the module of the
        //source, check, if there is already an edge between v's module
        //and w's module.
        //If not, create an edge.
        if ((*module)[w] != (*module)[v]) {
          list<node> l= g_new->get_graph().adj_nodes((*rep)[(*module)[v]]);
          if (l.search((*rep)[(*module)[w]]) == NULL) {
            edge e = g_new->new_edge((*rep)[(*module)[v]], (*rep)[(*module)[w]]);
    } } } }
    g_new->display();
    L.call(GN, rankN);
    sugiyama.ext_call(GN, rankN, A);
    g_new->zoom_graph();
    return g_new;
}
```

Table 7.7: C++ implementation of Data Modules analysis.

layout work is done and the graph is drawn. Additionally, in the tree view a more structured representation of the identified data modules is given. Therefore, each data module is represented in the tree control as a top-level node. If it is expanded, all the cells that are contained in the data module are shown as its children.

Further information that is given in the tree view involves the size of the data module and if it contains any sink nodes. This information is retrieved from the data structures that are built up during the partitioning process (see Table 7.7 on the preceding page). The cells that are represented by the currently selected item in the tree view will be highlighted on the spreadsheet UI.

There is some basic support for fish eye-views offered by this component. The buttons `btn_zoom_in` and `btn_zoom_out`, that are represented by the zoom-in and zoom-out icons in the dialog, will either expand or collapse the data module,that is currently selected in the tree view. Expanding a data module, i.e. replacing it in the *SRG* with its member cells, consists of three steps:

- replacing the data module by the cells it contains,
- reconstructing the inter-cell edges from the *DDG*, and
- redirecting the edges that target to the expanded data module to the node that represents their target cell.

Collapsing an expanded data module in the *SRG* means to replace the member cells of the data module by a single node, i.e. the result node of the data module. This is done in a straightforward manner, as only the nodes that were added during the expansion have to be removed from the graph, except for the result node of the data module. As the result node is the only source of edges that should remain after collapsing the module, only the edges targeting into the module have to be redirected to the remaining result node.

7.3 Limits and Improvement

The toolkit that is described in this chapter was developed in order to demonstrate the effectiveness of the auditing strategies suggested in the previous chapter. Thus, it has to be considered merely a prototype that is not targeted to fit the needs of end-users. However, it was well suited for demonstrating the feasibility of the visualization approaches.

Of course, there are still some remaining problems. The desired degree of integration between the plug-in and the spreadsheet system could be reached, but the integration between the plug-in and `LEDA`, the graph visualization toolkit, needs further improvement.

There are also some missing features, because not all of the desired features, e.g. fish-eye views or the feedback between the components, were implemented

	Logical Areas	Semantic Classes	Data Modules
Coordination graph window - spreadsheet UI	F	-	-
Coordination graph window - toolkit	-	-	F
Coordination toolkit - graph window	I	I	F
Coordination toolkit - spreadsheet UI	F	F	F
Coordination spreadsheet UI - other components	I	I	I
Coloring of cells	-	-	-

Table 7.8: Special features and implementing components. F stands for *the feature is fully implemented*, I for *Improvement needed*, - for *missing*

for all components. Fish-Eye views, for instance, are available with logical areas and data modules, but only to a certain degree with semantic classes. Another feature that would be nice to have, is automatic coloring of cells depending on the membership in a logical area, semantic class or data module. In Table 7.8 a brief summary of implemented features and needed improvement is given.

Summary

The following issues have been addressed in this chapter:

- A prototype has been implemented to show the feasibility of the auditing techniques.
- The prototype is a plug-in for the gnumeric spreadsheet system.
- It consists of three loosely connected analysis components and a graph visualization component.
- Each desired feature, e.g. fish eye-views and seamless integration with the spreadsheet UI, is demonstrated by one or more component(s).
- The prototype is not suitable for end-users yet.

Chapter 8

Outlook

In this chapter, the spreadsheet visualization technique introduced in Chapter 6 is discussed in the broader context of other spreadsheet auditing techniques (see Chapter 5). Although the visualization technique has been developed as an auditing technique, possible applications in the fields of spreadsheet maintenance and spreadsheet development will also be discussed in the following section.

The auditing technique and toolkit have turned out so far to be cost effective and efficient for finding errors. Its efficiency shows specifically when it is applied to the kind of spreadsheet programs it has been designed for, i.e. large spreadsheets with regularly re-occurring patterns of formula usage. The logical areas and semantic classes are a helpful tool to reduce the complexity of size that is hidden in large spreadsheet programs. Data modules can support checking of the *DDG* of a spreadsheet program by drastically reducing the number of nodes and edges that have to be considered without losing any information.

Nevertheless, this approach is not suitable for checking spreadsheets with complicated calculation, where formulas do not reoccur. For this kind of spreadsheet programs, the spreadsheet testing approaches have to be used. To combine the spreadsheet visualization approach suggested here with different spreadsheet testing approaches is also feasible and might be a good idea. Therefore, the formulas in a representative semantic unit (or cell) of each semantic class (or logical area) are checked by the conventional testing approach, and consecutively auditors check for whether the semantic class (or logical area) occurs in the right places.

During the practical tests of the auditing technique and several discussions, possible extensions have been figured out and will be described in Section 8.2.

8.1 Discussion

There are already several tools for the visual auditing and testing of spreadsheet programs. Usually, the testing approaches will yield good results if the region to test is small and the spreadsheet tester is familiar with the testing approach. Spreadsheet testing considers a spreadsheet more or less conventional software and it is tried to find wrong results by comparing the calculated results to a test oracle.

8.1.1 Position of the Work

Spreadsheet visualization is a totally different approach to the identification of errors. The particularities of the spreadsheet program and the spreadsheet programmers are taken into account by combining the formulas with the layout of the spreadsheet. Different visualization techniques rely on different kinds of combinations, but they have all in common that their users do not check the spreadsheet program on a cell-by-cell level, but rather through examining the spreadsheet UI to find peculiarities in the patterns that the visualization indicates.

Thus, the visualization approaches usually give immediate feedback to the users on the spreadsheet UI. Although this feature is desirable at first sight, it has the drawback that the visualization technique does not scale up for spreadsheet programs that do not fit on a single screen any more. All spreadsheet visualization techniques that have been examined during the development of this work (see Section 5.5) share this inconvenience.

Although logical areas, semantic classes and data modules can still be projected to the spreadsheet UI in order to find patterns, there are some major differences:

- The auditing approach introduced does not only use the spreadsheet UI for visualization. There are also *SRG* and structural views of the spreadsheet program.

- Visualization is not performed on a cell-by-cell level, but cells are aggregated to form more abstract units, i.e. logical areas, semantic units and semantic classes or data modules.

The first property makes the visualization technique suit for large spreadsheets. Patterns that are larger than the size of the screen become visible and it is easy to identify ruptures in regular structures. The second feature establishes scalability of the technique. There are some auditing approaches, e.g. the **S2** and **S3** visualization (see Sajaniemi [Saj00]) and *SpACE* (see Butler [But00]), that support the aggregation of identical formulas into abstract units.

However, the homogeneity criterion is very strict, i.e. copy-equivalence, and only geometrical compact areas with homogeneous formulas can form such an abstract unit. Thus, regular patterns of formulas that repeat e.g. every third cell, will not be recognized. Furthermore, it has turned out that the cross section of different equivalence criteria is a very worthy help. This feature is not supported at all by alternative visualization approaches, but an important part of the auditing approach introduced.

A further abstraction step, as it is implemented by semantic classes, turned out to reduce the complexity of auditing large spreadsheet programs. The *SRG* becomes much more compact, and there are less entities for the auditor to check. A corresponding feature has not been offered by any other spreadsheet visualization approach so far.

Beside these visualization tools that are based on coloring cells, there are also tools to visualize data dependencies among cells. Well known examples are the built-in auditing tools of Excel, that allow users to trace dependents and predecessors of a given cell, or improvements of this technique (see Davis [Dav96]). These approaches are suitable for bug tracing, but as a matter of fact, they are not suitable as auditing tools.

Chan et al. [CC00] presents a data flow based auditing and spreadsheet visualization toolkit that is suitable for auditing. There are some data flow based auditing strategies, that are based on the assumption that data flow in the spreadsheet should be directed the way people read the spreadsheet, i.e. from the top left to the bottom right corner. Deviations in this direction will be highlighted to the auditor and are classified as a hot-spot.

A further auditing strategy that is suggested by Chan et al. requires data flow to be only amongst spatial neighbors. Anomalies are again classified as an error. The second strategy obviously relates to the structured spreadsheet modeling approach that is suggested by Rajalingham et al. (see Subsection 5.3.3). Nevertheless, this technique does not scale up. Both strategies can be easily transferred to data modules:

1. The data flow in a data module has to be oriented from top-left to bottom-right, and data dependencies between result cells and other data modules have to follow the same rule.

2. Members of data modules have to be in a physical area.

Nevertheless, there are some fundamental differences, as data modules are the more flexible visualization tool. Apart from the strategies suggested by Chan, there are three further auditing strategies (see Subsection 6.4.3). Here, the users are not confronted with the spreadsheet on a cell-by-cell level any more. Instead, they start on the level of data modules, and each data module can be checked separately. Thus, this approach is more suitable for larger spreadsheet programs.

8.1.2 Further Applications

So far, the advantages and short-comings of this visualization technique have only been discussed in the context of spreadsheet auditing. Nevertheless, the identification of structures and patterns could also be exploited in other areas of spreadsheet development. Obviously, software maintenance in general, and spreadsheet maintenance in particular tend to increase the number of errors in software.

Reverse Engineering and Maintenance

The approach presented here leads to a better understanding of a spreadsheet program by the spreadsheet programmer and maintainer. Misconceptions are a very common source of errors (see the results of the field audit in Section 4.3). They result from missing documentations of both spreadsheets and their changes. It is common that spreadsheets with a maintenance cycle above 6 months are not even understood by their creators. Therefore, maintenance is often based on a conceptual model reconstructed by assumptions of how the conceptual model of the spreadsheet could have looked like. The visualization of the semantic classes and the logical areas can be a means of supporting the reconstruction of the conceptual model of a spreadsheet. Knowing this structure, maintenance might be done without bluring the structure, i.e. by

- copying and pasting not individual cells, but semantic units,
- correcting errors not in a single cell, but in a logical area,
- considering the pattern of occurrence of members of a semantic class or a logical area, before modifications are made.

The drawback that the spreadsheet maintainers' freedom is restricted if structure-preserving maintenance is done, is out-weighted by the advantages of a consistent model throughout several maintenance cycles. Many errors discovered in field audits (see Chapter 4) that were due to structure destroying maintenance, e.g. the value instead of formula errors, can be avoided.

Programmer Guidance

As for the application in spreadsheet maintenance, the information about semantic classes and logical areas can be used to guide the spreadsheet programmer during the implementation stage. The following ways of guidance can be outlined so far:

- copy and paste on the level of semantic units,
- warning programmers if a new cell or modification does not correspond to a given pattern, and
- warning programmers, if they are about to introduce formulas that are structurally or logically equivalent to others, as this might be the introduction of outliers.

Of course, the programmers can only be warned, because there might be well founded reasons to break regular structures or to introduce structurally equivalent formulas.

One open issue that has to be solved before is the automatization of the parametrization of the semantic class algorithm (see Section 8.2), as the algorithm has to be re-executed after each change in the spreadsheet program in order to have a valid abstraction of the structure. Obviously, it is not feasible to force the spreadsheet programmer to adjust the parameters of the analysis algorithm after each change.

8.2 Further Work

In this section some extensions of the spreadsheet visualization and auditing strategies are discussed. One issue commonly criticized to be missing is that currently labels and formatting information is not considered. Possible ways to improve the visualization with this extra information are shown in Subsection 8.2.1. As the main application of the visualization technique is spreadsheet auditing, possible improvements of the auditing strategies are suggested in Subsection 8.2.2.

8.2.1 Using Layout Information to Improve the Visualization

As a matter of fact, the users' model of the spreadsheet is not only determined by the spreadsheet program, i.e. the set of formula cells and constants. Indeed, labels and formatting issues bear some additional semantics, too. Users tend to mark out blocks they consider to be related by certain layout characteristics, e.g. frames, fonts or format templates.

The same is true for labels. If the same concept is repeatedly used throughout the spreadsheet program, it is expected that the labels will also be identical. Having a look at real world spreadsheets furthermore reveals, that final sums are generally formatted in a specific way, e.g. they are displayed in a bold font or they are underlined. Additionally, there are some format templates that bear specific semantics, e.g. a result template.

For the identification of semantic units, the denseness of a geometrical area on the spreadsheet UI plays an important role. Nevertheless, as the meaning of labels cannot be automatically determined, label cells are considered to be empty. However, for users there might be a difference between label cells that carry certain information, too, and empty cells, that are generally used to separate different areas on the spreadsheet UI. Considering the assumptions made above, the layout information can be used for the issues that are discussed subsequently.

Selecting Parameter Values

One of the crucial issues when spreadsheet analysis is performed by the semantic classes approach is the specification of the correct parameters for the partitioning algorithm. End users will need a certain degree of training and/or experience in order to become familiar with the meaning of each parameter. As this is a drawback in the quest of creating auditing and visualization tools that are suitable for end-users, automatically selecting parameters would be a big step forward.

The manual specification of parameters has also a further drawback, as different regions of a given spreadsheet program might require different parameters. Thus, the upper part that deals with data collection, is column-oriented, whereas the bottom part, that calculates business figures, is row-oriented. By now, the spreadsheet has to be manually decomposed in different regions, and each of the regions is analyzed with different parameters.

The information that is supplied through labels and formatting might be exploitable to smoothly adjust the algorithms' parameter. However, labels and formatting issues also depend on the taste of the spreadsheet programmer. Identical spreadsheet programs can be formatted in many different ways, without changing the semantics. Thus, possible heuristics for choosing the parameters have to be carefully examined and tested before they are used.

Identification of Result Cells

As it has been stated in the introduction of this section, the final sums and results of a spreadsheet program are usually highlighted by special formats. Thus, developing heuristics to deduce from the format guidelines to the meaning of a specific cell, i.e. if a cell is a *real* result of a spreadsheet, or a check-sum. Hence, when the spreadsheet auditor decides to decompose the spreadsheet into data modules, the manual removal of *DDG* sink nodes can be accelerated.

A system that is aware of certain heuristics might look for those cells that are near the sinks, and are specifically highlighted. Usually, it can be expected that a check-sum has some of the *real* results of the spreadsheet as input. Thus, sink nodes, depending on many highlighted neighbors, can be considered candidates for removal. Nevertheless, automatic removal is dangerous, as the spreadsheet programmer is free to break this conventions, e.g. by highlighting intermediate results and not highlighting the final result.

Further Considerations

However, in large spreadsheets, cells are sometimes only incidentally similar. The auditor has to check for each irregularity, whether it has been introduced on error, on purpose, or there is no irregularity at all, but only an incidental similarity. If formatting issues are important in the analyzed spreadsheet program, the formatting of a cell can also be used to get a better insight into the degree of cohesion of a semantic class or a logical area. In highly cohesive semantic classes, outliers that do not share the formatting of the other members might be considered incidental members and, thus, not so important.

8.2.2 Automatic Identification of Errors

Automatic error detection in spreadsheet programs is also a very common suggestion. As it was discussed in Chapter 6, irregularities in the geometrical pattern or the equivalence criteria of members of semantic classes or logical areas are an important approach towards the identification of errors.

Although the abstract representation of the spreadsheet supports the auditor in finding irregularities, the so-called hot-spots, it is still a tedious task to check the distribution of all semantic units of a given semantic class or the members of a data module. There is a limited number of kinds of irregularities, e.g. ruptures of the geometrical pattern or outliers in the structural representation.

Surveying these kinds of irregularities and improving the toolkit to check the abstract representation of the spreadsheet program for these irregularities would significantly accelerate the auditing process. Nevertheless, there might also be some drawbacks. If auditors get used to the feature that their attention is drawn to the hot-spots by the toolkit, they will not check the visualization for themselves for any unrecognized hot-spots.

Thus, in order to be efficient, a rule base that contains nearly all kinds of irregularities is necessary. This rule base cannot be supplied in advance, as errors are partly domain specific. Nevertheless, this support is of course worthwhile and can surely prevent some errors from being overseen.

Summary

The following issues have been addressed in this chapter:

- The here presented spreadsheet visualization approach
  - is well suited for large, regular spreadsheet,
  - can reduce auditing effort can when dealing with abstract units, above the level of single cells, and
  - is scalable.
- Further applications in the fields of
  - Reverse Engineering and Maintenance, and
  - Programmer Guidance

  are possible.

- Possible Improvements include the
  - consideration of layout issues in order to
    - find parameters for the semantic class algorithm,
    - find a measure for the cohesion of a semantic class or logical areas, and
    - identify result cells of the spreadsheet,
  - categorization and automatic identification of irregularities.

Chapter 9

Conclusion

The aim of this thesis is to help spreadsheet users to improve the quality of spreadsheet programs. Although many different spreadsheet development and testing techniques were developed, they were not widely accepted by end-users. Often, this is due to the fact that most approaches rely too much on software engineering methodologies. This contradicts the fact that spreadsheet programs are not created by a defined process.

However, as it was pointed out in panel discussions between software engineers and end-users, e.g. during the 1^{st} EuSpRIG symposium, spreadsheet programers are often not aware of the fact that they are programming at all. Nevertheless, they are aware that they have to check the results of their spreadsheets.

A group of tools that emerges consider this fact and aim to offer the spreadsheet programmer support for testing and auditing by visualizing certain structural aspects of spreadsheet programs. These tools, surveyed in Section 5.5, are most likely to be accepted by end-users. An indication for the acceptance is that some visual spreadsheet auditing tools, e.g. the *spreadsheet detective* or *SpACE*, are already commercially distributed.

In Chapter 6 of the thesis, scalable approaches to spreadsheet auditing were introduced. The three approaches are based on visualizing different structural aspects of spreadsheet programs and help the spreadsheet auditor to reconstruct a conceptual model of the spreadsheet. These visual auditing approaches introduce the following extra benefits:

Scalability: The visualization techniques found in the literature recover certain structural aspects and highlight them on the spreadsheet UI by coloring cells. Thus, the part of the spreadsheet analyzed at a time is limited by the size of the screen. The scalable approaches introduced here offer novel visualization capabilities that go beyond the spreadsheet UI.

Different degrees of abstraction: Logical areas, semantic classes and data module aggregate *similar* cells, whereby the definitions of similarity vary, into abstract units. Other visualization approaches that were discussed in

Section 5.5 offer only one definition of similarity, usually the one referred to here as copy equivalence.

Not limited to geometrical areas: Because of the definition of logical areas, abstract units are not forced to be formed by spatial neighbors anymore. They can be scattered throughout the whole spreadsheet. Thus, pattern based auditing can be introduced.

Auditing strategies: For each of the three visualization approaches discussed, different auditing strategies with a varying focus are suggested. Depending on the interest of spreadsheet auditors they can use either *pattern-*, *structure-*, or *SRG*-based auditing strategies.

As it was aimed to support end-users, one of the goals was to keep the auditing technique simple and free of IT-vocabulary for users without neglecting a solid theoretical foundation. Effective application of the auditing toolkit should be possible for end-users without extra training.

Visual auditing with logical areas meets this requirement to a high degree. Discussions and presentations for spreadsheet users have shown that it is easy to communicate node equivalence classes and the different auditing strategies. Although logical areas are not suitable for auditing very large spreadsheets, they are well suited for the application by end-users with only little experience because the concept is easy to explain.

The same is true for data modules. Both, logical areas and data modules, can be applied straightforwardly without any parameters. Thus, spreadsheet users do not have to deal with any issues specific to the methodology, but can concentrate on the auditing strategies.

Data modules and logical areas or semantic classes can be combined in many ways. Not only can they be applied on different regions of spreadsheets, or spreadsheets of different kinds, but will also point out different aspects when they are applied to the same part of a spreadsheet. The two concepts can be considered orthogonal, because logical areas and semantic classes group cells that repeatedly fulfill the same task, whereas data modules group cells that contribute to the same task.

Semantic classes evolved out of logical areas. They turned out to be very efficient for the analysis of large spreadsheets because a high level of abstraction is introduced. The concept is still simple to explain to end-users, but due to required parameters, i.e. $\vec{d}$ and b, the benefit of the analysis depends on the match between parameters and the supposed model. Thus, a certain degree of training and experimenting is needed.

On the one hand, it can be argued that the required training will hinder the success of the visualization approach. But on the other hand the efficiency and the variety of possible applications of semantic classes (see Subsection 8.1.2) might also justify a certain amount of training required.

For semantic classes two different target groups are considered. The first target group consists of end-users who have already successfully applied logical areas and data modules. At a given time they might notice that they require the extra abstraction capabilities offered by semantic classes. As they are already familiar with the auditing methodology and are also well motivated, it is likely that they will not shy away from some extra training.

The second target group are professional consultants that examine spreadsheets produced by other persons. As these consultants are not end-users, but, to a certain degree, IT-professionals, the additional training will not hinder them.

To sum up, it can be said that the auditing approaches presented here offer novel abstraction and visualization capabilities. In contrast to other spreadsheet visualization techniques, spatial limitations due to the spreadsheet UI are not imposed any more. Cells can be categorized according to different criteria, and the users can combine different auditing strategies. Although there is a high degree of flexibility, e.g. different similarity criteria or pruning of the DDG, two of the three approaches can be applied with a minimum of training.

The three approaches are not suitable for all kinds of spreadsheet programs. The abstraction technique yields only a minimal benefit for spreadsheet programs that are computationally very complicated, but small. However, they are well suited to reduce the complexity of size that is immanent in huge spreadsheet programs of a certain category. The testing of these spreadsheet programs is very expensive and usually only done in a superficial way.

By the identification of hot-spots in these huge spreadsheet programs, the auditing process can be accelerated and costs are reduced. Although there are already different suggested approaches to improve spreadsheet quality, they all require changes in the process of developing spreadsheet programs or restrict the way spreadsheet users layout their spreadsheet programs.

The spreadsheet visualization approach aims to offer a varying degree of help for spreadsheet users to improve their spreadsheets without changing the way spreadsheets are created or imposing any restrictions on the spreadsheet programmers freedom of how to build up a spreadsheet.

Appendix A

Installation Guide

The spreadsheet visualization toolkit described in this thesis is a plug-in for the *Gnumeric* spreadsheet system which is a part of the *GNOME*-Desktop Environment for *Linux*. For the development of the plug-in, the following versions were used:

- *SuSE*-Linux 8.0, with the linux-kernel version $2.4.18 - 4GB$
- *GNOME* 2.0
- *Gnumeric* 1.1.9

A.1 Installing and Compiling GNUMERIC and the toolkit

In order to compile and run *Gnumeric* properly, other dependent libraries have to be installed. As the here described plug-in does not depend on them, detailed instructions can be found on the homepage of the *Gnumeric* project [Gol03].

The plug-in is integrated into the spreadsheet system by means of a *CORBA* interface that is supplied by *Gnumeric*. A CORBA implementation that is integrated in the *GNOME* Desktop Environment is given by the *bonobo*-libraries, that have to be installed. As some major changes to these libraries have taken place, a version higher than 2.0.0 is required. To integrate bonobo support into Gnumeric, it has to be configured with the option *–with-bonobo*.

Although it is recommended to compile Gnumeric with a recent version of the gnu C compiler (gcc), the plug-in does not work properly with gcc 3.0 or later versions. On the development system, gcc 2.95.3 was used.

In order to compile and install Gnumeric properly, the following steps have to be followed after installing all the depending libraries, the Gnumeric source code has been downloaded and copied to the directory `gnumeric_home`:

1. Install AGD and LEDA (see next paragraph).

2. The plug-in's source code and makefile have to be copied to the directory `gnumeric_home/plugins/vis`.

3. In the file `gnumeric_home/plugins/Makefile.am`, the plug-in has to be added. Hence, the word `vis` has to be added to the `SUBDIRS_FUNCTIONS` variable.

4. The plug-in has to registered in the file `gnumeric_home/configure`. Therefore, the location of the plug-in's makefile, i.e. `plugins/vis/Makefile`, has to be added to the variable `ac_config_files`.

5. The plug-in's makefile has to be registered as a `CONFIG_FILE`. Therefore, the line `CONFIG_FILES=$CONFIG_FILES plugins/vis/Makefile` has to be appended to the `ac_config_target` section of the file `gnumeric_home/configure`.

6. Gnumeric has to be configured. Let `dest_dir` be the target of the installation, e.g. `/usr/local`. In this case, the directory has to be changed to `gnumeric_home` and the command `./configure --prefix=dest_dir --with-bonobo` has to be executed. If all needed libraries are properly installed, *Gnumeric* can be compiled with the command `make`. In the next step, `make install` will copy the executable files to the destination directory and set the required permissions.

7. Finally, the dialog layouts, i.e. the file `gnumeric_home/plugins/vis/dialog.glade` of the plug-in have to be copied manually to the directory `dest_dir/share/gnumeric/`$VERSION$`-bonobo/glade`. The interface to `AGD`, i.e. the file `gnumeric_home/plugins/vis/libGraphs.so` has to be copied into the user's library path.

After performing these steps, the *Gnumeric* spreadsheet system as well as the plug-in are properly installed on the target system. For further trouble shooting, either the *Gnumeric* homepage (see [Gol03]) or the file `gnumeric_home/README` can be consulted.

Further, the *Library of Efficient Datatypes and Algorithms (LEDA)*, Version 4.4 with *AGD* has to be installed correctly on the system, before the spreadsheet visualization toolkit can be executed. *LEDA* can be purchased from *Algorithmic Solutions*.

Bibliography

[Ack82] William B. Ackermann. Data Flow Languages. *IEEE Computer*, pages 15–40, February 1982.

[ACM00] Yirsaw Ayalew, Markus Clermont, and Roland Mittermeir. Detecting errors in spreadsheets. In *Spreadsheet Risks, Audit and Development Methods*, volume 1, pages 51–62, AAAAAA, 7 2000. EuSpRIG, University of Greenwich.

[AGMN97] D. Alberts, C. Gutwenger, P. Mutzel, and S. Näher. AGD-library: A library of algorithms for graph drawing. In G. F. Italiano and S. Orlando, editors, *Proceedings of the Workshop on Algorithm Engineering (WAE '97)*, Sept. 1997.

[Ale96] Robin A. Alexander. Teaching Good Systems Design for Spreadsheet Projects. *Journal of Accounting Education*, 14(1):113–122, January 1996.

[Ama92] Makoto Amamiya. A new parallel graph reduction model and its machine architecture. In John A. Sharp, editor, *Data flow computing: Theory and Practice*. Ablex Publishing Corporation, Norwood, NJ, 1992.

[Ang93] Boon Seon Ang. Efficient implementation of sequential loops in dataflow computation. In *Proceedings of the conference on Functional Programming Languages and Computer Architecture*, pages 169–178. ACM, 1993.

[App97] B. Appleton. Patterns and software: Essential concepts and terminology, 1997.

[Asi84] Isaac Asimov. *Asimovs New Guide to Science*. Basic Books, Inc., New York, 1984.

[Aya01] Yirsaw Ayalew. *Spreadsheet Testing Using Interval Analysis*. PhD thesis, Universität Klagenfurt, Universitätsstrasse 65–67, A-9020 Klagenfurt, Austria, November 2001.

[BB93] James C. Brancheau and Carol V. Brown. The Management of End-User Computing: Status and Directions. *ACM Computing Surveys*, 25(4):437–482, December 1993.

[BBL76] Barry W. Boehm, J. R. Brown, and M. Lipow. Quantitative Evaluation of Software Quality. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605. IEEE-CS-ACM, 1976.

[Bei90] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, second edition, 1990.

[Ber86] T. Berry. How to structure spreadsheets. *Business Software*, pages 56–58, October 1986.

[BG87] Polly Brown and John Gould. An experimental study of people creating spreadsheets. *Transactions on Office Information Systems*, 5(3):258–272, July 1987.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, (4):61—72, April 1988.

[BP93] V.R. Basili and B.T. Perricone. Software Errors and Complexity: An Empirical Investigation. In *Software Engineering Metrics (Vol. 1: Measures and Validation)*, pages 168–183. McGraw-Hill International, 1993.

[BR00] Keith H. Bennett and Vaclav T. Rajlich. Software Maintenance and Evolution: A Roadmap. In Anthony Finkelstein, editor, *The Future of Software Engineering*, pages 73–87. ACM Press, 2000.

[Bra98] Neil Bradley. *The XML Companion.* Addison-Wesley, 1998.

[Bra99] Mary Brandel. Technology Flashback – 1983: As Easy as Lotus 1-2-3. *Computerworld*, 1999.

[Bri00] Daniel Bricklin. Software arts & visicalc. http://www.bricklin.com/history/intro.htm, 2000. visited on 6th February 2002.

[Bro95] Frederick P. Jr. Brooks. *The Mythical Man-Month: Essays on Software Engeering.* Addison Wesley Professional, 1995.

[Bro02] Christopher Browne. Linux spreadsheets. http://www.ntlug.org/ cbbrowne/spreadsheets.html, January 2002. visited on 10th January 2002.

[Bud94] David Budgen. *Software Design.* Addison-Wesley, 1994.

[But00] Ray Butler. Is This Spreadsheet a Tax Evader ? How H. M. Customs & Excise Test Spreadsheet Applications. In *Proceedings of the 33rd Hawaii International Conference on System Sciences - 2000*, volume 33, 2000.

[Cas92] Rommert Casimir. Real programmers don't use spreadsheets. *ACM SIGPLAN Notices*, 27(6):10–16, June 1992.

[CC00] Hock Chuan Chan and Ying Chen. Visual checking of spreadsheets. In *Spreadsheet Risks, Audit and Development Methods*, volume 1, pages 75–85. EuSpRIG, University of Greenwich, 7 2000.

[Cha01] Hock Chuan Chan, editor. *Easy Steps to Design & Check Your Excel spreadsheets.* Federal Publications, 2001.

[Cha02] David Chadwick. Training Gamble leads to Corporate Grumble? In *Spreadsheet Risks, Audit and Development Methods*, volume 3, pages 2–10. EUSPRIG, 7 2002.

[CHM02] Markus Clermont, Christian Hanin, and Roland Mittermeir. A Spreadsheet Auditing Tool Evaluated in an Industrial Context . In *Spreadsheet Risks, Audit and Development Methods*, volume 3, pages 35–46. EUSPRIG, 7 2002.

[CK93] P. G Cragg and M. King. Spreadsheet modelling abuse: An opportunity for OR. *Journal of the Operational Research Society*, 44(8):743–752, August 1993. Summary by R. Panko.

[CK02] Martin Campbell-Kelly. The rise and rise of the spreadsheet. Talk at EUSPRIG 02, July 2002.

[Cop00] James O. Coplien. *Software Patterns.* SIGS Books & Multimedia, New York, 2000.

[CR98] David Chadwick and Kamalasen Rajalingham. Integrity Control of Spreadsheets: Organisation & Tools. In *Proceedings of the IFIP TC11 WG11.5 Second Working Conference on Integrity and Internal Control in Information Systems*, pages 147–168, November 1998.

[CRKE99] David Chadwick, Kamalasen Rajalingham, Brian Knight, and D. Edwards. An Approach to the Teaching of Spreadsheets Using Software Engineering Concepts. In *Proceedings of the 4th International Conference on Software Process Improvement, Research, Education and Training INSPIRE'99*, pages 261–273, 1999.

[CS96] Yolande E. Chan and Veda C. Storey. The use of spreadsheets in organizations: Determinants and consequences. *Information & Management*, 31:119–134, 1996.

[DAFP93] Sergio T. Mujica David A. Fuller and Jos A. Pino. The Design of an Object-Oriented Collaborative Spreadsheet with Version Control and History Management. In *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*, pages 416–423. SIGAPP, ACM, 1993.

[Dav96] J. S. Davis. Tools for spreadsheet auditing. *International Journal of Human-Computer Studies*, 45(4):429–442, 1996.

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and Charles A.R. Hoare, editors. *Structured Programming*, chapter Notes on structured programming. Academic Press, 1972.

[DI87] N. Davies and C. Ikin. Auditing spreadsheets. *Austrialian Accountant*, pages 54–56, December 1987.

[Doe99] Mark Doernhoefer. Surfing the net for software engineering notes. *ACM SIGSOFT Software Engineering Notes*, 24(3):15–24, 1999.

[DW90] Weichang Du and William Wadge. The Eductive Implementation of a Threedimensional Spreadsheet. *Software-Practice and Experience*, 20(11):1097–1114, November 1990.

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, second edition, 1994.

[Fle02] Adam M. Flemming. Daniel bricklin. http://ei.cs.vt.edu/BRICKLIN.Flemming.HTML, January 2002. visited on 10th January 2002.

[Fur86] G. W. Furnas. Generalized fisheye views. In *Conference proceedings on Human factors in computing systems*, pages 16–23. ACM, April 1986.

[Gil90] D. J. Gilmore. Methodological Issues in the Study of Programming. In *Psychology of Programming*, pages 83–96. Academic Press, Ltd., 1990.

[GN99] Emden R. Ganser and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software Practice and Experience*, 1(5), 1999.

[gnu03] The gnu homepage. http://www.gnu.org, January 2003. visited on 25th January 2003.

[Gol03] Jody Goldberg. The gnumeric project. http://www.gnumeric.org, January 2003. visited on 10th January 2003.

[Gor98] Antonio Augusto Gorni. Spreadsheet applications in materials science. In Gordon Filby, editor, *Spreadsheets in Science and Engineering*, chapter 8, pages 229–260. Springer, Berlin, Heidelberg, 1998.

[Gro02] Thomas Grossman. Spreadsheet Engineering: A Research Framework. In *Spreadsheet Risks, Audit and Development Methods*, volume 3, pages 23–34. EUSPRIG, 7 2002.

[Har00] Mary Jean Harrold. Testing: A roadmap. In Anthony Finkelstein, editor, *The Future of Software-Engineering*. ACM Press, 2000.

[HNX90] J.-M. Hoc and A. Nguyen-Xuan. Language Semantics, Mental Models and Analogy. In *Psychology of Programming*, pages 139–157. Academic Press, Ltd., 1990.

[Hua75] J.C. Huang. An Approach to Program Testing. *Computing Surveys*, 7(3):113–128, September 1975.

[Hud89] Paul Hudak. Conception, Evolution and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[ISL95] Thomas Isakowitz, Shimon Shocken, and Henry C. Lucas. Toward a Logical/Physical Theory of Spreadsheet Modeling. *ACM Transactions on Information Systems*, 13(1):1–37, 1995.

[Jac75] M. A. Jackson. *Principles of Program Design*. Academic Press, 1975.

[JM00] Diane Janvrin and Joline Morrison. Using a structured design approach to reduce risks in End user spreadsheet development. *Information and Management*, 37:1–12, 2000.

[Jul98] Frank M. Julian. Using spreadsheets in chemical engineering problems. In Gordon Filby, editor, *Spreadsheets in Science and Engineering*, chapter 6, pages 171–202. Springer, Berlin, Heidelberg, 1998.

[Kan95] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 1995.

[KBB86] Krishna M. Kavi, Bill P. Buckles, and Narayan Bhat. A Formal Definition of Data Flow Graph Models. *IEEE Transactions on Computers*, C-35(11):940–947, November 1986.

[KCPR00] Jennifer Kreie, Timothy Cronan, John Pendley, and Janet Renwick. Applications development by end-users: can quality be improved? *Decision Support Systems*, 29:143–152, 2000.

[KCR00] Brian Knight, David Chadwick, and Kamalesen Rajalingham. A structured methodology for spreadsheet modelling. In *Spreadsheet Risks, Audit and Development Methods*, volume 1, pages 43–50. EuSpRIG, University of Greenwich, 7 2000.

[KE01] Stacy E. Kovar and Kristin Evans. Case: The Bakery, a cross-functional case study for introductory managerial accounting. *Journal of Accounting Education*, 19:113–122, 2001.

[KFN93] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software*. Van Nostrand Reinhold, second edition edition, 1993.

[Kok87] Peter Kokol. Some Applications of Spreadsheet Programs in Software Engineering. *Software Engineering Notes*, 12(3):45–50, July 1987.

[LC01] Daeyeol Lee and Marvin M. Chun. What are the Units of Visual Short-term Memory: Objects or Spatial Locations. *Perception and Psychophysics*, pages 253–257, 2001.

[Leh98] Stefan Leharne. Spreadsheet applications in aquatic chemistry. In Gordon Filby, editor, *Spreadsheets in Science and Engineering*, chapter 5, pages 141–170. Springer, Berlin, Heidelberg, 1998.

[Lin37] Carolus Linnaeus. *Systema Naturae*. 1737.

[LRO98] Jean Paul Le Roux and R.D. O'Brien. Applications of spreadsheets in earth sciences. In Gordon Filby, editor, *Spreadsheets in Science and Engineering*, chapter 4, pages 115–140. Springer, Berlin, Heidelberg, 1998.

[MA70] Zohar Manna and Pnueli Amir. Formalization of Properties of Functional Programs. *Journal of the Association for Computing Machinery*, 17(3):555–569, July 1970.

[MAC00] Roland Mittermeir, Yirsaw Ayalew, and Markus Clermont. User Centered Approaches for Improving Spreadsheet Quality. Technical Report TR-ISYS-MAC-1, Institut für Informatik-Systeme, Universität Klagenfurt, July 2000.

[Mat02] Richard Mattesich. Spreadsheet: Its first computerization(1961–1964). http://www.j-walk.com/ss/history/spreadsh.htm, January 2002. visited on 10th January 2002.

[MC02] Roland Mittermeir and Markus Clermont. Finding High-Level Structures in Spreadsheets. In *Proceedings of the 9th Working Conference on Reverse Engineering*, 2002.

[MCA00] Roland Mittermeir, Markus Clermont, and Yirsaw Ayalew. User Centered Approaches for Improving Spreadsheet Quality. Technical Report TR-ISYS-MCA-1, Institut für Informatik-Systeme, Universität Klagenfurt, July 2000.

[MOTU93] Hausi Müller, Mehmet Orgun, Scott Tilley, and James Uhl. A Reverse Engineering Approach To Subsystem Structure Identification. *Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[MWT94] Hausi Müller, Kenny Wong, and Scott Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Colloquium on Object Orientation in Databases and Software Engineering*, volume 62. Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS), 1994.

[Mye79] G. J. Myers. *The Art of Software Testing.* Wiley-Interscience, 1979.

[Neu98] Erich Neuwirth. Spreadsheets as tools in mathematical modelling. In Gordon Filby, editor, *Spreadsheets in Science and Engineering*, chapter 3, pages 87–114. Springer, Berlin, Heidelberg, 1998.

[Nev87] John M. Nevison. *The Elements of Spreadsheet Style.* Brady Book, 1987.

[NM90a] Bonnie Nardi and James Miller. An Ethnographic Study of Distributed Problem Solving in Spreadsheet Development . In *Proceedings of the conference on Computer-supported cooperative work* , pages 197–208. ACM, October 1990.

[NM90b] Bonnie A. Nardi and James R. Miller. The Spreadsheet Interface: A Basis for End User Programming. Technical Report HPL-90-08, HP Software Technology Laboratory, March 1990.

[NO01] David Nixon and Mike O'Hara. Spreadsheet auditing software. In *Spreadsheet Risks, Audit and Development Methods*, volume 2. EuSpRIG, University of Greenwich, 7 2001.

[ON88] J. R. Olson and E. Nilsen. Analysis of the cognition involved in spreadsheet interaction. *Human-Computer Interaction*, 3:4:309–349, 1988.

[Orm90] T. Ormerod. Human Cognition and Programming. In *Psychology of Programming*, pages 63–83. Academic Press, Ltd., 1990.

[Orv98] Wiliam Jay Orvis. Applying spreadsheets in physics and electronic engineering. In Gordon Filby, editor, *Spreadsheets in Science and Engineering*, chapter 2, pages 37–86. Springer, Berlin, Heidelberg, 1998.

[OW96] Gerald J. O'Brien and W. David Wilde. Australian managers' perceptions, attitudes and use of information technology. *Information and Software Technology*, 38:783–789, 1996.

[Pai97a] Jocelyn Paine. MODEL MASTER: Making Spreadsheets Safe. In *Proceedings of CALECO97*. CTI, 1997.

[Pai97b] Jocelyn Paine. Web-O-Matic: using System Limit Programming in a declarative object-oriented language for building complex interactive Web applications. In *Proceedings of the 8th REXX Symposium*. IBM, 1997.

[Pai01] Jocelyn Paine. Ensuring spreadsheet integrity with model master. In *Proceedings of EuSpRIG 2001*, volume 2. EuSpRIG, July 2001.

[Pai02] Jocelyn Paine. Modelmaster demonstration. http://users.ox.ac.uk/p̃opx/, July 2002.

[Pan97] Raymond R. Panko. Applying code inspection to spreadsheet testing. *Working Paper*, November 1997.

[Pan98a] Raymond Panko. What we know about spreadsheet errors. *Journal of End User Computing*, 10(2):15—21, 1998.

[Pan98b] Raymond R. Panko. What we know about spreadsheet errors. *Journal of End User Computing: Special issue on Scaling Up End User Development*, 10(2):15–21, Spring 1998.

[Pan00] Raymond Panko. Spreadsheet errors: What we know. what we think we can do. In *Spreadsheet Risks, Audit and Development Methods*, volume 1, pages 7–17. EuSpRIG, University of Greenwich, 7 2000.

[Pan02a] Raymond Panko. Spreadsheet research homepage. http://www.panko.com, July 2002.

[Pan02b] Raymond Panko. Spreadsheet research homepage. http://panko.cba.hawaii.edu/HumanErr/ProgNorm.htm, July 2002.

[Par94] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software Engineering*, volume 16, pages 279–287. IEEE, IEEE, 1994.

[Per95] William E. Perry. *Effective Methods for Software Testing.* John Wiley and Sons, Inc., 1995.

[Pfa01] John F. Pfaffenberger. Improve your spreadsheet. http://spreadsheetstyle.com/style/10tips.htm, 2001. visited on 1th August, 2002.

[PH96] Raymond R. Panko and Richard P. Halverson,Jr. Spreadsheets on trial: A survey of research on spreadsheet risks. *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, January 2-5 1996.

[PH97] Ray Panko and Richard P. Halverson. Are Two Heads Better than One? (At Reducing Errors in Spreadsheet Modeling). *Office Systems Research Journal*, 1997.

[Pow02] D.J. Power. A brief history of spreadsheets. http://www.dssresources.com/history/sshistory.html, January 2002. visited on 10th January 2002.

[Pre92] R. S. Pressman. *Software Engineering: A practitioner's Approach.* McGRAW-HILL, third edition, 1992.

[PS99] Ray Panko and Ralph H. Sprague, Jr. Hitting the Wall: Errors in Developing and Code Inspecting a "Simple" Spreadsheet Model. *Decision Support Systems*, 22:337–353, 1999.

[RCB$^+$00] Karen Rothermel, Curtis Cook, Margaret Burnett, Justin Schonfeld, T. Green, and Gregg Rothermel. Wysiwyt testing in the spreadsheet paradigm: An empirical evaluation. In *ICSE 2000 Proceedings*, pages 230–239. ACM, 2000.

[RCK02] Kamalasen Rajalingham, David Chadwick, and Brian Knight. Efficient Methods for Checking Integrity: A Structured Spreadsheet Engineering Methodology. *Informatica: An International Journal of Computing and Informatics*, 26(1), February 2002.

[RCKE00] Kamalasen Rajalingham, David Chadwick, Brian Knight, and Dilwyn Edwards. Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development. In *Proceedings of the 33rd Hawaii International Conference on System Sciences 2000*, volume 33. IEEE, 2000.

[Rea90] J.T. Reason. *Human Error*. Cambridge University Press, Cambridge, UK, 1990.

[RKC00] Kamalesen Rajalingham, Brian Knight, and David Chadwick. Classification of spreadsheet errors. In *Spreadsheet Risks, Audit and Development Methods*, volume 1, pages 23–34. EuSpRIG, University of Greenwich, 7 2000.

[RLDB98] Gregg Rothermel, L. Li, C. DuPuis, and Margaret Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *ICSE 1998 Proceedings*, volume 20, pages 198—207. IEEE, April 1998.

[RP00] Cliff T. Ragsdale and Donald R. Plane. On modeling time series data using spreadsheets. *ω-The international Journal of Management Science*, (28):215–221, 2000.

[RPG94] Jens Rasmussen, Annelise Mark Pejtersen, and L. P. Goodstein. *Cognitive Systems Engineering*. John Wiley & Sons, Inc., 1994.

[RPL89] Boaz Ronen, Michael Palley, and Henry Lucas. Spreadsheet analysis and design. *Communication of the ACM*, 32(1):84–93, January 1989.

[RRB00] James Reichwein, Gregg Rothermel, and Margret Burnett. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. In *Proceedings of the 2nd Conference on domain-specific languages*, volume 2, pages 25–38. ACM, 2000.

[Saj00] Jorma Sajaniemi. Modeling spreadsheet audit: A rigorous approach to automatic visualization. *Journal of Visual Languages and Computing*, 11(1):49–82, 2000.

[Sha92] John A. Sharp, editor. *Data flow computing: Theory and Practice*. Ablex Publishing Corporation, Norwood, NJ, 1992.

[Sha98] Gerry Shaw. Spreadsheets in molecular biology. In Gordon Filby, editor, *Spreadsheets in Science and Engineering*, chapter 7, pages 203–228. Springer, Berlin, Heidelberg, 1998.

[SK86] Mary Summer and Robert Klepper. End-user application development : Practices, policies, and organizational impact. In *Proceedings of the conference on Computer Personal Research*, pages 102–116. ACM, 1986.

[Sof02a] Southern Cross Software. Operis group plc. http://www.operis.com/oak.htm, 2002. visited on 11th September 2002.

[Sof02b] Southern Cross Software. The spreadsheet detective. http://www.uq.net.au/detective/home.html, May 2002. visited on 11th September 2002.

[Sta93] Marc Stadelmann. A Spreadsheet Based on Constraints. In *Proceedings of the sixth annual ACM symposium on User interface software and technology*, pages 217–224. ACM, ACM, November 1993.

[TBH82] Philip Treleaven, David Brownbridge, and Richard Hopkins. Data-Driven and Demand-Driven Computer Architecture. *Computing Surveys*, 14(1):93–143, March 1982.

[TMO92] Scott Tilley, Hausi Müller, and Mehmet Orgun. Documenting Software Systems with Views. In *Proceedings of the SIGDOC'92*, pages 211–219. ACM, 1992.

[TT96] Martin Tampoe and Bernard Taylor. Strategy software: Exploring its potential. *Long Range Planning*, 29(2):239–245, 1996.

[TT00] Thompson Teo and Margaret Tan. Spreadsheet development and 'what-if' analysis: quantitative versus qualitative errors. *Accounting Management And Information Technologies*, 9:141–160, 2000.

[VF92] George E. Vlahos and Thomas W. Ferratt. The use of information technology by managers of corporations in greece to support decision making. In *Proceedings of the conference on Computer Personal Research*, pages 136–151. ACM, 1992.

[VFK00] George E. Vlahos, Thomas W. Ferratt, and Georg Knoepfle. Use and perceived value of computer-based information systems in supporting the decision making of german managers. In *Proceedings of the conference on Computer Personal Research*, pages 111–123. ACM, 2000.

[Vos00] Gottfried Vossen. *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme.* Oldenburg, München, Wien, 4. korrigierte und erg. aufl. edition, 2000.

[WH93] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP.* Addison-Wesley, Reading, Massachusetts, third edition, 1993.

[Wil93] Nicholas P. Wilde. A WYSIWYC (What You See Is What You Compute) Spreadsheet. In *Proceedings of the 1993 Symposium on Visual Languages*, pages 72–76. IEEE, IEEE Computer Society Press, 1993.

[WM97] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau.* Springer, second edition, 1997.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *Computing Surveys*, 29(4):366–427, December 1997.

[ZZ93] W.M. Zage and D.M. Zage. Evaluating Design Metrics in Large-Scale Software. *IEEE Software*, 10(4):75–81, April 1993.