Comparing Spreadsheets

A New Strategy for Analysis, Detection and Aggregation of Spreadsheet Differences

Willem-Jan Meerkerk

TU Delft

Comparing Spreadsheets

A New Strategy for Analysis, Detection and Aggregation of Spreadsheet Differences

by

Willem-Jan Meerkerk

to obtain the degree of Master of Science
in Computer Science
at the Delft University of Technology,
to be defended publicly on Thursday June 27, 2019 at 1:00 PM.

Student number:	4142160	
Project duration:	September 3, 2018 – June 27, 2019	
Thesis committee:	Prof. Dr. A. van Deursen,	TU Delft, supervisor
	Dr. A.E. Zaidman,	TU Delft
	Dr. C.C.S. Liem,	TU Delft
	Mateo Mol,	Infotron, supervisor

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, The Netherlands
www.ewi.tudelft.nl

Infotron
Linnaeusstraat 2C
Amsterdam, The Netherlands
www.infotron.nl

Abstract

Comparing spreadsheet files is a new, unexplored research domain in computer science. Methods for regular file comparison are not straightforwardly applicable to spreadsheet files, because they are fundamentally different. Spreadsheets are binary files, the structure of spreadsheets is two-dimensional, they contain both data and calculations, and the content exists on different abstraction levels. Fundamental challenges in the spreadsheet comparison problem include: change propagation, performance, 2D alignment, the grouping of data, and movement detection. One simple modification can change the whole structure and model of the spreadsheet. The aim of a good file comparison method is to show the actual changes made by an end-user, not the propagated changes.

In this thesis, a new pipeline-based approach for comparing two spreadsheet files is proposed. The spreadsheet comparison is solved in three phases: (1) structure analysis, (2) change detection, and (3) change aggregation. The essential element in this approach is passing on information from the structure analysis to the change detection. In addition, the detected changes are forwarded to the aggregation such that changes are converted into clear, understandable differences. The final comparison result, therefore, provides information at different levels of abstraction. New solutions like cell hashing, an optimized 2D alignment using longest common subsequences, and different algorithms for comparing worksheets, defined names, rows, columns and cells have resulted in a state-of-the-art spreadsheet comparison approach.

In addition, this thesis presents a prototype demonstrating the proposed concepts in practice. This tool, called CompareXL, is a stand-alone application that can compare two spreadsheet files. The user interface is built with a multi-level interaction design, showing the comparison results intuitively to the end-user.

Spreadsheets are commonly used and many problems arise from spreadsheet versioning issues. Spreadsheet users have user needs related to overview, validation, completeness, error resolving, visualization, and evolution. A spreadsheet comparison program is helpful to address these user needs. This research shows that it is beneficial to involve users in the exploration of solutions, because comparing spreadsheets is not only a technical problem but also a user problem. The outcome of this research offers multiple directions for future work, with the ultimate goal that all spreadsheet version problems will soon belong to the past.

Preface

Looking back at the end of this graduation project I am grateful for the last 9 months of research. What started as a large and complex project has developed into a relevant and concrete end product. I was surprised how quickly I had a first working prototype. Apparently, the spreadsheet comparison problem was solvable. I am very pleased that I was able to carry out this project in the context of Infotron. For me, good theoretical knowledge always goes hand in hand with smart applications in practice. During my study Computer Science, I have learned many fundamentals to develop intelligent solutions. I really enjoyed putting these into practice during this challenging graduation project.

There are a lot of people I would like to thank. First of all, thank you Arie for being my thesis supervisor and for all your valuable input during our meetings. Thank you Mateo, and all the colleagues at Infotron, for your help and feedback during my research, the opportunity to build a nice spreadsheet compare tool, and all the enjoyable board game sessions ;). Thanks to my family, and especially my parents, for the warm home and all the freedom you have given me. Thanks to my roommates and friends, for all the fun we had during this last year of my student time. Angeline, thank you for your love and unconditional support.

Finally, I want to give thanks to God the Father, for his infinite love, grace, joy, and support in my life. His glorious presence was my greatest inspiration and motivation last year. I hope that my work on this graduation project may reflect some of His great work of creation.

Willem-Jan Meerkerk
Delft, June 2019

Contents

1 Introduction

1.1 Motivation

File comparison is a topic that has existed ever since the earliest times in the history of computing [1–3]. For example, the Unix `diff` utility was developed in the early 1970s. It is in the nature of working with computers, that multiple versions of computer files arise. In general, many operations users perform on computers are reflected back in creating a new version of a computer file. Ever since humans work with computer files, there has been the need to compare different versions of their files.

File comparison tools support users to understand the differences they made and help to fix errors. It is often unclear what has changed between different versions of a computer file. A user cannot memorize all the modifications that were made, and often a computer file is edited by multiple users. A file comparison tool gives insight into the changes and can help users to understand the evolution of their (possibly shared) computer file. Furthermore, a clear overview of all differences between versions can help to identify faults.

Extensive research in the field of file comparison has been conducted. Many methods have been developed, mostly for comparing text files. Most file comparison tools compute the longest common subsequence of two files and highlight differences for all parts that are not in the longest common subsequence. As a result, many file comparison software algorithms and tools are available, and the problem of comparing text files is now well understood.

Comparing spreadsheets however is a much less explored domain. The comparison of spreadsheet files is complex, due to various reasons.

- First of all, a spreadsheet file differs from a regular text file, because the data is stored in binary format.
- While the structure of regular files is one-dimensional (collection of lines), the structure of a spreadsheet file is two-dimensional (collection of rows and columns), and data exists on different levels (workbook, worksheets, and cells).
- Regular files have clear atomic data items (characters in a line), but data items in spreadsheet files are non-atomic (cells either contain a value or formula, or both, and have optional display and formatting properties).

- Regular files only contain pure data, spreadsheet files contain combinations of data and calculations.
- A good file comparison shows the actual changes an end-user has made. In the context of comparing spreadsheets, one simple user action (e.g. inserting a column) can change the whole structure and model of the spreadsheet file (e.g. all cells to the right are moved, and all formulas dependent on these cells are changed).

In the current research, spreadsheets in general and spreadsheet versioning are well understood. Still, there is no complete, state-of-the-art spreadsheet comparison approach described in the literature. The limited number of papers only provide small pieces of the whole puzzle of spreadsheet comparison.

Besides the scientific work, the available commercial spreadsheet comparison tools all have their shortcomings. State-of-the-art tools lack the ability to give clear overviews, have problems comparing large and complex spreadsheets, and are unable to make a complete diff between two spreadsheet files. None are built with the aim to be an all-purpose compare tool, that can efficiently compare two spreadsheet files of all different types.

1.2 Research Context

This graduate project is performed at the company Infotron, located in Amsterdam. The ultimate goal of Infotron is to drastically increase the quality of spreadsheets worldwide. Infotron serves clients varying from accountants, consultants, and financial experts as well as professionals from the high tech, energy, logistics, transport, and engineering industries. In all the experience gained by Infotron in the last years, it appeared that many spreadsheets end-users are struggling with multiple spreadsheet versions. Several problems arise and there is no simple, stable and well-performing solution to compare spreadsheets.

Infotron has many connections with end-users and Excel experts, which are of great significance in this research. Furthermore, the whole team of colleagues with Excel experience are available to support this research. The collaboration of the TU Delft and Infotron has earlier led to a publication about analyzing and visualizing spreadsheets [4], and more research is still going on. The combined effort in this graduate project will make this research of both commercial and scientific value.

1.3 Research Objective

The goal of this thesis is to get insight into the complex comparison of spreadsheet files, and to develop a prototype tool that can actually compare two spreadsheets files efficiently. To achieve these goals, we need:

1. to fundamentally understand the spreadsheet comparison problem,
2. to develop a strategy to compare the differences on different levels of detail, and
3. to find a smart way to present the comparison results.

We therefore investigate the following research question:

How to develop an intelligent and applicable strategy for comparing two spreadsheet files?

1.4 Research Approach

We decide to use prototyping as the main research tool [5]. Since it is difficult to predict the main research challenges beforehand, the best method is to work with comparing spreadsheets in practice. After an initial stage of background research in literature and existing tools, we start the development of a running prototype: a tool that can compare two spreadsheets. Along the road, we expect to face fundamental challenges that make the comparison difficult. New findings and solutions addressing these challenges are of scientific significance and will be reported in this thesis.

We involve different Excel end-users in this research, in the form of user interviews. The interviews are held at companies, at the university, and with Excel experts. The goal is to obtain more insight into the context: find out how users maintain different spreadsheets versions and which problems they face. In the interviews, we ask users to explain their user needs for having a compare tool. Because the result of a spreadsheet comparison is not trivial, we talk about the expected outcome and how users can best be helped with a comparison tool. We expect that the interviews will clarify the challenges from a user perspective.

Based on the input of end-users and our own experience during the development of the compare tool, we define a list of requirements related to the spreadsheet comparison problem. A requirement describes underlying user needs or technical challenges that exist when comparing spreadsheets. We study the literature and existing tools, and expect to conclude that the current research and available tools are unable to give sufficient solutions to the requirements. In this thesis, we will propose new solutions to the underlying user needs and technical challenges. Furthermore, we will present a prototype tool, called CompareXL, to demonstrate the outcome of this research in practice.

The evaluation of this research is two-fold. First, we set up experiments to test the technical aspects of our comparison approach. These automated tests evaluate criteria like correctness, stability, and performance. For instance, we test the performance of the compare tool on very large spreadsheet files. Secondly, we perform user tests to test the user aspects of our comparison approach. We present the prototype to end-users and investigate the quality of the comparison side by side. We also ask to what extent the new solutions meet the goals of the user for comparing spreadsheets. These user tests will evaluate criteria like correctness and completeness and validate the user needs.

We conclude this research with a reflection on the spreadsheet comparison problem, based on 9 months of experience with this subject. We discuss the results of the evaluation, to conclude to what extent the new contributions of this thesis solve user needs and technical challenges. Finally, we end with a list of recommendations, to encourage further exploration of this research domain on spreadsheet comparison.

1.5 Thesis Structure

To answer the research question, the rest of this thesis is structured as follows. In Chapter 2, we give an overview of the current literature and existing tools, to understand the background of the spreadsheet comparison problem. In Chapter 3, we present an analysis of the spreadsheet comparison problem, to find out why it is so difficult to compare two spreadsheet files. We summarize the problems in terms of technical challenges and user needs. In Chapter 4, we propose nine solutions for comparing spreadsheets, together representing our new spreadsheet comparison strategy. The solutions solve technical challenges and address user needs. In Chapter 5, we show our prototype tool, demonstrating the proposed concepts in practice. In Chapter 6, we evaluate to what extent our strategy is intelligent (solving technical challenges) and applicable (fitting the user needs). Finally, in Chapter 7, we wrap up our research and present the contributions, conclusions, and suggestions for future work.

2 Background

2.1 Related Work

In this section we explain the prerequisite knowledge to understand the concepts introduced in this thesis. Furthermore, we summarize the related work that has been done in the field of spreadsheet versioning and comparison.

We will see that there is a lot of literature available for file comparison in general, but limited work has been done for spreadsheet comparison. The ideas introduced in the available papers are only pieces of the whole puzzle of spreadsheet comparison. They help to understand the context of spreadsheets better, and some pieces are re-used and optimized in our solution for spreadsheet comparison.

2.1.1 File Comparison

The goal of a file comparison program is to report differences between two files A and B. A regular text file can be seen as a collection of lines. The lines are called A_i and B_j, with $i = 1,...,n$ and $j = 1,...,m$. The differences are a collection of line changes between the two files. This is frequently called the 'diff' between A and B. The diff will tell how to change the lines in the first file to make it match the second file.

A diff does not only provide insight in the changes, it is also an important building block of version control systems. If a complete diff between two files can be made, it is sufficient to only save the original file A and the diff between A and B. When more versions of a file are created, only the diffs need to be stored. The benefit is that the space required to store the different files is reduced: only the original file (probably large) and the diffs (probably small) are saved. Other than that, version control systems provide access to intermediate versions by applying diffs to the original file. Because the difference information is so important for version control systems, a complete and efficient file comparison algorithm is required.

The simplest approach to compute a diff is to scan files A and B line by line, reporting differences for the lines that are unequal. This method will work if only individual lines are changed. However, with inserted or deleted lines the files will become out of sync and all the other lines will be reported as different.

Another straightforward approach is to scan both files A and B until two unequal lines are found. At that point the files can be scanned forward until again a matching pair of lines is found. If that occurs, the lines in between can be reported as differences. Lines skipped in A are deletions and lines skipped in B are insertions. However, this method fails if there are lines deleted in A and inserted in B at the same position. It is not clear how to resynchronize, resulting again in a large set of differences. It appears that there is no trivial strategy for comparing simple text files.

In 1976, Hunt and McIlroy [1] introduced one of the first file comparison algorithms. This algorithm formed the starting point for the `diff` utility in Unix. The outcome is a minimal list of line changes to bring the first file into agreement with the second. The line changes that are reported are delete (<) and append (>), with the corresponding line number. An example output of the `diff` algorithm is shown in Figure 2.1.

```
$ cat file1.txt         $ cat file2.txt         $ diff file1.txt file2.txt
alpha                   whiskey                 0a1
bravo                   alpha                   > whiskey
charlie                 bravo                   3,4c4,6
delta                   xray                    < charlie
echo                    yankee                  < delta
foxtrot                 zulu                    ---
golf                    echo                    > xray
                                                > yankee
                                                > zulu
                                                6,7d7
                                                < foxtrot
                                                < golf
```

Figure 2.1: Output of the `diff` utility on two simple text files.

The key part in their algorithm is to solve the *longest common subsequence* (LCS) problem. The longest common subsequence is the sequence that contains the maximum number of lines in A and B that are equal. In other words, finding the LCS gives the lines in A and B that are unchanged. When the LCS is found, it is easy to extract the line changes. Lines that are in A but not in the LCS are deleted, and lines that are in B but not in the LCS are appended. The result of the Hunt-McIlroy algorithm is a list of all deletes and appends, summarized per segment of consecutive lines.

To gain practical efficiency, various techniques are described by Hunt and McIlroy [1]. Matching lines from the beginning and the end are stripped, to reduce the search space. There is no need to compare identical parts at the beginning or the end of the files. Furthermore, each line of a file is hashed into one computer word to make comparison of reasonably large files possible. A hash is a 'fingerprint' of a line. Not all individual symbols in a line will be considered, but only the hash. This will speed up the comparison and reduces the required memory space. More optimizations like presorting into equivalence classes, merging by binary search, and dynamic storage allocation are used to obtain better performance.

Miller and Myers [2] present a different algorithm to compare two files. Again, the lines are treated as indivisible objects, so a file with n lines is considered as a string of n symbols. The approach brought up here finds the *edit distance* between the two strings of symbols. This edit distance is the length of the shortest sequence of insertions and deletions that will convert the first string to the second. They propose a systematic approach using three rules, to build up a solution from the solutions to subproblems. The algorithm starts by finding identical prefixes of A and B, the case where the edit distance is 0. Then, three rules are applied to determine all combinations with an increasing edit distance (1, 2, and so on). This continues until the algorithm has found a shortest possible edit script (sequence of edit commands) for converting the first input string into the second.

This method is particularly efficient when the difference between the two files is small compared to the files' lengths. Miller and Myers [2] observed that their program often ran four times faster than the `diff` command. However, for files with no lines in common, the program ran out of memory and reported only a lower bound on the edit distance.

Their paper brings up the important point that an efficient file comparison algorithm depends on the content of the files to compare. It can help to select a file comparison algorithm when assumptions can be made about the expected overlap. One can even build a mechanism to dynamically switch to the most efficient algorithm based on predictions on the number of differences. For regular file comparison, one would expect at least some, and probably a large amount of overlap between the files. This is also true for spreadsheets. When almost everything has changed it can be more useful to report that, than to create a huge set of differences.

Of course, further optimizations are possible to the file comparison approach. In practice, most file comparison tools find the longest common subsequence between the two files. Any line not in the longest common subsequence is reported as an addition or deletion. Optimizing the file comparison problem is in that sense equivalent to optimizing the longest common subsequence problem. This problem will be regarded in the next section.

2.1.2 Longest Common Subsequence

There is extensive literature available about the LCS problem. A valuable survey of LCS algorithms is given by Bergroth et al. in [6]. We repeat the formal definition and paraphrase the overview of the approaches to solve this problem, summarized in [6] and their follow-up paper [7].

We are comparing two strings $X[1..n]$ and $Y[1..m]$, with length n and m respectively. A *subsequence* $S[1..s]$ of $X[1..n]$ is obtained by deleting $n-s$ symbols from X. A *common subsequence* of $X[1..n]$ and $Y[1..m]$ is a subsequence that occurs in both strings. The *longest common subsequence*, denoted by $lcs(X,Y)$ is a common subsequence of maximal length. The length of the $lcs(X,Y)$ is denoted by r. The longest common subsequence does not need to be unique. There may be several common subsequences with maximum length r.

Example 1. Let $X = abcdbb$ and $Y = cbacbaaba$. Then $lcs(X,Y) = acbb$, or $lcs(X,Y) = bcbb$. The length of the longest common subsequence $r = 4$.

There are two variants of the LCS problem: sometimes only the length r should be determined, sometimes the sequence itself has to be produced. Basically, every algorithm calculating r only can be modified to solve the LCS also by introducing additional bookkeeping that records the algorithm progression. After r is known, the LCS can be constructed by backtracking the selections made.

The very first feasible approach to solve the LCS problem is introduced by Wagner and Fischer [8] in the year 1974. Their method is based on a dynamic programming technique. Each character lying in the input string X is compared with characters from each position of Y. This leads to the calculation of a LCS for all possible prefixes of X and Y. After processing all prefixes, the length r can be found in the bottom right corner of the dynamic programming table. With backtracking, the actual LCS can be found. The Wagner-Fischer algorithm has time and space complexity of $O(nm)$.

Although the time and space complexity of this dynamic programming technique is 'only' quadratic, it tends to be too large for many applications. Due to this, several heuristics have been devised. Sometimes it is not the time complexity that is crucial but space becomes the limiting resource. The whole set of LCS algorithms, which improve the dynamic programming approach, fall into three categories: row-by-row methods, contour methods and diagonal methods. More details about the different algorithms and a performance evaluation can be found in the survey paper [6].

A recent theoretic study on the LCS problem is presented by Bringmann et al. [9]. They conclude that a textbook algorithm, like Wagner-Fischer, solves LCS in time $O(n^2)$, but although much effort has been spent, no $O(n^{2-\epsilon})$-time algorithm is known. Recent work shows that such an algorithm would refute the Strong Exponential Time Hypothesis (SETH). Despite the quadratic-time barrier, for over 40 years an enduring scientific interest produced faster algorithms for the LCS problem and its variations. Particular attention was given to identification and exploitation of input parameters that yield strongly subquadratic time algorithms for special cases of interest, e.g., differential file comparison. This line of research was successfully pursued until 1990, when significant improvements came to a halt.

We conclude that for comparing larger sequences, it is important to select a fast algorithm. It will be useful to start with an implementation of the 'naive' Wagner-Fischer algorithm [8]. Then, other algorithms can be implemented to optimize the performance. The correctness can be compared with the original implementation. Based on the survey paper [6], it seems fruitful to test different implementations: Hunt-Szymanski [10], Kuo-Cross [11] and Wu, Manber, Myers, and Miller [12].

2.1.3 Spreadsheet Versioning

Nowadays, spreadsheets are one of the most commonly used programming tools [13]. Schmitz et al. [14] present a survey done among 180 spreadsheet users. One of their survey results is that spreadsheet users do not formally use version control. 74% of the respondents typically has two or more versions of a spreadsheet, and 64% of the respondents tracks different spreadsheet versions by embedding dates or version numbers in the filenames. Only 4% of the respondents save their spreadsheets on OneDrive, which includes rudimentary, but functional, version tracking. Despite the popularity of version control systems like Git, it has not been adopted by spreadsheet users. This is verified in another study [15], where it is estimated that 70% of the spreadsheet users performs manual version control. Spreadsheet users often create new versions, but do not use version control systems.

Jansen et al. [16] obtained more understanding on how spreadsheets evolve over time. Based on two case studies on two different sets of spreadsheets, they conclude that most spreadsheets grow over time, both in the number of non-empty cells as unique formulas. The growth of the data is caused by adding more data points to the analysis. Growth in unique formulas is caused by adding new functionality to the spreadsheet. Unique formulas were changed in almost every version, only the number of changes differed per version. The motivation for most formula changes is to implement a new feature request, to improve the maintainability of the spreadsheet, or to correct errors that are made in previous versions.

The results of the evolution study were discussed with the creators of the spreadsheet, and suggestions were made that could support a spreadsheet user in making a better model. First of all, they advice to summarize the changes made in a new version. For example, choosing to only show the changes in unique formulas instead of all formulas helps to present the changes in a concise way to the creator of the spreadsheet. A summary helps the creator of the spreadsheet to detect if all changes are intended and correctly implemented. Secondly, it will be useful if a list of formulas that were frequently changed in earlier versions of the spreadsheet is generated. This could function as a checklist to make sure that all necessary changes have been made. Thirdly, formulas that are candidates for refactoring should be suggested. Formulas that have to be changed in every new version can often be rewritten in such a way that changes are not necessary. Lastly, sudden drops or spikes in changing and growing rates should be highlighted. They sometimes indicate anomalies in the spreadsheet.

The representativeness of the selected set of spreadsheets analyzed in the paper is limited. Still, general conclusions about spreadsheet evolution, based on real-world files from industry, can be made. This study shows that a compare tool would help end-users in the maintainability of their spreadsheets. All the suggestions require a comparison strategy that is efficient, able to summarize, able to detect structural changes, and able to process the detected changes. Another contribution of this paper is an algorithm called FormulaMatch, that can match unique formulas of two different versions of the same spreadsheet.

Xu et al. [17] study a concrete example of spreadsheet versions: spreadsheet templates. Spreadsheet templates are semi-finished spreadsheets with predefined formulas and table layouts. New spreadsheets based on these semi-finished spreadsheets are created. These templates are used to help end-users create spreadsheets efficiently. The research collected 47 predesigned templates and 490 instances, based on spreadsheets from the Enron corpus. The researchers observed four kinds of errors introduced in the instances: missing formulas, range errors, unnecessary formulas and inconsistent formulas. In the usage of 79% of the predesigned templates, at least one error is introduced, and missing formulas are the most common type of errors. Spreadsheet templates are often used by end-users, and their usage reflects how end-users work with spreadsheet versions in general. Therefore, it is reasonable to conclude that often errors are introduced when new spreadsheet versions are created. A compare tool can help to identify errors made in spreadsheet versioning.

Dou et al. [18] introduced a versioned spreadsheet corpus, called VEnron, which was extracted from the Enron Corporation email archive. They developed a semi-automated approach where they cluster spreadsheets that likely evolved from one to another into evolution groups. The clustering was made on various fragmented information, such as spreadsheet filenames, spreadsheet contents, and spreadsheet-attached emails. In each evolution group, they recovered the version information of the spreadsheets. That made it possible to identify interesting issues that arise from spreadsheet evolution. One step of their approach is to check whether all the spreadsheets in a group share similar table structures and formulas. For that, they use the Microsoft Spreadsheet Compare tool to compare two versions of spreadsheets. This tool is included in our existing tools survey in Section 2.2. They have also used this tool to investigate what changes happened during the evolution in each evolution group. The changes inspected are: structural changes, entered value changes, formula changes and calculated value changes.

The most important conclusions based on the VEron evolution analysis are the following. First of all, 72.2% of the evolution groups involve more than one committer (user who modified the spreadsheet). This suggests that spreadsheets were often maintained by multiple users. Most (87.5%) evolution groups have less than 5 committers. This indicates that spreadsheets were often updated by a small group of users. Secondly, users deploy various ways to represent version information of spreadsheets in the filename. Time is the most common type (66.7%), for example users use the months (May, Jun, July, or Aug) to distinguish different versions. Sometimes (18.9%) indices in the filenames' prefix or suffix are used (e.g. v2, v3, v4). But often (39.4%) users did not use any version representation to distinct different versioned spreadsheets. It also appeared that in 18.3% of the evolution groups users adopted more than one version representation across multiple versions. Thirdly, various spreadsheet changes occurred in all evolution groups. For cell changes in general, entered value changes are much more common than formula changes (20.3% vs. 4.2%). Still, formula changes occur often (in 19.0% of all formula cells) during evolution. For structural changes, row changes are much more common than column changes (11.7% vs. 3.5%). Lastly, in 16.9% of all groups a later spreadsheet version contained more errors than the previous version. New errors are easily introduced during evolution, which we also concluded from [17].

An improved versioned spreadsheet corpus, called VEnron2, was introduced in [19]. The authors observed that the versioned spreadsheets in each evolution group of VEnron exhibit certain common features, like similar table headers and worksheet names. Based on this observation, they proposed an automatic clustering algorithm, SpreadCluster. This algorithm learns the criteria of features from the versioned spreadsheets in VEnron, and then automatically clusters spreadsheets with similar features into the same evolution group. The evaluation result shows that SpreadCluster could cluster spreadsheets with higher precision and recall rate than the filename-based approach used by VEnron. They also applied SpreadCluster on the other two spreadsheet corpora: FUSE and EUSES. The result of their research are multiple datasets with versioned spreadsheets, which are available online. The datasets are not only useful to study spreadsheet evolution, they also constitute a rich resource to evaluate a compare tool on many versions of real-world spreadsheets.

2.1.4 Comparing Spreadsheets

The essential part of comparing spreadsheets is the comparison of two worksheets. A worksheet is a 2D collection of cells, indexed by row and column. Every cell can either contain a value or a formula. Not only individual cells can be changed, but also structural changes can occur in worksheets, such as row/column insertions and row/column deletions. Currently, there are two papers that describe an algorithm to compare the content of two worksheets, SheetDiff [20] and RowColAlign [21].

For SheetDiff [20], the goal is to develop a method for identifying the changes between two spreadsheets, with the explicit purpose to present them to users in a concise form. The built-in 'track changes' functionality of Microsoft Excel is mentioned, but this has two limitations.

Firstly, change tracking lacks the ability to compare two spreadsheets in general, because it can only track changes that are recorded. If it is not turned on, none of the changes will be reported. Secondly, the display of the changes is rather poor according to the authors. There is no clear overview of all the differences. Two other commercial tools are mentioned: DiffEngineX and Synkronizer. According to Chambers et al. [20], one of the major problems with these systems is that they represent the differences by coloring two different spreadsheets. This requires users to continuously switch between the two sheets and rely on their memory to judge the differences. It would be better if the changes are shown in context on one sheet, then the comparison could be done in place.

SheetDiff is a greedy, iterative algorithm based on the idea of finding edit operations that each transform worksheet A into B as much as possible. SheetDiff compares A and B to find row r and column c that contain the highest number of different cells. It computes four spreadsheets ($A-r$, $A-c$, $A+r$, $A+c$) formed by deleting r or c from A, or by inserting B's version of r or c into A. Of these four options, it chooses the one that minimizes the number of cells that remain different relative to B. This choice effectively selects a row or column insertion or deletion (which SheetDiff appends to its output, a list of edit operations) and also yields a worksheet $A^{'}$ that is more similar to B than A was. SheetDiff then uses this worksheet $A^{'}$ for the next iteration, and the algorithm continues iterating until a similarity metric threshold between A and B is achieved. Then it computes all remaining non-identical cells by pairwise comparison and appends an edit operation for each differing cell. The output is a list of edit operations, which are row/column insertions, row/column deletions and individual cell changes.

Harutyunyan et al. [21] present a new algorithm that can identify differences between two spreadsheets. The algorithm, RowColAlign, is a dynamic programming algorithm based on the idea of finding maximal subsequences of rows and columns that are nearly unchanged in A compared to B. RowColAlign takes A and B and first attempts to find what they have in common. This common subsequence of rows and columns is referred to as the target alignment $T(A,B)$ of A and B. Everything they do not have in common must be an edit operation and must therefore appear in the output list (the list of edit operations that transforms A into B). This is either an insertion if the row/column in question is in B but not A, or a deletion if the row/column is in A but not B. The areas in common need not be identical, a small number of cell-level differences are tolerated and represented as cell-level edits.

The algorithm acts in four phases. In the first, it compares A and B to identify which rows should be deleted. In the second phase, it compares A and B to identify which columns should be deleted. In the third phase, it deletes the rows and columns and performs pairwise comparisons on the remaining individual cells inside $T(A,B)$ and $T(B,A)$. Finally, if any row or column pair in T remains mostly different, it is represented as a row or column edit, rather than multiple cell-level edits.

An empirical evaluation of both algorithms is presented in the same paper. A planted model is used to generate test cases (randomly mutated spreadsheets), for a range of parameters: spreadsheet width and height, alphabet size, probability of row and column deletion, and probability of cell-level edit. The authors found that the SheetDiff algorithm made errors on moderately difficult spreadsheets with different suggestive patterns. Overall, the error rates of SheetDiff are widely-scattered, indicating that unpredictability is a consistent trait of SheetDiff. In contrast, the new RowColAlign made no errors. This algorithm always recovered precisely the correct target alignment and cell-level edits. For extremely difficult test cases (large spreadsheets with many changes) SheetDiff has error rates of 52%, 60% and 96% for different alphabet sizes. RowColAlign did not make any errors even on these more difficult test cases.

Assuming that A and B are $n \times n$ arrays, the space complexity of RowColAlign is $O(n^2)$. The time complexity is $O(n^4)$, since filling an entry in the dynamic programming table takes $O(n^2)$ time. This run time is quadratic in the input area. This is better than the worst case asymptotic algorithmic complexity for SheetDiff, which occasionally fails to terminate. It sometimes makes a row or column edit and then, in a subsequent step, makes another edit that cancels out the first edit (e.g. first a row insertion, then a row deletion). This problem is caused by the fact that when SheetDiff selects a row or column to insert or delete, this operation not always improves the final similarity between the spreadsheets. The fundamental issue is that similarity is not always a monotonically increasing function as successive edits transform one spreadsheet into another. Therefore, a greedy, local optimization algorithm such as SheetDiff cannot be guaranteed to terminate, and cannot be guaranteed to find the right answer every time. A dynamic programming algorithm, such as RowColAlign, is better suited to the problem at hand.

Essentially, RowColAlign is a two-dimensional generalization of the classic dynamic programming algorithm for solving the one-dimensional longest common subsequence problem. In other words, RowColAlign solves the problem of aligning two 2D arrays. A worksheet in a spreadsheet file, with rows and columns, can be seen as a 2D array. Therefore, this algorithm seems to be a good candidate to compare the content of two worksheets.

However, there are still some limitations left. The RowColAlign algorithm is not able to capture all operations in a spreadsheet, such as copy-paste (where affected regions might not subtend an entire row or column) or cell-fill (where a cell formula is dragged over and applied to other cells). Moreover, spreadsheets with cells containing formulas are not considered in the paper. One row insert can change all the formulas in the cells below. Another important issue is the space and time complexity for large spreadsheets. Consider two worksheets with $n = 10{,}000$ rows, which is not uncommon in practice. Then the algorithm will run out of memory, as it will require $O(n^2)$ space, which are 100,000,000 array entries. Basically, every row is compared with every other row and every column is compared with every other column. Hence, heuristics or other optimizations are required to compare the contents of two very large worksheets.

2.1.5 Other Approaches

There are a few other attempts done in the field of comparing spreadsheets.

Jansen et al. [16] developed an algorithm called FormulaMatch, able to detect and visualize changes in formulas. The researchers begin with the statement that analyzing changes between two spreadsheets is difficult. Simple structural changes to a worksheet, like inserting a row or a column, can lead to a myriad of changes. The number of detected changes can be reduced if only changes in the individual formulas are taken into account; all changes in structure, data and formatting are ignored in their algorithm. However, it is still possible that this results in thousands of changes. More complex spreadsheets contain a lot of formulas. For example, one of the models in their case studies contained about 100,000 formulas. In such a spreadsheet, a single change like moving a cell, can lead to a change in thousand related formulas. Presenting all these changes to the user, will not help them to understand the risks induced by the applied changes. For this reason, only the unique formulas are considered in the FormulaMatch algorithm. The R1C1 notation of a formula is used to detect the unique formulas in a spreadsheet. In the default A1 notation all formulas have absolute references, e.g. `=SUM(B1:B10)`. In the R1C1 notation all formulas have relative references, e.g. `=SUM(R[-10]C:R[-1]C)`.

When comparing two spreadsheets, FormulaMatch starts with detecting the unique formulas. For every formula in the first version V0, the equivalent in the second version V1 has to be found. To find a matching formula the similarity of the formulas in their R1C1 notation is analyzed, together with additional properties. This gives good results when the number of formulas in both versions is the same, but not when formulas have been added or deleted. To correctly handle the addition or deletion of formulas, a more sophisticated matching algorithm is applied. The result of FormulaMatch gives for each formula in V0 a matching formula in V1. Based on the similarity scores it is known if the formula has been changed or not.

This paper shows that it is useful to consider unique formulas using the R1C1 notation for determining formula differences in spreadsheets. Furthermore, the matching algorithm can serve as a good heuristic in detecting movements of blocks of formula cells.

Schmitz et al. [22] proposed a method and tool to identify real-world formula errors. After identifying the versions of the same spreadsheet, an automated analysis of the differences between the files is performed. Two types of differences are detected: modified cells and moved cells. The detection of modified cells focuses on changes in formulas, changes in values are not considered. In the calculation of formula differences the R1C1 notation is used, in which copy-equivalent formulas share the same formula. The detection of moved cells uses a heuristic regarding the surrounding of the changed cells. If the formula of a changed cell is found at a different location in the changed worksheet, with an identical surrounding area, it is assumed that the whole area was moved to the new location.

This paper shows that it is often useful to use heuristics for determining differences in spreadsheets. Another important contribution is the Enron Error Corpus, a dataset of 30 spreadsheets with errors. Two versions are available: the faulty spreadsheet and the corrected spreadsheet. It has been documented which cells have been changed between the two versions.

Moreira [23] introduced the tool SheetGit, which aims to be a version control solution for spreadsheet end-users. The paper claims that a set of fundamental features is implemented: creating versions, switching between them, the option to see the differences between two versions, and sharing them through the Internet. However, it appears that this publication is more a proposal of the ideas of the author; the figures are still mock-ups, no details about the spreadsheet comparison are given and no evaluation of their tool is presented. The tool should be an add-in of Excel, which offers the ability of version management in a similar way of using Git. The differences between versions will be shown inside Excel itself. If the user selects a particular version, the mouse/cursor will recreate the actions required to turn one version into the other. This will be impractical for a large number of differences. We conclude that the paper about SheetGit only initially explores the idea of version management for spreadsheets, but it is incomplete and lacks sufficient details.

2.2 Existing Tools

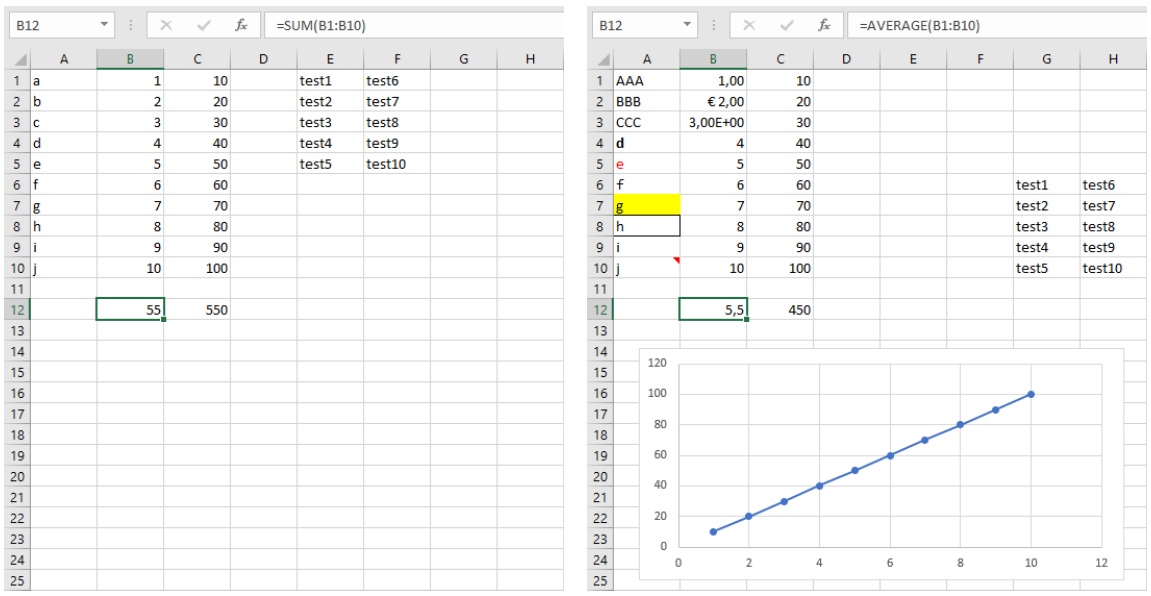
Apart from the literature, some commercial spreadsheet comparison tools are already available. We conducted a review to test to what extent these tools can successfully compare two spreadsheet files. In this section, we present the currently known spreadsheet compare tools, explain their approach and compare their features. We will see that all tools have their shortcomings. We think these weak points reflect the underlying problems that make the spreadsheet comparison difficult. Therefore, we will end this section with a reflection on the shortcomings. We conclude what needs to be solved to develop a better spreadsheet compare tool.

2.2.1 Approach

The goal of this review is two-fold: (1) to identify how the tools work from a user perspective, and (2) to test the quality of the comparison based on some test files. These test files are created beforehand, so that all tools can be compared using the same sample files.

Firstly, we created a sample spreadsheet S_1 with dummy content covering most of the Excel features. It contains features like cells with values, formulas, display properties, number formatting and comments, graphs, hidden rows/columns, defined names, and hidden worksheets. After this we applied a range of modifications: inserted/changed/deleted values and formulas, inserted rows/columns, a new worksheet, changes in the display and formatting, and more. The resulting spreadsheet is called S_1'. A screenshot of both files is shown in Figure 2.2. Spreadsheet S_1 contains 3 worksheets, 96 allocated cells, whereof 78 value cells and 12 formula cells.

Furthermore, we have used two versions S_2 and S_2' of an existing large spreadsheet, with a file size of 10.7 MB. Spreadsheet S_2 contains 202 worksheets, and a total number of 1,108,880 allocated cells, whereof 60,474 values cells and 890,111 formula cells. This set of files is used to test the performance for large, complex files.


B12 =SUM(B1:B10)

	A	B	C	D	E	F	G	H
1	a	1	10		test1	test6		
2	b	2	20		test2	test7		
3	c	3	30		test3	test8		
4	d	4	40		test4	test9		
5	e	5	50		test5	test10		
6	f	6	60					
7	g	7	70					
8	h	8	80					
9	i	9	90					
10	j	10	100					
11								
12		55	550					

B12 =AVERAGE(B1:B10)

	A	B	C	D	E	F	G	H
1	AAA	1,00	10					
2	BBB	€ 2,00	20					
3	CCC	3,00E+00	30					
4	**d**	4	40					
5	e	5	50					
6	f	6	60				test1	test6
7	g	7	70				test2	test7
8	h	8	80				test3	test8
9	i	9	90				test4	test9
10	j	10	100				test5	test10
11								
12		5,5	450					

Figure 2.2: Screenshot of the first worksheet in sample spreadsheets S_1 and S_1'

In Table 2.1 a complete list of the tools, their tested version, and the website link is given. There are three types of tools. An add-in runs in Excel and is available when working with the spreadsheet. A standalone tool runs as a separate application and is not dependent on Excel itself. The third type is an online tool, running in the web browser.

Table 2.1: List of examined existing spreadsheet compare tools

	Tool	Type	Version	URL
T1	Synkronizer	Add-in	11.2.809.0	`https://www.synkronizer.com/`
T2	Ablebits Compare Sheets	Add-in	2018.4	`https://www.ablebits.com/`
T3	Microsoft Spreadsheet Compare	Standalone	16.0	`https://support.office.com/`
T4	xlCompare	Standalone	6.2.3	`https://xlcompare.com/`
T5	DiffEngineX	Standalone	3.12	`https://www.florencesoft.com/`
T6	4TOPS Compare Spreadsheets	Standalone	3.2.0.1	`http://www.4tops.com/`
T7	Excel Compare	Standalone	3.8	`http://www.formulasoft.com/`
T8	xltrail	Online	2.5.0	`https://www.xltrail.com/`
T9	CloudyExcel	Online	N/A	`http://cloudyexcel.com/`
T10	XL Comparator	Online	N/A	`https://www.xlcomparator.net/`

2.2.2 Results

The ten tools offer a different user experience, mostly depending on the type of tool. Table 2.2 summarizes the approach of all the compare tools. The scope of a comparison shows whether the whole workbook, or only one individual worksheet is considered. Surprisingly, one tool (T10) cannot even compare a whole workbook or worksheet, but only one column. Furthermore, the steps a user should take to start a comparison and the comparison output of each tool are summarized in the table.

All tools have varying capabilities in comparing spreadsheets. Table 2.3 lists a comparison of the compare tools, showing how well they perform on different criteria. The tools either have support (✓), limited support (◇) or no support (–) for a feature. The performance on large, complex spreadsheets is reported on a qualitative scale. The symbol ○ ○ ○ indicates that the tool cannot handle large, complex files; the symbol ● ● ● indicates that the tool can handle large, complex files very well.

Large and Complex Files

The two add-in tools, T1 and T2, require you to open both versions of the spreadsheets in Excel. This has a clear disadvantage for large files, because Excel can become slow when working with complex files. The large sample files S_2 and $S_2^{'}$ are so complex that Excel is constantly recalculating, becoming a little unresponsive. This blocked the comparison process for tool T1. We had to disable the *auto calculate* function in order to complete the comparison. Tool T2 is saving the files and processing them intermediately, resulting in a bad comparison performance. Furthermore, it does not compare complete workbooks and is therefore not able to handle complex files.

The stand-alone tools also have limited support for large and complex files. Tool T3 is slow in loading and comparing such files, it took over 10 minutes to finish the comparison of S_2 and $S_2^{'}$. In the meantime there is no clear progress indication, resulting in an application that appears to

Table 2.2: Approach and output of the existing tools

	Scope	Approach	Output
T1	Workbook	Open two files in Excel, select worksheets, start comparison	Side-by-side view, summary, list of differences, merge tool, Excel report
T2	Worksheet	Open two files in Excel, go to worksheet, start comparison	Side-by-side view, merge tool
T3	Workbook	Start tool, load two files, start comparison	Side-by-side view, list of differences, graph with differences, Excel report
T4	Workbook	Start tool, load two files, start comparison	Side-by-side view, merge tool, difference explorer, Excel report
T5	Workbook	Start tool, load two files, select worksheets, start comparison	Excel report, Excel sheets with marked differences
T6	Workbook	Start tool, load two files, start comparison	HTML report
T7	Workbook	Start tool, load two files, start comparison	Excel report
T8	Workbook	Create project, upload initial file, add second file to version control, select versions to compare	File version history, list of worksheets and VBA modules, visual worksheet diff, VBA diff, summary
T9	Worksheet	Upload input files, select worksheets, compare	Side-by-side view
T10	Column	Upload input files, select one column in worksheet, compare	Summary (same or different rows)

be crashed. After comparison, the user interface is unresponsive, because it cannot handle the complexity and large amount of data in the worksheets. Tool T4 simply crashes when loading large files. Tool T5 also gets stuck, with no indication that it is doing anything. Even though this is a stand-alone tool, Excel is still used in the background during the comparison, which is not so efficient. Tool T6 already takes a lot of time for comparing small files, it runs endlessly on large files.

The online version management tool T8 analyzes the uploaded files in a background process on the server. This tool does not become unstable on large, complex files, however, it still takes forever for such files to be processed. This is confusing to end-users, as there is no indication when the server-side processing eventually will complete. In the end, the tool was not able to finish the comparison S_2 and $S_2^{'}$, the files were stuck on 'Processing'.

Other tools have a simple comparison strategy. Tool T7 only reports the rows that have been inserted or deleted. It is therefore able to finish the comparison of large files, but the outcome is not useful to determine what exactly has been changed. Tool T9 is an online tool and gives a timeout when uploading large files, probably due to the file size. Tool T10 has a predefined limit of 5 MB per uploaded file, and can only compare one column in one worksheet. These tools are clearly designed for simple scenarios. The comparison of large, complex files requires a compare tool with a more advanced strategy.

Table 2.3: Feature comparison of the existing tools

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Cell values	✓	✓	✓	✓	✓	✓	◇	✓	◇	◇
Cell formulas	✓	✓	✓	✓	✓	–	–	✓	–	–
Cell movements	–	–	–	–	–	–	–	–	–	–
Row/column insert	✓	–	✓	–	–	–	–	✓	–	–
Row/column hide	–	–	–	–	–	–	–	–	–	–
Worksheet insert	–	–	✓	–	–	–	–	✓	–	–
Worksheet hide	–	–	✓	–	–	–	–	–	–	–
Number formatting	✓	✓	✓	✓	–	–	–	–	–	–
Text formatting	✓	✓	✓	✓	–	–	–	–	–	–
Defined names	✓	–	✓	–	✓	–	–	–	–	–
Charts	–	–	–	–	–	–	–	–	–	–
Cell comments	✓	–	–	–	✓	–	–	–	–	–
VBA code	–	–	✓	✓	◇	–	–	✓	–	–
Large, complex files	●●○	○○○	●○○	○○○	○○○	○○○	●○○	●○○	○○○	○○○

Features

In Table 2.3 we see that all tools can compare the cell values. This is the most basic concept in a spreadsheet; a value is for example a number, date, text or currency. Even for this simple task, the results of tool T9 are inaccurate. It reports that almost all cells in sample files S_1 and S_1' are changed, because the cells are incorrectly aligned. If this tool would have done a direct cell comparison (e.g. compare cell A1 with A1, cell B1 with B1, etc.) the results would have been better. Tools T7 and T10 provide limited functionality in comparing cell values. The output of T7 is only an indication which rows have been inserted and deleted, not the individual cells. And tool T10 only compares one column in two versions and reports the rows that are the same or different based on this column. Most other tools are able to compare the values correctly, and show which data have been added, changed or deleted in the cells.

Most tools (except T6, T7, T9 and T10) consider the formulas in cells. Some of them make a distinction in the actual formula and the calculated value. If a formula depends on data of other cells, the formula can remain the same, but the outcome of the formula can be different. It can be useful to know which calculated values are changed. But on the other hand, this can lead to a large number of differences. When only one cell value is changed and many cells use this value in their formulas, many differences will be reported for a single user action. It depends on the scenario if this is expected or not. In general, most tools show the inserted, changed and deleted formulas correctly.

None of the tools can report cell movements. Consider the block of 5×2 cells that is moved from E1:F5 to G6:H10, as shown in Figure 2.2. This is the result of a cut-and-paste action. All tools doing the cell comparison correctly report this as 10 value deletions and 10 value inserts. But for large blocks of moved information this can be confusing and result in a large set of differences. The comparison result would be more insightful if cell movements can be detected.

Three tools (T1, T2 and T8) report row and column insertions and deletions. The detection of row and column changes is important, because otherwise cells get misaligned and many false positive differences are reported. Consider for example the scenario that a new column B is inserted. All cells in the original columns B, C, D, etc. are shifted one place to the right. A adequate cell comparison should compare cell B1 with C1, cell B2 with C2, and so on. In order to make such an alignment, it is required to know which rows are inserted and deleted, and which columns are inserted and deleted. We see in the tools that do not detect row and column changes, many cells are misaligned, resulting in many cells reported to be changed. This makes the comparison result in most cases completely impossible to read, because a user cannot see which are the relevant and which the irrelevant changes.

None of the tools detect hidden rows and columns. In principle the cells are still available, but they are not shown in the spreadsheet. If the user does not explicitly look at the row or column number, a user will have the illusion that data is deleted when rows or columns are hidden, It will be useful to report to the user which rows or columns are made hidden or visible, when this is changed.

Only two tools (T3 and T8) report worksheet changes. In all tools a clear overview of worksheet differences is missing, which is relevant in many situations. Especially when the number of worksheets is large, it is difficult to figure out what has been changed on worksheet level. Users should get to know which worksheets are inserted and/or deleted. Furthermore, there are more changes related to worksheets. Worksheets are often made hidden or even made *very hidden*, which again will give the illusion that content has been deleted. Worksheets can also be renamed, moved, or made protected. An explicit overview of the worksheets in both versions, with their properties, will give provide immediate overview on the worksheet changes.

Some advanced tools can optionally report display changes like text formatting and number formatting (T1, T2, T3 and T4). This can be useful, but is not required in most situations. Some tools can optionally report defined name changes (T1, T3 and T5). This is essential, because calculations or references can be in the defined names of a spreadsheet. The calculation model is not only in the formulas, it can also be in the defined names. No tool reports changes in objects, like charts or images. Some of the tools optionally report changes in cell comments (T1 and T5). There are tools that can report changes in the VBA code (T3, T4 and T8). In T5 this is also possible, but only if a specific macro setting is enabled in Excel. VBA code is often used in more advanced spreadsheets, and it can therefore be essential to know the differences in the macros used.

2.2.3 Conclusion

What did we learn from the survey of existing tools? First of all, there is no tool that covers all the aspects of spreadsheets. In other words, no existing tool can make a *complete* diff between two spreadsheet files. The list of features considered in Table 2.3 is not exhaustive, but it obviously shows that the current tools do not aim to be a full diff tool. This is confirmed by the output of the existing tools. There is no tool that generates a complete *edit script* of two spreadsheets, containing all modifications between the files. All the output are lists, overviews, Excel or HTML reports, or interfaces that are designed to give the user more or less insight in the changes. They are not built to generate a complete diff, and in that sense different from file comparison tools for regular files.

Furthermore, we conclude that most of the tools are designed with a specific use case in mind. Some simple tools (T6, T9 and T10) can only report changes in one column or one worksheet and are created essentially to give a quick overview of changes in a simple table or data sheet. There is a tool (T7) that is built to quickly identify individual rows that have been changed.

Other tools (T2 and T4) are focused on merging changes between two versions. They provide the required user interface to perform these tasks, such as a side-by-side view and navigation between the differences. Another tool (T8) is designed to be a version management tool and even integrates with Git. However, none of the tools is built with the use case to be an all-purpose compare tool, meaning that it can be used on all spreadsheets in general. There are clearly different use cases in which a compare tool is relevant, but the most general use case (give a user-oriented, summarized overview of all changes in two spreadsheets) is currently not supported by the existing tools.

We conclude that based on the currently available tools, there is enough motivation to develop a new solution to address the shortcomings of the existing tools.

- First of all, such a new spreadsheet compare tool should be able to compare as many types of spreadsheet files as possible. Spreadsheets are widely used in many domains, and there are different variations: small and large, simple and complex, and so on. It is clear that all tools have problems comparing spreadsheets with many values or many formulas.
- Secondly, the new tool should be able to summarize the differences and provide insight in the changes on different levels. Most of the current tools lack the ability to give clear overviews, for example: which of all worksheets in a workbook have been changed?
- Thirdly, it is valuable if a compare tool can give a complete diff between two spreadsheet files. Such a compare tool can give guarantees, for example that no formulas have been changed between two spreadsheets versions. Also, a complete diff makes integration with version control systems possible.

In the end, more insight in the context of spreadsheet versioning will help to build a compare tool that suits the user needs. The current tools clearly do not support all scenarios for which end-users need a spreadsheet comparison tool. It is helpful to involve users in the development process, because every user will start using a compare tool with a certain goal. In the next chapter we will therefore present an extensive analysis of the spreadsheet versioning domain and its challenges.

3 Spreadsheet Versioning Challenges

Comparing spreadsheets is not a trivial task. Unlike regular file comparison, there are fundamental issues that make the spreadsheet comparison problem difficult. The goal of this chapter is to gain understanding in these issues. We will obtain insight in the spreadsheet versioning problem, and define concrete requirements for this problem. The requirements are building blocks for developing a suitable spreadsheet comparison approach. Two types of requirements are identified: (1) user needs and (2) technical challenges for which a solution needs to be found. By identifying the challenges, we can split the difficult problem into smaller subproblems. With clear subproblems it is easier to invent small and applicable solutions, and the whole spreadsheet comparison problem becomes more manageable.

The method we applied is two-fold. First, we interviewed six Excel end-users, interested in a spreadsheet comparison tool, from different business and education contexts. The people interviewed are listed in Table 3.1. The interviews had two goals. First, they serve to provide more insight about the usage of spreadsheets in practice, e.g. when new versions are created, how they are saved and shared, and so on. Furthermore, the interviews are used to investigate problems of users regarding spreadsheet versioning. The outcome of the interviews is described in Section 3.1.

Secondly, we started early with the development of a prototype. During six months of research and implementation of the actual spreadsheet comparison tool, we experienced several fundamental issues. More insight in these issues and the underlying characteristics of spreadsheet files are described in Section 3.2.

We will finalize the chapter with a list of requirements in Section 3.3. The problems from Section 3.1 are formulated in terms of user needs, reflecting the requirements for a spreadsheet compare tool from a user perspective. The issues from Section 3.2 are formulated in terms of technical challenges, reflecting the requirements for a spreadsheet compare tool from an implementation perspective. The outcome of this problem analysis, a list of requirements, provides a relevant summary of the fundamental challenges that exist for the comparison of spreadsheets.

Table 3.1: List of people interviewed for the problem analysis

Person	Position	Context
A	Employee Financial Control	University
B	Finance Manager	University
C	Purchase & Logistics Officer	Small company
D	Software Engineer	Large investment company
E	Performance Manager	Large insurance company
F	IT Manager	Large insurance company

3.1 Spreadsheet Context

We asked the following questions about version organization:

- What role do spreadsheets play in your organization?
- How and when are new versions created?

We asked the following questions about version problems:

- What problems do you experience regarding different spreadsheet versions?
- How can a compare tool help to solve the problems?

3.1.1 Version Organization

Person A uses spreadsheets mainly as financial calculation models. Most spreadsheets contain worksheets with input data, worksheets with calculations, and worksheets with output. The spreadsheets are shared among different users using SharePoint. Every week a new version is created with new data, the previous file is then overwritten. Old versions of the calculation models are not maintained. The spreadsheets are shared with 40-50 end-users.

Person B also works with financial calculation models, shared using SharePoint, mostly with a version number in the filename. Here one employee, the administrator, is responsible for the spreadsheets. The structure is that there is one spreadsheet template, and many filled in versions of the template by other users. Sometimes a new version of the spreadsheet template is created. Filled in spreadsheet versions are saved in a subdirectory, which represents a group (e.g. one month). The number of files can become very large: there is a case where 700 versions are created in a subdirectory. The spreadsheets are shared with 30-40 end-users.

Person C uses spreadsheets for business cases and forecasts. Since the company is small, most of the spreadsheets are small and relatively simple. All sheets have one owner. New versions of spreadsheets are created by the end-user, and spreadsheets are shared to others via email. It is often described in the email what has been changed in the newer version.

Person D mentions that spreadsheets play an important role in their financial company. Their spreadsheets are classified in four levels of confidence; 160 spreadsheets are in the highest category (business critical). He estimates that 450 employees are using spreadsheets in their company. Many of them are managing spreadsheets versions: spreadsheets are often changed, copied and shared. New versions appear when modifications to the spreadsheets are required or when a data source is updated. The old versions are maintained, versions are organized on file level (e.g. a new folder or a new file name is created). There is no version management system used. Versions are stored centrally on a network drive, and thus accessible by other employees. The type of spreadsheets varies greatly, but most spreadsheets are calculation models (load incoming data, perform calculations and produce output).

Person E describes that large and many spreadsheets are used in their financial company, with many different applications. He himself is using many reports, which frequently contain input, selection/calculations and output. There are two situations where new spreadsheet versions arise: when a spreadsheet is remodeled or when a spreadsheet is delivered with new data. The new versions are indicated with a new file name, mostly with a different date. The spreadsheets are stored on a network drive, where each employee has his or her own access. Some spreadsheets are shared using SharePoint, but there is no real collaboration on spreadsheets. One person is responsible for a spreadsheet and has knowledge about the content.

Person F works for the same company as person E and confirms the fact that spreadsheets are used on a large scale, also in crucial processes like business decision making. The type of spreadsheets he is responsible for are budget and forecast spreadsheets, each having around 40-50 versions per year. New versions are created on new insights. Some spreadsheets are connected to a database for data import, and contain a complex calculation of the impact of costs. Spreadsheets can be gigantic: a spreadsheet with 85 worksheets and a file size of around 82 MB is mentioned. The spreadsheets are stored on SharePoint: the built-in version management is sometimes used to revert back to a previous version when something went wrong in the calculations. There is collaboration on the spreadsheets: 20 employees are working on the same spreadsheet with budgets, but never at the same time. This interviewee is generally satisfied with version organization of spreadsheets.

3.1.2 Version Problems

Person A describes the problem that end-users sometimes continue to work with an old version instead of the newest one. Because versions of the same spreadsheet are overwritten, data in between the versions can be lost. The way to solve this is to revert back to a previous version, and then manually obtain the missing data and add it to the newest spreadsheet. A compare tool however can help to identify the intermediate changes and possibly merge the different versions. Another problem is that once in a while formulas are changed unwantedly. Despite all spreadsheets are protected, meaning that no formula cells can be changed, some 'experienced' users remove the protection and do change the formulas at their own discretion. In such cases, it is often unclear what has been changed. The main reason for using a compare tool is to have an overview of all things that has been changed. Furthermore, the comparison results can be used to check if all data has been filled in, and to validate if the calculation model is unchanged. A comparison of two spreadsheet files is sufficient. It depends on the type of spreadsheet to see which sort of differences are relevant.

The main issue of person B is to manage a large number of spreadsheet versions. Because new versions of the main spreadsheet template can arise, one problem is that the resulting spreadsheet files can originate from different versions of the source template. A compare tool that can identify the version of the source file in batch will be helpful. Another problem is unwanted changes in spreadsheets, because from time to time formulas are incorrectly or accidentally changed. The third problem is that there are sometimes multiple owners of a spreadsheet.

A compare tool that can identify changes between different versions is helpful for both problems. In this context (financial calculation models), it is important to be able to see only formula changes. Most errors are created in the calculation model, while the data changes are uninteresting. Because there is one spreadsheet administrator, the compare tool may be optimized for one or a small set of users.

Person C concludes that there are issues with spreadsheet versions, even in her small company with one user per spreadsheet. Regularly, many spreadsheet versions are saved, causing a bit of confusion. The indication in the filename is often not sufficient to determine the contents of the spreadsheet and the last version. The end-user would be supported by a quick overview of the changes using a compare tool. Furthermore, the communication via email about changes in spreadsheets is often incomplete or incorrect. For a newer version, it would be helpful if a summary of the changes with respect to a previous version can be created. This would save the end-user time in describing the changes, and always guarantees that a complete overview of the changes is listed. Ideally, for both scenarios, the compare tool should integrate in the current workflow of the end-users, for example by integration in the file system, email program or spreadsheet software.

Person D explains that after adjusting a spreadsheet, it is often unclear what precisely has been changed. Often an Excel user forgets what he or she did exactly after modifying a spreadsheet. There is fear that accidentally too much has been changed, or that accidentally something has been overlooked. A complete overview of all changes between two versions will give the user insight in their adjustments. If such a compare tool is available, it will probably be used on a daily basis to check if all modifications are intended before saving the new version. Additionally, it is advantageous that an export of all changes between two versions can be used for auditing purposes. In this company, all modifications to business critical spreadsheets need to be justified. A comparison report is a proof for what exactly has been changed, and will save a lot of time in the auditing process. For a large number of changes, it should be possible to start with the main differences and to zoom in on the details. It would also be helpful if the compare tool can answer questions like 'are all formulas unchanged?' or 'are there only new data cells inserted?'. Another problem is that after a longer period, it is not possible to see how a spreadsheet functioned a number of versions ago. It would be helpful if turning points can be detected in the evolution of a spreadsheet. Instead of manually inspecting versions, it is informative to know that many formulas are changed in e.g. v4 and v9, probably because another calculation method has been used.

Person E describes that version management problems are not always caused by incorrect spreadsheets. Sometimes issues arise in the workflow. For example, a colleague was working on an old spreadsheet version locally. He uploaded his modified version on the network drive, overwriting the most recent version. It was not possible to go back to the previous version, causing loss of data. A compare tool would not solve this issue, but possibly an analysis of the two versions before uploading can detect this situation and raise some warning. Another problem that often occurs is that formula references become invalid (#REF!) after adjusting the spreadsheet. This is caused by a human error, because cells are incorrectly moved or deleted or data is missing. After complex modifications, it is often not possible to find which cell modifications are causing the reference errors. In such case, the person can only revert to an earlier backup and has to do his work again. A compare tool would be able to easily detect the cell changes and can help to identify the source of the error. This will save a lot of time. The main advantage of a compare tool for this user is that it can provide insight in the changes, so that the end-user can check if he has changed what he wanted to change. Besides the overview of changes, it will be useful if the compare tool can detect errors (e.g. caused by incorrect cell formatting, or formulas changed from correct to incorrect). An idea to accomplish this is to run a risk analysis on the comparison result.

The main problem of person F is that it is difficult to find what went wrong when a spreadsheet contains errors. Because of the complexity of spreadsheets, it is noticed too late that new errors are introduced during the development of the spreadsheet. Often a lot has already been adjusted, making it impossible to find the cause and to fix the errors. A tool that can compare the current version with a previous back-up can help to quickly identify the source of the errors. In the context of multiple spreadsheet users, it will be helpful if a compare tool can validate if a user only changed the part of the spreadsheet he is responsible for. When a collaborator uploads a new version, it is not possible to verify his contribution to the spreadsheet. A comparison report or a version history in a worksheet in the spreadsheet itself will help to prevent version problems in the future.

3.2 Spreadsheet Characteristics

Independent how users operate with spreadsheets, there are conceptual problems regarding spreadsheet comparison. In this section we will mention them in terms of typical characteristics specific for spreadsheet files. Most of the observations emerge from practice: while investigating sample spreadsheets, while designing a suitable comparison algorithm, and during the actual development of the prototype. For each characteristic, we will explain why this is a complicated factor of the spreadsheet comparison problem.

3.2.1 Binary File Format

Spreadsheets in general are stored as binary files. Unlike text files, binary files are not human-readable. In Excel 97-2003, Microsoft used a proprietary binary file format (.xls) as its primary file format. Since Excel 2007, the newer Office Open XML format (.xlsx) is used. In this format, the complete spreadsheet is saved as a series of XML files. In order to save disk space, the collection of XML files is zipped into a single file. Macro-enabled spreadsheets (.xlsm) have the exact same file format, with the addition that it contains an additional binary container with the VBA code. There are alternative spreadsheet file formats (for instance .xlsb, .xlam, .ods). All formats have in common that the spreadsheet is a binary file on disk.

A direct comparison of the binary spreadsheet files is not meaningful. Every time a spreadsheet file is saved, the content of the binary file will be completely different. That is because the underlying XML files are compressed. After compression, the chunks of the new binary file do not correspond to chunks of the previous binary file. The idea of a regular text comparison approach is to find the parts of the files that are the same, and create a diff for the parts that are changed. Because all chunks of a new spreadsheet file will be different, a similar comparison approach on the binary files is meaningless.

However, it is easy to see the raw data in a modern spreadsheet file. One can change the .xlsx extension to .zip and extract the file. The result is a set of individual XML files, organized in a predefined structure with folders and file names. Also media items like images are stored inside the package. In Appendix A a tree of all XML files in the sample spreadsheet S_1 (introduced in Section 2.2) is shown. Most files in the package have a clear meaning, like `workbook.xml`, `sheet1.xml`, and so on. The raw data of the workbook and the raw data of the first worksheet are also listed in Appendix A.

Even though the content of those XML files looks insightful, it is not particularly helpful to compare the raw XML data. Just like the problem with compression, it is not guaranteed that a new version follows the exact same XML structure. Consider for example a spreadsheet that contains one cell with the world 'hello'. This cell value can be stored in the file `sheet1.xml`. Now consider a second version of this spreadsheet that contains 10 cells with the word 'hello'. As an optimization, Excel decides to store the string 'hello' in `sharedStrings.xml` and to put references to that string in `sheet1.xml`. The structure of the XML files is now completely changed, rendering a comparison on the raw XML data meaningless. Many more examples can be constructed where the XML structure changes dramatically. For example, after reordering the worksheets, `sheet1.xml` of the first spreadsheet no longer matches `sheet1.xml` of the second spreadsheet. This all indicates that more intelligence required to interpret changes in the raw XML data, and that we cannot rely on existing text comparison approaches for binary spreadsheet files.

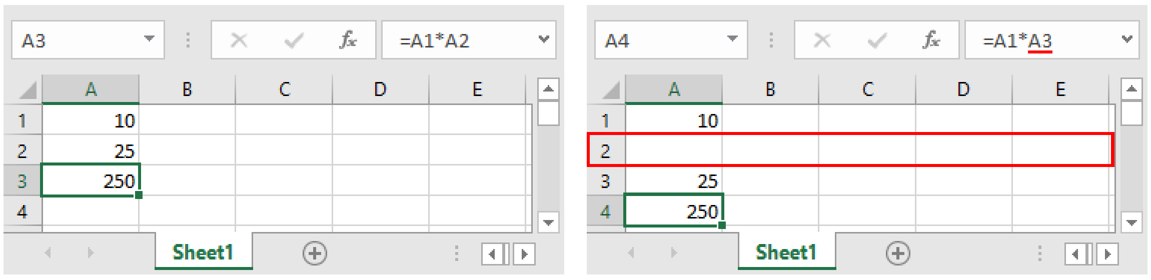
3.2.2 Change Propagation

One edit operation in a spreadsheet can result in potentially many changes. The underlying reason is that in fact a spreadsheet is a combination of data and calculations. Calculations are present in the form of formulas, and formulas often reference other cells with data. Excel allows easy manipulation of the spreadsheet, and some of the user actions cause substantial changes in the cell references.

There are five operations that cause side-effects: row inserts, row deletes, column inserts, column deletes and worksheet renames. In Figure 3.1a the effect of inserting an empty row is demonstrated. Cells A2 and A3 are shifted downwards, and because of that, the reference to A2 in the formula is updated to A3. Deleting a row will result in similar side-effects. Figure 3.1b shows the effect of inserting an empty column. The cells B1, C1 and D1 are shifted to the left, resulting in an update of the cell range in the shown formula. Equivalently, deleting a column will result in similar side-effects. Finally, in Figure 3.1c the effect of renaming a worksheet is displayed. Because the formula depends on a cell in worksheet 'Sheet1', the reference to that worksheet is updated to 'Blad1'. All these operations update the structure of the worksheet or the workbook, and therefore we will call them structure changes. The amount of additional changes caused by these side-effects can be tremendous, because formulas often rely on many other cells. One structure change can lead to hundreds or thousands of extra formula changes.

An end-user is only interested in the actual actions he or she performed in the spreadsheet. If we classify inserted/deleted rows, inserted/deleted columns and worksheet renames as actual edit operations, the end-user should be informed about such a single operation, and not the side-effects. Therefore, it is important to detect the structure changes beforehand. If it is known which structure changes have occurred, this can be taken into account in later stages of the comparison.

The problem of change propagation is specific for spreadsheets files, due to the combination



A3 =A1*A2

	A	B	C	D	E
1	10				
2	25				
3	250				
4					

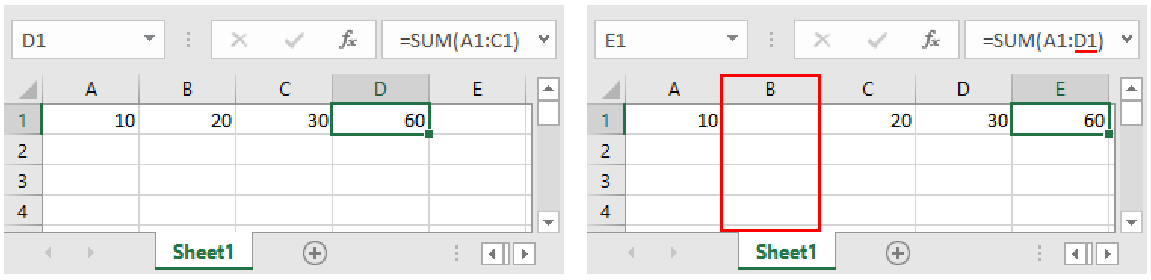
Sheet1

A4 =A1*A3

	A	B	C	D	E
1	10				
2					
3	25				
4	250				

Sheet1

(a) Effect of inserting empty row 2



D1 =SUM(A1:C1)

	A	B	C	D	E
1	10	20	30	60	
2					
3					
4					

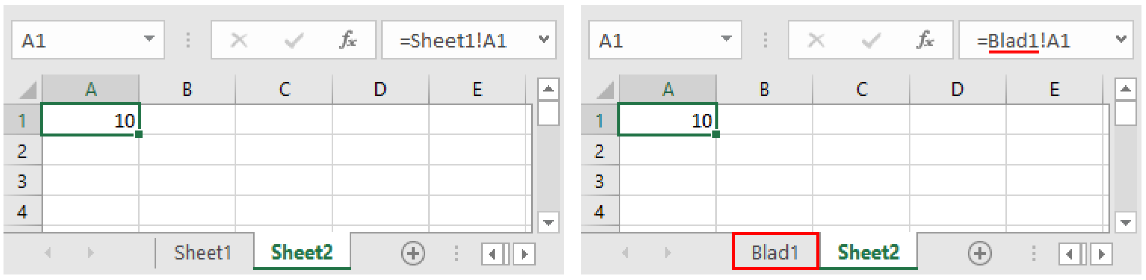
Sheet1

E1 =SUM(A1:D1)

	A	B	C	D	E
1	10		20	30	60
2					
3					
4					

Sheet1

(b) Effect of inserting empty column B



A1 =Sheet1!A1

	A	B	C	D	E
1	10				
2					
3					
4					

Sheet1 Sheet2

A1 =Blad1!A1

	A	B	C	D	E
1	10				
2					
3					
4					

Blad1 Sheet2

(c) Effect of renaming worksheet ‘Sheet1’ to ‘Blad1’

Figure 3.1: Illustration of the effect of change propagation in formulas

of data and calculations. The fact that spreadsheet data is two-dimensional (rows and columns) makes this issue even trickier. The root cause is that spreadsheet software allows easy manipulation of the data, resulting in shifted rows and columns. This is beneficial for the end-user, but difficult when developing a comparison approach, which then needs to include some sort of alignment algorithm. We have seen that existing tools often fail when structure changes occur, and that sometimes many false positive changes are reported. For example, one simple row insertion can result in 10,000 cell changes. For spreadsheet comparison, it is important to find the real changes performed by the user, not the propagated changes.

3.2.3 Many Edit Operations

Regular text files have clear and simple edit operations. An edit operation is an action to convert one file to another. Most text comparison algorithms consider two operations: line inserts and line deletes. It is important to have clearly defined edit operations, not only for showing changes between files, but also for switching and merging between files. Most version management systems execute the set of edit operations to go from one version to another. This requires that the set of edit operations is complete.

For spreadsheets, there is a whole variety of edit operations. In the initial phase of this research, we did a manual inventarisation of all actions that are possible within Excel. Every action is a modification of the underlying spreadsheet, so every action can be seen as an edit operation. We found 112 different edit operations, listed in a table in Appendix B. Probably the list is not totally complete, and it is not always clear how to represent certain actions. Nevertheless, this inventarisation is valuable, because it reflects the order of magnitude of the spreadsheet comparison problem. If the goal is to develop a complete spreadsheet comparison approach, this approach should consider cells, rows, columns, worksheets, charts, conditional formatting, named ranges, pivot tables, styles, number formatting, and many more. It is challenging, if not impossible, to invent a general algorithm that can compare all spreadsheet aspects at once. With so many different edit operations, it seems that a tailor-made comparison for all the different spreadsheet aspects is required.

3.2.4 Different Levels of Granularity

Another characteristic of spreadsheets is that the content of spreadsheets exists at different levels of detail. Generally speaking, there are three different levels in a spreadsheet: (1) workbook, (2) worksheet, and (3) cell. Every spreadsheet action will take place at one of these levels. For example: an external data connection is defined at workbook level; a chart is added at worksheet level; and a value or formula is entered at cell level. The inventarisation of all spreadsheet edit operations in Appendix B includes a column at what scope (level of detail) every action occurs. The levels of granularity can be made more specific, for instance: rows, columns, blocks of cells, and cell details.

It is difficult to represent and report changes at different levels of detail. A regular text comparison approach outputs changes with a clear indication of the location: the corresponding line number. Because there is one level of granularity, the changes can easily be reported line-by-line. A spreadsheet comparison approach should operate on different levels of detail, and therefore generate changes with different types of locations. The comparison result will contain workbook changes, worksheet changes and cell changes. If the level of detail is made more specific, there will also be row changes, column changes, blocks of cell changes, and cell detail changes. It is not trivial to find a representation of the changes on different levels of granularity, and to build an interface where such diverse changes are intuitively reported.

3.2.5 Large and Complex Files

There are two aspects that are not uncommon for spreadsheets: large files and complex files. Besides small and normal spreadsheets, we have seen a lot of spreadsheets with file sizes in the range of 1-80 MB. The cause of such substantial file sizes is the amount of data that is contained in the spreadsheets. For example, the largest spreadsheet in our sample set contains more than 10 million allocated cells. All the individual cell values are stored in the spreadsheet. Theoretically, Excel allows very huge spreadsheets. The limits for a worksheet are 1,048,576 rows by 16,384 columns, and the number of worksheets in a spreadsheet is limited by the available memory.

We have also seen many spreadsheets that are complex in terms of calculations. The cause of the complexity is the amount of formulas and the difficulty of formulas contained in the spreadsheet. Formulas are difficult when they have many different operations, many different cells references, many conditional operations, or a long calculation chain [24]. Furthermore, it is important to notice that in the raw data, a formula cell contains both the actual formula (in text) and the outcome of that formula (the calculated value).

Both aspects are a complicating factor for comparing spreadsheets. Large files are difficult because all raw data needs to be loaded into memory, and then all that data needs to be processed to some kind of object model. The object model contains the whole content of the spreadsheet (e.g. all worksheets, all cells in a worksheet, all properties of a cell, and so on) and is potentially very large. It can take a lot of time to traverse through many cells, and more advanced algorithms will not perform well on large amounts of rows and columns.

Complex files are difficult to compare because of the propagation of changes. We have seen in Section 3.2.2 that structure changes can lead to formula changes, caused by worksheet renames or row/column insertions and deletions. When such structure changes occur in complex files, many formulas will be different in the two versions and many formula comparisons need to be performed. In addition, changes in all calculated values of formula cells need to be ignored. It is only relevant to report the actual data and formula changes, not the propagation of such changes.

Usually, it is impossible to have an overview of the content of large and complex files, because such files contain too many worksheets, cells and formulas. Precisely for these files a compare tool is useful, because a human is unable to browse through all the content themselves. Therefore, it is important to optimize the comparison approach for large and complex files.

3.3 List of Requirements

We conclude our problem analysis with a summary of the problems regarding spreadsheet versioning, both from a user perspective and a technical perspective. Each requirement reflects a specific user need or a technical challenge that should be solved.

The following requirements summarize the user needs:

Req 1. Overview
The main problem in spreadsheet versioning is that it is unclear what has changed. There are potentially many changes between two spreadsheet versions. Users want a summary and be able to zoom in, to view the changes in increasing level of detail.

Req 2. Validation
Users always start the compare tool with a certain goal in mind. They want to use the comparison result to validate a certain hypothesis. For instance, are the formulas unchanged, or what is the cause of the error in Sheet1? It must be possible to easily answer such questions regarding different spreadsheet versions.

Req 3. Completeness
Spreadsheets can differ in many ways, because Excel contains many features. Sometimes changes occur in the deep details of a spreadsheet. A spreadsheet comparison must be complete, since users want to be sure that this is the full set of changes. A complete diff is also required when the comparison should be used for merging spreadsheet versions.

Req 4. Error Resolving
When a new version of a spreadsheet contains errors, users want to fix the errors using the compare tool. The erroneous version is compared with a previous version without errors. The compare tool should be able to handle spreadsheets with errors.

Req 5. Visualization
It if often not sufficient to give an extensive list of changes. Users want to view the changes as close as possible to their original spreadsheet. A visualization or a display of the changes in the context of the actual spreadsheet will be helpful.

Req 6. Evolution
Besides the one-by-one spreadsheet comparison, spreadsheet evolution is an important extension. For instance, users want to view changes over time, or the introduced errors over time.

The following requirements summarize the technical challenges:

Req 7. Change Propagation
One single user action can lead to potentially many changes. Structure changes, for instance inserting an empty row or renaming a worksheet, are propagated. Only the actual user actions should be reported, not the propagated changes.

Req 8. Performance

Spreadsheets can be very large and very complex. Such files are important to compare. The comparison must be smart and efficient, in order to get results in acceptable time.

Req 9. 2D Alignment

The content of a spreadsheet is 2-dimensional, worksheets contain rows and columns. The cells in both versions needs to be aligned, before individual cells can be compared.

Req 10. Grouping of Data

The fundamental data items in spreadsheets are usually combinations of cells. The comparison should report changes in terms of larger blocks, not only the changes for single cells. For instance, one database import with many inserted rows should be reported as one insert of a large cell range.

Req 11. Detect Movements

The data in a spreadsheet can easily move from position, for instance by a copy/paste action. These changes should be reported as movements, instead of inserts and deletions.

4

Finding Spreadsheet Differences

In this chapter, we present our solutions to the spreadsheet comparison problem. Each section describes a new contribution or approach that we invented during the development of our prototype. The proposed solutions include argumentation, design choices and technical details. This chapter contains the main research work of this thesis.

In Sections 4.1 and 4.2 we introduce and motivate a new spreadsheet comparison approach. In Sections 4.3 and 4.4 we present two core ideas that are essential to solve the spreadsheet comparison problem. In Sections 4.5 to 4.8 we describe how we actually detect changes in two spreadsheets. In Section 4.9 we propose a method to summarize the outcome of the spreadsheet comparison. Finally, we conclude in Section 4.10 which requirements from the previous chapter are realized by the solutions proposed in this chapter.

4.1 Change Categorization

We propose a comparison approach where the final comparison result contains a list of categorized changes. Initially, we propose three categories: Data, Model and Structure. The idea behind this is based on the fact that many spreadsheets can be seen as calculation models. A calculation model in general consists of input (data), formulas (model) and the position of the data/model items (structure) in the spreadsheet. A categorized approach has the advantage that the comparison result is easy to interpret, and that it is easy to zoom in on changes that are relevant for the goal for which a compare tool is used.

It is reasonable to divide the changes in these three categories, because it corresponds to scenarios how spreadsheets are changed. A common use case for modifying a spreadsheet is the situation where data of a calculation is updated. For example, someone wants to compute the profit of a new month using a spreadsheet. In that case, the actual calculation model (the formulas) remain unchanged, while new outcome is produced based on updated data. With the proposed categorized approach, it is easy to validate that this scenario has happened: one would expect some data changes and no model changes. When the set of detected model changes is complete, meaning that all formulas are checked, there is guarantee that no formulas have been changed based on this comparison result. In that sense, the comparison result does not only make the actual changes clear, it additionally gives guarantees on what has happened.

This approach fits the user need for validation. In multiple interviews with end-users it came forward that spreadsheets errors are frequently introduced by users who edited formulas accidentally or incorrectly. Often an Excel user is only supposed to update the data, and to leave the formulas untouched. With the categorization of data and model, it is easy to see if formulas were changed. If that is the case, one can directly inspect the details of the (possibly incorrect) updated formulas. The opposite scenario is also covered: it is possible to check if only formulas and no values have been updated. If this categorized approach is combined with a detection per worksheet, one can even validate if in one worksheet only formula changes occurred, and in another worksheet only data changes.

As mentioned in Section 3.2.3, in the first stage of this research, we conducted an inventarisation of all possible Excel actions. The complete list of all 112 actions is shown in Appendix B. We assigned a category to all listed actions, shown in the same table. Because spreadsheets allow for such a variety of edit operations, it is required to define which changes should be detected. We decided to focus on the most important aspects of a spreadsheet, which are the categories Data, Model, and Structure. Other details, like display or protection changes, are omitted in this first proposal of a comparison approach.

We define data changes, model changes and structure changes as follows.

- **Data changes** are changes in the actual values of a spreadsheet: value inserts, value changes and value deletes. Values can be of any type, like numbers, text, dates, booleans, and so on. Values are always in cells. When values are updated, the data of a spreadsheet is changed.
- **Model changes** are changes in the formulas of a spreadsheet: formula inserts, formula changes and formula deletes. Formulas are in cells, but can also occur in the defined names of a spreadsheet. Constants in formulas are also considered as model changes (e.g. the formula '=42'), because such values are defined in a formula and not in an actual cell value. When formulas are updated, the model of a spreadsheet is changed.
- **Structure changes** are changes that do not effect any data or model changes. For example, the insertion of an empty row will change the structure of a spreadsheet, but does not change any value or formula.

We have created a collection of 28 change types that will be detected in our comparison approach. The complete list is shown in Table 4.1, which includes the scope (level of detail), and the category of each change. We constructed the categorization with the purpose that the sets of data changes and model changes are complete. In other words, if no data changes are reported, there is a guarantee that no values are inserted, updated or deleted. And if no model changes are reported, there is a guarantee that no formulas are inserted, updated or deleted. Notice that we intend manual updates by the user here. For example, when a worksheet is renamed (structure change), all formulas in the spreadsheet that refer to cells in that worksheet are also updated accordingly. In our comparison method, only one structure change will be reported and no model changes, because no manual update to any formula has occurred. It is our objective that this approach, where we focus on the real user actions that are performed, is intuitive to end-users and provides a clean comparison result.

Table 4.1: Complete list of all detected changes

#	Change	Scope	Category
1	WorksheetInsert	Global	Structure
2	WorksheetDelete	Global	Structure
3	WorksheetHidden	Global	Structure
4	WorksheetVisible	Global	Structure
5	WorksheetProtected	Global	Structure
6	WorksheetUnprotected	Global	Structure
7	WorksheetOrderChange	Global	Structure
8	WorksheetRename	Global	Structure
9	DefinedNameInsert	Global	Model
10	DefinedNameChange	Global	Model
11	DefinedNameRename	Global	Model
12	DefinedNameDelete	Global	Model
13	RowInsert	Worksheet	Structure
14	RowDelete	Worksheet	Structure
15	ColumnInsert	Worksheet	Structure
16	ColumnDelete	Worksheet	Structure
17	RowHidden	Worksheet	Structure
18	RowVisible	Worksheet	Structure
19	ColumnHidden	Worksheet	Structure
20	ColumnVisible	Worksheet	Structure
21	FormulaInsert	Cell	Model
22	FormulaChange	Cell	Model
23	FormulaChangeToValue	Cell	Data, Model
24	FormulaDelete	Cell	Model
25	ValueInsert	Cell	Data
26	ValueChange	Cell	Data
27	ValueChangeToFormula	Cell	Data, Model
28	ValueDelete	Cell	Data

4.2 Comparison Pipeline

There is no natural way to start a simple comparison algorithm on two spreadsheets. Unlike regular file comparison, there is no trivial ‘start’ and ‘end’ of a spreadsheet. Initially it is unknown which parts of a spreadsheet belong to each other, information that is required to start the comparison. For instance, when worksheets are renamed, it is unclear which pairs of worksheets should be compared. Or, when rows or columns are inserted, it is unclear which pairs of cells in a worksheet should be compared. In such case it is incorrect to compare cell A1 with A1, cell B1 with B1, and so on.

We noticed that this problem can be solved when different steps in the comparison approach are distinguished. Therefore, we propose a comparison pipeline that contains three phases: (1) Structure Analysis, (2) Change Detection and (3) Change Aggregation. Each consecutive phase is using results from the previous phase. An illustration of the complete comparison pipeline is shown in Figure 4.1.

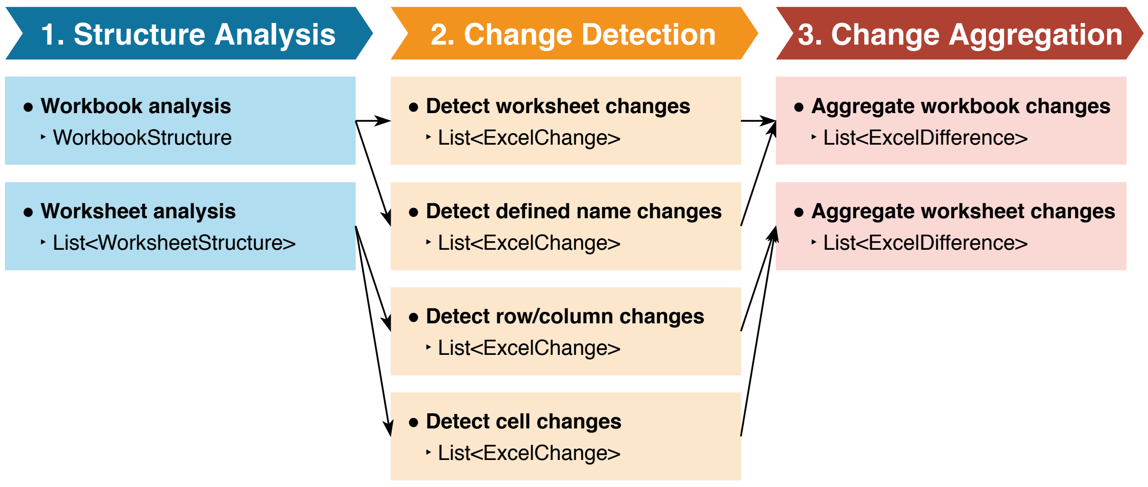


Figure 4.1: Overview of our proposed comparison approach

In the first phase, all prerequisite structure information required for the next phase is gathered. A workbook analysis and a worksheet analysis is performed. In the workbook analysis, all inserted, deleted and matching worksheets are identified. In the worksheet analysis, for every pair of matching worksheets the sheet name, the used cell range and the inserted and deleted rows/columns are identified. The outcomes of the workbook and worksheet analyses, a `WorkbookStructure` and multiple `WorksheetStructures`, contain all the necessary structure information to begin with the actual comparison.

In second phase, four types of changes are detected: worksheet changes, defined name changes, row/column changes and cell changes. The change detection outputs corresponding changes as defined in Table 4.1. Each `ExcelChange` contains information about the change, for example the cell position, as well as the old and new formula in a `FormulaChange`. The worksheet change detection returns changes 1–8, the defined name change detection returns changes 9–12, the row/column change detection returns changes 13–20, and the cell change detection returns changes 21–28. More details about the comparison of these aspects will be given in Sections 4.5 to 4.8.

In the final phase, the changes found in the previous phase are aggregated to differences. We define a **change** as a single, atomic modification in a spreadsheet. A **difference** is defined as a group of changes. Two types of changes are aggregated: global workbook changes and worksheet changes. More details about the aggregation of changes will be given in Section 4.9. The final outcome of our comparison pipeline is a collection of `ExcelDifferences`, describing all the modifications between the two spreadsheet files.

This comparison approach has several advantages. First of all, it provides more structured and better organized code. The pipeline approach creates phases that all have a well-defined goal and a well-defined outcome. The entire spreadsheet comparison problem is divided in three smaller tasks, namely (1) analyze, (2) detect and (3) aggregate. Each task has a number of similar sub-tasks (e.g. analyze workbook and analyze worksheet). The sub-tasks, the smaller units of the spreadsheet comparison, are easier to implement. Secondly, the sub-tasks are easier to test too. Every part of the comparison can be individually tested with small unit tests. Thirdly, the different phases of the comparison are easier to debug. Because every step of the pipeline has a clear outcome, it is easy to validate which steps are correct and which steps are incorrect. For example, if the cell detection is missing some change, it is helpful to validate if the structure analysis was correct. Then the error must be in the cell detection itself. Finally, we will see in Section 5.3 that this pipeline approach is straightforward to parallelize. That means that our proposed comparison approach is in addition ready to be optimized.

4.3 Cell Hashing

To optimize the performance, we apply hashing on cells to speed-up the spreadsheet comparison process. An efficient comparison algorithm should 'zoom in' as fast as possible on the parts of both spreadsheets that are changed, and exclude the parts that are unchanged. The bottleneck for large spreadsheets is the huge number of cells. It will help a lot if all cells, or even complete worksheets, that are the same in both versions can be skipped for further inspection. For instance, assume that some sort of 'fingerprint' is available for worksheets. Then the fingerprints can be compared, and with two equal fingerprints it is certain that the content of these worksheets is the same, so they can be skipped immediately. But when only one cell in a worksheet is updated, the complete fingerprint of the worksheet is changed, and the benefit is gone. Therefore, it would be better to have a more localized fingerprint. Because the structure of worksheets is two-dimensional, a fingerprint for every row and column would be an outstanding idea. Unfortunately, there is no such fingerprint available in the raw XML representation of the spreadsheet. Nevertheless, this idea did inspire us to create an intelligent pre-processing step, leading to a substantial performance gain, namely the hashing cells, rows and columns.

Before we start the comparison pipeline, the data of both spreadsheets should be loaded into memory. Instead of collecting all the cells (with all their details), it is enough to save a more compact representation of all cells. We do that by computing a hash for every cell. A hash function maps an input cell with arbitrary size (e.g. cell location, cell formula, cell formula, and/or additional properties) to a single output with a smaller, fixed size. We decided to use the xxHash algorithm[1], which aims at being 'an extremely fast non-cryptographic hash algorithm, working at speeds close to RAM limits'. For our application, no cryptographic guarantees are required for the hash function. We only need a deterministic, quick hash function that is able to summarize cell objects. Accordingly we selected the xxHash algorithm, which is also used for many other applications like databases, games, filters, file transfer, etc. The output of the hash function is a 32-bits unsigned integer. In other words, every cell can be summarized to one single number, which is unique based on the contents of the cell.

For every worksheet, we create three arrays: one to store the cell hashes, one to store the row hashes and one to store the column hashes. Then, in a single loop through all allocated cells in the current worksheet, every cell with corresponding row i and column j is considered. When the cell contains a formula, the actual formula is hashed; when the cell contains a value, the actual value is hashed. The hash is stored in the 2-dimensional array with cell hashes, which is indexed on row i and column j. Additionally, the same hash is added to the array with row hashes at index i and is added to the array of column hashes at index j. We include the column/row index in the addition operation to guarantee that the location of the cell in the corresponding row/column is included. Otherwise, a row with one cell '42' at position A1 will get same the same hash of a row containing the same cell '42' on another position B1, for instance.

The pseudo-code of the algorithm for hashing all data in a worksheet is listed in Algorithm 1. After looping through all cells, we have a complete summary of all cell hashes, row hashes and column hashes. When two rows contain exactly the same cells, the row hashes are constructed to be equal. And when two columns contain exactly the same cells, the column hashes are constructed to be equal. The cell hashes are used to speed up further comparison steps. It is

[1] https://cyan4973.github.io/xxHash/

no problem to have all cell, row and column hashes of all worksheets in both spreadsheets in memory, because each hash fits in four bytes.

Algorithm 1 Extract hashed data from a worksheet

Input: A worksheet with a dimension of n rows and m columns
Output: The cell hashes, row hashes, column hashes of this worksheet

```
1:  function ExtractHashedData($sheet$)
2:      Let $cells[0 \dots n, 0 \dots m]$, $rows[0 \dots n]$ and $columns[0 \dots m]$ be new arrays
3:      for all $cell \in sheet.Cells$ do
4:          if $cell.Formula \neq$ "" then
5:              $hash \leftarrow$ xxHash($cell.Formula$)
6:          else if $cell.Value \neq$ "" then
7:              $hash \leftarrow$ xxHash($cell.Value$)
8:          else
9:              continue
10:         end if
11:         $i \leftarrow cell.Row$
12:         $j \leftarrow cell.Column$
13:         $cells[i, j] \leftarrow hash$
14:         $rows[i] \leftarrow rows[i] + hash \times (j + 1)$
15:         $columns[j] \leftarrow columns[j] + hash \times (i + 1)$
16:     end for
17:     return $(cells, rows, columns)$
18: end function
```

Ideally, we want the comparison algorithm to traverse at most once through all allocated cells in the spreadsheet. We constructed this pre-processing step such that the output allows us to draw conclusions which parts of the worksheets are unchanged. A major advantage of this approach is that we can immediately skip the matching parts of the two spreadsheets, based on the same row/column hashes. After a relatively cheap pre-processing step (calculating the hashes), the whole spreadsheet comparison problem is reduced to the comparison of only the parts that are changed.

A requirement for correct results is that all the cell information must be included in the hash. Otherwise the cell hashes, and the corresponding row/column hashes, will remain the same even if the represented cells are changed. In the initial approach, it is enough to focus on cell formulas and values. But in the future, if more advanced comparison is required, additional attributes (e.g. style, protection, number formatting) should be appended to the hashes. The hash function should be constructed such that it always creates a unique hash for every cell.

Every hashing approach has the potential risk of 'hash collisions': the situation where two distinct pieces of data get the same hash value. Collisions can also occur by the fact that we add cell hashes to generate our row and column hashes. In theory, a hash collision has impact on the correctness of our comparison approach. If two distinct cells get the same hash value, they are considered equal in the further processing steps. However, because the hashes are 32-bits integers, there is space for 2^{32} possible numbers, which are more than 4 billion possible hash values. The chance of a hash collision is therefore very small. Moreover, we skip parts of the spreadsheets based on row and column hashes, and not on individual cell hashes. Hence, our approach will only be incorrect if a hash collision has occurred in every cell in a row or column. Generally, rows and columns contain multiple cells, and a hash collision is already exceptional, so the risk of incorrect results is therefore very low. In the case that a hash collision occurs in one cell of a row or column, different row/column hashes will produced. In a further comparison step, we inspect the actual cells and still conclude that the rows are equal. So this scenario, that is unlikely to happen, will only have an impact on the performance, not on the correctness.

Another issue is that when structure changes occur (e.g. inserting a new row), many formulas are updated. That will create many different row hashes, even if the actual content of these rows is unchanged. We solved this problem by calculating the hash on the formula in R1C1 notation. In the default A1 notation all formulas have absolute references, e.g. `=SUM(B1:B10)`. In the R1C1 notation all formulas have relative references, e.g. `=SUM(R[-10]C:R[-1]C)`. For instance, an inserted column A will update the formula in A1 notation, but in R1C1 notation the formula will be exactly the same. A downside is that it costs time to convert all formulas to R1C1 notation, because only the formula in A1 notation is saved in the raw XML. We observed that for performance reasons, it is better to use the formula in A1 notation for very large sheets (> 10,000 rows). Furthermore, one type of structure change, a worksheet rename, will still cause many updated formulas in both formula notations. But even when many formulas are updated, our approach will still produce valid results. Only the optimization of excluding unchanged rows is omitted. In general, formulas are mostly updated due to structure changes of row/column insert and row/column delete. Our approach is optimized for these cases.

Finally, a more fundamental issue is the fact that it is difficult to deal with combinations of row and column inserts/deletes. In our hashing approach, the insertion of one new row and one new column will change all the row and column hashes. In this case, we cannot exclude any parts of the worksheet that are unmodified. Therefore, our further comparison steps must be able to compare complete worksheets and compare all cells efficiently. The performance gain of having all cell hashes, instead of full cell objects, is still present.

After all, the idea of calculating cell hashes, row hashes and column hashes is an important optimization. For example, when only 10 out of 10,000 rows in a worksheet have been changed, the cell hashing approach reduces the number of cells to be inspected with a factor of 1000. This pre-processing step has effect on every spreadsheet comparison, because it is likely that some, often large, parts of the spreadsheets are unchanged. This enhancement is constructed to be general, such that it will speed up the comparison approach for all sorts of spreadsheets.

4.4 2D Alignment

The most difficult part of the spreadsheet comparison is the analysis of the structure of two worksheets. We are given two worksheets containing a collection of cells, indexed by row and column. In fact the two worksheets can be seen as two 2D matrices A and B, with size $n_A \times m_A$ and $n_B \times m_B$. In general, the number of rows $n_A \neq n_B$ and the number of columns $m_A \neq m_B$, therefore the two matrices need to be aligned. The goal of the alignment is to detect which rows r_A and columns c_A are deleted, and which rows r_B and columns c_B are inserted. Then the target alignment $T(A,B)$ is known, which contains the elements that are both present in matrix A and matrix B. We will refer to this target alignment as the collection of aligned cells in the two worksheets. Before we can compare the cells in a worksheet, we need to align the corresponding matrices A and B to find r_A, c_A, r_B, c_B and $T(A,B)$. The structure analysis of two worksheets is therefore equivalent to the problem of aligning two 2D matrices with unequal size.

4.4.1 Overview

We propose a novel approach that solves the 2D alignment of two worksheets efficiently. The idea is that we calculate the longest common subsequence lcs on row hashes of A and B. The row hashes of A and B that are in the lcs are the matching rows, and the row hashes of A and B that are not in the lcs are the unique rows. At this point, we conclude that all the matching rows can be skipped as they are exactly equal. No structure changes occurred in these rows.

For the unique rows, we compute consecutive row segments: rows that are at corresponding positions in A and B. A row segment contains unique rows only in A, unique rows only in B, or combinations of unique rows in A and B. Based on the row segments, we conclude that inserted rows or deleted rows are found.

For row segments that contain both different rows from A and B, we run a fine-tuned, but complex 2D alignment algorithm (based on RowColAlign [21]) that accepts parts of the matrices A and B. This algorithm finds two aligned sub-matrices, where the rows and columns are selected such that they have maximized similarity, based on 1-dimensional LCS on rows and columns. These two sub-matrices represent the collection of aligned cells in the current segment, and have identical dimensions. Additionally, the 2D alignment algorithm reports the rows and columns of A to delete, and rows and columns of B to insert, to obtain the aligned sub-matrices. This is the required structure information we are looking for. After processing all the row segments, we conclude which rows r_A and columns c_A are deleted, and which rows r_B and columns c_B are inserted, and concatenate the collections of aligned cells to form the target alignment $T(A,B)$. Now the alignment of the two 2D matrices A and B is efficiently solved.

Our approach is efficient, because we run the complex 2D alignment algorithm only on the segments of A and B that have been changed. We avoid running the complex algorithm on the full matrices A and B, which have probably substantial overlap and are possibly large in dimension. The key part in our approach is that we first calculate the 1-dimensional longest common subsequence of the two arrays of row hashes. This computation is fast, for three reasons.

- First, we already optimized the row hashes: every row is represented by a single number of four bytes.
- Secondly, we select an efficient implementation of the LCS algorithm, based on our findings in Section 2.1.2.

- Thirdly, we apply a trimming technique where we skip the start and end parts of the two arrays of row hashes that are equal.

Even for large worksheets with more than 10,000 rows, the computation of the 1D LCS can be executed in a few milliseconds.

Moreover, at the start of our approach, we determine not only the LCS on the row hashes, but also on the column hashes. Based on that, we count the number of matching rows and the number of matching columns. If the columns turn out to be more similar than the rows, we continue with an exact analogous approach as described in the previous paragraph, but now on the columns rather than on the rows. For example, when column modifications are performed all row hashes are changed. In that case we flip the approach and continue the alignment based on column segments. This prevents that the complex 2D alignment algorithm is executed on the whole matrices A and B. Finally, at multiple points in our approach we check for trivial cases and draw conclusions as fast as possible.

4.4.2 Detailed description in 10 steps

Now that the general idea of our 2D alignment approach is presented, we explain it in more detail. Our complete approach consists of 10 steps, from input (two worksheets A and B) to output (inserted/deleted rows, inserted/deleted columns, aligned cells).

1. **Extract the data from A and B.** In a single loop over all allocated cells in A and B, compute the used cell range, cell hashes, row hashes and column hashes. The algorithm for extracting the hashed data is described in Algorithm 1. In the same loop, the minimum and maximum row and column index is continuously being updated. After processing all cells, these indices define the used cell range in the worksheet.
2. **Check for trivial cases.** If B is empty and A is not empty, report all rows in A as deleted rows r_A. If A is empty and B not empty, report all rows in B as inserted rows r_B.
3. **Determine the unique rows in A and B.** First compute trimmed start and end bounds on the two arrays of row hashes. Scan both arrays of row hashes simultaneously from the start, proceeding to the next item until the two hashes are unequal. Repeat the same scan from the end of both arrays. Then a start offset and end offset are found: the number of items that are equal from the start and the end. Those parts of the two arrays can be skipped, because they are the same. An array segment is defined on the two arrays of row hashes, starting from the start offset and ending at last item minus the end offset.
   Then, compute the longest common subsequence on the two array segments of row hashes. The result are two lists of matching row indices, rows that are exactly equal. The unique rows of A are calculated by enumerating until the height of A, excluding the matching rows. Similarly, the unique rows of B are calculated by enumerating until the height of B, excluding the matching rows.
4. **Determine the unique columns in A and B.** Repeat the same procedure for calculating the longest common subsequence, now on the two arrays of column hashes. The result are the matching column indices, columns that are exactly equal. The unique columns of A are all columns until the width of A, excluding the matching columns. Similarly, the unique columns of B are all columns until the width of B, excluding the matching columns.

5. **Check for trivial cases.** Based on the four collections (unique rows in A and B, unique columns in A and B) conclusions can be drawn. If all collections are empty, all rows and columns in A and B are the same and nothing has been changed. If there are only unique rows in A and no unique rows in B, all these rows are reported as deleted rows r_A. If there are only unique rows in B and no unique rows in A, all these rows are reported as inserted rows r_B. Equivalently, if there are only unique columns in A and no unique columns in B, all these columns are reported as deleted columns c_A. And finally, if there are only unique columns in B and no unique columns in A, all these columns are reported as inserted columns c_B. In all these trivial cases, the 2D alignment method is finished.
6. **Select a strategy**, in case of both unique rows and columns in A and B. The ratio of unique rows compared to the worksheet height and the ratio of unique columns compared to the worksheet width is computed. The default case is to continue with the row-wise approach, if the rows are more similar than the columns. However, if the columns are more similar than the rows, continue with the column-wise approach. The next steps will be explained in terms of rows. The case with columns is analogous.
7. **Calculate separate row segments.** For all the unique rows, find consequetive segments of rows that correspond to the same location in A and B. For example, assume that unique rows in A are $[1,2,3,20,21,22]$ and unique rows in B are $[11,12,19,20,21,32,33,34]$. Then the individual row segments are $([1,2,3],[])$, $([],[11,12])$, $([20,21,22],[19,20,21])$, and $([],[32,33,34])$. Three cases can be distinguished: unique rows that are only in A, unique rows that are only in B and combinations of unique rows in both A and B.
8. **Check for trivial cases.** All the individual row segments are inspected. When there are unique rows in A and no corresponding rows in B, all rows are reported as deleted rows r_A. When there are unique rows in B and no corresponding rows in A, all rows are reported as inserted rows r_B.
9. **Run fine-tuned 2D alignment algorithm.** For every row segment that contains both unique rows in A and B, the content needs to be aligned. Since the number of rows is possibly unequal, the goal is to find an alignment with the same dimensions, so that later individual cells can be compared. The unique rows of A and B in the current segment, together with the overall unique columns of A and B are forwarded to a RowColAlign algorithm. The inner workings of the RowColAlign algorithm are described in Section 2.1.4, more details about the modifications are given below. The algorithm outputs a target alignment and the corresponding deleted and inserted rows and columns to reach that target alignment from the input. A structure analysis for every individual row segment is now available.
10. **Collect results.** After processing all row segments, the rows marked as deleted or inserted in every individual row segment are combined to the overall collections r_A and r_B. Furthermore, only the columns marked as deleted or inserted in all row segments are added to the overall collections c_A and c_B. The aligned cells detected in every row segment are appended to the overall target alignment $T(A,B)$.
    After all, the 2D alignment approach is finished here. All relevant structure changes in the two worksheets are detected and are stored in collections r_A, r_B, c_A and c_B. In addition, the final target alignment $T(A,B)$ contains all the aligned cells in the two worksheets.

4.4.3 Fine-tuned 1D LCS algorithm

The longest common subsequence algorithm plays an important role in the alignment of the two worksheets. For every worksheet, the LCS is at least calculated on the two arrays of row hashes (step 3) and the two arrays of column hashes (step 4). Additionally, for different row/-column segments the LCS is calculated multiple times in the RowColAlign algorithm (step 9), to measure the similarity of rows and columns. It is therefore fruitful to optimize the performance of the LCS algorithm, as it will improve the performance of the spreadsheet comparison in general. The LCS calculation is one of the bottlenecks in terms of run time.

In Section 2.1.2 we gave an overview of the LCS algorithm and concluded that different variants should be compared to the naive implementation. We implemented four variants of the LCS algorithm: the ‘naive’ Wagner-Fischer approach [8], and the algorithms of Hunt-Szymanski [10], Kuo-Cross [11] and Wu, Manber, Myers, and Miller [12]. Our findings match the conclusion of the survey paper [6], namely that the Kuo-Cross algorithm performs very well for varying input types. We further optimized the implementation of this algorithm, in combination with the ideas of the follow-up paper [7], to perform at its best for our specific input (row, column and cell hashes). Our final implementation of the 1-dimensional longest common subsequence algorithm is listed in Algorithm 2.

4.4.4 Fine-tuned 2D alignment algorithm

An optimized version of the RowColAlign algorithm solves our problem of aligning row and column segments with unequal dimensions. In Section 2.1.4 we have seen two methods in the literature that solve this problem: SheetDiff [20] and RowColAlign [21]. We built further on the RowColAlign approach, for two reasons. First, the SheetDiff algorithm is a greedy, local optimization algorithm that can fail to terminate or can get stuck in a local optimum. Secondly, the dynamic programming approach of the RowColAlign algorithm has proven to improve the results of SheetDiff for aligning two 2D matrices.

We improved the RowColAlign algorithm with several modifications. Firstly, we adapted the algorithm such that it can compare sub-matrices instead of the whole matrices A and B. This is done by providing row indices of A and B and column indices of A and B that should be considered. This modification allows us to ‘zoom in’ on specific row or column segments, as we can input collections of unique rows and columns in A and B to align. Hereby we solve the problem that RowColAlign runs out of memory for large matrices, as it requires $O(n^2)$ space.

Secondly, we added a threshold to the value in the *LCS1D* matrix, which contains the similarity scores for all combinations of rows and columns. It requires a minimum number of cells in a row or column to match, otherwise the rows or columns are considered to be different. This threshold is added to prevent that for example an inserted row with one cell is matched to an already existing row with one cell. We need at least a number of cells in a row or column to confirm that two rows or columns are comparable. The threshold is set to $0.05 \times$ the row width or the column height.

Thirdly, we defined a different cell weight for empty cells and non-empty cells. This weight is introduced to improve the similarity score of rows and columns. We observed that rows containing many empty cells were often incorrectly matched, because they have so many cells in common. Empty cells count with a factor of 0.5, non-empty cells count with a factor of 1.0.

Algorithm 2 Optimized implementation of the 1D LCS algorithm

Input: Two arrays X and Y with length m and n

Output: The length r of the longest common subsequence and the corresponding *items*

```
 1: function CalculateLCS($X, Y$)
 2:     Let $MinYPrefix$ be a new array of length $m + 1$
 3:     Let $items$ be a new array of length $m + 1$
 4:     $MinYPrefix[0] \leftarrow 0$
 5:     $MinYPrefix[i] \leftarrow n + 1, \forall i \in 1 \dots m + 1$
 6:     $items[0] \leftarrow \epsilon$
 7:     $r \leftarrow 0$
 8:     for $i \leftarrow 0$ to $m - 1$ do
 9:         $updates \leftarrow \varnothing$
10:         for $k \leftarrow 0$ to $r$ do
11:             $r_1 \leftarrow MinYPrefix[k]$
12:             $r_2 \leftarrow MinYPrefix[k + 1] - 1$
13:             for $j \leftarrow r_1$ to $r_2$ do
14:                 if $X[i] = Y[j]$ then
15:                     $updates \leftarrow updates \cup \{(k + 1, j)\}$
16:                     break
17:                 end if
18:             end for
19:             for all $(k, j) \in updates$, in reverse order, do
20:                 $MinYPrefix[k] \leftarrow j + 1$
21:                 $items[k] \leftarrow (i, j, items[k - 1])$
22:                 if $k > r$ then
23:                     $r \leftarrow r + 1$
24:                 end if
25:             end for
26:         end for
27:     end for
28:     return $(r, items)$
29: end function
```

Finally, the default RowColAlign algorithm does not consider formulas. We adapted the structure of the algorithm to operate on cell hashes. In the cell hash the value or formula (either in A1 or R1C1 notation) is encoded, depending on the type of cell. Now that the algorithm operates on general cell hashes, it can align cells with any features. It is therefore easy to extend the algorithm in the future.

Beyond these improvements, the general technique of RowColAlign presented in [21], to align two 2D matrices using dynamic programming, is still unchanged.

4.4.5 Conclusion

Our complete proposal for the 2D alignment contains specific refinements for the comparison of worksheets in a spreadsheet. Our method is applicable to align all 2D matrices of unequal size, in general. It performs especially well on large matrices that have many items in common. One of the weaknesses is that it is still hard to solve the 2D alignment for matrices having both row and column modifications. All the optimizations of considering row hashes and column hashes are then irrelevant, because all rows and column hashes are changed in this scenario. We earlier concluded that the combination of row and column modifications is a more fundamental issue, with no trivial solution. Our approach will still produce correct results, but it will run the fine-tuned 2D alignment algorithm on the whole matrices A and B. This operation can therefore be costly in terms of time and space.

However, for realistic spreadsheets, we will see in Chapter 6 that our optimized approach performs very well. Many optimizations that we built in came up from testing the approach on real spreadsheets in practice. We argue that a good structure analysis, which is the 2D alignment of worksheets, provides the foundation of a correct spreadsheet comparison. Therefore this section represents one of the most important solutions for solving the spreadsheet comparison problem.

In the next part of this chapter, we propose solutions on how to actually address the comparison of two spreadsheets S_1 and S_2. We explain the comparison on four different aspects, corresponding to the four change detection tasks of the comparison pipeline. The solutions are described in Sections 4.5 to 4.8. Each solution will result in one or more change objects, as defined in Table 4.1. After each section we will mention which type of changes are the result of the described solution. In total 28 change types will be outputted by the comparison of all aspects of the spreadsheet.

4.5 Comparing Worksheets

We start by explaining how to compare the collections of worksheets in two spreadsheets. In the next subsections we describe three methods for detecting all worksheet changes.

4.5.1 Finding matching worksheets

Initially, we have two lists of worksheets in spreadsheet S_1 and S_2. The first step is to find the matching worksheets. Matching worksheets are the worksheets that are present in both spreadsheet S_1 and S_2, possibly with a different name. When the matching worksheets are recognized, the deleted worksheets in S_1 and the inserted worksheets in S_2 are also known.

We start by matching worksheets of S_1 and S_2 that have exactly the same name. Then, we compute lists of unique worksheets in S_1 and S_2, which are the worksheets that do not match. When there are unique worksheets in both lists, we try to match worksheets by their internal identifier. In the raw XML data, it is possible to retrieve the '`xr:uid`' attribute in the worksheet XML. This internal identifier is created when a new worksheet is added. When a worksheet is renamed, this 'uid' attribute remains the same. Luckily, we can match the renamed worksheets based on this identifier, instead of having to inspect the contents of the worksheets. If this step produces any matches, they are added to the collection of matching worksheets and removed from the collections of unique worksheets. Finally, the collection of deleted worksheets are the remaining unique worksheets in S_1 and the collection of inserted worksheets are the remaining unique worksheets in S_2. The pseudo-code for analyzing the worksheets is listed in Algorithm 3.

When the inserted, deleted and matching worksheets are known, it is easy to report the corresponding changes. An worksheet inserted is reported for every inserted worksheet, and a worksheet delete is reported for every deleted worksheet. All the matching worksheets are checked, for any of them whose name is changed a worksheet rename is reported.

Result 1. WorksheetInsert, WorksheetDelete, WorksheetRename

4.5.2 Finding worksheet state modifications

For the collection of matching worksheets, the state of both worksheets is checked. We examine the hidden state and the protected state. For any worksheet that is visible in S_1 and hidden in S_2, a worksheet made hidden is reported. For any worksheet that is hidden in S_1 and visible in S_2, a worksheet made visible is reported. Note that a worksheet can also be very hidden. This is a specific trick in Excel to programmatically hide worksheets. Very hidden worksheets are really hidden in the user interface and are not shown in the 'unhide' option. The changes for very hidden sheets are also reported. Equivalently, we check the state of protected worksheets.

Algorithm 3 Analyze worksheets for insertions, deletions and matches

Input: List of worksheets W_1 and W_2 of two spreadsheets
Output: List of inserted worksheets I, deleted worksheets D and matching worksheets M

```
 1: function CompareWorksheets($W_1, W_2$)
 2:     $M \leftarrow \varnothing$
 3:     for all $(s_1, s_2) \in$ MatchByName($W_1, W_2$) do
 4:         $M \leftarrow M \cup \{(s_1, s_2)\}$
 5:     end for
 6:     $U_1 \leftarrow W_1 - \{s_1 \in M \rightarrow (s_1, s_2)\}$        ▷ unique worksheets in $W_1$
 7:     $U_2 \leftarrow W_2 - \{s_2 \in M \rightarrow (s_1, s_2)\}$        ▷ unique worksheets in $W_2$
 8:     if $U_1 \neq \varnothing \wedge U_2 \neq \varnothing$ then
 9:         for all $(s_1, s_2) \in$ MatchByUid($U_1, U_2$) do
10:             $M \leftarrow M \cup \{(s_1, s_2)\}$
11:             $U_1 \leftarrow U_1 - \{s_1\}$
12:             $U_2 \leftarrow U_2 - \{s_2\}$
13:         end for
14:     end if
15:     $I \leftarrow U_2$
16:     $D \leftarrow U_1$
17:     return $(I, D, M)$
18: end function
```

For any worksheet that is unprotected in S_1 and protected in S_2, a worksheet made protected is reported. For any worksheet that is protected in S_1 and unprotected in S_2, a worksheet made unprotected is reported. Because we use the collection of matching worksheets from the structure analysis, we are able to correctly detect state changes for worksheets that are renamed.

Additionally, we inspect the collection of inserted worksheets and examine the state of those sheets. If a newly inserted worksheet is also made hidden, very hidden or protected, this is reported.

Result 2. WorksheetHidden, WorksheetVisible, WorksheetProtected, WorksheetUnprotected

4.5.3 Finding worksheet order changes

Every worksheet in a spreadsheet has a position. In the XML, the first worksheet has position 0, the second worksheet position 1, and so on. The detection of reordered worksheets is performed on the visible sheets in S_2. It is unnecessary to consider deleted sheets in S_1 or hidden sheets in S_2, because order changes are only relevant for worksheets that are visible to the user. For the visible sheets, we find the matching worksheets in S_1 and S_2 and determine the *aligned* worksheet order. This is the position of all matching worksheets, corrected for inserted and deleted worksheets. The result are two collections P_1 and P_2 containing the aligned positions of both matching worksheets, starting with 0 and ending with n (the number of matching worksheets). There are no gaps for inserted or deleted worksheets.

We check the two collections of aligned worksheet orders, and return no changes if they are exactly the same. In the case that the worksheet orders are different, we run a greedy algorithm to find the minimum set of worksheet order changes between the two files. In every iteration, the current positions in P_1 are compared with the final positions in P_2. The action that has the highest impact to improve the current positions is then selected. This reorder action is then recorded and simulated, updating the current positions in P_1. This process is repeated until the current positions P_1 match the final positions P_2. The pseudo-code for the greedy algorithm that finds the worksheet order changes is listed in Algorithm 4.

Algorithm 4 Find the minimum set of worksheet order changes

Input: List of initial worksheet positions P_1 and final worksheet positions P_2
Output: List of movements M for all worksheets to move from positions P_1 to P_2

```
1: function FINDWORKSHEETORDERCHANGES($P_1, P_2$)
2:     Let $M$ be a new array with same length of $P_1$ and $P_2$
3:     repeat
4:         $D \leftarrow P_2 - P_1$
5:         $i \leftarrow \text{argmax}_k |D[k]|$
6:         $move \leftarrow D[(i]$
7:         if $move > 0$ then
8:             $start \leftarrow P_1[i] + 1$
9:             $end \leftarrow start + move$
10:            $step \leftarrow -1$
11:        else
12:            $start \leftarrow P_1[i] + move$
13:            $end \leftarrow P_1[i]$
14:            $step \leftarrow 1$
15:        end if
16:        for all $\{j \in P_1 \mid P_1[j] \geq start \wedge P_1[j] < end\}$ do
17:            $P_1[j] \leftarrow P_1[j] + step$
18:        end for
19:        $P_1[i] \leftarrow P_1[i] + move$
20:        $M[i] \leftarrow M[i] + move$
21:    until $P_1 = P_2$
22:    return $M$
23: end function
```

For every reordered sheet, we report a worksheet order change with the original position and the new position.

Result 3. WorksheetOrderChange

4.6 Comparing Defined Names

A spreadsheet can contain defined names to simplify the references in formulas. A defined name consists of a name and value. The value is always a formula. An example is the defined name '`Numbers`' with value '`=Sheet1!$B$1:$B$10`'. Now a valid formula '`=SUM(Numbers)`' can be written, to calculate the sum of the ten numbers in B1 to B10. Defined names are easy to manage and allow abstraction and re-use of ranges in the formulas of the spreadsheet. By default defined names are created in the workbook (global scope), but it is also possible to create defined names in the workbook (local scope).

To compare the defined names, we first collect all the defined names in the workbook (global scope) and worksheets (local scope), both for spreadsheet S_1 and S_2. We group the defined names by their scope. Then we inspect the defined names, first on global scope, and then for every local scope. The reason we do this is because multiple defined names can occur with the same name but in a different scope. And a defined name can be deleted from the global scope and added to a local scope. Both cases must be detected by the comparison.

The inspection of two collections of defined names D_1 and D_2 in a certain scope starts with a grouping based on the name. The defined names in D_1 and D_2 with equal names are considered to be matching defined names, the rest are unique defined names in D_1 and unique defined names in D_2. For the pairs of matching names, the values are compared and a defined name change is reported when the formula is updated. However, before the actual formulas are checked, a formula transformation based on the detected structure changes is applied. The formula transformer takes formula f_1, applies a correction for all row inserts/deletes, column inserts/deletes and worksheet renames (if applicable), and returns a transformed formula $f_1^{\prime}$. Then, formula $f_1^{\prime}$ is compared with f_2 and only when the formulas are really different, we know that a manual formula update has taken place. Only in these cases a defined name update is reported. We skip hereby redundant changes caused by structure changes, as we are only interested in the actual formula updates by users themselves. The pseudo-code for the algorithm to transform formulas based on a set of structure changes is listed in Algorithm 5. The exact same strategy for comparing formulas is applied when comparing the cells, as we explain in Section 4.8.

After processing the matching defined names, the unique defined names in D_1 and D_2 are considered. If a pair of unique defined names is found with exactly the same value, a defined name rename is reported. This only happens when one matching value is found in D_1 and one matching value is found in D_2; values might not be unique. This implicates that defined name renames are only detected if the name is changed and the value is unchanged. For the rest of the unique defined names, a defined name delete is reported if the defined name is only in D_1, and a defined name delete is reported if the defined name is only in D_1.

Result 4. DefinedNameInsert, DefinedNameChange, DefinedNameRename, DefinedNameDelete

Algorithm 5 Transform a formula to correct for structure changes

Input: Original formula f, and a list of structure changes S. S can contain: modified sheet names, inserted rows, inserted columns, deleted rows, deleted columns

Output: Transformed formula f' that is updated for the given structure changes

```
function TRANSFORMFORMULA($f, S$)
    if $S = \emptyset$ then
        return $f$
    end if
    $T \leftarrow$ TOKENIZE($f$)                ▷ split formula (e.g. '=', 'SUM', '(', 'B1:B10', ')')
    $f' \leftarrow$ ""
    for all $t \in T$ do
        if $t$.TokenType is no ExcelAddress then
            $f' \leftarrow f' + t$
            continue
        end if
        $address \leftarrow t$.Address                ▷ e.g. 'A10' or 'Sheet1!$B$3'
        if $address$.Worksheet ≠ "" ∧ $address$.Worksheet ∈ $S$.ModifiedSheets then
            $s \leftarrow \{(s_1, s_2) \in S.\text{ModifiedSheets} \mid s_1 = address.\text{Worksheet}\}$
            $address$.UPDATEWORKSHEET($s$)                ▷ update worksheet, if applicable
        end if
        for all $i \in S$.DeletedRows, in reverse order, do
            $address$.DELETEROW($i, 1$)                ▷ delete one row, if applicable
        end for
        for all $i \in S$.InsertedRows do
            $address$.INSERTROW($i, 1$)                ▷ insert one row, if applicable
        end for
        for all $i \in S$.DeletedColumns, in reverse order, do
            $address$.DELETECOLUMN($i, 1$)                ▷ delete one column, if applicable
        end for
        for all $i \in S$.InsertedColumns do
            $address$.INSERTCOLUMN($i, 1$)                ▷ insert one column, if applicable
        end for
        if $address$ is invalid then
            $f' \leftarrow f'$ + "#REF!"
        else
            $f' \leftarrow f' + address$
        end if
    end for
    return $f'$
end function
```

The next two steps, comparing rows and columns (Section 4.7), and cells (Section 4.8) in a worksheet, are performed only for all matching worksheets s_1 and s_2.

4.7 Comparing Rows and Columns

For the detection of row and column changes, we rely on the previously detected worksheet structure. Every worksheet is already analyzed and at the moment of comparison, the collections of inserted rows and columns, deleted rows and columns and the used cell range of both worksheets are known. In the next two subsections we describe how row/column inserts and deletes and row/column state modifications are detected.

4.7.1 Finding row/column insertions and deletions

Because most of the difficult work is already done in the structure analysis, we can directly start with inspecting the collections of inserted rows/columns and deleted rows/columns. It is not correct to immediately report row/column inserts and row/column deletions for every result from the structure analysis. Consider for example the case that s_1 has an used cell range of A1:D10, and that a new value is entered at cell E1000. The used cell range of s_2 will then be A1:E1000, and 990 inserted rows will be returned by the structure analysis. In this case it is expected to only report one value insert, and no row or column modifications. The reporting of such changes should therefore depend on the used cell range of the other corresponding worksheet.

We argue that it is only required to report row and column changes for modifications that fall into the used range of the other version. In the example just given, it was relevant to show that an empty row at position 4 was inserted, because row 4-10 are shifted downwards. But an empty row at position 11 would have no effect, the used cell range ends at position 10. Hence, an inserted row should only be reported if the row index is within the height of worksheet s_1. A deleted row should only be reported if the row index is within the height of worksheet s_2. Equivalently, an inserted column should only be reported if the column index is within the width of worksheet s_1. And a deleted column should only be reported if the column index is within the width of worksheet s_2.

Furthermore, it can happen that the same row (or column) is deleted and inserted. The structure analysis will return the index of such a row both in the inserted rows and deleted rows. Now we have to make a design choice. We can report this as two changes, a row insert and a row delete, or ignore the row changes and let the cell comparison detect updated cells. In the first case a user will see all deleted cells in the row and all inserted cells in the row, reported as separate changes. In the second case a user will see updated cells, or cell inserts or cell deletes when the original or new cells are empty. It is not obvious which scenario has been occurred. Here, we come across another fundamental issue about comparing spreadsheets. Because we only have the original and new spreadsheet file, we do not know which intermediate steps have been taken. It is impossible to decide if the user really deleted and inserted a row at the same position, or updated all cells in that row manually. The end result in the raw XML is exactly the same.

We think that it is more relevant to a user to report updated cells (second case), because it is confusing to report both an insert and a delete at the same position but in separate changes.

Therefore, we compute the intersection of all inserted rows in s_1 and deleted rows in s_2, and mark them as changed rows. We report for all inserted rows, except the ones that are changed or do not fall into the used cell range, a row insert. We report for all deleted rows, except the ones that are changed or do not fall into the used cell range, a row delete. Equivalently, the same is done for the inserted and deleted columns. The final result of this approach is that row and column modifications are only reported when they changed the structure of the worksheet.

Result 5. RowInsert, RowDelete, ColumnInsert, ColumnDelete

4.7.2 Finding row/column state modifications

Every row or column in a worksheet can be made hidden. It is important to detect such changes, because it is often confusing what has happened in these scenarios. A hidden row looks like a deleted row, but that is not the case. For the detection of state modifications, we cannot simply traverse through all rows in s_1 and s_2, comparing row 1 with row 1, row 2 with row 2, and so on. Structure changes affect the alignment of rows, and therefore we rely on the information about inserted and deleted rows. The same holds for the columns.

To detect state modifications we need to know the matching rows and columns in s_1 and s_2. For s_1, the matching rows are all rows until the height of the used cell range, except the deleted rows. For s_2, the matching rows are all rows until the height of the used cell range, except the inserted rows. We loop over all pairs of matching rows and compare the state of both rows. If one row in s_1 is visible and the corresponding row in s_2 is hidden, we report a row made hidden. If one row in s_1 is hidden and the corresponding row in s_2 is visible, we report a row made visible. Again, the identical is done for the columns. Additionally, we inspect the inserted rows/columns and report a row/column made hidden if the newly inserted row/column is hidden as well.

Surprisingly, there are performance considerations for the relatively simple task of detecting row/column state changes. Consider the example where worksheet s_1 has a used cell range of A1:D10. Now suppose a value is inserted in cell A1048576, the last cell that will be visited with the shortcut CTRL + DOWN. The used cell range of s_2 is now changed to A1:D1048576, where the height is increased with more than a million rows. Our approach for detecting the state changes will examine all rows until the last cell, which is a lot of work in this case. In general, it is difficult to avoid this issue. It was possible that one row was made hidden in this enormous amount of rows. If the whole used cell range is not examined, such state changes will not be reported. Of course this is an exceptional situation and the detection of row/column state changes on normal worksheets is rather fast to check. Possibly the detection can be more optimized by a direct inspection of the raw XML data.

Result 6. RowHidden, RowVisible, ColumnHidden, ColumnVisible

4.8 Comparing Cells

The strategy for the detection of cell changes relies once more on the detected worksheet structure of worksheet s_1 and s_2. The outcome of the structure analysis was a list of inserted rows, deleted rows, inserted columns, deleted columns, and a collection of aligned cells.

First, a comparison of all aligned cells is performed. The collection of aligned cells consists of pairs of cell references (c_1, c_2) that belong to each other in s_1 and s_2. For example, (A1, A1), (B1, C1), (C1, D1), (A2, A2), (B2, C2), (C2, D2), and so on. References to empty cells are also present in this collection. It simply defines the mapping between all cells in the complete used cell range of s_1 and s_2; cells that are not related to an inserted or deleted row/column. It is required to compare pairs of *aligned* cells, because structure modifications change the alignment of cells. Incorrect results will be obtained if all cells were simply compared based on their location (e.g. A1 with A1, A2 with A2, and so on).

Additionally, we inspect all the cells in the inserted and deleted rows. Although a row insert or row delete is already reported, it is likely that additional data or model changes have occurred in the cells of inserted and deleted rows. For an empty row insert, we expect only one change to be reported: a row insert (structure change). But for an inserted row with content, we expect multiple changes to be reported: a row insert (structure change), plus one or more value inserts (data change) or formula inserts (model change). Hence, we should look into the cells of the inserted and deleted rows. It can happen that the same row is deleted and inserted. Therefore, we first calculate the collection of the changed rows, which are the indices of rows that are both deleted and inserted. We loop over all cells in the changed rows and compare the contents of these cells, just like we did for the aligned cells. Then, for all cells in the inserted rows, excluding rows that are changed, we report value inserts and/or formula inserts. And for all cells in the deleted rows, excluding rows that are changed, we report value deletes and/or formula deletes.

Equivalently, we follow the same procedure for the inserted and deleted columns. To prevent the detection of duplicate cell changes, we check the current row index for every cell while processing the inserted and deleted columns. If an inserted/deleted row is also detected at the current row index, we do not report a cell change for the inserted/deleted column. Consider for example the case that both row 4 and column D are inserted, and that values are inserted at all cells in that row and column until D4. The inserted values are then A4, B4, C4 and D4, and D1, D2, D3 and D4. In fact, only seven values are present in the new row and new column, so we want to report the value insert at D4 only once. Therefore, while processing the inserted cells in column D, we detect that row 4 is already inserted and do not report a duplicate value insert of D4.

The comparison of two cells depends on the state of each cell. We noticed that every cell is either an:

- empty cell (neither the 'formula' and 'value' field are set)
- value cell (only the 'value' field is set)
- formula cell (both the 'formula' and 'value' field are set)

We conclude that there are nine possible cases when two cells c_1 and c_2 are compared. For instance, when cell c_1 is empty and cell c_2 is a formula cell, we should report a formula insert. When c_1 only has a value and c_2 has a value and a formula, we should report a value changed to formula. And when c_1 and c_2 both have a value and no formula, we should compare the values and report a value change. All scenarios are listed in Table 4.2. The table shows the type of change that should be reported depending on the state of both cells.

Table 4.2: All possible cases for detecting cell changes in two cells

		Cell 1		
		× Value × Formula	✓ Value × Formula	✓ Value ✓ Formula
Cell 2	× Value × Formula	-	ValueDelete	FormulaDelete
	✓ Value × Formula	ValueInsert	ValueChange	FormulaToValue
	✓ Value ✓ Formula	FormulaInsert	ValueToFormula	FormulaChange

In the method that compares a pair of cells, all possibilities are checked, and the corresponding change is reported. In the case of two value cells, we check the values v_1 and v_2 and only report a value change if $v_1 \neq v_2$. In the case of two formula cells, we inspect formula f_1 and f_2. Now we use the same strategy of transforming formula f_1 into $f_1^{'}$ based on structure changes, as we earlier described in Algorithm 5. We compare $f_1^{'}$ with f_2 and only report a formula change if $f_1^{'} \neq f_2$. Note that we ignore the values of two formula cells, we only compare the formulas. That is because we are not interested in the change of calculated values of formulas. One change in the spreadsheet can lead to many updated calculated values. We reason that it is important to only report the manual actions performed by an end-user With this cell comparison strategy, the user will only see the cause of his actions, and not the propagated result. As shown in Table 4.1, three cell change types are a data change, three cell change types are a model change, and for two changes types both the data and model are affected.

We argue that our cell comparison strategy is efficient, because we only look into the details of a small subset of cells, namely the aligned cells. By examining the row and column hashes in the structure analysis, we already excluded many cells that are unchanged. Therefore, only small parts of the worksheet that contain changes are inputted to our cell change detection. Only for these cells the details are checked. For all other cells, outside the cell detection, the optimized cell hashes are used. Furthermore, inside the cell change detection, all cells are visited only once.

Note that it is straightforward to extend the cell comparison. More cell properties to be checked, for instance display or protection properties, can easily be added to the method that compares two cells. The approach is flexible, in the sense that multiple changes can be reported during the comparison of one pair of cells.

Result 7. ValueInsert, ValueChange, ValueChangeToFormula, ValueDelete,
FormulaInsert, FormulaChange, FormulaChangeToValue, FormulaDelete

4.9 Change Aggregation

In addition to the presented comparison steps, we propose a method to aggregate all detected changes. This method will solve the user need for having an overview of all changes in increasing level of detail. Consider for example the case where a user creates a new worksheet and imports 100,000 rows with 10 columns of data from a database. Our comparison approach up to now will report 1 worksheet insert and 1,000,000 value inserts. In fact, this is a common scenario and can be seen as one edit operation: insert a new worksheet with content. The idea of our aggregation method is that it is much more informative to describe a scenario to a user, than presenting a large list of changes.

We took a high-level overview on our comparison approach and defined 32 scenarios that describe actual differences on spreadsheets. A difference is defined to be a group of changes. Changes are detected by the previously presented comparison steps. In other words, our detection method outputs changes, and our aggregation method outputs differences. The differences are more general than the changes.

In Table 4.3 all 32 types of differences are listed. Every difference contains a short text describing the operation, shown in italics. Just like the changes, every difference is categorized into data, model and structure. For some differences, it depends on the underlying changes to which category it belongs. The last column shows the actual aggregation that is made: the change types that are combined. These change types correspond to the changes we earlier defined in our change categorization. A collection of changes is denoted with an asterisk *, a single change has no asterisk. For example, the 'WorksheetInserted' difference consists of one 'WorksheetInsert' change and multiple 'ValueInsert' and/or 'FormulaInsert' changes.

We came up with the 32 scenarios based on spreadsheet user patterns we could think of. We were inspired by the answers from users in the interviews (Section 3.1), where they indicated which operations they usually perform to create new spreadsheet versions. The idea is that after comparison, initially a summary of all high-level differences between the two spreadsheets will be shown. The user can then zoom in on the details by inspecting the underlying changes in a difference. For example, a user can see that he deleted a worksheet with content (summary). He can expand this difference and see all the values and formulas that were present in this worksheet (details). And for instance, a user can see a list of cell ranges that were modified in a specific worksheet (summary). He can expand each cell range to see the changed values and formulas in this block of cells (details).

Table 4.3: Complete list of all aggregated differences

#	Difference	Category	Aggregation
1	EmptyWorksheetInserted *Empty worksheet 'Sheet4' inserted*	Structure	WorksheetInsert
2	EmptyWorksheetDeleted *Empty worksheet 'Sheet1' deleted*	Structure	WorksheetDelete
3	WorksheetInserted *Worksheet 'Sheet4' inserted*	Data, Model	WorksheetInsert + ValueInsert*, FormulaInsert*
4	WorksheetDeleted *Worksheet 'Sheet1' deleted*	Data, Model	WorksheetInsert + ValueDelete*, FormulaDelete*
5	WorksheetsHidden *2 worksheets are made hidden*	Structure	WorksheetHidden*
6	WorksheetsVisible *2 worksheets are made visible*	Structure	WorksheetVisible*
7	WorksheetsProtected *3 worksheets are made protected*	Structure	WorksheetProtected*
8	WorksheetsUnprotected *3 worksheets are made unprotected*	Structure	WorksheetUnprotected*
9	WorksheetsOrderChanged *4 worksheets are reordered*	Structure	WorksheetOrderChange*
10	WorksheetsRenamed *4 worksheets are renamed*	Structure	WorksheetRename*
11	EmptyRowsInserted *Empty rows 4-8 are inserted*	Structure	RowInsert*
12	EmptyRowsDeleted *Empty rows 4-8 are deleted*	Structure	RowDelete*
13	EmptyColumnsInserted *Empty columns B-C are inserted*	Structure	ColumnInsert*
14	EmptyColumnsDeleted *Empty columns B-C are deleted*	Structure	ColumnDelete*
15	RowsInserted *Rows 100-202 are inserted*	Data, Model	RowInsert* + ValueInsert*, FormulaInsert*
16	RowsDeleted *Rows 100-202 are deleted*	Data, Model	RowDelete* + ValueDelete*, FormulaDelete*
17	ColumnsInserted *Column X-Z are inserted*	Data, Model	ColumnInsert* + ValueInsert*, FormulaInsert*
18	ColumnsDeleted *Column X-Z are deleted*	Data, Model	ColumnDelete* + ValueDelete*, FormulaDelete*
19	RowsHidden *Rows 10-20 are made hidden*	Structure	RowHidden*

#	Difference	Category	Aggregation
20	RowsVisible *Rows 10-20 are made visible*	Structure	RowVisible*
21	ColumnsHidden *Columns F-H are made hidden*	Structure	ColumnHidden*
22	ColumnsVisible *Columns F-H are made visible*	Structure	ColumnVisible*
23	DefinedNamesInserted *3 defined names are inserted*	Model	DefinedNameInsert*
24	DefinedNamesChanged *3 defined names are changed*	Model	DefinedNameChange*
25	DefinedNamesRenamed *3 defined names are renamed*	Model	DefinedNameRename*
26	DefinedNamesDeleted *3 defined names are deleted*	Model	DefinedNameDelete*
27	CellRangeInserted *Cell range A4:B10 is inserted*	Data, Model	ValueInsert*, FormulaInsert*
28	CellRangeChanged *Cell range A4:B10 is changed*	Data, Model	ValueChange*, FormulaChange*
29	CellRangeDeleted *Cell range A4:B10 is deleted*	Data, Model	ValueDelete*, FormulaDelete*
30	SingleCellInserted *Cell A1 is inserted*	Data, Model	ValueInsert*, FormulaInsert*
31	SingleCellChanged *Cell B2 is changed*	Data, Model	ValueChange*, FormulaChange*
32	SingleCellDeleted *Cell D4 is deleted*	Data, Model	ValueDelete*, FormulaDelete*

Two type of changes are aggregated: changes on workbook level and changes on worksheet level. In the next subsections we explain our aggregation method. We will often refer to the numbered change types defined in Table 4.1, and the numbered difference types defined in Table 4.3.

4.9.1 Aggregation of workbook changes

The first step of aggregation is applied on the list of all global changes (change type 1-12). This complete list is grouped by changes of the same type. For instance a group with all inserted worksheets is created, a group with all worksheets made hidden is created, and so on. Then, we inspect each group and transform the grouped changes into differences. Three types of differences are constructed: worksheet differences, worksheet state differences and defined name differences.

For the worksheet changes, we consider every change individually. For a worksheet insert and worksheet delete, we retrieve all corresponding cell changes in that worksheet. These are either value/formula inserts and value/formula deletes. If no cell changes are found, an empty worksheet inserted or an empty worksheet deleted difference is created. If some cell changes are found, a worksheet inserted or a worksheet deleted difference is created. This is an important distinction, because it determines the difference category. Empty inserted or deleted worksheets are considered to be structure differences, and inserted or deleted worksheets with content are considered to be data or model differences. The actual category depends on the underlying cell changes. If a worksheet only contains value inserts or deletes, it is a data difference. If a worksheet only contains formula inserts or deletes, it is a model difference. If it contains both, it is both a data and model difference. We conclude that for every worksheet change (change type 1-2), a new worksheet difference (difference type 1-4) it created.

For the worksheet state changes, we simply create one new difference for every group. It is informative enough for a high-level difference to describe how many worksheets are made hidden, made visible, made protected, and so on. All corresponding changes in the current group are added to the difference. That allows to expand the details for every worksheet state change, for example to inspect all the worksheets that are renamed. The details about the actual changes are still stored in the change objects, for example the old and new worksheet name. Hence, every difference object allows zooming in on all the underlying details that happened in the described scenario. We apply the same method to defined name changes: all changes of the same type are grouped into one difference. We conclude that for every group of worksheet state changes (change type 3-8), one worksheet state difference (difference type 5-10) is created. And for every group of defined name changes (change type 9-12), one defined name difference (difference type 23-26) is created.

4.9.2 Aggregation of worksheet changes

The second step of aggregation is applied on the list of all row and column changes (change type 13-20) and cell changes (change type 21-28), for every matching worksheet. The goal is to create differences that combine as many as possible changes logically belonging to each other. It is intuitive to group the changes based on their location in the worksheet. Therefore, we create three type of differences in a worksheet: consecutive blocks of rows and columns, consecutive blocks of cells, and individual cells.

First, the row and column changes are grouped by the same change type, just like we did in the previous step. Then, for every group we calculate the consecutive blocks of these changes. For example, the grouping of inserted rows 1, 2, 3, 10, and 11 is transformed to inserted rows 1-3 and inserted rows 10-11. Dependent on the change type, we apply an approach to create a new row/column difference for every consecutive block. For every block of rows/columns that are inserted or deleted, we retrieve all cell changes in the corresponding rows/columns. If no cell changes are found, an 'empty rows inserted' or an 'empty columns inserted' difference is created, which is a structure difference. If some cell changes are found, a 'rows inserted' or a 'columns inserted' difference is created. The corresponding cell changes are added to the difference. The category of the difference (either data or model, or both) depends on the type of cell changes. In addition, we mark the cell changes that are added to the row/column difference as 'visited'. This action is important because we track for all cell changes in a worksheet whether they are already aggregated or not. Changes that are included in a difference are marked as aggregated. Furthermore, for every block of rows/columns that are made hidden or made visible, the corresponding difference containing all the rows/columns is created. We conclude that for every block of inserted or deleted row/columns (change type 13-16), a new row/column difference (difference type 11-18) is created. And for every block of row/column state changes (change type 17-20), a new row/column state difference (difference type 19-22) is created.

Next, we continue on the list of cell changes in the worksheet that are not yet visited. We devised a clustering algorithm for all the remaining cell changes in a worksheet, based on their cell location. The algorithm is listed in Algorithm 6. The aim is that all cell changes lying next to each other based on direct neighbors (top, bottom, left, right) are grouped together. The result of the algorithm are consecutive blocks of cell changes, where unchanged cell locations divide the different blocks. For example, changes in A1, A2, B1, B2, F4, F5, F6 and H100 will be clustered into three blocks: {A1, A2, B1, B2}, {F4, F5, F6} and {H100}. After running the clustering algorithm, we determine for every resulting block the change count. For a single change, a single cell inserted/changed/deleted difference is created. For multiple changes, a cell range inserted/changed/deleted difference is created. We conclude that for every block of remaining cell changes (change type 21-28), a new cell difference (difference type 27-32) is created. Note that we combine the cell changes based on their edit action, which is either an insert, change or delete. All the cell differences are therefore classified as insertion, change or deletion. The category of the cell difference (either data or model, or both) again depends on the type of the underlying cell changes.

The final outcome of the comparison pipeline is a list of global workbook differences and lists of worksheet differences. All the differences are segregated into 32 difference types, three categories (data/model/structure), and three actions (insert/change/delete). In combination with the descriptive text, an overview with all the constructed differences will provide a useful summary of all modifications between two spreadsheet files.

Algorithm 6 Cluster cell changes into consecutive blocks

Input: A list of all non-aggregated cell changes C in a worksheet
Output: A collection of change groups, clustered by their cell location

```
 1: function ClusterCellChanges($C$)
 2:     Let $D$ be a new dictionary, indexed by a key of two numbers $(i, j)$
 3:     Let $Q$ be a new queue
 4:     Let $result$ be a new list
 5:     for all $c \in C$ do
 6:         $D[(c.\text{Row}, c.\text{Column})] \leftarrow c$
 7:     end for
 8:     while $D \neq \emptyset$ do
 9:         $B \leftarrow$ new Block()                          ▷ create empty block with changes
10:         $Q$.Enqueue(some item from $D$)          ▷ start with a random, non-visited change
11:         while $Q \neq \emptyset$ do
12:             $c \leftarrow Q$.Dequeue()
13:             $i \leftarrow c.\text{Row}$
14:             $j \leftarrow c.\text{Column}$
15:             $B$.Add($c$)                                ▷ expand block with this change
16:             $D[(i, j)] \leftarrow \epsilon$                  ▷ mark this change as visited
17:             if $D[(i-1, j)] \neq \epsilon$ then            ▷ check top neighbor cell
18:                 $Q$.Enqueue($D[(i-1, j)]$)
19:             end if
20:             if $D[(i+1, j)] \neq \epsilon$ then         ▷ check bottom neighbor cell
21:                 $Q$.Enqueue($D[(i+1, j)]$)
22:             end if
23:             if $D[(i, j-1)] \neq \epsilon$ then            ▷ check left neighbor cell
24:                 $Q$.Enqueue($D[(i, j-1)]$)
25:             end if
26:             if $D[(i, j+1)] \neq \epsilon$ then           ▷ check right neighbor cell
27:                 $Q$.Enqueue($D[(i, j+1)]$)
28:             end if
29:         end while
30:         $result \leftarrow result \cup \{B\}$
31:     end while
32:     return $result$
33: end function
```

4.10 Reflection

In this chapter we proposed nine solutions to the spreadsheet comparison problem. We look back at the requirements defined in Section 3.3. The next Table 4.4 summarizes which requirements are realized by our solutions, including the corresponding section.

Table 4.4: Status of the requirements, realized by solutions in this chapter

	Requirement	Status	Section
1	Overview	✓ By design	
2	Validation	✓ Solution	4.1
3	Completeness	~ Solution (partly)	4.5-4.8
4	Error Resolving	✓ By design	
5	Visualization	× Not in this research	
6	Evolution	× Not in this research	
7	Change Propagation	✓ Solution	4.2
8	Performance	✓ Solution	4.3
9	2D Alignment	✓ Solution	4.4
10	Grouping of Data	✓ Solution	4.9
11	Detect Movements	× Not in this research	

Some requirements are satisfied by the design choices we made. For example, we defined the differences such that they provide insight at multiple levels of detail, addressing Requirement 1. Furthermore, we built the comparison approach such that it accepts all types of spreadsheet files (including the ones with errors), addressing Requirement 4 .

Requirement 3 is solved to a certain degree. The provided comparison approach does not yet compare the full set of possible edit operations on spreadsheets. We have seen that it will require a lot of effort to create a complete spreadsheet diff tool. However, the described comparison methods in Sections 4.5 to 4.8 are complete for the spreadsheet aspects they apply to. While not all spreadsheet parts are compared, we claim that the comparison of worksheets, defined names, rows and columns, and cells is complete. In addition, the categories 'data' and 'model' are complete. When no data changes are detected, it is guaranteed that no values have been updated. And when no model changes are detected, it is guaranteed that no formulas have been updated.

At the end of this chapter, we conclude that the main research work of this thesis solved many problems regarding spreadsheet versioning. Both four user needs and four technical challenges are resolved. After the theoretical description and motivation of our solutions, we want to present the application of these concepts in practice. Therefore, we describe our prototype tool in the next chapter.

5 Tool: CompareXL

Beyond the theoretical solutions explained in the previous chapter, we engineered a full spreadsheet compare tool: CompareXL. This prototype is a stand-alone tool and has been implemented simultaneously during the research work. CompareXL is able to analyze and compare two spreadsheet files, and output a comparison result. Our implementation is two-fold: it consists of a front-end (the user interface) and a back-end (the comparison engine), described in Sections 5.1 and 5.2. Here we show screenshots of our final result, explain the design choices and mention the technical details. Moreover, we spent a lot of time optimizing the implementation. The goal was to develop a finished prototype that can be used particularly by end-users 'in the wild', not only being a simple proof-of-concept. Many enhancements have been made, based on testing and improving the comparison on self-created and real spreadsheets, described in Section 5.3. We conclude with a reflection on the development process in Section 5.4.

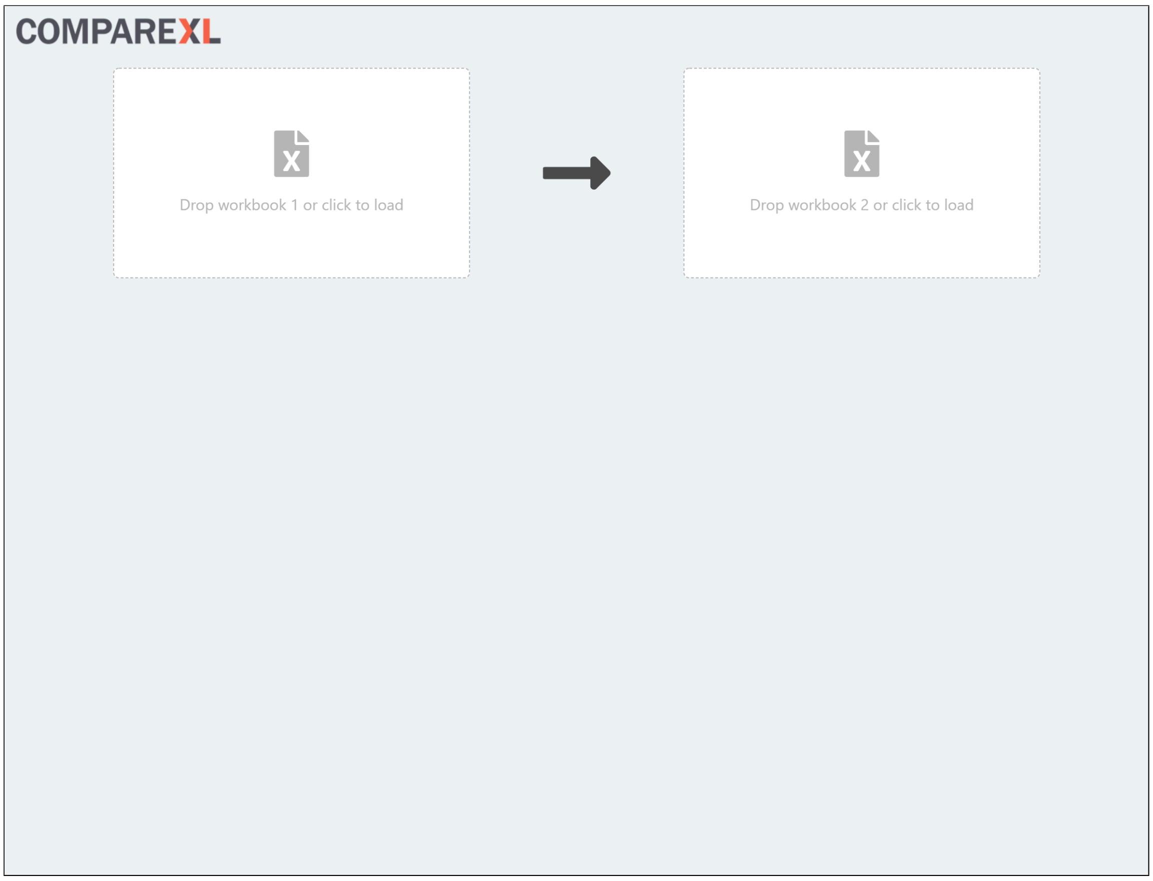
At the time of the publication of this thesis, Infotron is exploring options for the commercial exploitation and further development of CompareXL.

5.1 User Interface

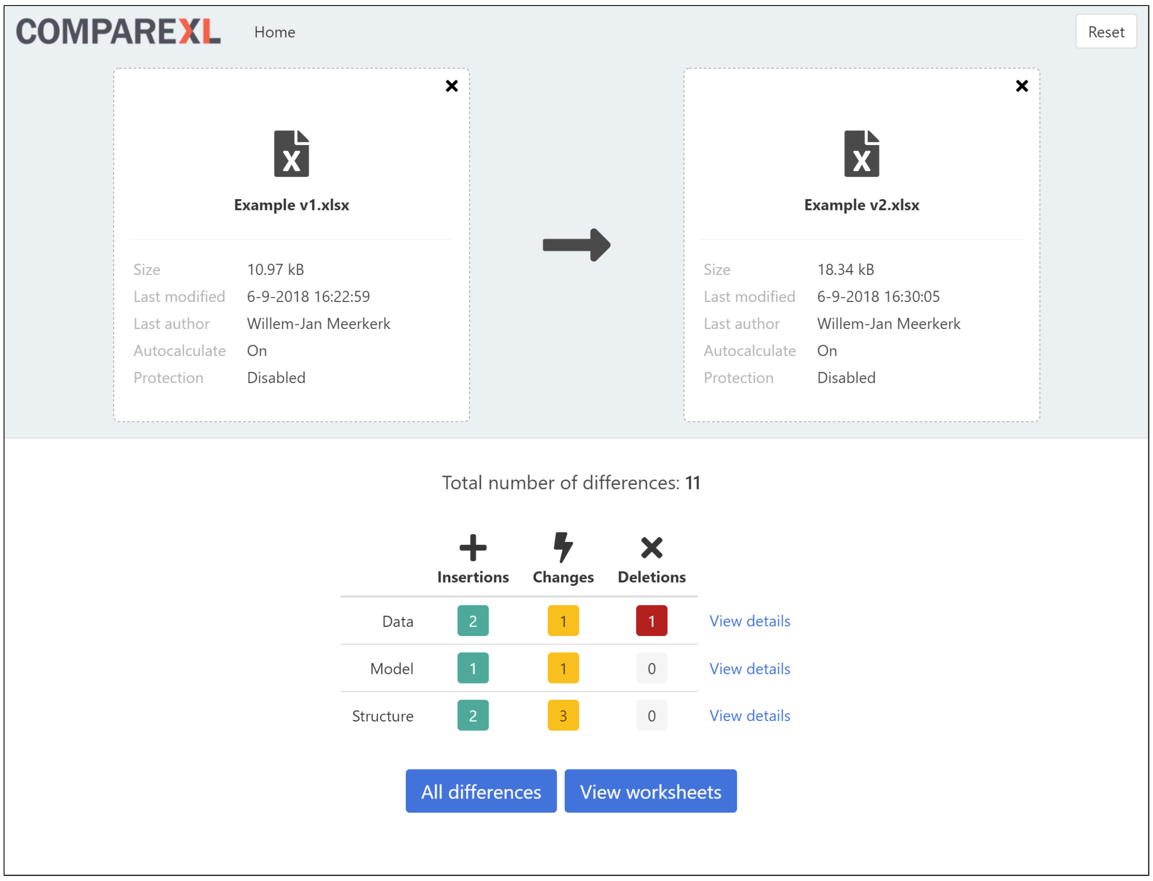
More than 20 years ago, Shneiderman [25] introduced a useful starting point for designing advanced graphical user interfaces. It is called the Visual Information Seeking Mantra: *Overview first, zoom and filter, then details-on-demand.* In our project, we followed this principle and we were inspired by it for visualizing the comparison result in the user interface. Our final result is in line with this proven guideline.

The user interface is designed with the user needs of Excel end-users in mind. We show the comparison result on multiple levels of abstraction, for two reasons. First, users use the compare tool with different goals: some ask high-level questions (e.g. are any formulas changed?) and some ask detailed questions (e.g. what exactly has been changed in Sheet1?). Secondly, the comparison of different spreadsheets can give many different outcomes (e.g. small and large amount of differences). The user interface is built so that it is helpful in all these scenarios. Both a high-level overview (summary) and detailed views (list of expandable differences) are available. Moreover, the user interface supports users to obtain insight in their spreadsheet versions.

On the next pages, we demonstrate CompareXL with screenshots of the full user interface.


COMPAREXL
Drop workbook 1 or click to load
Drop workbook 2 or click to load

(a) Initial screen



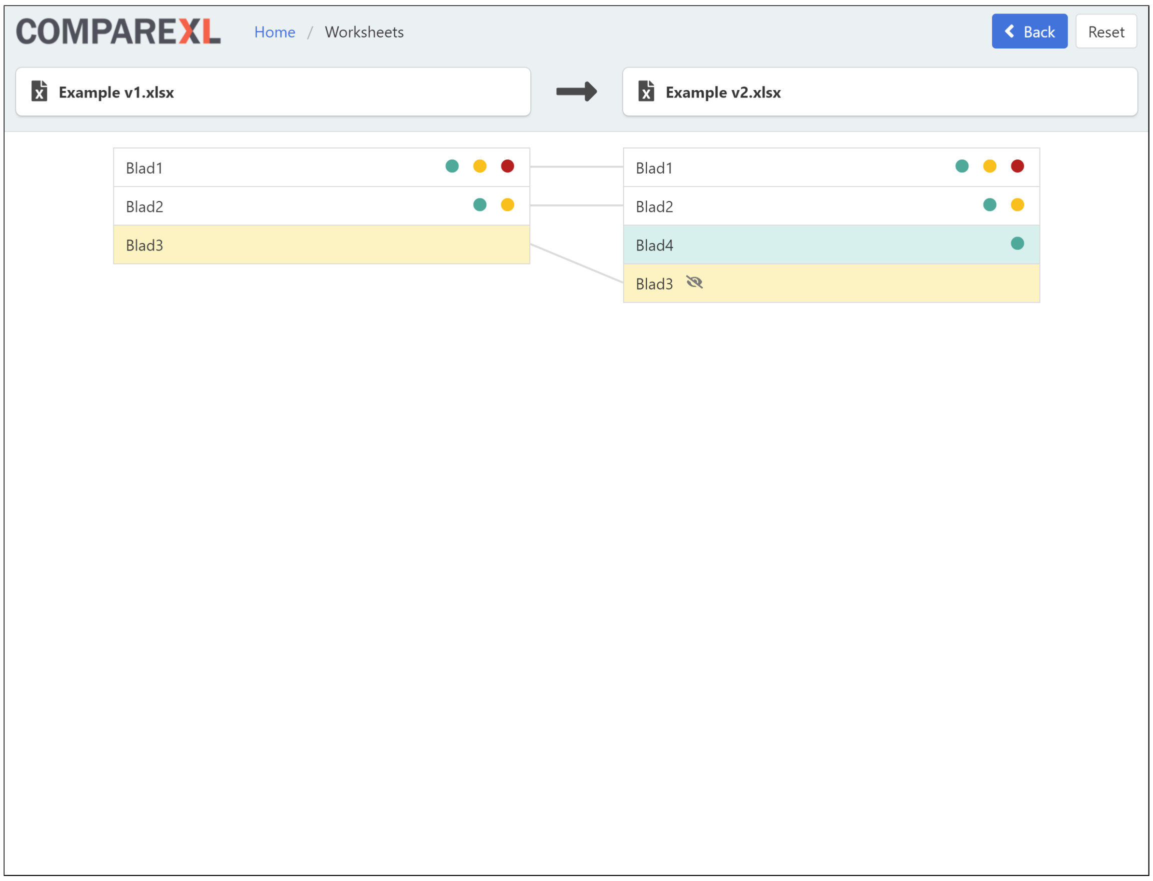
	Example v1.xlsx	Example v2.xlsx
Size	10.97 kB	18.34 kB
Last modified	6-9-2018 16:22:59	6-9-2018 16:30:05
Last author	Willem-Jan Meerkerk	Willem-Jan Meerkerk
Autocalculate	On	On
Protection	Disabled	Disabled

Total number of differences: **11**

	Insertions	Changes	Deletions	
Data	2	1	1	View details
Model	1	1	0	View details
Structure	2	3	0	View details

(b) Comparison result, shown after selecting two spreadsheet files

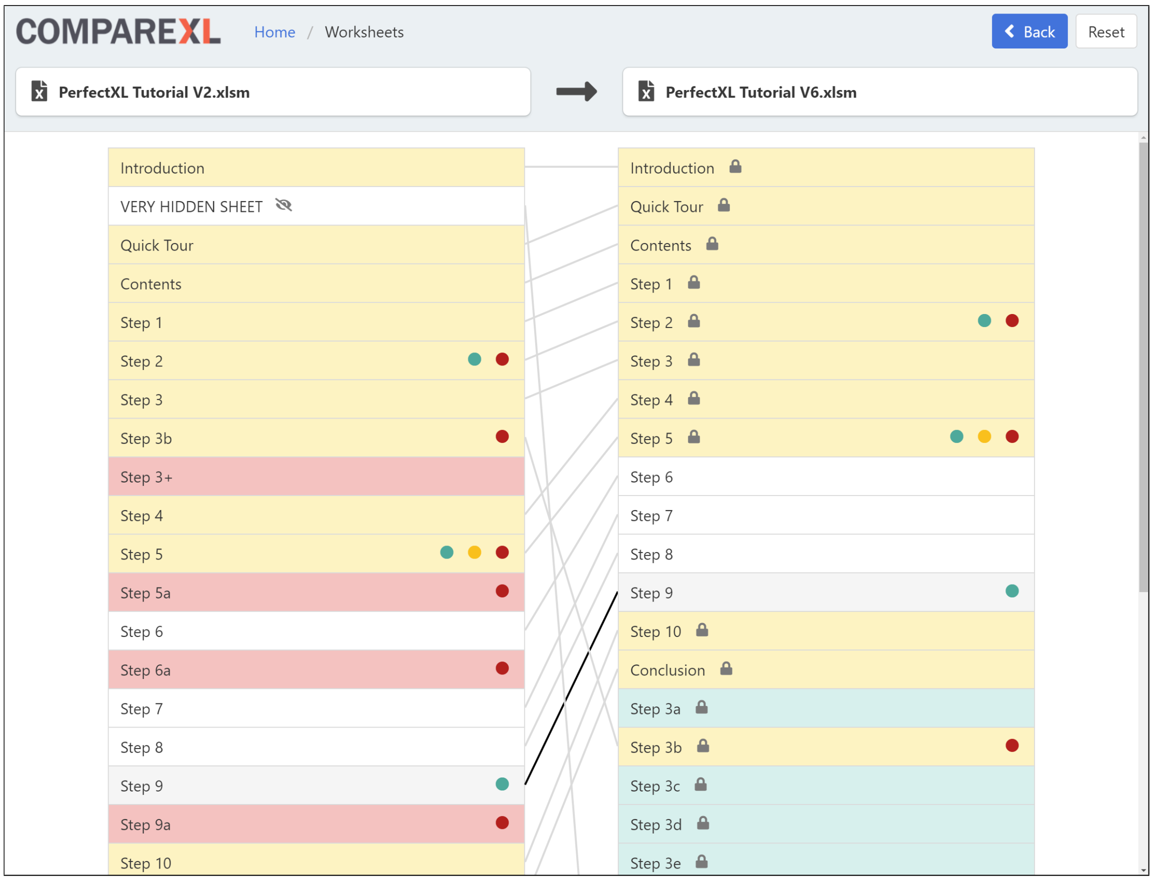
Figure 5.1: Screenshots of CompareXL (1)



Example v1.xlsx → Example v2.xlsx

Example v1.xlsx	Example v2.xlsx
Blad1 ● ● ●	Blad1 ● ● ●
Blad2 ● ●	Blad2 ● ●
Blad3	Blad4 ●
	Blad3

(a) Worksheet overview

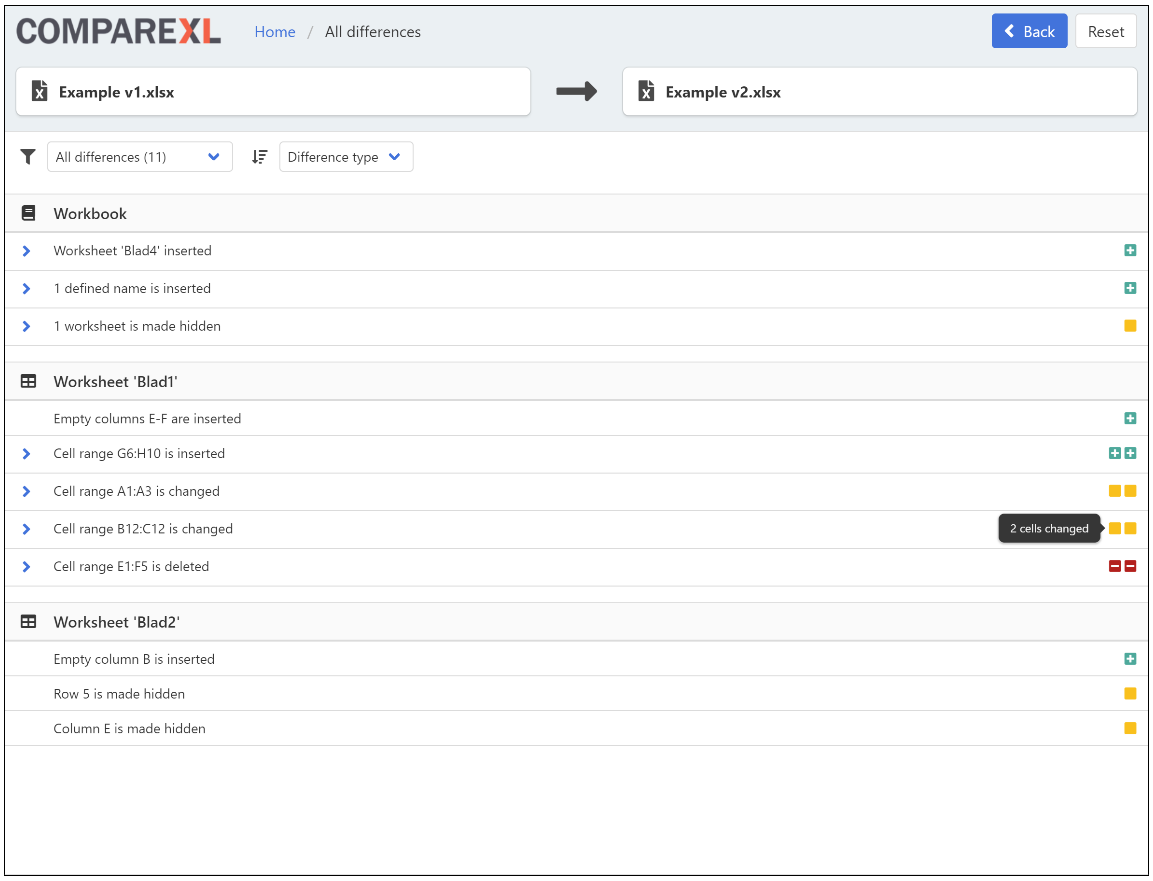


PerfectXL Tutorial V2.xlsm → PerfectXL Tutorial V6.xlsm

PerfectXL Tutorial V2.xlsm	PerfectXL Tutorial V6.xlsm
Introduction	Introduction
VERY HIDDEN SHEET	Quick Tour
Quick Tour	Contents
Contents	Step 1
Step 1	Step 2 ● ●
Step 2 ● ●	Step 3
Step 3	Step 4
Step 3b ●	Step 5 ● ● ●
Step 3+	Step 6
Step 4	Step 7
Step 5 ● ● ●	Step 8
Step 5a ●	Step 9 ●
Step 6	Step 10
Step 6a ●	Conclusion
Step 7	Step 3a
Step 8	Step 3b ●
Step 9 ●	Step 3c
Step 9a ●	Step 3d
Step 10	Step 3e

(b) More advanced worksheet overview, for another comparison

Figure 5.2: Screenshots of CompareXL (2)



Workbook

Worksheet 'Blad4' inserted	⊞
1 defined name is inserted	⊞
1 worksheet is made hidden	🟨

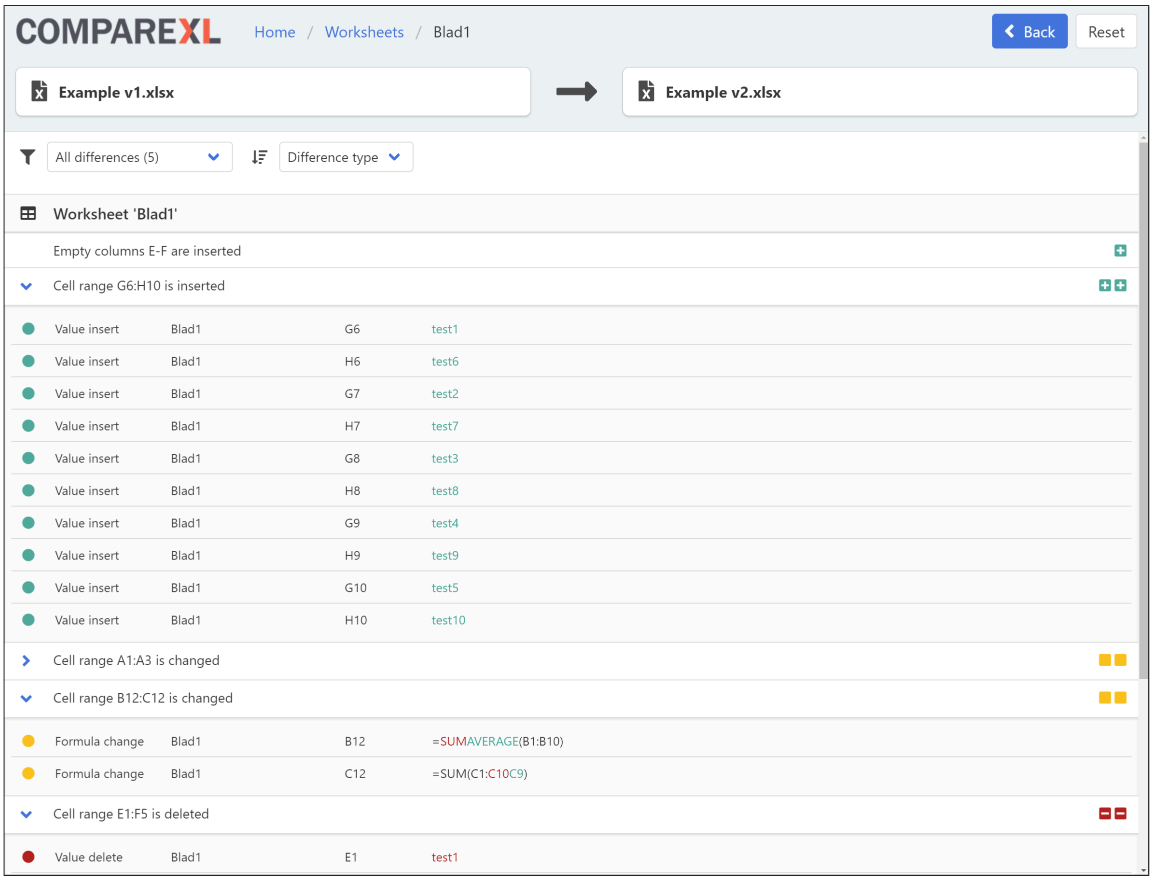
Worksheet 'Blad1'

Empty columns E-F are inserted	⊞
Cell range G6:H10 is inserted	⊞⊞
Cell range A1:A3 is changed	🟨🟨
Cell range B12:C12 is changed	2 cells changed 🟨🟨
Cell range E1:F5 is deleted	⊟⊟

Worksheet 'Blad2'

Empty column B is inserted	⊞
Row 5 is made hidden	🟨
Column E is made hidden	🟨

(a) Overview of all differences



Worksheet 'Blad1'

Empty columns E-F are inserted ⊞

Cell range G6:H10 is inserted ⊞⊞

●	Value insert	Blad1	G6	test1
●	Value insert	Blad1	H6	test6
●	Value insert	Blad1	G7	test2
●	Value insert	Blad1	H7	test7
●	Value insert	Blad1	G8	test3
●	Value insert	Blad1	H8	test8
●	Value insert	Blad1	G9	test4
●	Value insert	Blad1	H9	test9
●	Value insert	Blad1	G10	test5
●	Value insert	Blad1	H10	test10

Cell range A1:A3 is changed 🟨🟨

Cell range B12:C12 is changed 🟨🟨

●	Formula change	Blad1	B12	=SUMAVERAGE(B1:B10)
●	Formula change	Blad1	C12	=SUM(C1:C10C9)

Cell range E1:F5 is deleted ⊟⊟

●	Value delete	Blad1	E1	test1

(b) Overview of differences in a worksheet, with expanded details

Figure 5.3: Screenshots of CompareXL (3)

Figure 5.1 shows the basic usage of CompareXL. A user can select or drop two spreadsheet files. The comparison is immediately started when two files have been selected. First, a summary of the comparison result is shown. The total number of differences is displayed, as well as a simple 3×3 matrix that gives an overview of the result. This matrix summarizes horizontally on category (data, model and structure), and vertically on type: insertions (green), changes (yellow) and deletions (red). The matrix is constructed such that is easy to draw conclusions, valid on the whole comparison result. For instance, it can be easily seen if only formulas modifications are made: then only numbers in the second row are expected. Furthermore, it can be easily identified if only insertions are made: then only numbers in the first column are expected.

There are three ways to view more details about the comparison result. Firstly, there is a button to go to the list of all differences. Secondly, there is a button to go the overview of all worksheets in the two files. Finally, the rows in the 3×3 matrix are clickable. That allows a user to zoom in on the data, model and structure differences. This is nothing more than navigating to the list of all differences, filtered on the corresponding category.

Figure 5.2 shows the worksheet overview of CompareXL. Two lists of worksheets are shown: all worksheets in the first file and in the second file. Matching worksheets have a connection in between, that is highlighted when a user hovers over a certain worksheet. Unique worksheets, both in the first and second file, are shown without a connection. The background color of a worksheet indicates modifications on the *state* of the worksheet. For instance, a worksheet is inserted (green), deleted (red), made hidden (yellow), and so on. The bullets on the right of the worksheet indicate modifications *inside* the worksheet. For instance, values or formulas in a worksheet can be inserted (green bullet), changed (yellow bullet) or deleted (red bullet). Hence, the insertion of an empty worksheet (only green background) is distinguished from the insertion of a worksheet with content (green background and green bullet).

This worksheet overview is constructed with two goals. First, it should provide a direct insight in the matching worksheets of both files. For spreadsheets with many worksheets the worksheets correspondence is often unclear. The connections easily identify the same worksheets, even if worksheets are renamed or reordered (see the example in Figure 5.2b). Secondly, it should be straightforward to know in which worksheets differences have been detected. The bullets directly indicate that either insertions, changes or deletions are detected in a certain worksheet. It is guaranteed that there are no modifications in worksheets without a bullet. Consider for example a spreadsheet that contains 100 worksheets, and in only one worksheet some cells are changed. Then this is immediately clear in the worksheet overview: one worksheet will have a yellow bullet and the others not.

Figure 5.3 shows the list of differences in CompareXL. Two views are available: an overview of all the differences, and an overview of differences per worksheet. In the complete overview, shown after clicking on the button ‘all differences’ on the home page, we summarize all differences on workbook level and then enumerate all differences per modified worksheet. In the overview per worksheet, shown after clicking on a worksheet in the worksheet overview, we summarize only the differences in the selected worksheet. Hence, depending on the goals of a user, both a complete or a zoomed-in version of the differences is available. The complete list will be helpful if the number of differences is relatively small. For large numbers of differences, it will be easier to select a worksheet and inspect the differences in that worksheet.

The list of differences corresponds to the aggregated differences outputted by the comparison pipeline. As explained in Section 4.9, each difference can contain several underlying changes. For example, an inserted cell range contains the inserted values or inserted formulas. Each difference with underlying changes can be expanded, as shown in the Figure 5.3b. Then all details about the actual changes can be seen. The list of changes contains several columns, depending on the type of change. For value changes and formula changes, a combined inline diff between the old and new value or formula is computed. That is done to show a compact and clear representation of the change. For example, when only a small part of a large formula is changed, it is not that helpful to display the complete old and new formula. With the inline diff, the unchanged part of the formula is shown only once (in black), together with the deleted parts (in red) and the inserted parts (in green).

For a convenient inspection of the differences, a filter and sort option is added. Every difference list can show (1) all differences, or only (2) data differences, (3) model differences, or (4) structure differences. For instance, a user can easily find all formula updates in a certain worksheet using the filter option on 'model differences'. The sort option allows sorting on (i) difference type, (ii) rows and then columns, or (iii) columns and then rows. By default the sorting is set on difference type, ordering the differences first on insertions, then on changes and then on deletions. By changing the sorting to rows/columns, the differences are ordered based on their cell location. This sort option follows the natural order of cells in the worksheet itself, from top to bottom. It allows a user to inspect the differences in the same order how he would scroll through the worksheet. The sort option on columns/rows is added equivalently.

CompareXL is implemented as a stand-alone tool using the Electron framework[1]. This framework allows to build cross platform desktop apps with JavaScript, HTML, and CSS. CompareXL runs hereby independently, operates alongside Microsoft Excel, comes with an easy installer and is ready to support multiple platforms (Windows, Linux, macOS) in the future. For the development of the user interface inside the Electron application, we used the Vue.js framework[2]. The combination of a desktop application with an integrated web interface was the most flexible approach for our goals. Both frameworks provided a modern infrastructure, supporting the development process of the user interface. The final prototype of CompareXL is a modern, scalable and well-organized stand-alone application.

[1] https://electronjs.org/

[2] https://vuejs.org/

5.2 Comparison Engine

The back-end of CompareXL is written in C# and is developed on the .NET Core framework. Its software architecture is divided in several projects. Each project has number of namespaces, each with their own purpose. An overview of the architecture of CompareXL is shown in Figure 5.4.

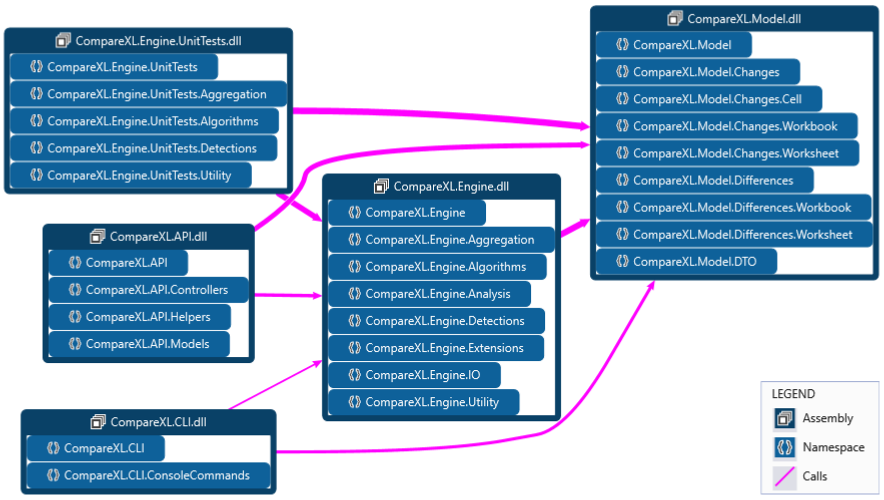

CompareXL.Engine.UnitTests.dll
CompareXL.Engine.UnitTests
CompareXL.Engine.UnitTests.Aggregation
CompareXL.Engine.UnitTests.Algorithms
CompareXL.Engine.UnitTests.Detections
CompareXL.Engine.UnitTests.Utility
CompareXL.Model.dll
CompareXL.Model
CompareXL.Model.Changes
CompareXL.Model.Changes.Cell
CompareXL.Model.Changes.Workbook
CompareXL.Model.Changes.Worksheet
CompareXL.Model.Differences
CompareXL.Model.Differences.Workbook
CompareXL.Model.Differences.Worksheet
CompareXL.Model.DTO
CompareXL.Engine.dll
CompareXL.Engine
CompareXL.Engine.Aggregation
CompareXL.Engine.Algorithms
CompareXL.Engine.Analysis
CompareXL.Engine.Detections
CompareXL.Engine.Extensions
CompareXL.Engine.IO
CompareXL.Engine.Utility
CompareXL.API.dll
CompareXL.API
CompareXL.API.Controllers
CompareXL.API.Helpers
CompareXL.API.Models
CompareXL.CLI.dll
CompareXL.CLI
CompareXL.CLI.ConsoleCommands
LEGEND
Assembly
Namespace
Calls

Figure 5.4: Overview of the projects and namespaces of CompareXL

In the main project, `CompareXL.Engine`, the complete comparison pipeline is implemented. This project contains the analysis, change detection and change aggregation steps, including all underlying algorithms, business logic, utility methods and I/O operations. In addition, there are several projects that interface with the comparison engine. First of all, individual components of the comparison approach are tested in the `CompareXL.Engine.UnitTests` project. Next, we have a command line interface, `CompareXL.CLI`, where the comparison is executed on two given files and the comparison result is printed in the console output. Finally, we have an application programming interface, `CompareXL.API`. This project contains a small web server, Kestrel[3], that is the default web server implementation in .NET Core. This web server provides one API endpoint to compare two given spreadsheet files. The API endpoint accepts two paths to the files, and returns the comparison result in JSON format.

The project that is not shown here is `CompareXL.Desktop`, which is the stand-alone Electron application. This project is developed outside the .NET Core environment, and contains a Node.js project for building the user interface. Upon compilation of the user interface, we package the `CompareXL.API` executable within the stand-alone application. When the CompareXL application is started, we start the web server as a separate process in background. The web interface communicates for every comparison with the back-end via an HTTP request. The comparison result in JSON format is then stored and processed in the user interface. The advantage of this approach is that we can develop the front-end and back-end completely independent of each other.

[3] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel

The implementation of the comparison engine relies on the `CompareXL.Model` project, where the model classes are defined. For all 28 changes (described in Section 4.1) and all 32 aggregated differences (described in Section 4.9) corresponding C# classes are created. The type of the class describes type of operation that occurred (e.g. a value change), the fields of the class describe the content of the operation (e.g. value '42' at cell A10 in worksheet 'Sheet1'). We defined a com-

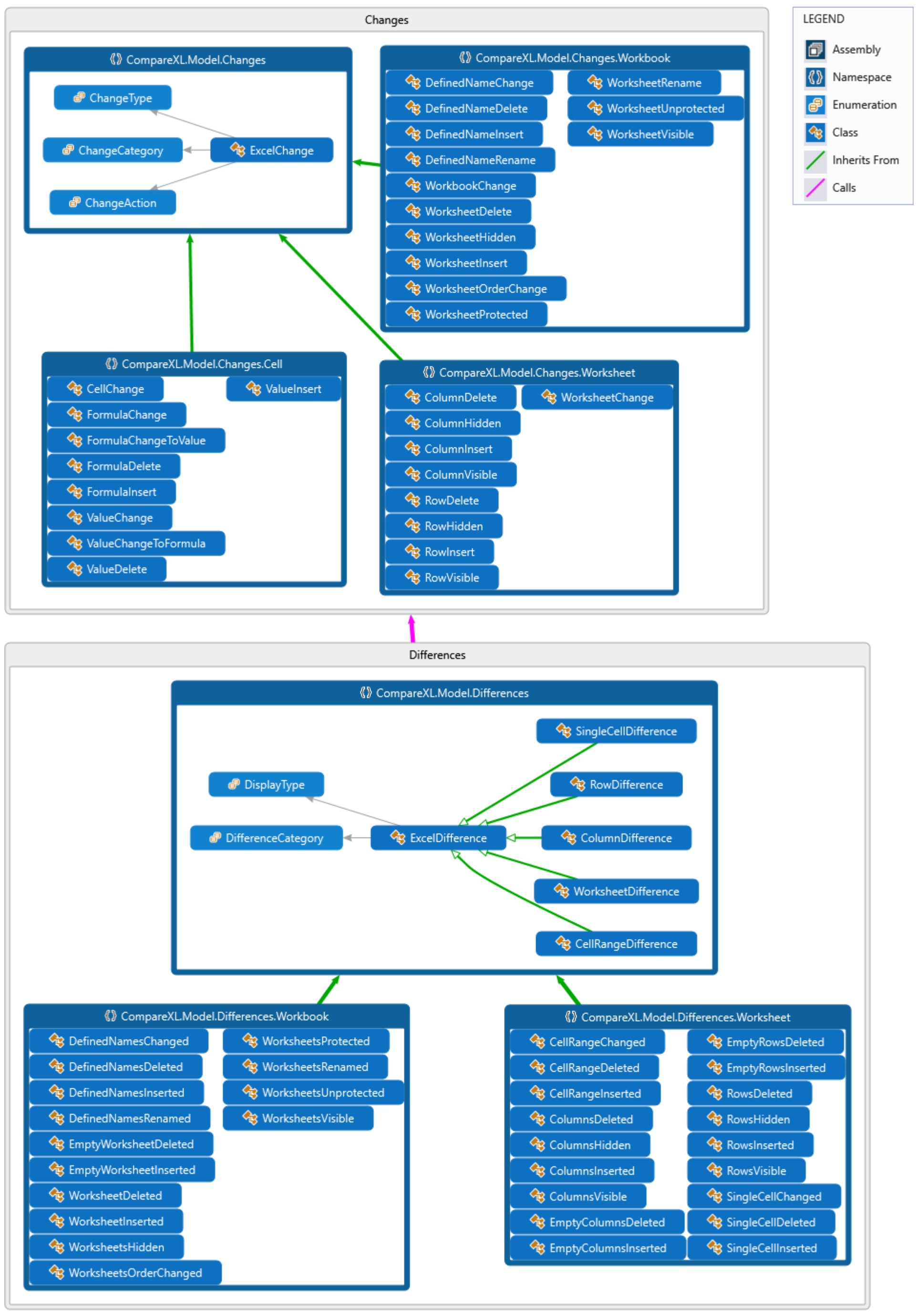


Figure 5.5: Overview of the model classes of CompareXL

plete class hierarchy, to represent changes and differences at multiple levels in a spreadsheet. For instance, the `ExcelDifference` classes are divided in workbook, worksheet, row, column, cell range and cell differences. An overview of all the model classes is shown in Figure 5.5.

The main class in our comparison engine is `ExcelComparator`, which is responsible for the comparison of two spreadsheet files. Figure 5.6 illustrates this class, with all the other implementation classes in the `CompareXL.Engine` namespace. The constructor reads two spreadsheet files into memory. Then the 'Compare' method starts the actual comparison and executes all three phases of the comparison pipeline. First, the workbook and worksheet structure is analyzed, then changes are detected in the workbook and worksheets, and finally the comparison result is constructed where the changes are aggregated to differences.

It is impossible to describe all the implementation details in this thesis. However, Figures 5.4 to 5.6 provide a good summary of the architecture of CompareXL.

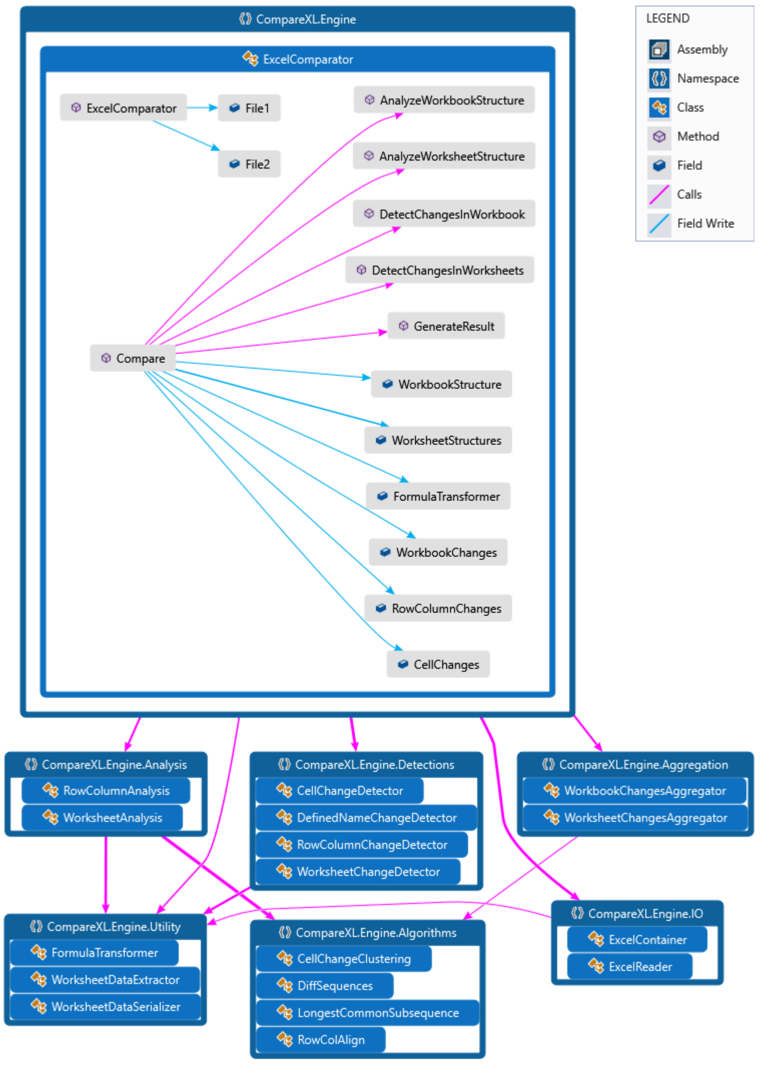


Figure 5.6: Overview of the implementation classes of CompareXL

5.3 Performance Optimizations

An important phase in the development of our prototype is the time we spent on performance optimizations. We noticed that there is a lot of diversity in spreadsheet files, and the ultimate goal is run the comparison of all possible spreadsheet files as fast as possible. In this section we describe three improvements. To test the effect of the enhancements, we ran two test suites on the different versions of the code. The first was the existing unit test suite, consisting of 120 unit tests at that time. The second was a benchmark test suite, consisting of 9 tests on real-world, large, spreadsheet files (file size between 300 kB and 35 MB). The spreadsheets were distinctive, some with large amounts of data, some with many complex formulas. We were interested in optimizations that improve the overall run time, while keeping all unit tests completing successfully. The results of the are summarized in Table 5.1.

Table 5.1: Run time of two test suites after different optimizations

Code version	Unit test suite	Benchmark suite
1. Initial codebase	6.489 s	245.4 s
2. Switched to single library: EPPlus	5.983 s	183.6 s
3. Implemented parallelization	5.853 s	132.5 s
4. Optimized RowColAlign algorithm	3.463 s	98.1 s

Initially, we decided to use two third-party libraries for analyzing the spreadsheet files: Gembox Spreadsheet[4] and OpenXML SDK[5]. The motivation was that Gembox is fast in reading a spreadsheet file and creating an object model of the spreadsheet. Such a high-level object model is created for fast looping over all allocated cells in a worksheet, or retrieving the cell data at a specific index. But in addition, we needed some specific entries from the raw XML representation of the spreadsheet, not available in the model of the Gembox library. Therefore, we added the OpenXML SDK to read out the required information, for instance the internal worksheet UID. It turned out that it is inefficient to use two libraries, because every spreadsheet needs to be opened twice. This is costly for large spreadsheets, every spreadsheet should be uncompressed twice and multiple I/O operations are performed. We noticed that the OpenXML SDK had another problem: it locks the spreadsheet file, causing issues while the spreadsheet is still opened in Excel or when a new comparison is started. In the end, we decided to switch to one single library for analyzing the spreadsheet files: EPPlus[6]. EPPlus has the advantage that it both creates an efficient object model and allows direct access to the underlying raw XML of the spreadsheet, exactly what was needed for our implementation. Furthermore, the code of EPPlus is open-source, licensed under LGPL. EPPlus is still used as spreadsheet libary in the final version of CompareXL. Switching to a single spreadsheet library resulted in a 8% run time improvement for the unit test suite and a 25% improvement on the benchmark suite. Moreover, our source code is better organized with this improvement.

Another improvement is parallelization of parts of the code. For efficient parallelization, it

[4] https://www.gemboxsoftware.com/spreadsheet

[5] https://docs.microsoft.com/en-us/office/open-xml/open-xml-sdk

[6] https://github.com/JanKallman/EPPlus

is important to find independent tasks that are not that small, and not that large to execute. We made two parts of our implementation run in parallel. First, the reading of the two spreadsheet files at the start of the comparison. Secondly, the structure analysis that is executed for each worksheet, with the single loop over all allocated cells to calculate the hashes and to perform the 2D alignment. Both tasks are logical units to parallelize, because they operate on separated parts of the spreadsheet. The reading of the two spreadsheet files is completely independent, the structure analysis of worksheets only needs to store its result in a tread-safe list. The implementation of parallelization resulted in a 2% run time improvement for the unit test suite and a 28% improvement on the benchmark suite.

The last enhancement is a particular improvement in the RowColAlign algorithm, related to the implementation of row and column segments. After a lot of debugging, we found that the RowColAlign was comparing a redundant number of cells for row and column segments of comparable size. With the introduction of row and column segments, the RowColAlign algorithm was often executed to align a single column with another single column, or to align a single row with another single row. These are trivial cases and an early outcome can be given, preventing a costly LCS calculation for large rows or columns. For one sample spreadsheet we were debugging, this enhancement reduced the comparison time from 15 minutes to 30 seconds. In general, this optimization of the RowColAlign algorithm resulted in a 41% run time improvement for the unit test suite and a 26% improvement on the benchmark suite.

We found it enlightening to test the prototype on real-world spreadsheets during the development process. It gave us more insight in the bottlenecks of our comparison, and it resulted in a more efficient implementation.

5.4 Development Process

We want to conclude this chapter with a brief summary of the engineering process. From the first day we organized and planned this graduation project using a large Trello board, keeping track of all todo items, meetings, and so on. We prepared 12 official meetings, and performed 6 interviews and 6 user tests. In total we created 265 Trello cards, including 149 archived, finished todo items. The Trello board really helped to streamline the graduation project and to be always up to date on the project status.

After an initial stage of research, we made a plan for the implementation of the compare tool using 11 milestones. Each milestone was given a title and had a list of features to be implemented. During the project, we were able to implement the following milestones:

- Milestone 1: Detect value changes
- Milestone 2: Detect worksheet changes
- Milestone 3: Detect row and column changes
- Milestone 4: Detect formula changes
- Milestone 5: Create user interface
- Milestone 6: Aggregate changes

The following milestones are left for future work:

- Milestone 7: Visualization of changes
- Milestone 8: Apply pattern recognition
- Milestone 9: Detect movements
- Milestone 10: Detect display changes
- Milestone 11: Compare VBA code

The code was organized on a private GitHub repository. In the period of 9 months, we created 340 commits, with 39,064 additions and 17,781 deletions (lines of code). We released 7 versions of the prototype tool: 4 versions as command-line interface, 3 versions as complete stand-alone application.

6
Evaluation

In the previous chapters, we proposed our solutions to the spreadsheet comparison problem and we showed our prototype tool operating in practice. Now, we want to validate our solutions and measure the quality of our work. We performed different experiments to evaluate our proposed solution. In this chapter we present their methods and results.

First we state our evaluation goals. We want to validate the comparison approach on five different aspects: correctness, completeness, user needs, stability, and performance. The corresponding evaluation questions are listed in Table 6.1.

Table 6.1: List of evaluation questions

	Evaluation Question	Experiment
Q1	Does the comparison algorithm produce a correct comparison result?	I, II, V
Q2	Does the comparison algorithm produce a complete comparison result?	V
Q3	Do the comparison approach and interaction design address user needs?	V
Q4	Can the comparison algorithm run on many different spreadsheets without problems?	III
Q5	Does the comparison algorithm scale to large and complex spreadsheets?	IV

We came up with five different experiments to measure these aspects quantitatively or qualitatively. We describe each experiment in Sections 6.1 to 6.5, by explaining their method and summarizing their results. Then we answer the evaluation questions in Section 6.6, where we discuss and critically analyze our results. The conclusions follow in the next chapter.

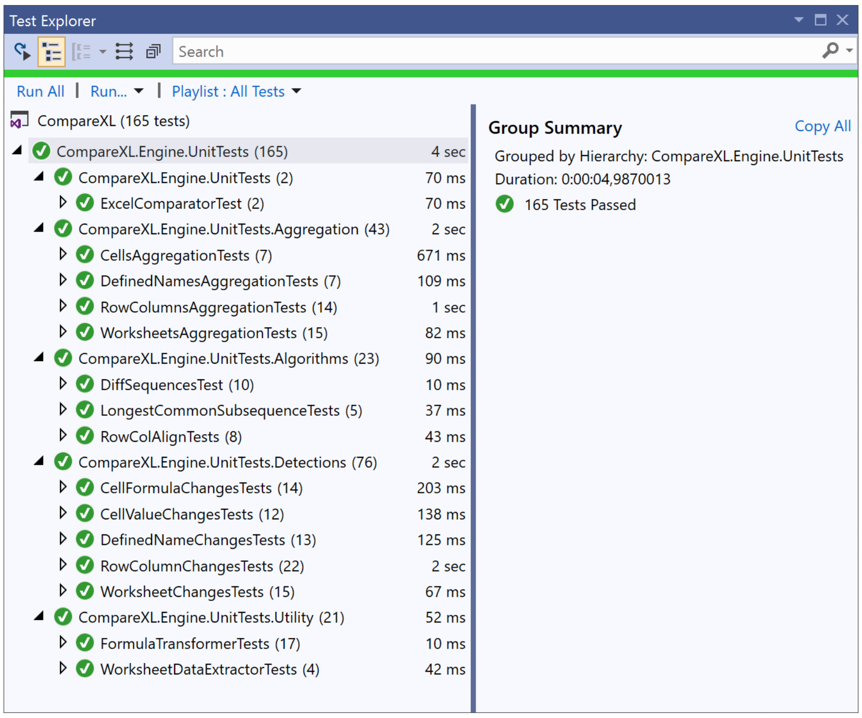
All experiments are executed on a Dell XPS 15 9550 laptop, with an Intel Core i7-6700HQ CPU and 16 GB of memory.

6.1 Experiment I: Unit Tests

During the development of the prototype, we created a large set of unit tests to validate the correctness of our implementation. This experiment is not performed at the end of the implementation phase, but continuously during development. The idea is that all unit tests should pass, so that it is guaranteed that the code gives the outcome as defined in our unit tests. As all tests are successful in the final prototype, this experiment will not find more weaknesses in our final implementation. However, many incorrect results were fixed during implementation. This set of unit tests makes sure that the comparison result always gives the exact result as defined in the tests. Therefore, this serves to validate the correctness of our implementation continuously during many development iterations.

Our test method was as follows. Every milestone in our development process had a list of features to be implemented. For instance, the second milestone about worksheet changes contained features like worksheet insert, worksheet rename, and so on. For every feature we created two small spreadsheets that test this individual feature. For example, one spreadsheet with one empty worksheet, and another spreadsheet with two empty worksheets. The unit test validates that one worksheet insert is detected, and no other changes. The unit tests were created before the features were implemented. That forced us to structurally think about normal cases, corner cases and their expected outcome. For some cases the expected outcome was not even clear, and more discussion or research was required. Besides the small unit tests, we created a number of larger functional tests. These tests validate the full outcome of a comparison result on self-modified spreadsheets or real-world spreadsheets with two versions. The functional tests validate the correctness of a complete comparison result, and contain in general many assert statements in one test method.

The result are shown in Figure 6.1. Our test suite contains a total number of 165 unit tests, and runs in less than 5 seconds. The code coverage is 97.7% of all code in the `CompareXL.Engine` namespace. Inspection of the test code shows that a total number of 1181 assert statements are written in the unit tests. 100% of the unit tests pass.



```
CompareXL (165 tests)
◢ ✓ CompareXL.Engine.UnitTests (165)                          4 sec
   ◢ ✓ CompareXL.Engine.UnitTests (2)                          70 ms
      ▷ ✓ ExcelComparatorTest (2)                              70 ms
   ◢ ✓ CompareXL.Engine.UnitTests.Aggregation (43)              2 sec
      ▷ ✓ CellsAggregationTests (7)                           671 ms
      ▷ ✓ DefinedNamesAggregationTests (7)                    109 ms
      ▷ ✓ RowColumnsAggregationTests (14)                       1 sec
      ▷ ✓ WorksheetsAggregationTests (15)                      82 ms
   ◢ ✓ CompareXL.Engine.UnitTests.Algorithms (23)              90 ms
      ▷ ✓ DiffSequencesTest (10)                               10 ms
      ▷ ✓ LongestCommonSubsequenceTests (5)                    37 ms
      ▷ ✓ RowColAlignTests (8)                                 43 ms
   ◢ ✓ CompareXL.Engine.UnitTests.Detections (76)               2 sec
      ▷ ✓ CellFormulaChangesTests (14)                        203 ms
      ▷ ✓ CellValueChangesTests (12)                          138 ms
      ▷ ✓ DefinedNameChangesTests (13)                        125 ms
      ▷ ✓ RowColumnChangesTests (22)                            2 sec
      ▷ ✓ WorksheetChangesTests (15)                           67 ms
   ◢ ✓ CompareXL.Engine.UnitTests.Utility (21)                 52 ms
      ▷ ✓ FormulaTransformerTests (17)                         10 ms
      ▷ ✓ WorksheetDataExtractorTests (4)                      42 ms

Group Summary
Grouped by Hierarchy: CompareXL.Engine.UnitTests
Duration: 0:00:04,9870013
✓ 165 Tests Passed
```

Figure 6.1: Results of the unit tests of CompareXL

6.2 Experiment II: Correctness Test

After finishing our prototype with a definitive version, we want to test the compare tool on new, unseen spreadsheet files. To automatically test the correctness, we need a test set with labeled data. That is, two versions of the same spreadsheet with a list of modifications that are made. It is difficult to obtain that data, and much work to produce itself. We in fact developed such a set as part of the tool development process (which served as a 'training set'), but now we are searching for a new data set.

Fortunately, the Enron Error Corpus [22] provides us with a new dataset containing two versions of spreadsheets with labeled modifications. In this research, a method is presented to automatically find errors in real-world spreadsheets from the Enron Corpus. The original (faulty) version and a modified (corrected) version are made available for different spreadsheets. It has been documented which cells have been changed between the two versions. Different types of modifications have been made, e.g. the correction of wrong calculations, wrong formulas, wrong labels, and so on (see the first column of Table 6.2). In total 31 labeled modifications are listed, in 28 different spreadsheet files.

Our test method was as follows. For every modification in the dataset, the two corresponding spreadsheet versions are compared using the user interface of the compare tool. We manually inspect the comparison result and check if the listed modification is recognized and presented correctly. This includes the correct worksheet and cell location of the modification, and the formula or value update if applicable. If everything is correct, we count this as a correct detection of the compare tool. Otherwise, we report the modification as not detected and write down the reason for it.

The results are shown in Table 6.2. In total 29 of the 31 modifications are identified correctly by our compare tool. Two modifications are not detected, both related to small bugs in the implementation. In the case of Error Nr 22 a value change is reported instead of a formula change. This is an incorrect outcome of the comparison, caused by an error in the formula comparison. Error Nr 26 is not detected because the comparison does not check external references. This bug can be fixed by considering the external references during the formula comparison. All the other types errors are identified successfully. We conclude that 94% of the modifications in the Enron Error Corpus are correctly detected by the comparison tool.

Table 6.2: Results of correctness test on the Enron Errors Corpus dataset

Modification	Detected	Not detected	Correctness
File reference not working	0	1	0%
Fixed value	1	0	100%
Range error	12	0	100%
Wrong calculation	9	1	90%
Wrong formula	1	0	100%
Wrong labels	1	0	100%
Wrong reference	3	0	100%
Wrong values	2	0	100%
Total	29	2	94%

6.3 Experiment III: Stability Test

Another important quality aspect is how stable the compare tool is on new, unseen spreadsheet files. If the compare tool will be used 'in the wild', we want to make sure that it performs successful on many different files, not only the spreadsheets that we tested the compare tool on. We already concluded that there is a lot of variety in spreadsheet files. For instance, spreadsheets vary in file size, complexity, Excel version, encoding, and can contain many specific features like embedded objects, encryption, macros, and so on. It might be that specific circumstances raise errors in the implementation of the comparison method. We want to test if an actual comparison result is produced on many different spreadsheets. In this test, we do not study the actual content of the comparison result. Instead, we only focus on if a comparison result is produced, without any unexpected exceptions raised or timeouts.

For this experiment, we used three different datasets containing versioned spreadsheet files: VEUSES, VFUSE and VEnron2. All three datasets are produced by SpreadCluster [19], an automatic clustering algorithm that generates spreadsheet evolution groups (versioned spreadsheets of the same source). VEUSES is built on EUSES [26], a spreadsheet corpus with spreadsheets from a variety of sources. It contains 177 evolution groups and 363 spreadsheets. VFUSE is built on FUSE [27], a spreadsheet corpus that contains a diverse set of spreadsheets from the public web. It contains 188 evolution groups and 1,143 spreadsheets. VEnron2 is built on the Enron email archive [28], with spreadsheets extracted from email messages of the Enron corporation. It contains 1,609 evolution groups and 12,254 spreadsheets. We decided to use VEnron2 [19] instead of VEnron1.1 [18], as this set contains more versioned spreadsheets and the grouping is more accurate. With this selection of three datasets, we tried to test the applicability of our approach to a set of spreadsheets exhibiting a wide variation.

Our test method was as follows. All datasets are extracted, resulting in a separate folder for every evolution group. An automatic test script inspects all folders and selects the first two spreadsheets for every evolution group. The comparison is executed on these two versions, and the result is recorded. We mark a test case as successful if a comparison result is returned without any exception. A test case is erroneous if an exception is raised during the comparison. We additionally record the type of error and the error message. If a test case runs for more than 5 minutes, we end the comparison and report a timeout.

The results are shown in Table 6.3. In total 1868 of the 1973 comparisons finished successfully. In total 97 comparisons ended with an error, and 8 comparisons ended with a timeout. We conclude that the approach works well on a wide variety of spreadsheets: given a random pair of evolved sheets, in 95% of the cases the tool can successfully produce a comparison.

Table 6.3: Results of stability test on VEUSES, VFUSE and VEnron2 datasets

Dataset	*N*	Successful	Error	Timeout
VEUSES	177	167 (94.4%)	10 (5.6%)	0 (0.0%)
VFUSE	188	186 (98.9%)	2 (1.1%)	0 (0.0%)
VEnron2	1608	1515 (94.2%)	85 (5.3%)	8 (0.5%)
Total	1973	1868 (94.7%)	97 (4.9%)	8 (0.4%)

A summary of all errors is listed in Table 6.4. Some errors are specific shortcomings in third-party libraries. For instance, all test cases are .xls files, and are internally converted to .xlsx by the Gembox library. Apparently some support is missing for encrypted files. Furthermore, some errors are specific to content and encoding of the tested spreadsheet files. This is reflected in the errors with invalid characters and buffer errors, caused by very exotic spreadsheet files. Finally, some errors identify real bugs in the implementation of the comparison method. Examples are 'null reference' exceptions and 'argument out of range' exceptions.

A first inspection of these errors suggests that these are not fundamental problems in the comparison algorithm, but engineering issues in the current implementation. Further implementation effort is needed to resolve these issues in future versions of the tool.

Table 6.4: Error details and their count in the stability test

Errror Type	Count
GemBox.Spreadsheet.SpreadsheetException	
Current version of GemBox.Spreadsheet can't read encrypted workbooks.	3
System.ArgumentException	
Name contains invalid characters	32
The worksheet name cannot contain any of the following characters: …	11
We don't support specified formula token: Empty	10
System.ArgumentOutOfRangeException	
Index was out of range. Must be non-negative and less than …	4
Zoom must be in range from 10 to 400.	1
System.IO.EndOfStreamException	
Unable to read beyond the end of the stream.	19
System.IO.IOException	
An attempt was made to move the position before the beginning of the stream.	1
System.NullReferenceException	
Object reference not set to an instance of an object.	14
System.Xml.XmlException	
'□', hexadecimal value 0x0B, is an invalid character. Line 1, position 7049.	1
'□', hexadecimal value 0x1B, is an invalid character. Line 1, position 9109.	1
Total	97

6.4 Experiment IV: Performance Test

It is also necessary to test the performance of the compare tool on new, unseen spreadsheet files. Some solutions are especially constructed such that the comparison method is efficient. Furthermore, during the implementation we incorporated several optimizations, for instance parallelization. With this experiment, we want to evaluate how fast the comparison is for different types of spreadsheets. Furthermore, we want to explore the limits of the compare tool by running it on very large and complex spreadsheets.

The test data we use for this experiment is a self composed dataset of 16 large and complex real-world spreadsheet files. The source of these files is initially the industrial spreadsheet archive of Infotron. Furthermore, we asked interviewed users if they would like to share versions of spreadsheets they use on their own or at their company. All 16 spreadsheets have two versions, vary in file size, and are real-world spreadsheets. Some of them are confidential.

Our test method was as follows. An automated test script loads the two versions of each spreadsheet in the dataset. It is recorded how long it takes to load the two files. The test script first runs an analysis to obtain statistics about the spreadsheets under test. The number of worksheets and the number of allocated cells, value cells and formula cells are obtained. We report the statistics and file size of only the first of the two spreadsheets versions, as both versions are comparable. Then the test script starts the comparison on both versions. If the comparison is successful, the amount of differences and changes is recorded, and the time it took to execute the full comparison. We have built in a time limit of one hour. If the comparison is not finished after that period of time, the comparison is ended and a timeout is reported.

The results are shown in Table 6.5. It took around 4 hours to complete this experiment. After all, 13 of the 16 comparisons (81%) finished within the time limits. We see a lot of diversity in the spreadsheets. For instance, the largest spreadsheet only contains 3 worksheets, another one has more than 200 worksheets. Some spreadsheets only have value cells, others have more formula cells, but most large spreadsheets have a combination of both. The performance really depends on the test case, the table shows that some large comparisons are done within seconds, some take more than a minute. The cause of the timeouts is not related to the large files itself. We compared the first version of file 13, 15 and 16 with the same first version of these files. All comparisons on the same version of these files is finished within one minute, with of course 0 differences and 0 changes as outcome. Therefore, we conclude that the timeout of the 3 spreadsheets is caused by the large amount of differences between the two versions. Finally, we did not have any memory issues during this challenging performance test.

Table 6.5: Results of the performance test on large and complex spreadsheet files

File	Size (MB)	Sheets	Allocated Cells	Formula Cells	Value Cells	T_{load} (s)	T_{comp} (s)	T_{total} (s)	Differences	Changes
1	1.11	22	175,025	96,137	38,852	0.98	4.61	5.59	48	552
2	3.49	27	163,971	17,931	20,740	1.10	21.22	22.32	65	5146
3	3.72	12	4,147	4	996	0.21	0.94	1.15	55	1557
4	4.22	22	208,104	86,338	35,206	1.82	6.13	7.95	35	61565
5	4.23	33	638,277	345,369	13,436	2.72	11.53	14.25	285	12072
6	4.86	46	730,679	277,433	133,320	3.71	6.30	10.01	7	13650
7	8.02	20	2,755	208	190	0.10	0.09	0.19	106	515876
8	10.05	9	56,143	0	19,947	0.12	0.35	0.47	45	25012
9	10.78	202	1,108,880	890,111	60,474	6.01	10.54	16.55	10	3593
10	12.93	11	1,083,524	1,081,540	982	7.82	15.52	23.34	1	10
11	18.25	114	3,016,593	527,998	1,522,562	10.57	571.49	582.05	8	168
12	21.48	119	4,570,060	10,703	2,677,354	18.17	34.25	52.41	325	2894664
13	27.51	113	2,678,908	1,016,752	787,012	12.08	timeout	timeout	-	-
14	35.19	36	28,527,494	211,741	1,084	2.69	164.42	167.11	415	1831
15	45.45	118	2,906,043	1,221,760	862,748	18.32	timeout	timeout	-	-
16	81.88	3	13,207,338	4,238	9,583,028	33.99	timeout	timeout	-	-

Besides the extreme test cases, we are interested in the average run time of our comparison method. Is our compare tool fast on spreadsheets with typical size and complexity? To answer this question we revisit the spreadsheets we used for our stability test in Section 6.3, focusing on the performance of the compare tool. In addition to the outcome, we also recorded the run time for each of the 1868 successfully executed comparisons. The results are summarized in Figure 6.2. The run time reported in this histogram is the total time of loading and comparing the two spreadsheets.

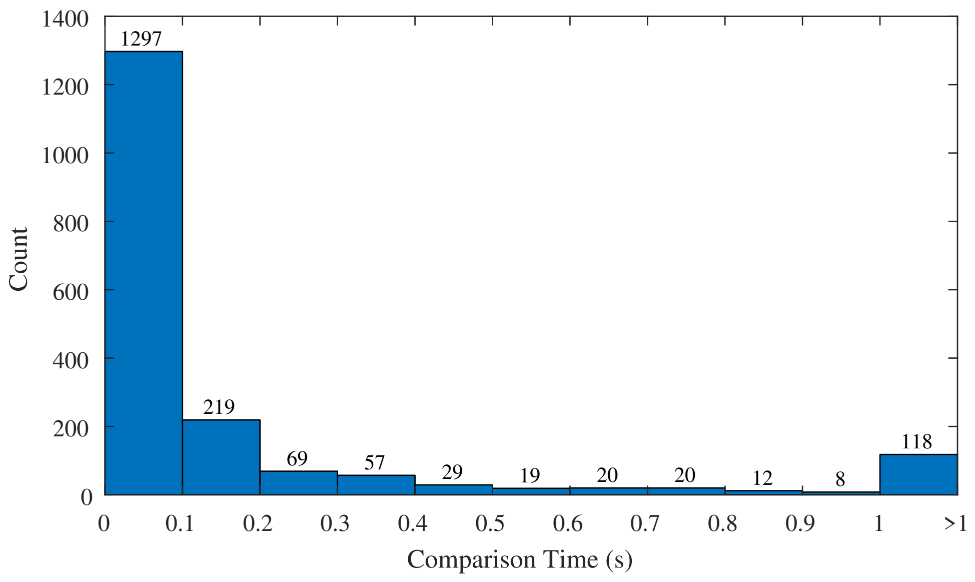


Figure 6.2: Histogram of the comparison time on VEUSES, VFUSE and VEnron2 datasets

In total 1297 of the 1868 spreadsheets are compared within 0.1 seconds. Only 118 of the 1868 comparisons take more than 1 seconds. There is a high variance in this last segment: 93 of these comparisons take 1–10 seconds, and 25 comparisons take 11–220 seconds. The complete distribution is highly skewed, most comparisons are very fast and some take a longer time. We conclude that our compare tool is able to compare 69.4% of the spreadsheets in the three datasets within 0–0.1 seconds, 24.3% within 0.1–1 seconds and 6.3% in more than 1 seconds.

Additionally, we recorded the file size of all test cases in the stability test. The average run time of all successful comparisons, categorized by their file size, is shown in Table 6.6. Many spreadsheets in these datasets have small file size (< 50 kB), and most of the spreadsheets have small or medium file size (< 500 kB). We see that on average the comparison method is performing fast on small and medium spreadsheets. For larger spreadsheets, the average run time is moderately fast. However, there is a higher variance: the comparison time really depends on the content, complexity and amount of modifications. We conclude that the file size has effect on the comparison time. The average performance of the compare tool is 1.06 seconds on all spreadsheets in the VEUSES, VFUSE and VEnron2 datasets.

Table 6.6: Summary of the comparison time per file size

File Size	Count	Average Time (s)	StdDev Time (s)
0 - 50 kB	1060	0.045	0.147
50 kB - 100 kB	308	0.120	0.390
100 kB - 500 kB	346	1.281	8.570
500 kB - 1 MB	72	9.181	32.601
1 MB - 10 MB	82	9.573	28.958
Total	1868	1.056	9.846

6.5 Experiment V: User Tests

In addition to the technical evaluation, an evaluation from the user perspective is also relevant. Some of our solutions are created to fit user needs, and our prototype tool with the complete user interface is designed especially for end-users. Therefore, at the end of the development process, we evaluated the final compare tool with six different Excel end-users. Three of them were also interviewed at the start of this research.

The goal is three-fold. First, we want to test how the prototype tool operates ‘in the wild’, in real-world usage scenarios. How end-users react on the user interface and how they interpret the comparison result is relevant. Secondly, we want to determine to what extent the comparison method meets correctness and completeness standards from end-users. Finally, we want to find out if the proposed solutions fit the actual user needs.

We followed the following method. The user test meetings were planned with an expected duration of 1–1.5 hours. One week before the user test, users were asked if they could save intermediate versions of spreadsheets they were currently working on. During the actual user test meeting, we began with a brief demonstration of the compare tool on some demo files. We gave users explanation about the comparison result and they inspected all the screens of the user interface. They were asked for their first reaction. After the introduction, we tested the prototype tool with the user in three steps.

First, the complete set of implemented differences was divided into 7 individual tasks (see Figure 6.3). Users were asked to test these tasks one by one, by creating modified spreadsheets versions in Excel themselves. After running a few comparisons for each task, users judged the implementation of each task on a scale of 1 (incorrect), 2 (sometimes correct) or 3 (always correct). This part of the user test focuses on the correctness of the comparison approach. After that, we showed a long list of all Excel features to the user. We asked if he or she could indicate which additional spreadsheet features should be implemented in order to use the compare tool in practice. The answers give us an idea of the completeness of our comparison approach.

Secondly, we let the user test the compare tool on real-world files they brought themselves. The users were given enough time (15–30 minutes) to explore the prototype tool themselves. During this period, we talked with the users and wrote down comments. After testing, the users were asked to judge the prototype tool on the following quality aspects:

1. **Overview**. The tool provides a useful high-level overview on what has changed
2. **Detail**. The tool reports enough details on what has changed
3. **Summary**. The tool is able to summarize large amounts of modifications
4. **Completeness**. The tool is able to compare all relevant aspects of my spreadsheet
5. **User-friendliness**. The tool is easy to work with
6. **Complexity**. The tool is able to handle large and complex files adequately
7. **Solution**. The tool fits my goals for comparing spreadsheets

The users gave each aspect a score on a scale of 1 (not at all), 2 (not so), 3 (somewhat), 4 (very) or 5 (extremely).

Finally, we concluded the user test with a reflection. We asked the strengths and weaknesses of the current prototype tool and asked where the user would use the current prototype tool for. We finished with a question if the user allows processing of the user test results in this thesis. Optionally, he or she could give permission to save the test data for follow-up research.

With all answers written down, the user test meeting was finished. In the next subsections we summarize the results.

6.5.1 Features

The results of the feature evaluation are shown in Figure 6.3.

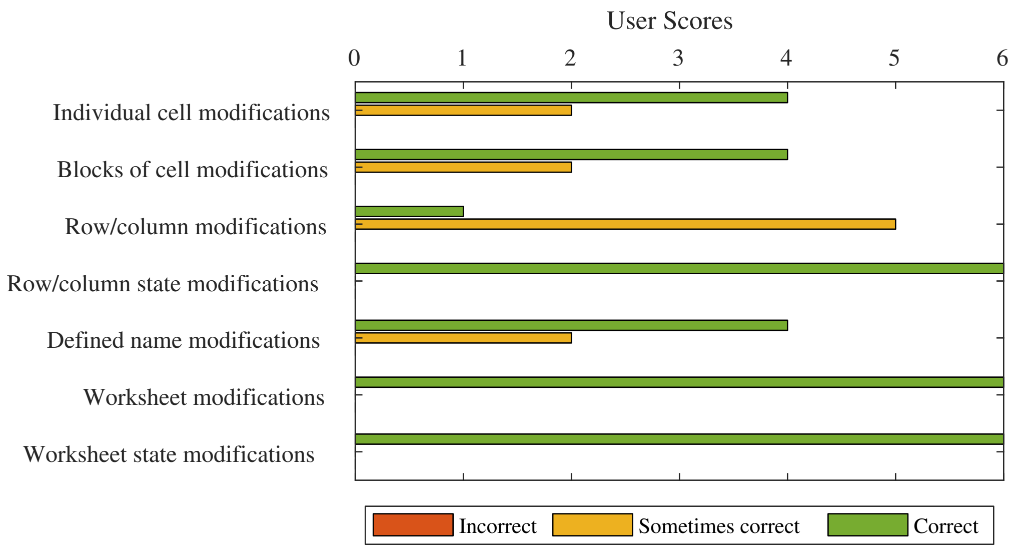


Figure 6.3: Summary of scores for the 7 implemented features

Two of the six users judged both individual and blocks of cell modifications as 'sometimes correct', because our comparison approach does not recognize an 'insert and shift' or 'delete and shift' action. This insert or delete action will result in many shifted cells to the right or to the bottom. When an empty cell is inserted in this way (user action), the result is that many cells are moved (propagated changes). This advanced kind of change propagation is difficult to detect, and therefore relevant future work. Our tool currently reports additional changes for the moved cells. Besides this specific case, all other normal cell modifications were correctly reported and aggregated.

Five of the six users judged row and column modifications as 'sometimes correct'. All incorrect results originate from test cases where combinations of row and column modifications were made, which we know are difficult to correctly detect. Individual changes to only rows and columns were always correct, and simple combinations also. Furthermore, the aggregation of underlying changes inside inserted/deleted rows and columns was considered intuitive and correct.

Two of the six users judged defined name modifications as 'sometimes correct'. This is caused by a simple bug which can easily be fixed. The aggregation of defined name inserts resulted in a defined name deleted difference, which is incorrect.

All other tasks were always correctly detected.

Table 6.7: List of additional features for the compare tool requested by users

Feature	Count	Feature	Count
Cell protection	6	Pictures	2
External data connections	6	Print options (header, footer, page breaks)	2
Pivot tables	6	Table filter state	2
Charts	5	Workbook protection	2
Conditional formatting	5	Cell comments	1
Data validation rules	5	Cell formula visibility	1
Global settings (e.g. auto calculate)	5	Row auto fit	1
Tables	5	Workbook styles	1
VBA code	5	Worksheet tab color	1
Cell number formatting	4	Cell style advanced (borders, alignment, etc.)	0
Cell merging	3	Cell style basic (background, color, font, etc.)	0
Table sort state	3	Cell style inline (font)	0
Cell hyperlinks	2	Column width	0
Embedded objects	2	Document properties	0
Ignored errors	2	Row height	0

We asked users which additional Excel features should be implemented in the comparison approach in order to be complete for the goals of the current end-user. The answers are summarized in Table 6.7. Users mentioned that the current prototype tool is already really helpful, and that the requirements they indicate are suggestions for improvements. Most users will happily use this version of the compare tool for their daily work. Only for one user the comparison is not complete enough yet to be a trusted tool. We can therefore conclude that the completeness requirement is relative, there is always room for extension of the supported features of the compare tool. Table 6.7 gives clear directions for further development.

6.5.2 Quality

After an extensive test of the prototype tool on real-world files, users concluded the user test with a quality evaluation based on their experience. All user scores are listed in Table 6.8, including the average score per criteria. The results of this quality evaluation are visualized in Figure 6.4.

Table 6.8: List of quality scores (1–5) given by in the user test

	Overview	Detail	Summary	Completeness	User-friendly	Complexity	Solution
User 1	4	4	5	4	5	5	4
User 2	5	5	5	4	4	5	5
User 3	4	5	5	4	5	5	5
User 4	4	4	5	3	4	5	3
User 5	4	5	5	3	5	4	4
User 6	5	5	4	4	4	4	4
Average	4.33	4.67	4.83	3.67	4.50	4.67	4.17

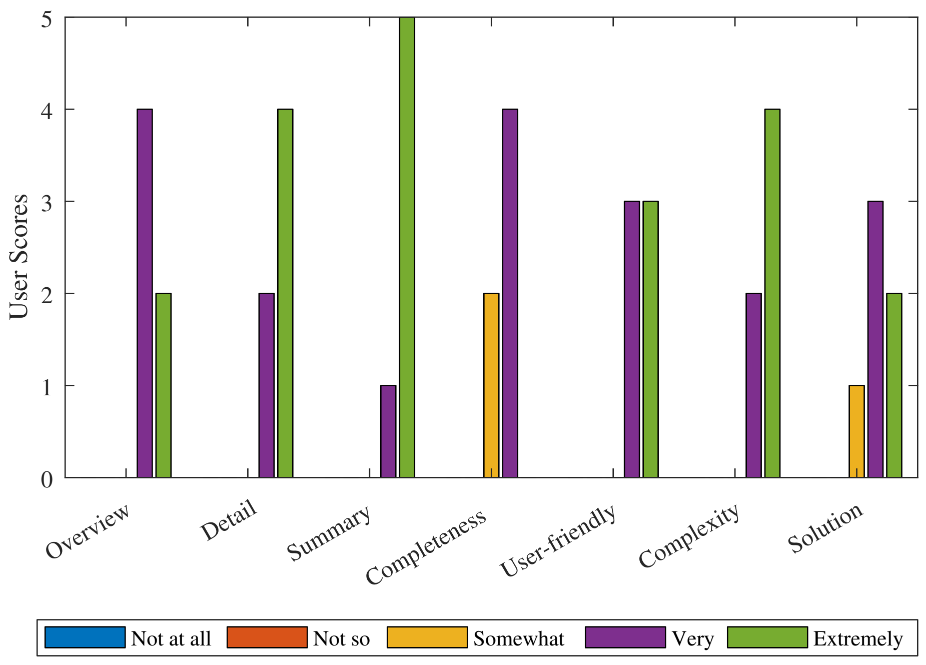


Figure 6.4: Summary of quality scores of the compare tool

The overall average score on all aspects is 4.4 of the maximum 5.0. No scores of 1 and 2 are given, and the score of 3 is also rare. The compare tool can improve the most on the 'completeness' aspect. User 4 (an Excel expert) will only use the tool regularly if he knows that it is complete (to a certain extent). His goals are clearly different than the goals of the other users, who gave a higher score on 'solution'. More explanation of the scores given by the users is summarized in the next subsection.

6.5.3 User Reflection

Users mentioned the following strengths of the current prototype tool (summarized):

- Clear and concise overview of changes.
- Impressive fast comparisons, very good performance.
- Well thought-out grouping of changes.
- Easy to zoom in on more details.
- Stand-alone application.
- Intuitive user interface.
- Smart inline diff of formulas.
- Helpful summary of comparison result.
- To be used by everyone.

Users mentioned the following weaknesses of the current prototype tool (summarized):

- No visual display of the changes.
- No export functionality.
- No integration with Excel.
- Implementation still contains some minor bugs.
- Not complete enough yet to be a trusted tool.
- Bad performance for some specific spreadsheets.
- Lack of overview for spreadsheets with enormous differences.

Users mentioned the following applications of the current prototype tool (summarized):

- General: instant overview of spreadsheet changes.
- Spreadsheet development: track changes.
- Spreadsheet validation: did I change what I wanted to change?
- Spreadsheet collaboration: find changes of other users.
- Spreadsheet accountability: create a report to justify all changes.
- Auditing of spreadsheets: find changes made since an earlier approved version.
- Debugging of spreadsheets: find the source of errors.

We conclude that the user test has given us more insight in the quality of the current prototype tool, both on the positive and negative aspects. The current version already meets the user needs of five of the six end-users. Further, we have gained more understanding how users operate with a compare tool.

Finally, many improvement suggestions have been identified. After the user tests, we did a status update of our Trello board. We created 14 new cards related to bugs in the prototype. 7 of them are simple bugs that are easy to fix. Furthermore, we created 21 new cards related to future improvements, 5 cards of cases that need further investigation and 11 cards of questions to think about. From an engineering perspective, that is a fruitful result of the user test.

6.6 Discussion

Now that we have all the results, what can we do with it? In this section we discuss the results of our research by reflecting on the evaluation questions. Furthermore, we provide a critical analysis of the threats to the validity of our results.

Q1. Does the comparison algorithm produce a correct comparison result?

The comparison method is well-tested with a unit test suite covering all the default and corner cases of all 28 implemented changes. Nonetheless, in practice, it turns out that sometimes incorrect results can be reported when combinations of row and column modifications are made. Our research identified this as one of the fundamental issues, and there is still room to improve this. Furthermore, our comparison approach does not detect 'insert and shift' and 'delete and shift' actions. This operation is rarely used, but will result in false positive changes. Not the user action is reported, but the propagated changes. An advanced method should be developed to detect this type of change. Finally, updates to external references in formulas are not yet detected by our approach, and there is a small bug in the aggregation of defined names. These issues can easily be fixed in a next version of the compare tool. In conclusion, three experiments validate that the outcome of our comparison is correct in almost all cases for spreadsheets comparison, except for the remarks up here.

Q2. Does the comparison algorithm produce a complete comparison result?

The comparison result is not complete in the sense that not all changes for every spreadsheet aspect are detected. As shown in Appendix B, the complete list of all possible spreadsheet modifications is huge. However, we claim that our comparison is complete in the sense that it detects all 28 defined changes listed in Table 4.1. The categories data and model are complete: we aim to detect all inserts, changes, and deletions of values and formulas. Furthermore, we aim to detect all structure changes for rows, columns, and worksheets. Because we completely implemented the comparison approach ourselves, this requirement is fulfilled by design. Our unit tests validate that we do not miss any changes for the sample spreadsheets.

Q3. Do the comparison approach and interaction design address user needs?

The enthusiastic responses in the user test prove that our compare tool meets the expectations of users. The quality scores given in the user tests are high. The tool can be further improved by supporting more spreadsheet features and by fine-tuning of the user interface. Because the goals of users for using a spreadsheet compare tool are different, it is difficult to validate if our approach suits all needs. For example, we do not provide a visualization of the changes, we have no integration with Excel and we do not offer export functionality. However, the current version of the compare tool does exactly what it is expected to do: compare two spreadsheet files and show an overview of the changes. All users in the user test mentioned multiple applications for which they would use the demonstrated version of the compare tool. In that sense, we conclude that the current prototype tool is already a supportive tool in many use cases regarding spreadsheet versioning. It solves the most important user need: to obtain an overview of all spreadsheet changes.

Q4. Can the comparison algorithm run on many different spreadsheets without problems?

The results of our stability test show that our approach is in general stable on new, unseen spreadsheet files: 94.7% of the spreadsheets are successfully compared. In 4.9% of the cases an error occurred, and in 0.8% a timeout occurred. We conclude that there is still room for improvement, ideally every spreadsheet should be compared without errors. The details about the errors give directions on how to improve the stability. But in general, we conclude that the compare tool is able to compare almost all sorts of new spreadsheets without problems.

Q5. Does the comparison algorithm scale to large and complex spreadsheets?

The average performance of our compare tool is 0.045 seconds for spreadsheets between 0 – 50 kB, 0.118 seconds for spreadsheets between 50 kB – 100 kB and 1.275 seconds for spreadsheets between 100 kB – 500 kB. For larger spreadsheets, the average compare time is more than 10 seconds. The interpretation of these results is tricky because there is a high variance in the running times. We conclude therefore that it really depends on the content of the spreadsheet how long the comparison will take. The file size is only a general indication for the comparison time. However, we can still conclude that in general our compare tool is extremely fast on small and medium spreadsheet files. Additionally, we have seen many examples of large spreadsheets that are rapidly processed by our comparison approach. Some files are loaded even faster than Excel itself does.

The results of our performance test show that our compare tool is able to process 81% of the most difficult spreadsheet files in our datasets. Spreadsheets with a tremendous amount of allocated cells or many worksheets are processed successfully. However, we cannot guarantee that all large and complex spreadsheets will be successfully compared. Our experiments showed three examples where the comparison was running infinitely. Again, the run time really depends on the content and the amount of modifications made in the spreadsheets. We conclude that in general the prototype tool is optimized to support very large and complex files. For some specific spreadsheets, no result can be produced; these cases need further investigation.

Threats to validity

The first factor threatening the internal validity is the limited set of Excel end-users we used for the interviews. This affects our results of the problem analysis about spreadsheet versioning. We expect that if more than 6 users were interviewed, more user needs could eventually be identified. A similar factor threatening the internal validity is the limited set of users asked for the user test. This affects our evaluation results about correctness, completeness and the quality scores. We tested the prototype tool with 6 end-users, and that is not enough to draw statistically significant conclusions. However, the aim of the user tests was to obtain a general idea of how users would use the prototype tool, and to inventory their reactions. These goals have been achieved. We have done our best to perform the interviews and user tests with different spreadsheet user groups: companies, universities, and Excel experts, with varying spreadsheet experience. We did not have time and opportunity to perform more interviews and user tests. The goal of this research was not to perform a complete user study.

Another factor jeopardizing both internal and external validity are the datasets we used for the evaluation. We know that we have a bias in our set of unit tests because we created all the sample spreadsheets ourselves. Furthermore, we ran the unit tests continuously during development, so our implementation is adapted to these examples. For both reasons, we have evaluated the correctness, stability and performance on new, unseen and different datasets. We think there is enough diversity in the spreadsheet files, because we selected spreadsheets both from other studies and from practice. Some are even confidential files from large companies. In the end, the comparison approach is tested on a large number of different spreadsheet files. However, we are aware that new spreadsheets can always give unexpected results in the comparison.

7 Conclusion

The goal of this graduation project was to investigate the problem of comparing two spreadsheets. To that end we obtained new insights about spreadsheet versioning, and proposed theoretical and practical solutions to the spreadsheet comparison problem. This chapter wraps up the research. First, we give an overview of the contributions of this thesis in Section 7.1. Then we answer the research question and draw conclusions in Section 7.2. Finally, our suggestions for future research work are summarized in Section 7.3.

7.1 Contributions

The main contributions of this thesis are:

- An extensive problem analysis about spreadsheet versioning. Based on user interviews and development experience, six user needs and five technical challenges related to spreadsheet comparison are identified.
- A new pipeline-based approach for comparing two spreadsheet files. This includes methods for (1) analyzing the spreadsheet structure, (2) detecting changes in spreadsheets, and (3) aggregating changes into clear, understandable differences. Improvements like cell hashing, an optimized 2D alignment using longest common subsequences, and different algorithms for comparing worksheets, defined names, rows, columns and cells are part of this approach.
- A new prototype tool, demonstrating the proposed approach in practice. This tool, called CompareXL, is a stand-alone application that can compare two spreadsheet files. The user interface is built with a multi-level interaction design, showing the comparison results intuitively to the end-user.
- Empirical evidence that the proposed approach is correct and scalable, that it can handle a wide variety of spreadsheets, and that it meets users' needs.

7.2 Conclusions

In this thesis we explored the following research question:

> **How to develop an intelligent and applicable strategy for comparing two spreadsheet files?**

We have seen in Chapter 2 that there is only little literature available about the spreadsheet comparison problem, despite the fact that a lot of research has been conducted on regular file comparison. We built on research about the longest common subsequence algorithm and the RowColAlign algorithm. Other papers gave us hints on how to approach a file comparison problem in general. We have seen in the same chapter that there are, according to our knowledge, currently ten existing spreadsheet compare tools. However, they come with all sorts of limitations. None of them cover all aspects of a spreadsheet, most of the tools are designed with a specific use case in mind, and most are not able to handle all types of spreadsheet files, especially large and complex spreadsheets. We therefore concluded that there is yet no sufficient, simple and all-purpose spreadsheet compare tool available.

In Chapter 3 we have seen that end-users face many problems regarding spreadsheet versions. Spreadsheet users have needs related to overview, validation, completeness, error resolving, visualization, and evolution. A compare tool that supports end-users on these aspects will solve a lot of spreadsheet versioning issues. We consider a spreadsheet comparison strategy *applicable* if it provides solutions to these user needs. Furthermore, we found that fundamental challenges related to spreadsheet comparison include: change propagation, performance, 2D alignment, the grouping of data, and movement detection. We consider a spreadsheet comparison strategy *intelligent* if it provides solutions to these technical challenges.

In Chapter 4 we have presented our new comparison strategy. We discovered the following insights about spreadsheet comparison methods.

- One essential element in a comparison approach is passing on information from a structure analysis step to a change detection step. It is necessary to first analyze the structure correspondence between the two spreadsheet files. Without knowledge of the matching worksheets and inserted/deleted rows and columns, it is impossible to make a correct comparison. Because spreadsheets are two-dimensional by nature, the structure analysis should always include a 2D alignment method.
- A second essential element in a comparison approach is the ability to quickly zoom in on parts of the spreadsheet that are different. Because spreadsheets can easily become large and complex, it is required to optimize the change detection step such that irrelevant comparisons are avoided. Otherwise, it is impossible to efficiently detect changes in medium or large sized spreadsheets.
- The third essential element in a comparison approach is forwarding results from a change detection step to an aggregation method. It is natural to spreadsheet comparison that the outcome contains a long list of changes. Without aggregation, the comparison result can quickly become unclear and difficult to interpret. We propose a method to summarize the changes into differences based on user scenarios. The user scenarios represent actual user operations and are defined at all levels of detail in the spreadsheet. The final comparison result, therefore, provides information at different levels of abstraction.

We concluded this chapter with the observation that four user needs and four technical challenges are addressed by the proposed solutions. The proposed solutions are designed to be *intelligent* and *applicable*.

We have demonstrated the application of these solutions in our prototype tool, shown in Chapter 5. We conclude that it is challenging to design an intuitive user interface for presenting the comparison results. We found that users use the compare tool with different goals: some ask high-level questions and some ask detailed questions. Furthermore, the comparison of different spreadsheets can give many different outcomes. A user interface should be designed such that it is helpful in all these scenarios. Our prototype tool shows the comparison result on multiple levels of abstraction. It provides both a high-level overview and detailed views of all the changes. The user interface was designed such that it is *applicable* to the user needs. Besides the user interface, we have built a solid code base for our comparison engine. We conclude that it was effective to invest in structured code, unit tests, and performance optimizations. Moreover, it was relevant to test the comparison engine during the development process on real-world spreadsheets. Hereby, the quality of the implementation was greatly improved, and the examples gave us more understanding on how to develop *intelligent* solutions.

In Chapter 6 we have evaluated our comparison strategy on correctness, completeness, user needs, stability, and performance. All 100% of our 165 unit tests were passing. 94% of the modifications in the Enron Error Corpus were correctly detected. The comparison approach has a stability of 95% on 1973 new, unseen spreadsheets in the VEUSES, VFUSE and VEnron2 datasets. 69% of these spreadsheets are compared within 0–0.1 seconds, 24% within 0.1–1 seconds and 6% in more than 1 second. Our comparison approach survived a challenging performance test and is able to compare 81% of the most difficult spreadsheet files in our datasets, without memory issues. Finally, the result of the user test is an overall average quality score of 4.4 out of 5.0. One moderate score was given on completeness (3.67), high scores were given on solution (4.17), overview (4.33), user-friendly (4.50), detail (4.67), complexity (4.67), and summary (4.83). We can summarize the reaction of users in the user tests the best with a quote from an Excel expert at the end of his user test: *"The current compare tool really shows great promise, with a valuable overview of all spreadsheet differences. It proves that the spreadsheet comparison problem is actually solvable."*

7.3 Future work

We suggest the following promising ideas for future research or applications:

- **Visualization of changes.** Regular file comparison tools show their results in context. It will be helpful if the spreadsheet comparison result is also shown in context, for example by using an abstract worksheet map. Cells in the worksheet map can then be indicated with a color, for instance for no change (grey), inserted (green), value changed (yellow), formula changed (blue), and deleted (red). The visualization will immediately provide overview and insight in all worksheet changes, better than a list of all changes. This idea requires research on how to visualize a worksheet in abstract, and how to add the change information to the visualization.
- **Apply pattern recognition.** Currently, the comparison approach is built from the spreadsheet perspective. It is interesting to see if pattern recognition or machine learning methods can create better spreadsheet comparisons. For instance, an idea is to automatically detect general blocks of information (features) using a pattern recognition technique. Then the features of version 1 and 2 can be compared, instead of checking all individual spreadsheet aspects itself. More research is required on the application of pattern recognition or deep learning on spreadsheets.
- **Detect movements.** Currently, a copy/paste action of cells is reported as cell deletions and cell insertions. A method should be developed to report this type of change in terms of movements. One possibility is to extend the aggregation method, by building a matching algorithm on the contents of the deleted and inserted cells. Another possibility is to detect labeled blocks of information in version 1 and 2 (e.g. using pattern recognition) and to compare the positions of the blocks of information.
- **Detect more changes.** The current research focused on data, model and structure changes. We implemented a list of 28 different change types. Users find it relevant to compare more spreadsheet features, see for example Table 6.7. More categories can be defined, for example, display changes, protection changes, and visibility changes. More research is required on how to compare all these aspects and how to present them in an intuitive way.
- **Compare VBA code.** Beside all spreadsheet features, many spreadsheets contain macros with VBA code. Regular text comparison algorithms can be applied to compare two versions of VBA code.
- **Spreadsheet evolution.** This research focused on the comparison of two spreadsheet files. It is interesting to apply this comparison method to multiple versions of spreadsheets. More research is required on how to create an overview of all spreadsheet changes over time, and how to run a comparison method in batch over multiple spreadsheets, for example to validate certain properties.
- **Risk analysis.** Now that we have a list of changes available between two spreadsheet files, it is interesting to run a risk analysis on the changes. For example, if one formula is deleted in a column of 1000 cells with the exact same formula, this is a potential risk. A whole new research topic is how to identify risks based on spreadsheet change information.
- **Excel integration.** The current prototype is a stand-alone application that runs indepen-

dently of Excel. Although this has many advantages, it can be helpful if the comparison can be executed directly from within Excel. For instance, a 'compare' button can be added to compare the current version with a previous version, selected by the user. The results can be shown in the stand-alone tool so that the user can continue working on his spreadsheet with the comparison result next to it. Other possibilities of Excel integration are: indication of all changes inside Excel, add an extra worksheet with a log of all changes or export an Excel or PDF report of all changes. More research is required on the technical consequences and applicability in Excel.

- **Spreadsheet merging.** A challenging idea is to build a spreadsheet merge tool. As a result, spreadsheets can be put under version control using existing version control systems, like Git. An important requirement is that the spreadsheet comparison must generate a complete diff between the two files. Most version control systems only use insertions and additions, so probably the change detection method must be rewritten. Furthermore, a method must be developed on how to deal with merge conflicts. We have seen that the list of all spreadsheet features available in Excel is very extensive. The biggest research question on this topic is therefore: how can we generate a complete diff between two spreadsheet files?

Finally, this project is continued at the company of Infotron with further development of the prototype tool CompareXL. The plan is to soon release a finished version of the spreadsheet compare tool, available to all Excel end-users.

A

Sample Spreadsheet File

```
Example v1.xlsx
├── [Content_Types].xml
├── docProps
│   ├── app.xml
│   └── core.xml
├── xl
│   ├── calcChain.xml
│   ├── sharedStrings.xml
│   ├── styles.xml
│   ├── workbook.xml
│   ├── theme
│   │   └── theme1.xml
│   ├── worksheets
│   │   ├── sheet1.xml
│   │   ├── sheet2.xml
│   │   └── sheet3.xml
│   └── _rels
│       └── workbook.xml.rels
└── _rels
    └── .rels
```

workbook.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<workbook xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
  xmlns:x15="http://schemas.microsoft.com/office/spreadsheetml/2010/11/main"
  xmlns:xr="http://schemas.microsoft.com/office/spreadsheetml/2014/revision"
  xmlns:xr10="http://schemas.microsoft.com/office/spreadsheetml/2016/revision10"
  xmlns:xr2="http://schemas.microsoft.com/office/spreadsheetml/2015/revision2"
  xmlns:xr6="http://schemas.microsoft.com/office/spreadsheetml/2016/revision6"
  mc:Ignorable="x15 xr xr6 xr10 xr2">
  <fileVersion appName="xl" lastEdited="7" lowestEdited="7" rupBuild="20325" />
  <workbookPr defaultThemeVersion="166925" />
  <mc:AlternateContent>
    <mc:Choice Requires="x15">
      <x15ac:absPath xmlns:x15ac="http://schemas.microsoft.com/office/spreadsheetml/2010/11/ac"
           url="D:/CloudStation/TU Delft/Afstuderen/Testsheets/" />
    </mc:Choice>
  </mc:AlternateContent>
  <xr:revisionPtr revIDLastSave="0" documentId="10_ncr:8100000_{DE330149-2F17-454D-8AC1-3E945D0E11FD}"
   xr6:coauthVersionLast="34" xr6:coauthVersionMax="34" xr10:uidLastSave="{00000000-0000-0000-0000-000000000000
  <bookViews>
    <workbookView xWindow="0" yWindow="0" windowWidth="28800" windowHeight="12225"
         xr2:uid="{5550BFEE-4B94-45EA-A11A-0A14ABC372C1}" />
  </bookViews>
  <sheets>
    <sheet name="Blad1" sheetId="1" r:id="rId1" />
    <sheet name="Blad2" sheetId="2" r:id="rId2" />
    <sheet name="Blad3" sheetId="3" r:id="rId3" />
  </sheets>
  <calcPr calcId="162913" />
  <extLst>
    <ext uri="{140A7094-0E35-4892-8432-C4D2E57EDEB5}">
      <x15:workbookPr chartTrackingRefBase="1" />
    </ext>
  </extLst>
</workbook>
```

sheet1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<worksheet xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
  xmlns:x14ac="http://schemas.microsoft.com/office/spreadsheetml/2009/9/ac"
  xmlns:xr="http://schemas.microsoft.com/office/spreadsheetml/2014/revision"
  xmlns:xr2="http://schemas.microsoft.com/office/spreadsheetml/2015/revision2"
  xmlns:xr3="http://schemas.microsoft.com/office/spreadsheetml/2016/revision3"
  mc:Ignorable="x14ac xr xr2 xr3" xr:uid="{7B26C0A5-59D8-40D8-96C4-4658D8877606}">
  <dimension ref="A1:F12" />
  <sheetViews><sheetView tabSelected="1" workbookViewId="0" /></sheetViews>
  <sheetFormatPr defaultRowHeight="15" x14ac:dyDescent="0.25" />
  <sheetData>
    <row r="1" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A1" t="s"><v>0</v></c>
      <c r="B1"><v>1</v></c>
      <c r="C1"><f>B1*10</f><v>10</v></c>
      <c r="E1" t="s"><v>47</v></c>
      <c r="F1" t="s"><v>52</v></c>
    </row>
    <row r="2" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A2" t="s"><v>1</v></c>
      <c r="B2"><v>2</v></c>
      <c r="C2"><f t="shared" ref="C2:C10" si="0">B2*10</f><v>20</v></c>
      <c r="E2" t="s"><v>48</v></c>
      <c r="F2" t="s"><v>53</v></c>
    </row>
    <row r="3" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A3" t="s"><v>2</v></c>
      <c r="B3"><v>3</v></c>
      <c r="C3"><f t="shared" si="0" /><v>30</v></c>
      <c r="E3" t="s"><v>49</v></c>
      <c r="F3" t="s"><v>54</v></c>
    </row>
    <row r="4" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A4" t="s"><v>3</v></c>
      <c r="B4"><v>4</v></c>
      <c r="C4"><f t="shared" si="0" /><v>40</v></c>
      <c r="E4" t="s"><v>50</v></c>
      <c r="F4" t="s"><v>55</v></c>
    </row>
    <row r="5" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A5" t="s"><v>4</v></c>
      <c r="B5"><v>5</v></c>
      <c r="C5"><f t="shared" si="0" /><v>50</v></c>
      <c r="E5" t="s"><v>51</v></c>
      <c r="F5" t="s"><v>56</v></c>
    </row>
    <row r="6" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A6" t="s"><v>5</v></c>
      <c r="B6"><v>6</v></c>
      <c r="C6"><f t="shared" si="0" /><v>60</v></c>
    </row>
    <row r="7" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A7" t="s"><v>6</v></c>
      <c r="B7"><v>7</v></c>
      <c r="C7"><f t="shared" si="0" /><v>70</v></c>
    </row>
    <row r="8" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A8" t="s"><v>7</v></c>
      <c r="B8"><v>8</v></c>
      <c r="C8"><f t="shared" si="0" /><v>80</v></c>
    </row>
    <row r="9" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A9" t="s"><v>8</v></c>
      <c r="B9"><v>9</v></c>
      <c r="C9"><f t="shared" si="0" /><v>90</v></c>
    </row>
    <row r="10" spans="1:6" x14ac:dyDescent="0.25">
      <c r="A10" t="s"><v>9</v></c>
      <c r="B10"><v>10</v></c>
      <c r="C10"><f t="shared" si="0" /><v>100</v></c>
    </row>
    <row r="12" spans="1:6" x14ac:dyDescent="0.25">
      <c r="B12"><f>SUM(B1:B10)</f><v>55</v></c>
      <c r="C12"><f>SUM(C1:C10)</f><v>550</v></c>
    </row>
  </sheetData>
  <pageMargins left="0.7" right="0.7" top="0.75" bottom="0.75" header="0.3" footer="0.3" />
</worksheet>
```

B

Spreadsheet Edit Operations

#	Name	Scope	Category
1	Workbook made protected	Workbook	Protection
2	Workbook made unprotected	Workbook	Protection
3	Workbook styles changed	Workbook	Display
4	Document properties changed	Workbook	Miscellaneous
5	Auto calculate option changed	Workbook	Miscellaneous
6	VBA code changed	Workbook	Miscellaneous
7	External data connection added	Workbook	Data
8	External data connection changed	Workbook	Data
9	External data connection deleted	Workbook	Data
10	Worksheet created	Workbook	Structure
11	Worksheet deleted	Workbook	Structure
12	Worksheet name changed	Workbook	Structure
13	Worksheet order changed	Workbook	Display
14	Worksheet made hidden	Workbook	Display
15	Worksheet made very hidden	Workbook	Display
16	Worksheet made visible	Workbook	Display
17	Worksheet made protected	Workbook	Protection
18	Worksheet made unprotected	Workbook	Protection
19	Worksheet protection changed	Workbook	Protection
20	Chart added	Worksheet	Display
21	Chart moved	Worksheet	Display
22	Chart changed	Worksheet	Display
23	Chart deleted	Worksheet	Display
24	Conditional formatting rule added	Worksheet	Display
25	Conditional formatting rule changed	Worksheet	Display
26	Conditional formatting rule deleted	Worksheet	Display
27	Data validation rule added	Worksheet	Display
28	Data validation rule changed	Worksheet	Display

#	Name	Scope	Category
29	Data validation rule deleted	Worksheet	Display
30	Embedded object added	Worksheet	Display
31	Embedded object changed	Worksheet	Display
32	Embedded object deleted	Worksheet	Display
33	Filter activated	Worksheet	Display
34	Filter deactivated	Worksheet	Display
35	Filter changed	Worksheet	Display
36	Header/footer added	Worksheet	Display
37	Header/footer changed	Worksheet	Display
38	Header/footer deleted	Worksheet	Display
39	Ignored error added	Worksheet	Display
40	Ignored error deleted	Worksheet	Display
41	Named range added	Worksheet	Model
42	Named range changed	Worksheet	Model
43	Named range deleted	Worksheet	Model
44	Page break added	Worksheet	Display
45	Page break changed	Worksheet	Display
46	Page break deleted	Worksheet	Display
47	Picture added	Worksheet	Display
48	Picture moved	Worksheet	Display
49	Picture deleted	Worksheet	Display
50	Pivot table added	Worksheet	Display
51	Pivot table changed	Worksheet	Display
52	Pivot table deleted	Worksheet	Display
53	Print options changed	Worksheet	Display
54	Protected range added	Worksheet	Protection
55	Protected range deleted	Worksheet	Protection
56	Sort state activated	Worksheet	Display
57	Sort state deactivated	Worksheet	Display
58	Sort state changed	Worksheet	Display
59	Tab color changed	Worksheet	Display
60	Table added	Worksheet	Display
61	Table changed	Worksheet	Display
62	Table deleted	Worksheet	Display
63	View options changed	Worksheet	Display
64	Column added	Column	Structure
65	Column moved	Column	Structure
66	Column deleted	Column	Structure
67	Column made hidden	Column	Display
68	Column made visible	Column	Display
69	Column width changed	Column	Display
70	Row added	Row	Structure
71	Row moved	Row	Structure

#	Name	Scope	Category
72	Row deleted	Row	Structure
73	Row made hidden	Row	Display
74	Row made visible	Row	Display
75	Row height changed	Row	Display
76	Row auto fit changed	Row	Display
77	Value added	Cell	Data
78	Value changed	Cell	Data
79	Value deleted	Cell	Data
80	Formula added	Cell	Model
81	Formula changed	Cell	Model
82	Formula deleted	Cell	Model
83	Cell moved	Cell	Structure
84	Cells merged	Cell	Display
85	Cells unmerged	Cell	Display
86	Hyperlink added	Cell	Display
87	Hyperlink changed	Cell	Display
88	Hyperlink deleted	Cell	Display
89	Comment added	Cell	Display
90	Comment changed	Cell	Display
91	Comment deleted	Cell	Display
92	Inline style changed	Cell	Display
93	Custom style set	Cell	Display
94	Default style set	Cell	Display
95	Style changed	Cell	Display
96	Borders changed	Cell details	Display
97	Background changed	Cell details	Display
98	Font changed	Cell details	Display
99	Formula made hidden	Cell details	Display
100	Formula made visible	Cell details	Display
101	Horizontal alignment changed	Cell details	Display
102	Indentation changed	Cell details	Display
103	Text made vertical	Cell details	Display
104	Text made horizontal	Cell details	Display
105	Cell locked	Cell details	Display
106	Cell unlocked	Cell details	Display
107	Style preset changed	Cell details	Display
108	Number format changed	Cell details	Display
109	Text rotation changed	Cell details	Display
110	Shrink to fit changed	Cell details	Display
111	Vertical alignment changed	Cell details	Display
112	Wrap text changed	Cell details	Display

Bibliography

[1] J. W. Hunt and M. D. Mcilroy, “An Algorithm for Differential File Comparison,” Bell Laboratories Computing Science, Tech. Rep. 41, jul 1976.

[2] W. Miller and E. W. Myers, “A File Comparison Program,” *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

[3] W. F. Tichy, “RCS – A System for Version Control,” *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[4] F. F. J. Hermans, “Analyzing and Visualizing Spreadsheets,” Ph.D. dissertation, Delft University of Technology, 2013.

[5] D. E. O’Leary, “Expert System Prototyping as a Research Tool,” *Applied Expert Systems, North-Holland, Amsterdam*, pp. 17–32, 1988.

[6] L. Bergroth, H. Hakonen, and T. Raita, “A Survey of Longest Common Subsequence Algorithms,” in *Proceedings of the Seventh International Symposium on String Processing and Information Retrieval*, ser. SPIRE ’00. IEEE, 2000, pp. 39–48.

[7] L. Bergroth, “Utilizing Dynamically Updated Estimates in Solving the Longest Common Subsequence Problem,” in *Proceedings of the 12th International Conference on String Processing and Information Retrieval*, ser. SPIRE ’05. Springer, 2005, pp. 301–314.

[8] R. A. Wagner and M. J. Fischer, “The String-to-String Correction Problem,” *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[9] K. Bringmann and M. Künnemann, “Multivariate Fine-Grained Complexity of Longest Common Subsequence,” in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2018, pp. 1216–1235.

[10] J. W. Hunt and T. G. Szymanski, “A Fast Algorithm for Computing Longest Common Subsequences,” *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[11] S. Kuo and G. R. Cross, “An Improved Algorithm to Find the Length of the Longest Common Subsequence of Two Strings,” in *ACM Sigir Forum*, vol. 23, no. 3-4. ACM, 1989, pp. 89–99.

[12] S. Wu, U. Manber, G. Myers, and W. Miller, “An O(NP) Sequence Comparison Algorithm,” *Information Processing Letters*, vol. 35, no. 6, pp. 317–323, 1990.

[13] C. Scaffidi, M. Shaw, and B. Myers, “Estimating the Numbers of End Users and End User Programmers,” in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’05)*. IEEE, 2005, pp. 207–214.

[14] J. Smith, J. A. Middleton, and N. A. Kraft, “Spreadsheet Practices and Challenges in a Large Multinational Conglomerate,” in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2017, pp. 155–163.

[15] S. Roy, F. Hermans, and A. van Deursen, “Spreadsheet Testing in Practice,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 338–348.

[16] B. Jansen, F. Hermans, and E. Tazelaar, “Detecting and Predicting Evolution in Spreadsheets: A Case Study in an Energy Network Company,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 645–654.

[17] L. Xu, W. Dou, J. Zhu, C. Gao, J. Wei, and T. Huang, “How Are Spreadsheet Templates Used in Practice: A Case Study on Enron,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 734–738.

[18] W. Dou, L. Xu, S.-C. Cheung, C. Gao, J. Wei, and T. Huang, “VEnron: A Versioned Spreadsheet Corpus and Related Evolution Analysis,” in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 162–171.

[19] L. Xu, W. Dou, C. Gao, J. Wang, J. Wei, H. Zhong, and T. Huang, “SpreadCluster: Recovering Versioned Spreadsheets through Similarity-Based Clustering,” in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 158–169.

[20] C. Chambers, M. Erwig, and M. Luckey, “SheetDiff: A Tool for Identifying Changes in Spreadsheets,” in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2010, pp. 85–92.

[21] A. Harutyunyan, G. Borradaile, C. Chambers, and C. Scaffidi, “Planted-model evaluation of algorithms for identifying differences between spreadsheets,” in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2012, pp. 7–14.

[22] T. Schmitz and D. Jannach, “Finding Errors in the Enron Spreadsheet Corpus,” in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2016, pp. 157–161.

[23] R. Moreira, “SheetGit: A Tool for Collaborative Spreadsheet Development,” in *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer, 2016, pp. 415–420.

[24] F. Hermans, M. Pinzger, and A. van Deursen, “Detecting Code Smells in Spreadsheet Formulas,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 409–418.

[25] B. Shneiderman, “The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations,” in *Proceedings 1996 IEEE Symposium on Visual Languages*. IEEE, 1996, pp. 336–343.

[26] M. Fisher and G. Rothermel, "The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[27] T. Barik, K. Lubick, J. Smith, J. Slankas, and E. Murphy-Hill, "Fuse: A Reproducible, Extendable, Internet-Scale Corpus of Spreadsheets," in *Proceedings of the 12th Working Conference on Mining Software Repositories.* IEEE Press, 2015, pp. 486–489.

[28] B. Klimt and Y. Yang, "The Enron Corpus: A New Dataset for Email Classification Research," in *European Conference on Machine Learning.* Springer, 2004, pp. 217–226.